Report

Dan Nash
40217045@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

1 Introduction

We were assigned to digitally recreate the game Checkers/Draughts, but with a focus on what algorithms and data structures were used to create the game, and for us to justify our use of them, within the context of the project. A list of mandatory features that were highlighted, to be implemented for the game are as follows; A game board with moveable and playable checker pieces which represents both team sides (White and Black), in which they can be played between two human players. Additionally, being able to state/calculate (within the code) the positions of what checker piece is where on the game board.

Once previously the stated functionality was finished, we could implement the following additional features to expand the project even further, for additional marks; An AI player which could compete against another AI or a human player. Extra credit for if the AI is an effective player. An Undo and Redo feature which enables the player to either undo a movement of a checker piece (for example if a mistake was made) and also being able to redo any recorded moves. And finally, a replay function which records the play history (which piece moves where) of the game, to which you can then watch and review.

2 Design

2.1 Overview

The language that was chosen was C-Sharp 7.1 witch would operate within the Unity Engine, version 5.0 and beyond. Reason for different builds of Unity being used was due to different machines unanimously not having the same version installed, between one another. But all version variants were 5.0 and above.

The features which have been implemented are all the basic ones required which were stated in the "Introduction". Unfortunately, Undo/Redo, Replay and an AI were not developed for the final build of the project.

2.2 Checker Pieces

Each checker piece corresponds with the class "Piece", which is an 8 by 8, 2D array, which equates to 64 coordinates. Which just so happen to be the same amount of tiles that is on a standard Checker Board. So, this is the data structure the checker piece's positions are stored. Within this class there are two methods which dictates how the piece moves around the board. They are known as "ForcedMove()" and "ValidMove()".

The method ForcedMove() enforces the player, that it has to take over and remove the opposing teams piece. This is a game rule which is triggered when the opposing player's checker piece, can be removed from play by one of your pieces. It essentially denies you as the player, to ignore an attack opportunity, which may actually put you in a vulnerable position, after killing the oppositions piece. The code below shows the "ForcedMove()" method in relation to if the checker piece is a member of the "White" team or is a "King" piece;

Listing 1: ForcedMove()

```
1  public bool ForcedMove(Piece[,] board, int x, int y)
2      {
3          if (isWhite || isKing)
4          {
5              // Diagonaly Top Left
6              if (x >= 2 && y <= 5)
7              {
8                  Piece p = board[x - 1, y + 1];
9                  if (p != null && p.isWhite != isWhite)
10                 {
11                     if (board[x - 2, y + 2] == null)
12                         return true;
13                 }
14             }
15
16             // Diagonaly Top Right
17             if (x <= 5 && y <= 5)
18             {
19                 Piece p = board[x + 1, y + 1];
20                 if (p != null && p.isWhite != isWhite)
21                 {
22                     if (board[x + 2, y + 2] == null)
23                         return true;
24                 }
25             }
26         }
27 }
28
```

As seen from line 6 of the code above, the method checks the range of the array index via the x and y coordinates. It then signals out that a forced move can only be made if x and y are within the acceptable range. The range is dependant on which direction the checker piece is moving. For instance, if the piece was moving diagonally top left, it would have to check if the piece's x coordinate was within 2 to 7 (of the array) and it's y coordinate within 0 to 5. This is because if this range was not stated, the checker piece could force jump out side of the 2D array's index, in which leads to the game breaking. This exact bug occurred during development and the solution to this problem is stated in section **4 Personal Evaluation**. The referenced tutorial here[1], gave insight to the author, on how to achieve this.

Another function this method provides is checking if there is is an empty coordinate when the possibility of "double jump" is being checked and if so, enable the player to move to said position. So by looking at code line 6 to 12 (Diagonally Top Left), it identifies that if you were to move in that direction, it would be - 1 to the x coordinate index (move to the left by 1) and then + 1 to the y coordinate index (move up by 1), which would essentially place the checker diagonally top left from the previous position on the board. But what it also does is it checks if that coordinate is either empty (null) or taken by another checker piece, but if that checker piece is not the same colour as the checker piece that's going to move and there is an available coordinate (not null) diagonally top left of the opposite team's checker piece, then the player is able to do a double jump in that direction. The opponents checker piece is not "destroyed" in this segment of code, but done in the "TryMove()" method within the "CheckerBoard" class.

The method ValidMove() checks on the board, if the move you are trying to do is legal within the game rules. For instance, denies player moving a piece on top of another piece, regardless of which team it is on, which is states on code line 4 to 5. And enabling a "double jump", which is invoked when legal to do so;

Listing 2: ValidMove()

```
1   public bool ValidMove(Piece[,] board, int x1, int y1, int x2, int y2)
2   {
3       // If you are moving on top of another piece, then deny that move
4       if (board[x2, y2] != null)
5           return false;
6       // Keep track of the number of tiles that have been jumped via x axis
7       int jumpedTilesX = Mathf.Abs(x1 - x2);
8       // Keep track of the number of tiles that have been jumped via y axis
9       int jumpedTilesY = y2 - y1;
10      // If jump 1 tile, it's a move. If jumped two tiles, it's a kill!
11
12      // WHITE TEAM MOVE SET
13      if (isWhite || isKing)
14      {
15          if (jumpedTilesX == 1)
16          {
17              if (jumpedTilesY == 1)
18                  return true;
19          }
20          else if (jumpedTilesX == 2)
21          {
22              if (jumpedTilesY == 2)
23              {
24                  Piece p = board[(x1 + x2) / 2, (y1 + y2) / 2];
25  // If piece is jumping and it's not null AND NOT over the same
26  // colour as our piece allowed to jump over by two tiles.
27                  if (p != null && p.isWhite != isWhite)
28                      return true;
29              }
30          }
31      }
32
```

The referenced tutorial here[2] gave insight on how this segment code could be achieved.

2.3 Players

There are two teams in the game. One team is White and the other team is Black. The White team is created by making it a public bool known as "isWHite", but the Black team is never truly defined. It is only ever defined as "!isWhite", which means is not White. There would of hardly been any design difference if Mr.Nash had decided to define the Black team as it's own boolean type.

There is also another boolean type which is defined as "isKing". This enables a checker piece to become a "king" piece and inherits the rules/move sets from both teams. How a checker piece is turned/promoted into a king, is demonstrated in the following code, which is a segment of the "EndTurn()" method, that's located in the "CheckerBoard" class.

Listing 3: EndTurn()

```
1   // Promotion to King!
2       // If white piece gets to the top of the board (7) then it becomes a "King" piece
3       if (selectedPiece != null)
4       {
5           if (selectedPiece.isWhite && !selectedPiece.isKing && y == 7)
6           {
7               selectedPiece.isKing = true;
8               selectedPiece.transform.Rotate(Vector3.right * 180);
9           }
10          // If black piece gets to the bottom of the board (0) then it becomes a "King" piece
11          else if (!selectedPiece.isWhite && !selectedPiece.isKing && y == 0)
12          {
13              selectedPiece.isKing = true;
14              selectedPiece.transform.Rotate(Vector3.right * 180);
15          }
16      }
17
```

This algorithm is done quite simply by first off checking if the selected checker piece is on y-7 coordinate (top of the board) and if the selected piece is a member of the White team. If both of those conditions are met then that selected piece is no longer "false" but is now "true". This is in regards to the "isKing" boolean type. That piece is then transformed (a function which interacts to the "WhitePIece" Game Object stated by Unity) into a king, which in return inherits the "Black" team move sets. same algorithm can be applied to a Black team checker piece, but swapping some of the values. For example changing the y coordinate position to 0 and stating the selected piece is Not (!isWHite) a member of the White team.

2.4 Game Board

The game board uses the same 2D Array as the class "Piece" does. From the coordinate positions of the individual checker pieces, it can position and generate where these exact GameObjects are placed, within the board's environment and it can identify where to generate the white and black team piece's.

Traditionally in regards to team placement, the White team is designated on the bottom 3 rows of the board, ranging from the y coordinates of 0-2. And the Black team is placed within the top 3 rows of the board, within the y coordinate range of 5-7. The following code shows how this was done within the "GenerateBoard()" method;

Listing 4: EndTurn()

```
1 // Generate White team, sets at bottom three rows of the board
2       for(int up2Down = 0; up2Down < 3; up2Down++)
3       {
4           // Determines if it's an odd row or not.
5           bool oddRow = (up2Down % 2 == 0);
6 // Makes pieces generate from left to right, going across the ↩
        bottom three rows of the board
7           for (int left2Right = 0; left2Right < 8; left2Right += 2)
8           {
9               GeneratePiece((oddRow)? left2Right : left2Right +1, ↩
        up2Down);
10          }
11      }
12
```

The algorithm here is for the placement of the White team checker pieces. It begins with a "For" loop where the condition ends when the loop reaches row 3 (y ["up2Down"] coordinate 2 of the array) on the checker board. While this For loop is incrementing towards row 3, a new boolean type is created which is named "oddRow". oddRow is assigned to a modulo operator (code line 5) which says that when int "up2Down" remainder (after dividing its first operand by its second) is equal to 0, oddRow condition is then "true". Then another For loop is made which goes from x array index 0 to 7 (int left2Right), and states that at coordinates 1, 3, 5, 7 (odd numbers), no checker piece will be generated, unless if the condition of an oddRow is met. So what ends up happening for the White team is, on the bottom (y = 0), the White pieces are generated on the following x coordinates; 0, 2, 4, 6. Which are all even numbers. On the second row (y = 1) however, the oddRow boolean condition is met and so pushes the generation of pieces by + 1 on the x coordinates (left2Right). This can be seen on line 9 from the code above. The third row (y = 2) generates the pieces the same as the first row (y = 0) and then the first instance of the For loop algorithm has finished it's cycle by incrementing the integer up2Down (y coordinates) to less than 3 (4th row). This is stated on code line 2.

In regards to how the Black team is generated, it is the same algorithm stated above, but with slight tweaks to the values of the first For loop, to generate pieces on the Black teams side of up board. Which are row 6 to 8 (y = 5 to y = 7). The code as follows shows this;

Listing 5: EndTurn()

```
1   // Generate Black team, sets at top three rows of the board
2   for (int up2Down = 7; up2Down > 4; up2Down−−)
3
```

As can be seen, instead of incrementing up from 0 to 2, as the White team does. This For loop decrements from row 8 (y = 7) to row 6 (y = 5). The rest of the code after the initial For loop is identical to the White team placement code.

2.5 Generating Checker Piece

In order for the board to be generated, first there must be code in place to dictate which "GameObject" is identified as as member of the White or Black team. The code below shows the process of this algorithm within the method "GeneratePiece()".

Listing 6: GeneratePiece()

```
1   // Associates int "left2Right" and "up2Down" to GameObject'↩
      s
2   private void GeneratePiece(int left2Right, int up2Down)
3   {
4       // States that if int up2Down is > 3; Then it's black team. If ↩
        less: White team. Use Ternary operator
5       bool isPieceWhite = (up2Down > 3) ? false : true;
6       // Ternary operator, if "isPieceWhite is white, then spawn "↩
        whitePiecePreFab". Else; Spawn "blackPiecePrefab"
7       GameObject go = Instantiate((isPieceWhite) ? ↩
        whitePiecePrefab : blackPiecePrefab) as GameObject;
8       go.transform.SetParent(transform);
9       Piece p = go.GetComponent<Piece>();
10
```

GeneratePiece() uses the same integers that GenerateBoard() uses, which are "left2Right" (x coordinates) and "up2Down" (y coordinates). A bool called "isPieceWhite" is then made which uses a ternary operator to sate if it's set to true or false, depending on the previously stated condition. The condition is stated on line 5 of the code "(up2Down > 3)". This means that if the integer up2Down is greater than 3 (y > 3) then the boolean is set to false. But if it's less than 3 (y < 3), then the boolean is set to true, which identifies that the checker piece is in fact White, not Black. Afterwards from code line 7, it "Instantiates" (Clones the object original and returns the clone[3]) the GameObject known as "go", with another ternary operator. But this time the condition being the previously stated boolean; "isPieceWhite". This then means it can define which GameObject belongs to the White team ("whitePiecePreFab") and Black team ("blackPiecePrefab") respectively. And with this being defined, the algorithm can then "transform" the GameObjects to the correct team's art assets.

Finally from line 9 of the code, the data type "Piece" (which is the class that creates the 2D array) is assigned to a local variable known as "p". "p" is then defined as the GameObject "go". "go" is then attached to the GetComponent function, which lets the GameObject "gain access to a different script[4] that outside of the current script that's being used i.e CheckerBoard. GetComponent is attached to the script/class called "Piece", which essentially means that the GameObjects are now attached to that class and so all the pieces of the class share the properties of what the GameObject inherits. For example what art assets or prefab's individual checker pieces get, dependent on board placement.

2.6 Scan For Move

The method "ScanForMove()" uses a List data structure with the class "Piece" as it's data type (instead of a generic like, int or string). This List is defined as "forcedPieces" and this is where any time a checker piece invokes the "ForcedMove()" method (which is situated inside the class "Piece", hence why the List's data type is "Piece"), then that instance is added to the List. The reason why a List was used instead of an Array is because the size of said list is not fixed, but in fact dynamic. So when it comes to adding and removing data to the List, it's less expensive than it would be, if it was an Array[5]. The forcedPieces List comes into play in various situations around the "CheckerBoard" script, like in the "SelectPiece()" method. The following code shows how the

List forcedPieces is used;

Listing 7: SelectPiece()

```
1    // Checks if there is no forced move rule that needs to be acted
2        if (forcedPieces.Count == 0)
3        {
4            selectedPiece = p;
5            startDrag = mouseOver;
6        }
7
```

Here when the player selects a piece, it checks the board if there is a ForcedMove() to be made, by checking the count (number of instances) of the forcedPieces List. If this list is empty, i.e 0; then the the piece that the player has their mouse cursor ("mouseOver"), can be selected. If the count was not 0 though, then it means the player is not allowed to play the piece their mouse cursor is over.

The following code shows how the algorithm within the "ScanForMove()" method works;

Listing 8: ScanForMove()

```
1    private List<Piece> ScanForMove()
2    {
3        forcedPieces = new List<Piece>();
4
5        // Left to Right of the 2D array
6        for (int dimensionX = 0; dimensionX < 8; dimensionX++)
7            // Up to Down of the 2D array
8            for (int dimensionY = 0; dimensionY < 8; dimensionY++)
9                // Checks X & Y cordinates on board (pieces), if there is a piece (!= null) and the piece is "White". Then it's your turn
10               if (pieces[dimensionX, dimensionY] != null && pieces[dimensionX, dimensionY].isWhite == isWhiteTurn)
11                   // Check if move has to be forced.
12                   if (pieces[dimensionX, dimensionY].ForcedMove(pieces, dimensionX, dimensionY))
13                       forcedPieces.Add(pieces[dimensionX, dimensionY]);
14
15       return forcedPieces;
16   }
17
```

The algorithm begins by creating a new instance of the List "Piece", which is tied to forcedPieces. Then a a standard For loop with a nested For loop is used to scan the x and y coordinates of the checker board 8 x 8 (64 tiles) environment. Very similar beginnings to the "EndTurn()" method, mentioned in section **2.4 GameBoard**. The "If" statement on code line 10 is a set of conditions to check if that the index of the 2D array is not equal to "null", (checker piece on that x and y coordinate) **AND** if the piece is a member of the White team, when it's the White team's turn to play, follow on to the next "If" statement on code line 12. Here it checks the pieces on the board and compares it to the "ForcedMove()" method found in section **2.2 Checker Pieces** and if there is any forced moves to be made, it is then added on to the list forcedPieces. Which as seen in the "SelectPiece()" that stored data can be used appropriately, when needed.

2.7 Remove Checker Piece

In the "TryMove()" method, there are various functions and algorithms taken place which achieve different objectives. But for this section of the report, we'll look into how a checker piece (GameObject) is removed from play. The following code demonstrates this;

Listing 9: TryMove()

```
1    // Check if it's a valid move by refering to the class "Piece" via "pieces"
2        if (selectedPiece.ValidMove(pieces, x1, y1, x2, y2))
3        {
4            // Did we kill?
5            // If this is a jump
6            if (Mathf.Abs(x2 - x1) == 2)
7            {
8                Piece p = pieces[(x1 + x2) / 2, (y1 + y2) / 2];
9                if (p != null)
10               {
11                   //Destorys piece that has been jumped over.
12                   pieces[(x1 + x2) / 2, (y1 + y2) / 2] = null;
13                   DestroyImmediate(p.gameObject);
14                   hasKilled = true;
15               }
16           }
17
```

The line of code at 2 is an "If" statement that takes in the parameters of the piece that's been selected, it's x and y coordinates that it is and was at, and finally checks against the "ValidMove()" method logic. Then at line 6 of the code, there is a nested "If" which checks that when there is a viable jump (by checking when x2 - x1 is == 2), it then does the required logic ([(x1 + x2) / 2, (y1 + y2) / 2]) which tells the code to check the coordinates just past the opposing teams piece they are about to jump over. Then on the final nested "If" statement, it states that the checker piece (which would be on the opposing team) that has just been jumped over, is immediately removed (DestroyImmmediate) from the board. A boolean for if the players team has removed a piece from play is then set to "true".

3 Enhancements

When comparing the end product of the project to the list of mandatory and non-mandatory features, Mr. Nash was able to achieve all mandatory features. But was unable to achieve the following optional features; Undo and Redo, Replay and an AI player. There are also two UI features that the author would have liked to have been included, which are discussed in section **3.4 UI**.

3.1 Undo and Redo

Mr. Nash would have created the undo and redo functionality with two development processes which would compliment each other. Firstly, he would redesign the project while implementing the "Command" pattern for it's architecture and secondly, he would have implemented two Stacks (data structures), with one for storing Undo movement actions and the other for Redo movement actions.

The "Command Pattern is a 'refied method call', what I mean is that's it's a method call wrapped in an object."[6], which is extremely useful for undo and redo functionality because it encapsulates the action (concept) and turns it into tangible data or object as you will. Which in return you can "stick into a variable, pass to a function, etc."[6].

Which is very convenient for when you want to store the location of a checker piece, lets say into a "Stack" because "the input handling code will be creating an instance of this every time the player chooses a move" [7], to which can then be stored in said "Undo" Stack. The Author has also noted that two of the advantages of using the command pattern are "Its maintainability is good and does not hold any redundant information and It's not memory intensive"[8], unlike other similar patterns like the memento which is "memory intensive" [9]

Stacks have "a 'First In Last Out' or LIFO, structure" so, they are great for storing a list of data which needs to be displayed in order of occurrence. The reason for creating two Stacks for the Undo and Redo (instead of just one) would be for the sole purpose of keeping the list of indexes, clean and robust. The Undo Stack would store any previous moves that were made while the Redo Stack would contain the move index's that the player has just "pop()" (a method to which removes the top/latest index of the stack), off of the Undo Stack. This enables the player to then actually revisit/reapply those moves again, if they decide they are happy with their original movement choice.

3.2 Replay

If the Undo and Redo functionality was implemented with regards to the command pattern, then Mr. Nash believes that creating a "Replay" function would have been simple enough because he could have stored all the move data (which was encapsulated to an object due to the command pattern) to a simple data structure like a List. Then the player could play a game, then at any point, replay the game from the very first instance, and review all moves that were made, as the pointer index goes from the very beginning of the List, to the very last (which would be the most recent player input) entry of said List. But unfortunately without the Undo and Redo functionality Mr. Nash wanted to design, this vision of a Replay system was not able to be achieved.

3.3 AI

A simple AI which is able to scan for available moves and then choose one at random would have been suffice. As long as it complies with the "ForcedMove()" methods that a regular human player has to abide too. But due to time constraints, was not able to develop this form of functionality. If given more time, Mr. Nash could have maybe established a more complex AI with a robust set of rules/behaviours like trying to get a checker piece to reach the opposite side of the board within as little steps as possible; Or actively try to manoeuvre it's checker pieces around the oppositions pieces, to avoid losing a piece. Of course there are more complex AI solutions like "Machine Learning", but that would have be no small feat to achieve.

3.4 UI

When the player wins the game as either White or Black team, a win condition is set, but it isn't visually presented to the player. Instead it's stated in a Debug.Log that can only be seen in Unity's inspector window. Which is not beneficial to the player at all. If given more time, Mr. Nash would have liked to have a UI (User Interface) element like a Text box or a graphical canvas, informing the player that they have won or lost, dependent on the outcome of the game. Finally in regards to the UI, Mr. Nash felt that having a grey backdrop (behind and around the checker board) was a rather dull and uninteresting visual stimuli for the player. Instead he wished to have a wood, tabletop-esque texture mesh, which will have been applied to the whole canvas behind the checker board. The Author felt that this would of helped with immersing the playing into the game.

4 Critical Evaluation

As stated in section **Enhancements**, Mr.Nash was unable to get the additional features that were requested, developed. And so for this section we will be critiquing what does and doesn't work well, in regards to the project.

4.1 Movement

Movement is relatively smooth and throughout play testing so far there has been no instances of opposite team pieces being able to move when it's the oppositions turn, or being able to do illegal moves such as; double jumping when there is no opponent piece to jump over (which enables the double jump). Another aspect of movement the game does well in, is it's ability to do multiple forced moves (double jumps) in a row. The game logic doesn't break when performing three double jumps in a row or it doesn't allow the player to perform half the sequence of forced moves and then move a different piece which does not have the forced move apply. This may be an odd thing to identify, but if that flaw in the games forced move logic existed, then it could be used by the player for their advantage. The example would be this; the player is forced to do two double jumps (removing two enemy pieces from play), but notices that by doing so, exposes that exact piece to being taken over by opposing player in their next turn. So in order to save that piece, he/she could do the first double jump removing that enemy checker piece from play and then move a completely separate piece move, to which saves the original. The point being; The forced move logic stops this from happening. Another highlight to point out is that during these multiple double jumps, if your piece gets promoted to a king before the final double jump, the logic doesn't stop. It will instantaneously allow the player to do forced moves, with the movement rule set that a King inherits (moving forwards and backwards).

4.2 Teams

There is no way in it's current build, to enable the choice for the two players, which team colour they want to be; White or Black. Instead the Player 1 has to take the side of the White team and Player 2 has to take the side of the Black team. Even though the lack of choice doesn't effect game play as such, it does effect the aesthetic preferences the players may have. The only credible thing

to mention in regards to the "Teams" is that transitioning from one team turn to the other is achieved seamlessly.

4.3 Victory Status

There is currently no visual notification to inform the player that they have won or lost. It's very lacklustre if the player just wins, after playing a tough opponent, and the game just remains inactive, unable to do anything because the oppositions has been eliminated. There is also no way to restart a new game session. The player has to close the game and then reboot it again. Not an effective way to encourage replayability.

5 Personal Evaluation

Mr.Nash has expressed that even though he is proud of the work that he has achieved thus far, such as having all the core mechanics of the game working and it being developed and ran on Unity; Effectively making this his first ever Unity game. He feels overly disappointed that he had spent a vast amount of time looking into Undo/Redo functionality and Replay; And was unable to achieve any demonstrative functionality, in regards to these features.

5.1 Bugs

In the development of the project, Mr.Nash encountered several bugs. But there were two bugs in particular which ceased development of the project for longer than he'd like to disclose. The two bugs ended up being tied to one another for reasons not fully known, apart from the fact that they were both related to movement of a white checker piece.

The first bug was this; Whenever two white pieces can both double jump (remove enemy piece from play) over the same Black Checker (**Appendix - Figure 1**), then the ForcedMove() method would forcefully decide which White checker piece could do the jump. It was biased towards the White Checker to the bottom right of the Black checker, making a diagonally left double jump. What I mean by this is, even though in the rules you are allowed to choose any checker to make the move, if multiple checkers are forced to do a move, then the player has free reign on which checker piece does said forced move. But the game denied the player of that choice within this specific instance. Now in saying that, this bug was fixed when the second bug was fixed. So unfortunately Mr. Nash is unable to comment on how this bug was in fact fixed.

The second bug was game breaking and so had higher priority to be fixed over the first identified bug. The bug goes as follows; Whenever a White checker piece jumped to the index of 0,6 to 6,6 or 0,7 to 6,7 (x and y coordinates of the 2D Array[**Appendix - Figure 2**]); Then an "IndexOutOfRangeException: Array index is out of range" would be thrown (**Appendix - Figure 3**). This would then proceed to stop the game, without being able to resume play, unless it was a brand new session. Upon further inspection (**Appendix - Figure 4**) it was identified that it was at code line 41 within the class "Piece" that the "Inspector" was catching the initial error. So, after reviewing the code where the error was originating from (**Appendix - Figure 5**), the Author could not see where the error was, on that line of code. Eventually Mr. Nash discovered that it was in fact code line 36 that was causing the error. The If Statement should of been;

Listing 10: ForcedMove()

```
1   if (x >= 2 && y <= 5)
2
```

Not this;

Listing 11: ForcedMove()

```
1   if (x >= 2 && y >= 5)
2
```

The reason for this being the game breaking problem as it is, is due to the fact that the y coordinate operator (y >= 5) enabled the player to jump beyond the range of the 2D array, especially in a double jump scenario. So when ever a white checker landed on what was visually seen as y coordinate 6 or 7 (Row 7 or 8), the code would identify it as y coordinate 8 or 9. Which is beyond the defined range of the 2D array which is 8 x 8 (0 to 7). But by simply changing the operator to y <= 5, it meant that the pieces were still landing within the range of the Array.

After the bug was fixed, the previous bug was then no longer apparent. Mr. Nash does not fully know the extent to why this is.

A Appendix

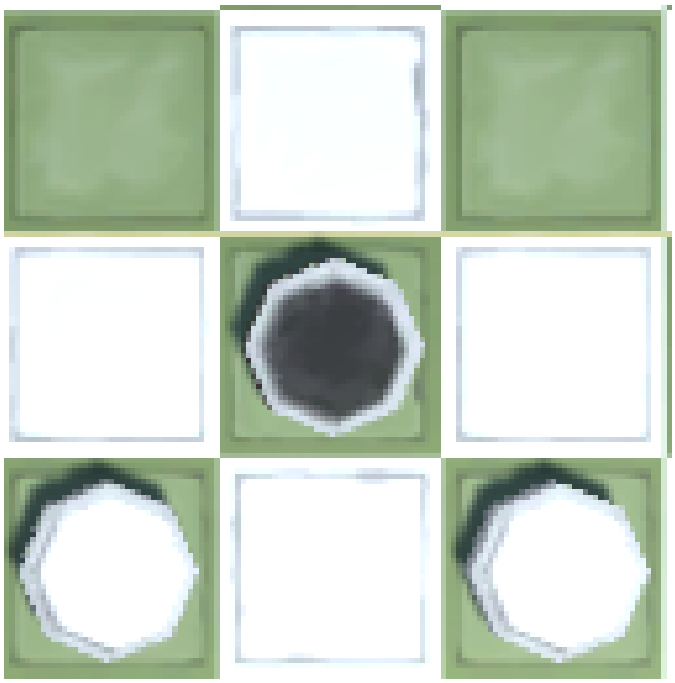

Figure 1: **Double Forced Move** - Two Choices

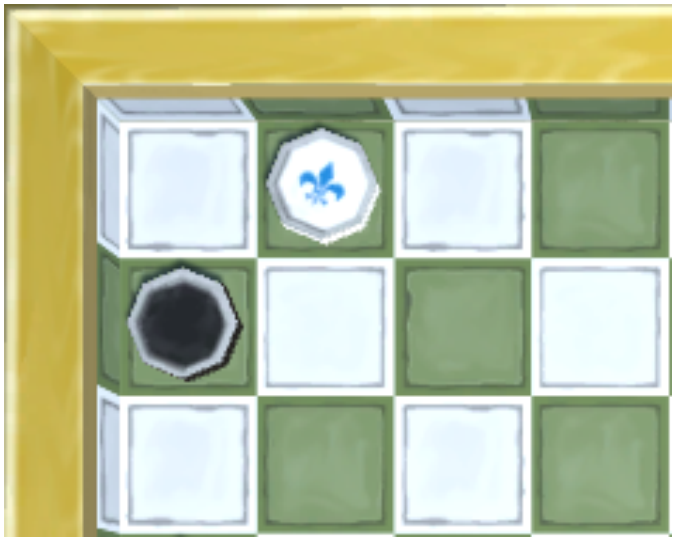

Figure 2: **Checker Placement** - Crash Occurs

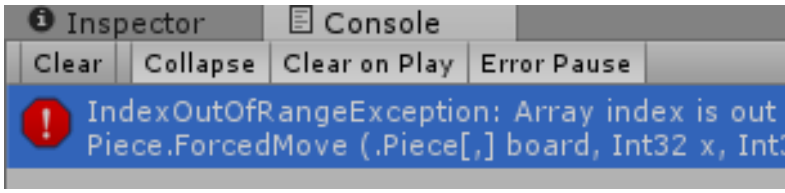

Figure 3: **Index Out of Range** - Console Error Message 1

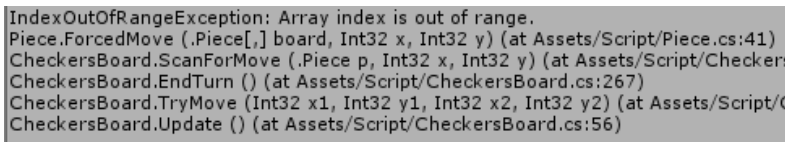

Figure 4: **Inspector** - Console Error Message 2

```
// Diagonaly Top Right
if (x <= 5 && y >= 5)
{
    // x + 1 means: 1 to the right (+) of your piec
    // y + 1 means: 1 up/above (+) of your piece
    // So it checks 1 to the Right then 1 above fro
    Piece p = board[x + 1, y + 1];
    if (p != null && p.isWhite != isWhite)
    {
        if (board[x + 2, y + 2] == null)
            return true;
    }
}
```

Figure 5: **Piece Class** - Code Line 41 Highlighted