Edinburgh Napier UNIVERSITY

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[**https://www.youtube.com/watch?v=z2QVD2n1fR8list=PLI** 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 [**https://www.youtube.com/watch?v=P78rfTOtG08index=4list=PLLH3mUGkfFCVXFGLRxfhst7pffE9o2SQO**] 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 pieces'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 [**https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/modulus-operator**]) 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 **https://docs.unity3d.com/ScriptReference/Object.Instantiate.h** 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[**https://www.youtube.com/watch?v=D2qM0fWBeHs**]" 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[**https://stackoverflow.com/questions/434761/array-versus-listt-when-to-use-which**] .

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 checks that if coordinate is **THIS IS ALL RUBBISH, WILL COME BACK TO IT TOMOROW!!!!!!!!!!!!**

3 Mock Introduction

Referencing You should cite References like this: [1]. The references are saved in an external .bib file, and will automatically be added to the bibliography at the end once cited.

4 Formatting

Some common formatting you may need uses these commands for **Bold Text**, *Italics*, and underlined.

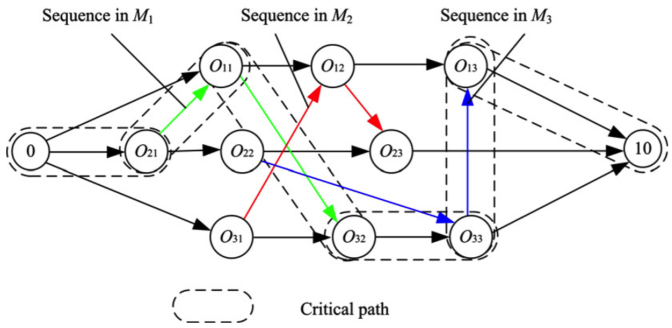

Figure 1: **ImageTitle** - Some Descriptive Text

4.1 LineBreaks

Here is a line

Here is a line followed by a double line break. This line is only one line break down from the above, Notice that latex can ignore this

We can force a break
with the break operator.

4.2 Maths

Embedding Maths is Latex's bread and butter

$$J = \left[\frac{\delta e}{\delta \theta_0} \frac{\delta e}{\delta \theta_1} \frac{\delta e}{\delta \theta_2}\right] = e_{current} - e_{target}$$

4.3 Code Listing

You can load segments of code from a file, or embed them directly.

Listing 10: Hello World! in c++

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello World!" << std::endl;
5     std::cin.get();
6     return 0;
7 }
```

Listing 11: Hello World! in python script

```
1 print "Hello World!"
```

4.4 PseudoCode

for $i = 0$ **to** 100 **do**
    print_number = true;
    if *i is divisible by 3* **then**
        print "Fizz";
        print_number = false;
    end
    if *i is divisible by 5* **then**
        print "Buzz";
        print_number = false;
    end
    if *print_number* **then**
        print i;
    end
    print a newline;
end

Algorithm 1: FizzBuzz

5 Conclusion

References

[1] S. Keshav, "How to read a paper," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 83–84, July 2007.