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. This is the data structure in which 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()

```
public bool ForcedMove(Piece[,] board, int x, int y)
  {
    if (isWhite || isKing)
    {
      // Diagonaly Top Left
      if (x >= 2 && y <= 5)
      {
        Piece p = board[x − 1, y + 1];
        if (p != null && p.isWhite != isWhite)
        {
          if (board[x − 2, y + 2] == null)
            return true;
        }
      }

      // Diagonaly Top Right
      if (x <= 5 && y <= 5)
      {
        Piece p = board[x + 1, y + 1];
        if (p != null && p.isWhite != isWhite)
        {
          if (board[x + 2, y + 2] == null)
            return true;
        }
      }
    }
}
```

As seen from 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 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. And enabling a "double jump", which is done when taking over an opponents piece and there is a free "tile" beyond it. The code below shows these two examples;

Listing 2: ValidMove()

```
public bool ValidMove(Piece[,] board, int x1, int y1, int x2, int y2)
{
    // If you are moving on top of another piece, then deny that move
    if (board[x2, y2] != null)
        return false;
    // Keep track of the number of tiles that have been jumped via x axis
    int jumpedTilesX = Mathf.Abs(x1 − x2);
    // Keep track of the number of tiles that have been jumped via y axis
    int jumpedTilesY = y2 − y1;
    // If jump 1 tile, it’s a move. If jumped two tiles, it’s a kill!

    // WHITE TEAM MOVE SET
    if (isWhite || isKing)
    {
        if (jumpedTilesX == 1)
        {
            if (jumpedTilesY == 1)
                return true;
        }
        else if (jumpedTilesX == 2)
        {
            if (jumpedTilesY == 2)
            {
                Piece p = board[(x1 + x2) / 2, (y1 + y2) / 2];
// If piece is jumping and it’s not null AND NOT over the same
// colour as our piece allowed to jump over by two tiles.
                if (p != null && p.isWhite != isWhite)
                    return true;
            }
        }
    }

```

2.3 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.

3 Mock Introduction

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

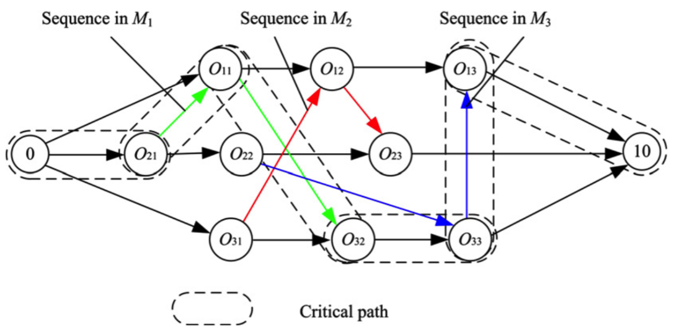

Figure 1: **ImageTitle** - Some Descriptive Text

4 Formatting

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

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 3: Hello World! in c++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    std::cin.get();
    return 0;
}
```

Listing 4: Hello World! in python script

```python
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