Tatedrez

by Marcos Rodrigues

My goal for this project was to write code that's easy to read, maintain, and build upon. All the rules and mechanics work for the 3x3 board, but if you increase the board size variable (and add more pieces) the game will still work perfectly. Another challenge I put upon myself was to implement the board logic on a one-dimensional array, instead of a more convenient two-dimensional array, slightly improving the game's performance.

There are three main classes: the Piece class handles the player's interaction with the pieces, the BoardTile class keeps track of where the player places the pieces and displays useful visual feedback to the player, and the BoardManager class enforces the game's rules and states. In retrospect, this last class got a little bloated, since everything from UI and sounds are handled inside it. It's a thing I could've improved on, but given my limited time for this project and wanting to keep the script complexity low, I put a larger effort into making the game work seamlessly, while having a nice aesthetic presentation.

And since there isn't an online mode, I made it so two people can play facing each other on the same phone!

Matchmaking

There are three ways to make a match in tic-tac-toe, columns, rows, and diagonals. After each play, the game will check if there's a match for the player who just moved a piece. To check if you made a column match, the game iterates over every element of the first row. If one matches the player's color, it iterates downward, increasing the index by *boardSize*. If every tile matches, the player wins, otherwise, the game moves over to the next tile in the row, repeating the process.

```
//Columns
for (int i = 0; i < boardSize; i++)
{
    if (board[i].CheckMatch(currentPlayerColor))
    {
        for (int j = 1; j < boardSize; j++)
        {
            if (!board[i + j * boardSize].CheckMatch(currentPlayerColor))
            {
                break;
            }

            if (j == boardSize - 1)
            {
                return true;
            }
        }
    }
}
```

For the rows it's similar, the game iterates over the first column, and checks in the right direction for matching pieces.

```
//Rows
for (int i = 0; i < boardSize; i++)
{
    if (board[i * boardSize].CheckMatch(currentPlayerColor))
    {
        for (int j = 1; j < boardSize; j++)
        {
            if (!board[i * boardSize + j].CheckMatch(currentPlayerColor))
            {
                break;
            }

            if (j == boardSize - 1)
            {
                return true;
            }
        }
    }
}
```

For the diagonals, we just have to check two lines: one starting from the top left, where the index increase is equal to *boardSize* + *1* and another one starting from the top right where the increase is *boardSize* - *1*. This will work on different size boards.

```
//Diagonals
if (board[0].CheckMatch(currentPlayerColor))
{
    for (int i = boardSize + 1; i < Mathf.Pow(boardSize, 2); i += boardSize + 1)
    {
        if (!board[i].CheckMatch(currentPlayerColor))
        {
            break;
        }

        if (i == Mathf.Pow(boardSize, 2) - 1)
        {
            return true;
        }
    }
}

if (board[boardSize - 1].CheckMatch(currentPlayerColor))
{
    for (int i = (boardSize - 1) * 2; i <= Mathf.Pow(boardSize, 2) - boardSize; i += boardSize - 1)
    {
        if (!board[i].CheckMatch(currentPlayerColor))
        {
            break;
        }

        if (i == Mathf.Pow(boardSize, 2) - boardSize)
        {
            return true;
        }
    }
}
```

Movement

Rook

The rook moves to tiles on the same row and column. To check the row of a tile on a one-dimensional array we just do the integer division of the tile's index over the board size. The same for the column, except we have to do the remainder of the division. We also have to check if no pieces are blocking the way, by going through each tile between the piece's position and the desired tile, and returning false if a piece is found.

```
private bool IsPieceBlockingRow(int a, int b)
{
    int direction = 1;
    int first = Mathf.Min(a, b) + direction;
    int last = Mathf.Max(a, b);

    for (int i = first; i < last; i += direction)
    {
        if (board[i].HasPiece())
        {
            return true;
        }
    }

    return false;
}
```

```
private bool IsPieceBlockingColumn(int a, int b)
{
    int direction = boardSize;
    int first = Mathf.Min(a, b) + direction;
    int last = Mathf.Max(a, b);

    for (int i = first; i < last; i += direction)
    {
        if (board[i].HasPiece())
        {
            return true;
        }
    }

    return false;
}
```

```
private void CheckRookValidMoves(int index)
{
    for (int i = 0; i < Mathf.Pow(boardSize, 2); i++)
    {
        bool validRow = IsOnSameRow(index, i) && !IsPieceBlockingRow(index, i);
        bool validColumn = IsOnSameColumn(index, i) && !IsPieceBlockingColumn(index, i);
        if (!board[i].HasPiece() && index != i && (validColumn || validRow))
        {
            validMoves[i] = true;
        }
    }
}
```

Bishop

The bishop moves diagonally, and the method for checking if it can move is similar. The target tile must be in the same diagonal and there can be no pieces blocking the way. On a board, two tiles share the same diagonal if the distance between their columns is the same as the distance between their rows.

```
private bool IsOnSameDiagonal(int a, int b)
{
    return Mathf.Abs(GetRow(a) - GetRow(b)) == Mathf.Abs(GetColumn(a) - GetColumn(b));
}
```

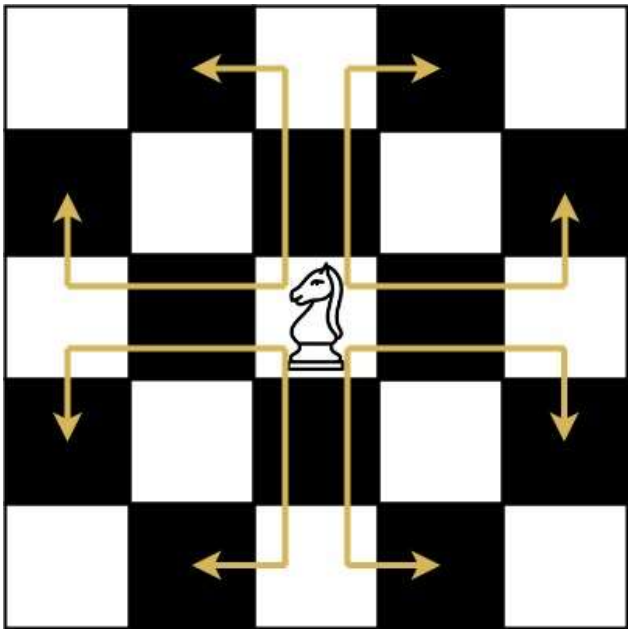
The rules for moving through diagonals work the same way as in the matchmaking part. We need to check the direction of the diagonal to know how much the index increases while going through it.

```
private bool IsPieceBlockingDiagonal(int a, int b)
{
    int top = Mathf.Min(a, b);
    int bottom = Mathf.Max(a, b);

    bool topLeftDiagonal = GetColumn(top) < GetColumn(bottom); //Otherwise we have a top-right diagonal
    int direction = boardSize + (topLeftDiagonal ? 1 : -1);
    top += direction;

    for (int i = top; i < bottom; i += direction)
    {
        if (board[i].HasPiece())
        {
            return true;
        }
    }

    return false;
}
```

```
private void CheckBishopValidMoves(int index)
{
    for (int i = 0; i < Mathf.Pow(boardSize, 2); i++)
    {
        bool validDiagonal = IsOnSameDiagonal(index, i) && !IsPieceBlockingDiagonal(index, i);
        if (!board[i].HasPiece() && validDiagonal)
        {
            validMoves[i] = true;
        }
    }
}
```

Knight

The knight is a bit different since it moves in L-shapes comprised of two tiles in one direction and one tile 90 degrees to either side. What's curious it's the knight works the opposite as the other pieces. It can't move to tiles on the same row, column, or diagonal. Furthermore, it will only move within a range of two tiles in every direction from the starting position, as you can see in the picture below.



Given this information, it's easy to write a condition for the knight's movement.

```
private bool IsInKnightRange(int a, int b)
{
    return Mathf.Abs(GetRow(a) - GetRow(b)) <= 2 && Mathf.Abs(GetColumn(a) - GetColumn(b)) <= 2;
}
```

```
private void CheckKnightValidMoves(int index)
{
    for (int i = 0; i < Mathf.Pow(boardSize, 2); i++)
    {
        if (!board[i].HasPiece() && !IsOnSameColumn(index, i) && !IsOnSameRow(index, i) && !IsOnSameDiagonal(index, i) && IsInKnightRange(index, i))
        {
            validMoves[i] = true;
        }
    }
}
```

All info on the available moves is stored on a boolean array the same size as the board array, which is checked against it to enable the tiles that the player can move the piece to.