Group Members:
Aidan Chaplin 6906952

Will Doran 6967392

Introduction

In our project, we coded a game of chess in unity and then coded an AI to work with the game

How to run / compile & operate

To compile the project, open the project folder in unity, and then click the run button. At the top of the editor.

To play the chess, double click on the 3P71Chess.exe in the builds folder. To move a piece, first click on the piece you want to move, then click on another space on the board. If that is not a valid space, the piece will not be moved there and you will have to select another piece. The square on the left of the UI tells you who's turn it is.

Clicking on the "custom board button" will cause a menu to pop up that allows you to customise the board layout. Entries in the text fields must be one character only. The first entry must be a letter from A-H, and the second must be a number from 1-8. Not all entries need to be filled (any left blank will assume their default)

The AI State button controls the influence of the ai. By default, the game is player vs player, with no AI control. Pressing the button once will cause the AI to play the white pieces. Pressing the button again will cause the AI to play the black pieces. Pressing it again will loop back to Player vs Player, and so on.

Overview

Class list:

- BoardState: The general board state of the game. It is the controller of who's where.

- BoardUI: The script on the buttons that allow the user to input there moves to the game
- BoardInitializer: The initializer that initialises the board state. Also heavily responsible for driving custom board states.
- AllPieces: The class that contains all the pieces of the game
- ColourPieces: A singular colour's pieces
- PieceMoves: Functions that return an array of positions from a selected position on the board
- AI: The AI controller
- BoardEvaluator: Evaluates the heuristic of a board.
- Node: A singular board the AI uses to compute it's next move
- ThreatEvaluator: Evaluates for both Check & Checkmate, and has static functions available to tell whether any given space is threatened by the opposite colour.

One of Aidan's favourite functions is the line function in BoardState. It takes in a board, a start point, a "vector" for where to go, a distance remaining, a boolean for if it can capture, and finally a colour. It then will recursively call itself, and return an array containing all the available positions the piece at the start coordinates can move into along the line of vector, up to the max distance, and can handle almost any move that can be seen in chess. In terms of line positions.

Heuristic & performance improvement

We were worried about our memory footprint with such a wide search range, so we made sure that behind the scenes, the board state was always represented by a simple 2d array of shorts, giving each board a memory footprint of 128 bytes.

Each entry in the array is either 0, or the value of a piece. A negative value means the piece is black, while a positive value means the piece is white.

The heuristic primarily evaluates based on the amount of material, using the following values:

- Pawns: 10
- Knights: 30
- Bishops: 32
- Rooks: 50
- Queen: 90
- King: 900

Since the values are modified in the array by a negative for black, simply summing them results in the total material difference.

These values are then slightly modified by their position, though position never has an impact of greater than ±5.0. These position weights were determined by manually set arrays derived from Laura Hartikka's freeCodeCamp article on AI Chess.

Finally, the heuristic looks for checkmate. It does this by first looking for check; if the king is not in check, it doesn't continue. If he is, however, then it sees if all the squares around the king are threatened. If any of them are both non-threatened and open for the king to move into, he isn't in checkmate. If this isn't the case, it finally checks to see if the checking piece(s) can be taken, or if a friendly piece can intercede. If none of these are the case, it returns the state as being a checkmate, which increases the heuristic evaluation by 10000 in favour of whichever side has made the checkmate.

This checkmate function is reused at higher levels to evaluate end-of-game conditions.