Manual to Chess Game – Unity Asset

Introduction

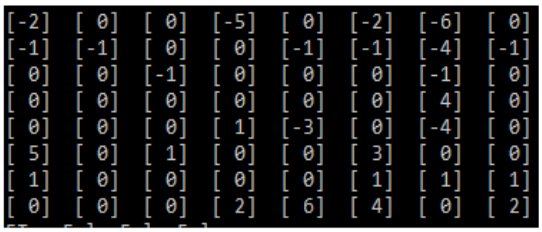
This document details how the unity asset called Chess Game works.
If you have any questions you can contact me at alekhine@hush.com
Code documentation: http://readytechtools.com/ChessGame/ManualChessGame.pdf

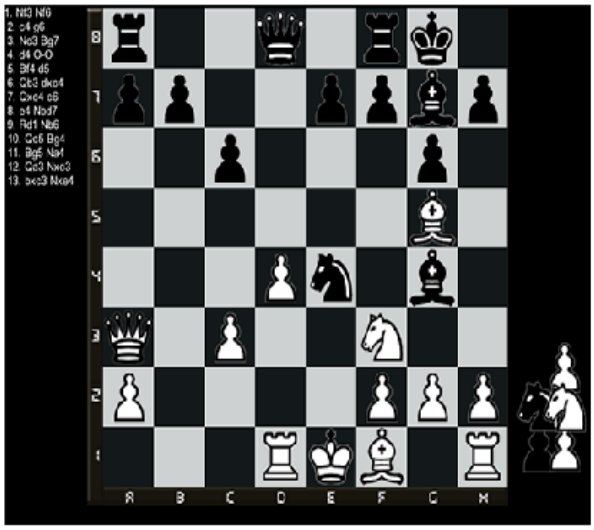
Content

Board representation

cgBoard
Abstract Board

cgChessBoardScript
Visual Board





The board is represented in two ways.
The **visual board** that the end user sees and interacts.
This visual board is primarily handled by **cgChessBoardScript.cs** which inherits from MonoBehaviour and has editable properties in the editor.

And secondly the **abstract board**, which stores the board state in as minimal a fashion as possible through a list of sbytes. This is handled primarly by **cgBoard.cs.**

The image above shows how cgBoard stores a board representation in a list of sbytes on the left, and how this list of sbytes is then read and represented visually by cgChessBoardScript on the right. cgChessBoardScript reads the state of the internal board to determine where to position pieces, which color can move, whether or not a move is legal etc.

If you are new to chess programming you may wonder why this separation into an abstract board and a visual board.
The reason is our AI Opponent needs to manipulate this board a lot(more than a million moves and reverts in some instances), so we need an optimized board version which doesn't strain memory and cpu nearly as much as manipulating the graphical board would to calculate the best move. Using the visual board is not feasible for our AI opponent.

The abstract board(cgBoard.cs) has a list called 'Squares' with sbytes(a whole number between -128 and 128). This list shows the current state of the board as seen in the above picture to the left.
This array has a capacity of 64 on a typical 8 x 8 board.
Each index corresponds with a position on the board, starting with index 0 as A8, 1 as B8, 2 as C8, you can see the full list of index to square name in cgBoard.SquareNames

Each element in the squares array of the abstract board(cgBoard.cs) indicates what occupies the square. 0 = empty square, 1 = white pawn, 2 = white rook, 3 = white knight, 4 = white bishop, 5 = white queen, 6 = white king, -1 = black pawn, -2 = black rook, -3 = black knight, -4 black bishop -5 = black queen and -6 = black king.

The script cgChessBoardScript looks at this list when determining what pieces should be located where at all times.
cgChessBoardScript stores an instance of cgBoard in cgChessBoardScript._abstractBoard, this instance of cgBoard dictates where pieces are placed, who is allowed to move and what moves are allowed.

Generating Moves

When cgBoard is instantiated it also generates a list of moves if this list of moves has never been instantiated while the program has been running.
This list is called "AllHypotheticalMoves" and is a dictionary over all hypothetical moves with complete disregard to blocking and the position of any other piece than the moving piece. This list is generated inside cgBoard._generateAllPossibleMoves();
Think of it as a database over all moves.
As we generate all moves for a piece on a given square on the board and then we save this information in a new instance of cgMoveSet, we then add this MoveSet to the dictionary list, with a string key built from its type + the index location on the board from which the piece is standing.
The bishop, knight, rook, queen and king are interchangeable regardless of color, only the pawn has color specific movement(the direction it can move, and where it gets promoted).
So we use white piece types for the white pieces and add one additional piece type for black pawn(-1).
After we have generated all the hypothetical moves we can look up what moves a piece can perform on any square, by using the key: piecetype+indexlocation.
If we want to know where the bishop can move when its standing on D8(index 3).
We would use the key string "43" 4 = bishop type, 3 = index location, and we would get the moveset corresponding to this piece on this location.
Since we have 7 different types(black pawn + all white types) and 64 squares the list will have 448 elements.
The idea behind this dictionary list is to save a huge amount of computation when the engine needs to generate thousands even millions of moves, as it can simply look in the dictionary list instead of computationally generating them.

Testing legality of moves

When all the hypothetical moves have been generated we must test their legality on the current board, as the moveset we generated does not take into account any other pieces which may block a move.

We collect all pieces and separate them into enemy and friendly pieces, we then find the moveset for each of our pieces in the previously created allHypothetical dictionary list and then we look at all potential moves and see which are blocked and which aren't.
We also go through all enemy pieces and look at all enemy moves to see if any of our pieces are pinned to our king(and thusly illegal to move) if so we remove all moves the pinned piece could make unless they capture the attacking enemy piece(if the piece is only pinned by 1 enemy piece – if its pinned by 2 it may never move.) or moves the pinned piece can make the position it such that it is still blocking the attack on the king
We also check if any enemy move attacks our king(in other words we are checked), if this is the case we revise the list to only include moves which block the checking move, or capture the checking piece, or move the king to a square which is not attacked.

Special Moves(Castling, EnPassant and Pawn promotions)

There are 3 special moves which we haven't taken into account yet, these three moves are En Passant, Castling and Pawn Promotions.

En Passant and Castling are dependent on what the previous states of the board are, if for instance we've moved the king at any earlier point, we may never castle, but if we only moved the rook and not the king, then we may not castle to the side where the rook was located, however we may still castle the opposite side. All this information is stored on the board, and we register any rook or king move inside special variables on the cgBoard to test Castling rights.
There's a few other caveats to castlingyou can look inside findLegalMoves theres a well commented walk through of all the scenarios where castling is disallowed.

En Passant is a move that is only legal for 1 turn immediately after a pawn has performed a double move, so whenever a pawn performs a double move we register this on cgBoard.enPassantSquare, if theres no such square we set enPassantSquare to 65 to indicate no square on the board is legal to en passant.
When we generate moves we look at enPassantSquare to see if our pawns should look out for a particular square.

Pawn promotions are handled on cgBoard, it auto-promotes to queen, the scenarios in which it is advantageous to underpromote are extremely rare, but ofcourse the player should have the option to select his promotion, however this is not currently the case, it would be simple to create, simply go to cgBoard.Squares at the index location of the promotion, and change the piece type to whichever piece type the player selected.
If the player promoted a pawn on b8 to a knight instead of a queen the code would be cgBoard.squares[1]=3.

AI Opponent

The AI Opponent I've spent a long time on, until I finally solved the problem and found a surprisingly simple way to analyze moves and look to a specified depth and ignore bad moves.

The solution is a regressive alphabeta search function, located at cgEngine.alfabeta

This seemingly simple search function is where the engine examines all board states and figures out which is the preferred move.
It does so by going to a specified depth and evaluating the board value at this depth (cgBoard.Evaluate), of the board and comparing this value to the other boards it has previously looked at, it uses alfabeta to prune and ignore certain branches in the search, as certain move can safely be ignored based on game theory.
Then it shifts the value it found upward to preceding parent moves.
We can then sort the original list of moves based on cgSimpleMove.val and find the best one in the list.

FAQ

Note: Contact me at alekhine@hush.com if you can't find an answer to your question.

How do I change the Behavior of the AI?
Modify the values inside cgValueModifiers.
Or change the searchdepths at cgEngine.SearchDepthWeak, cgEngine.SearchDepthStrong and/or cgEngine.SearchDepthEndGame

If you're feeling up for personalizing the AI even further, you can go to cgBoard.Evaluate and implement your own valuemodifiers, thus implementing your own patterns that you want the engine to look for and evaluate.

How can I see the second or third or fourth best move or worst move as judged by the engine?
After the analysis has been completed, the engine sorts the list at cgEngine.moves such that the best move and hence the most preferred move, is at index 0, second best at index 1, and so forth, descending onward through all legal moves available to it, the last move in the list being the least preferred move.

Now I should briefly note in case you are looking to find the worst possible move as judged by the engine: The engine currently does not examine 'how bad' a bad move is, once move scores below a certain threshold the branch is pruned, so the engine will not necessarily have done a very thorough analysis on bad moves. This means that the further down the list you go the more unclear it becomes how bad the moves are, the only thing that is clear is that the moves are bad.
You may ask why does it not thoroughly analyze bad moves? This is to save time, with the alphabeta algorithm that prunes bad branches, the time it takes to analyze can sometimes be more than halved, and the final prefered move will always be the same regardless, the only downside to using this alphabeta pruning is that we don't know how bad the bad moves are, we just know that its bad beyond a certain threshold, it may be way beyond the threshold or may be slightly below the threshold.

What are differences between the difficulty levels?
Here's a table that indicates the search depth differences between the difficulty levels.

	Strong Search Depth	Weak Search Depth
Difficulty 1	4	3
Difficulty 2	4	4
Difficulty 3	5	4

Why the weird data types like byte and sbyte?

We need our engine and everything it instantiates to be as highly optimized in terms of memory as possible. This is why the internal board and the engine, where useful and possible, use very simple data types like sbyte, byte and short.
For example if we were to use int(32bit) instead of byte as the data type on cgSimpleMove.from and cgSimpleMove.to then our total memory usage for just these variables when instantiating a million would be
2*32*1,000,000 / 800,000 = 80 megabytes
Whereas with bytes(8bit) the memory usage would be
2*8*1,000,000 / 800,000 = 20 megabytes

Which means a fourfold decrease in memory usage when using byte data types.
This is a signicant improvement which has a positive effect on the performance.