Isaac Montanez

Chessboard

A. General

For readability, I divided the chessboard file into several parts.

I designed the chessboard to work separately from the engine. Therefore, there are duplicate functions in both programs. Mixing functions between the chessboard and engine may create unnecessary bugs during future development.

The tile id format is: "tile-x y".

Tile name format is: color + name + "-" + player id + "." + (ability to castle or en passant).

The board state can be updated by: player moves, engine moves, undo button, new game button, past move dashboard, or by selecting no on the modal popup after changing the board state in the past move dashboard.

The white space on the right of the page is for future expansion.

B. Killed Piece Display

This section displays killed pieces to the right of the chessboard. After each move, the pieces are deleted and redisplayed to make the code simple and easy to read.

C. Past Move Dashboard

Displays past moves and allows the player to change the board state to a previous state.

The displayed board state is set to the current state when entering the box. The state updates to whatever the user is hovering over. Once the user clicks a board state, that state is updated, and the state no longer updates when hovering. After leaving the box, if the last clicked state is not the current move state, a modal appears to confirm the user selected the correct state. If the user does not click a state and leaves the box, the state reverts to the current move state.

Requires storing each previous game state and its weight.

Changing the game state does not clear Zobrist hashed values of different board state weights, improving board performance.

It is easier to accidentally change to the wrong state than to press the undo or new game button. Therefore, the past move dashboard requires confirmation to change state, but the undo and new buttons do not require confirmation.

D. Game Strength Display

Displays the strength of the game as calculated by the chess engine.

New weights are passed to the display after the engine has run. If the engine passes undefined, the value reverts to the last value passed.

E. Pawns

Pawns have their own rules and edge cases. Therefore, the pawn-related functions are together. Pawns are also handled outside of this block.

F. Get Moves

Get moves generates moves to check if the user's move is allowed. The move must be legal and cannot result in the player checking their King.

G. Game Over

Handles stalemates and checkmates.

Stalemates occur if:

1. the same board state has occurred three times;
2. both players have their King and a Knight, a Bishop, or no other pieces; or
3. whoever's turn it is has no legal moves.

The past move dashboard block adjusts for game state changes made in the dashboard.

Checkmate occurs if stalemate #3 is true and their King is checked.

The game ends once a stalemate or checkmate occurs. Once over, a modal prevents the user from making additional moves. Changing the board state in the past move dashboard removes this modal.

H. Engine

This block calls the engine and handles its output. For more information on the engine, please see the section below.

The only variables passed from the board to the engine are the bitmaps created in convertToChessEngineBoard and Zobrist Hashing information.

Moves are passed from the engine by a 3-variable array. This array includes the position the piece is moving from, the position the piece is moving to, and the game state weight. The positions are formatted as follows. The first number represents the y value, and the second represents the x value. Y values use 0 - 7 indexing, and x values use 1 - 8 indexing.

See chessEngineMoveUpdate to see how this array is processed.

I. Mouse

This block handles user interaction with the board.

When the mouse clicks a tile with a piece on it, the tile name changes to "emptyTile". The original tile name is stored, and the piece follows the pointer. If the player holds a piece and releases the mouse, one of two things happens. The piece name is returned to the original tile if the move is not allowed, or the name moves to the new tile, and the chess engine launches.

J. General

This section includes Chess which is the exported function sent to App.js.

Isaac Montanez

Chess Engine

A. General

I built the engine to improve my understanding of Javascript, not for performance. Therefore, the engine cannot evaluate as many scenarios as engines built in C++. Additionally, I have not finished building the engine. Therefore, the engine may make obvious mistakes.

Javascript processes Numbers in 32-bit form. Therefore, I decided to use Bigints for all bitwise operations since Bigints do not have a maximum bit limit.

I use an 80-bit bitboard to store board states because it is easier to understand and process the chess board as a 10 x 8 bitboard instead of an 8 x 8 bitboard.

Deep copying and discarding arrays like chessBoard and numOfAttacks allow for faster and simpler evaluation.

Visual Studio Code does not recognize BigInt() as a function, but it is a Javascript function. // eslint-disable-next-line no-undef fixes this.

Isaac Montanez

B. Chess Engine (file)

This file is the nerve center for the chess engine. It takes in the input from the chessboard, processes it with functions from other files, and returns a move to the chessboard.

Nodes are weighted by a combination of the current positions of each piece, the value of the remaining pieces, and numOfAttacks. numOfAttacks gives a bonus for the amount of the board each piece can reach, properly defended pieces, and possible successful attacks.

Weights are multiplied by -1 for the user in weightNextMoves. The user will attempt to reduce the weight while the ai attempts to maximize the weight. Each simulated player passes the weight of the best move it can make.

Since I do not display Merge Sort in the sorting Algorithms, I decided to use Merge Sort here. Merge Sort sorts in descending order for the ai and ascending order for the user.

RecursiveBoardEval makes use of alpha-beta pruning and iterative deepening. These topics are explained at:

- Alpha-beta pruning: https://www.youtube.com/watch?v=xBXHtz4Gbdo
- Iterative deepening: https://www.chessprogramming.org/Iterative_Deepening

C. Engine Find Moves

This file generates all legal moves. It moves each piece in all legal ways and deletes all moves that would result in a check on the player's King.

The chess engine will always promote a pawn to a Queen. The value of a Queen and the efficiency gained outweigh the possibility of promoting to the wrong piece.

D. Engine Get Board Weights / Fixed Board Weights / Pawn Structure Weights

These files are either set board weights or calculate previously set bonuses. These weights help the engine evaluate the best tiles first and improve ordering for alpha-beta pruning.

E. UpdateNumOfAttacks

NumOfAttacks allows the engine to see one move into the future by calculating where every piece can attack and move.

numOfAttacks guesses which pieces are threatened, can be threatened, and prioritizes mobility.

For performance, numOfAttacks updates after each simulated move in recursiveBoardEval().

Diagonal attacks can pass through the same player's Queens, Bishops, and Pawns for one tile. Straight attacks can pass through the same player's Queens and Rooks.

F. Zobrist Hashing

For an explanation of Zobrist hashing, visit:

https://www.chessprogramming.org/Zobrist_Hashing

I use an 81-bit Bigint instead of the traditional 64-bit. The extra time for an 81-bit Bigint is minimal. I use 81-bits because I use Math.random() to generate the bits for each number, and I do not know how random Math.random() is. The last bit determines if it is the user's or ai's turn.