Project Report

Chess Application

1. Introduction

This report presents a detailed overview of the development and implementation of a fully functional, browser based chess frontend website visualized using Canvas. The project simulates a full featured chessboard in which any games can be simulated, as well as offering users to practice and refine their understanding of chess tactics through interactive puzzles.

The project is written entirely in JavaScript using HTML5 Canvas for the visualization of the board and pieces, within a single script application.

2. Application Architecture

The architecture is centred around a single class called **`Board`**. This class encapsulates all game logic and state management. Supporting it is a static **`Piece`** class that defines constants for all piece types and colours using bitwise encoding.

2.1 The Pieces

```
class Piece {
   static King = 1;      // 001
   static Pawn = 2;      // 010
   static Knight = 3;    // 011
   static Bishop = 4;    // 100
   static Rook = 5;      // 101
   static Queen = 6;     // 110
   static White = 8;     // 1000
   static Black = 16;    // 10000
}
```

In order to store both the type of the piece and it's colour, I used **bitwise encoding.**
Each Piece and colour is assigned it's value, which when turned to binary and combined (**using a Bitwise OR**) can form a specific piece in a specific colour.
ex. If I wanted a White Queen,

=> let **`whiteQueen`** = Piece.*Queen* | Piece.*White* // (0110 | 1000 = 1110) or 14

This value is then placed into the array which represents the board state, and the pieces can be differentiated in the code by their value.
Similarly if I needed to check for the colour of a piece, (we would use a **Bitwise AND**)

=> piece & Piece.*White* !== 0 (will return true if white, false if black)

2.2 The Board

The **`Board`** class encapsulates all the logic that makes the game work. Inside it are the functions that draw the board on the screen, handle the ability to drag and drop the pieces, show where those pieces can be moved, check if moves are illegal, etc.

The Board class maintains:

1. A one-dimensional array (squares) which represents the 64 tiles of an 8x8 chessboard. The value of each index represents a specific pieces. Changes to the squares array represent

players making moves. Any change to the array, if it is a legal move will be visualized in the Canvas board.

- Game state variables such as:
    - **draggedPiece** and **draggedPieceIndex** which are used to enable the drag and drop functionality explained later.
    - **MoveNumber** is a variable which keeps track of the current turn, if the value is odd it's white's move and vice versa.
    - Several flags for castling rights, en passant state, and game-over conditions.
        - **enPassantTarget** checks whether there is target able pawn for en passant.
        - **enPassantSquare** represents the square behind the pawn that just took.
        - **enPassantCount** represents the number of turns since there has been a double move.
        - Variables for checking castling state (**whiteKingMoved**, **blackKingMoved**, **whiteRookMoved** and **blackrookMoved**).
    - **controlledSquares** which represents all squares controlled by the opposite side, depending on whose turn it is. Meaning all tiles that are under threat for the current player's move.
    - **moveHistory** which keeps track of all moved plays in an array. All moves are written in chess notation (ex. **Nxc6**).
    - **solution, opponentMoves** and **maxMoves** are used for the puzzle logic.
    - **kingChecked** which is self explanatory.
    - **flipped** which decides which perspective the game board is shown from.
    - **gameOver** checks for if the current king, or any other piece have legal moves. If not, the game ends.

3. Rendering

3.1 Drawing the Board

The board and pieces on the web page are drawn using the Canvas.

The **drawBoard** function draws the board, including each tile and the notation. Tiles are alternating dark and light squares. The notation includes letters (a-h) that appear at the bottom of the board and numbers (1-8) that appear at the side. If **flipped** is true, the board is drawn the other way aroud, meaning letters go from h to a and numbers from 8 to 1.

```
drawBoard() {
  c.clearRect(0, 0, canvas.width, canvas.height)
  for (let file = 0; file < 8; file++) {
    for (let rank = 0; rank < 8; rank++) {
      c.fillStyle = (file + rank) % 2 === 0 ? "#f6f6f6" : "steelblue";
```

```
        c.strokeStyle = "darkgray";

        let x = originX + rank * tileSize;
        let y = originY + file * tileSize;

        c.fillRect(x, y, tileSize, tileSize);
        c.strokeRect(x, y, tileSize, tileSize);

        if (rank === 0) {
            c.fillStyle = "ivory";
            c.font = "16px Arial";
            c.textAlign = "right";
            c.textBaseline = "middle";

            if (this.flipped) {
                c.fillText(""+ Math.abs(file + 1), x - 10, y + tileSize / 2);
            } else {
                c.fillText(""+(8 - file), x - 10, y + tileSize / 2);
            }
        }

        if (file === 7) {
            c.fillStyle = "ivory";
            c.font = "16px Arial";
            c.textAlign = "center";
            c.textBaseline = "bottom";
            let letter = null;
            if (this.flipped) {
                letter = String.fromCharCode(104 - rank);
            } else {
                letter = String.fromCharCode(97 + rank);
            }

            c.fillText(letter, x + tileSize / 2, y + tileSize + 24);
        }
    }
  }
}
```

3.2 Placing the Pieces

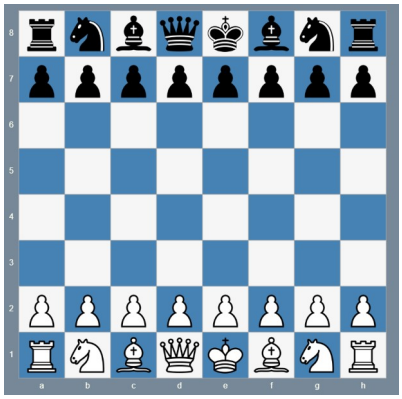
First, the state of the pieces is loaded in using **Fen Notation,** squares is transformed to represent what is written on the fen**:**

```
loadFromFen(fen) {
  let pieceFromSymbol = {
    "k": Piece.King,
    "p": Piece.Pawn,
    "n": Piece.Knight,
    "b": Piece.Bishop,
    "r": Piece.Rook,
    "q": Piece.Queen
  };

  let fenBoard = fen.split(' ')[0];
  let file = 0;
  let rank = 0;

  for (let symbol of fenBoard) {
    if (symbol === '/') {
      file = 0;
```

```
            rank++;
        } else if (isDigit(symbol)) {
            file += parseInt(symbol);
        } else {
            let color = isUpper(symbol) ? Piece.White : Piece.Black;
            let type = pieceFromSymbol[symbol.toLowerCase()];
            this.squares[rank * 8 + file] = type | color;
            file++;
        }
    }
}
```

Fen notation, is a string of letters and numbers that can represent any chess game state. It consists of 8 segments seperated by a '/' that each represent a row. The game is hardcoded to start with the starting position, but later the fen notation will be used to load each puzzle. Black pieces are represented in lower case, white pieces in uppercase, numbers represent number of blank squares.

'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR'



Next we preload each piece's image and then draw it on the board:

```
preloadImages() {
    let pieces = ["king", "pawn", "knight", "bishop", "rook", "queen"];
    let colors = ["white", "black"];

    pieces.forEach(piece => {
        colors.forEach(color => {
            let img = new Image();
            img.src = `pieces/${color}-${piece}.png`;
            this.pieceImages[`${color}-${piece}`] = img;
        });
    });

    setTimeout(() => this.drawPieces(), 100);
}
```

```
drawPieces() {
    this.squares.forEach((square, index) => {
        if (square !== 0) {
            let color = (square & Piece.White) === 8 ? "white" : "black";
            let piece = this.getPiece(square & 0b111);
            let image = this.pieceImages[`${color}-${piece}`];

            let x = originX  + (index % 8) * tileSize;
            let y = originY  + Math.floor(index / 8) * tileSize;

            if (image) {
                c.drawImage(image, x, y, tileSize, tileSize);
            }
```

```
        }
    });
}
```

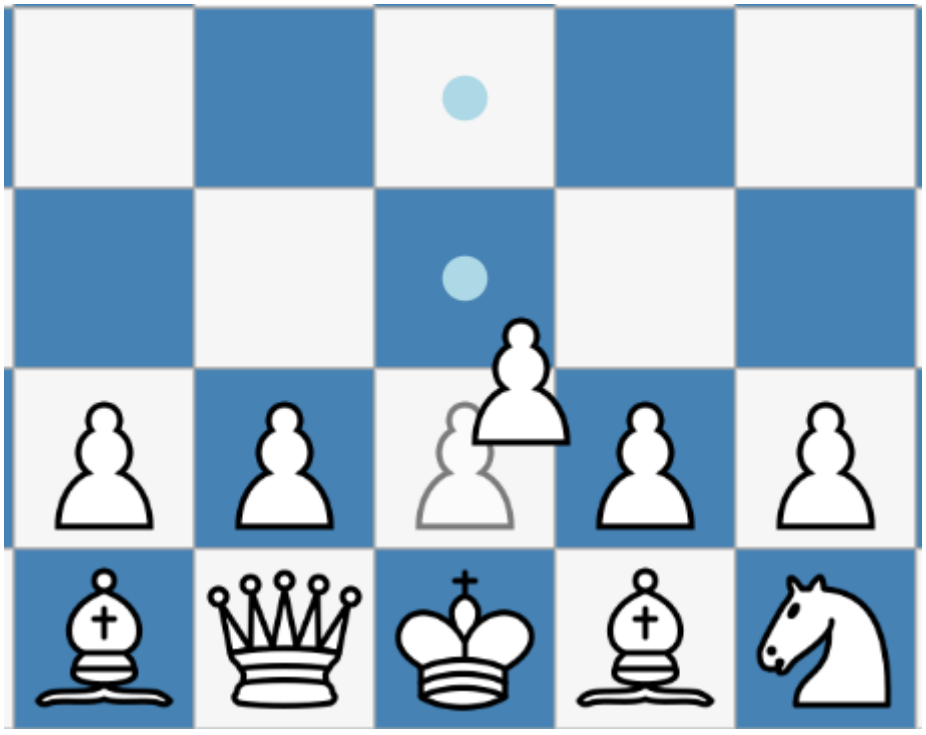
3.3 Highlighting legal moves

Next I will explain how the drag and drop is implemented, but before that here are the functions which are used to highlight to the player which moves are legal for which piece when they are picked up by drawing a circle.

```
highlightMoves(validMoves) {
    for (let move of validMoves) {
        let x = originX + (move % 8) * tileSize + tileSize / 2;
        let y = originY + Math.floor(move / 8) * tileSize + tileSize / 2;

        c.beginPath();
        c.arc(x, y, tileSize/8, 0, 2 * Math.PI);
        c.fillStyle = "lightblue";
        c.fill();
    }
}
```

```
displayMoves(currentPiece, currentIndex) {
    let validMoves = []

    for (let tile in this.squares) {
        if (this.isValid(currentPiece, currentIndex, parseInt(tile)) && this.simulateMove(currentPiece, currentIndex,
parseInt(tile))) {
            validMoves.push(parseInt(tile))
        }
    }

    this.highlightMoves(validMoves)
}
```

ex.



4. Drag and Drop Logic

The **handleDragaAndDrop**() method implements the full interaction logic for selecting, dragging, and dropping chess pieces on the board using mouse events. This function is central to the user experience, as it bridges the visual interface (Canvas) with the underlying game rules and logic.

It listens to three primary mouse events:

- **mousedown** for selecting a piece,
- **mousemove** for dragging the piece across the board,
- **mouseup** for releasing the piece and validating the move.

4.1 Mousedown – Selecting a Piece

```
canvas.addEventListener("mousedown", (e) => {
   if (this.gameOver) return;
   let mouseX = e.offsetX;
   let mouseY = e.offsetY;

   for (let i = 0; i < this.squares.length; i++) {
      let x = originX + (i % 8) * tileSize;
      let y = originY + Math.floor(i / 8) * tileSize;

      if (mouseX >= x && mouseX <= x + tileSize && mouseY >= y && mouseY <= y + tileSize &&
this.squares[i] !== 0) {
         this.draggedPiece = this.squares[i];
         this.draggedPieceIndex = i;

         let currentPlayer = (this.moveNumber % 2 === 0) ? Piece.White : Piece.Black;

         if ((this.draggedPiece & currentPlayer) === 0) {
            isDragging = true;

            this.drawBoard();
            this.drawPieces();

            this.displayMoves(this.draggedPiece, this.draggedPieceIndex);
         } else {
            this.draggedPiece = null;
            this.draggedPieceIndex = null;
         }
      }
   }
});
```

Triggered when the player clicks on the canvas near a piece. It:

Checks if the game is over (**gameOver**) so that you are not able to move any pieces after a checkmate or stalemate.

Calculates the mouse click coordinates relative to the canvas. Iterates over each square (0–63) and computes the actual pixel coordinates of each square based on originX, originY, and tileSize. If the mouse click is inside a square and the square contains a piece, it continues:

Saves the piece (**this.draggedPiece**) and its index (**this.draggedPieceIndex**)

Only allows dragging if the piece belongs to the player's color**.**

Triggers: **this.drawBoard()** and **this.drawPieces()** to refresh the board. t**his.displayMoves()** to visually highlight all valid destination tiles for the selected piece.

4.2 Mousemove - Dragging the Piece

First it checks if a piece is actively being dragged. Then it clears the canvas to avoid ghosting effects and finally redraws the board and all pieces.

Afterwards it calls **displayMoves()** again to re-highlight valid destinations during dragging.

Durign the dragging it draws a **semi-transparent** image of the piece at its original position, to make it easier to remember which piece you picked up.

It then draws the actual piece image under the mouse cursor. It uses **globaAlpha = 0.5** to give a transparent "dragging" effect for smoother UX.

```
canvas.addEventListener("mousemove", (e) => {

    if (this.draggedPiece !== null && isDragging) {
        let mouseX = e.offsetX;
        let mouseY = e.offsetY;

        c.clearRect(0, 0, canvas.width, canvas.height)

        this.drawBoard();
        this.drawPieces();

        this.displayMoves(this.draggedPiece, this.draggedPieceIndex)


        let originalX = originX  + (this.draggedPieceIndex % 8) * tileSize;
        let originalY = originY  + Math.floor(this.draggedPieceIndex / 8) * tileSize;
        c.globalAlpha = 0.5;

        let color = (this.draggedPiece & (Piece.White | Piece.Black)) === Piece.White ? "white" : "black";
        let piece = this.getPiece(this.draggedPiece & 0b111);
        let image = this.pieceImages[`${color}-${piece}`];

        if (image) {
            this.squares[this.draggedPieceIndex] = 0;
            c.drawImage(image, originalX, originalY, tileSize, tileSize);
        }
        c.globalAlpha = 1.0;

        c.drawImage(image, mouseX - tileSize  / 2, mouseY - tileSize  / 2, tileSize, tileSize);
    }
});
```

4.3 Mouseup – Placing the Piece

Mouseup is the most important and logest event. It handles everything that happens after placing a piece, including if placing the piece is even possible or legal. Here I handle everything including but not limited to move validation, castling, promotion, en passant, adding moves to the history, coloring of tiles, etc.

The first if checks if the placed piece is within the bounds of the board:

```
if (newRank < 0 || newRank > 7 || newFile < 0 || newFile > 7 || this.draggedPieceIndex === newIndex) {
    isDragging = this.colorTile("#f55538", this.draggedPieceIndex);
    this.draggedPiece = null;
    this.draggedPieceIndex = null;
    return;
}
```

Afterwards I check if a move is valid, meaning if the specific piece that is picked up can move to each tile on the board.

```
if (!this.isValid(this.draggedPiece, this.draggedPieceIndex, newIndex)) {
    isDragging = this.colorTile("#f55538", this.draggedPieceIndex);
    this.draggedPiece = null;
    this.draggedPieceIndex = null;
    return;
}
```

```
isValid(currentPiece, currentIndex, newIndex, checkingControl = false) {
    let piece = currentPiece & 0b111;

    if (piece === 1) {
        return this.kingMove(currentPiece, currentIndex, newIndex) && this.canCapture(currentPiece, currentIndex, newIndex)
    }

    if (piece === 2) {
        if (checkingControl) {
            let fromRank = Math.floor(currentIndex / 8);
            let fromFile = currentIndex % 8;
            let toRank = Math.floor(newIndex / 8);
            let toFile = newIndex % 8;

            let isWhite = (currentPiece & Piece.White) === Piece.White;

            if (isWhite && toRank === fromRank - 1 && Math.abs(toFile - fromFile) === 1) return true;
            return !isWhite && toRank === fromRank + 1 && Math.abs(toFile - fromFile) === 1;

        }
        return this.pawnMove(currentPiece, currentIndex, newIndex) && this.canCapture(currentPiece, currentIndex, newIndex)
    }

    if (piece === 3) {
        return this.knightMove(currentIndex, newIndex) && this.canCapture(currentPiece, currentIndex, newIndex)
    }

    if (piece === 4) {
        return this.bishopMove(currentIndex, newIndex) && this.canCapture(currentPiece, currentIndex, newIndex)
    }

    if (piece === 5) {
        return this.rookMove(currentIndex, newIndex) && this.canCapture(currentPiece, currentIndex, newIndex)
    }

    if (piece === 6) {
        return (this.rookMove(currentIndex, newIndex) || this.bishopMove(currentIndex, newIndex)) && this.canCapture(currentPiece, currentIndex, newIndex);
    }

    return false;
}
```

Each piece has its own function that returns true or false, depending on whether that specific piece can move in each tile.

Thirdly, **simulateMove()** is called to make sure that the move does not leave the king in check. For example if pawn is pinned by a bishop this if checks whether moving the pawn will leave the king in check. This is done by creating a seperate instance of the board where every move is checked and compared whether it returns a **checked king** or not.

```
if (!this.simulateMove(this.draggedPiece, this.draggedPieceIndex, newIndex)) {
    isDragging = this.colorTile("#f55538", this.draggedPieceIndex);
    let king = this.findKing(this.moveNumber % 2 !== 0);
    this.highlightTile("#f55538", king);
    this.draggedPiece = null;
    this.draggedPieceIndex = null;
    return;
}
```

```
simulateMove(currentPiece, currentIndex, newIndex) {
    let copy = this.copy();

    copy.squares[newIndex] = currentPiece;
    copy.squares[currentIndex] = 0;
```

```
    copy.moveNumber++;

    let isWhite = (currentPiece & Piece.White) === Piece.White;

    copy.controlledSquares = copy.getControlledSquares(isWhite);

    let kingIndex = copy.findKing(isWhite);

    // printBoard(copy.squares)

    return !copy.controlledSquares.has(kingIndex);
}
```

Afterwards there are checks for special moves like castling, pawn promotion and en passant.

The move notation is also handled here. I check what the move made was, convert it to chess notation and then store it.

```
let moveNotation = this.generateMoveNotation(piece, index, newIndex, captured);
this.moveHistory.push(moveNotation);
this.updateMoveList();
```

```
generateMoveNotation(piece, fromIndex, toIndex, capture = false) {
    let pieceType = piece & 0b111;
    let isPawn = pieceType === Piece.Pawn;
    let isKing = pieceType === Piece.King;

    let from = this.toAlgebraic(fromIndex, this.flipped);
    let to = this.toAlgebraic(toIndex, this.flipped);

    if (isKing && Math.abs(fromIndex - toIndex) === 2) {
        return toIndex % 8 === 6 ? 'O-O' : 'O-O-O';
    }

    let pieceChar = {
        [Piece.King]: 'K',
        [Piece.Queen]: 'Q',
        [Piece.Rook]: 'R',
        [Piece.Bishop]: 'B',
        [Piece.Knight]: 'N'
    }[pieceType] || '';

    let move = '';

    if (isPawn && capture) {
        let fromFile = from[0];
        move = fromFile + 'x' + to;
    } else {
        move = pieceChar + (capture ? 'x' : '') + to;
    }

    return move;
}
```

Lastly, there is puzzle logic i will explain later and a check whether the game has ended.

```
checkGameEnd() {
    let moves = this.availableMoves();

    let isWhite = this.moveNumber % 2 !== 0;
    let kingIndex = this.findKing(isWhite);
    let inCheck = this.controlledSquares.has(kingIndex);

    if (moves.length === 0) {
        if (inCheck) {
            document.getElementById("label").innerHTML = (isWhite ? "Checkmate! Black wins!" : "Checkmate! White wins!");
            this.gameOver = true;
        } else {
            document.getElementById("label").innerHTML = "Stalemate! It's a draw.";
            this.gameOver = true;
        }
        return true;
    }

    if (this.isInsufficientMaterial()) {
        document.getElementById("label").innerHTML ="Draw due to insufficient material.";
        this.gameOver = true;
        return true;
    }

    return false;
}
```

5. Piece Movement

Each piece has its own function that details whether a specific move given is legal for that piece. Except for the **Queen,** which uses both the **Rook's** and **Bishop's** functions for it's movement.

King:

```
kingMove(currentPiece, currentIndex, newIndex) {
    let controlled =  this.controlledSquares

    if (controlled.has(newIndex))
        return false;

    let row1 = Math.floor(currentIndex / 8);
    let col1 = currentIndex % 8;
    let row2 = Math.floor(newIndex / 8);
    let col2 = newIndex % 8;

    let rowDiff = Math.abs(row1 - row2);
    let colDiff = Math.abs(col1 - col2);

    if (rowDiff <= 1 && colDiff <= 1) {
        return true;
    }

    if (rowDiff === 0 && colDiff === 2) {
        let isWhite = (currentPiece & Piece.White) !== 0;

        if (isWhite) {
            if (this.whiteKingMoved) return false;
        } else {
            if (this.blackKingMoved) return false;
        }

        let rookCol = (col2 === 6) ? 7 : 0;
        let rookIndex = row1 * 8 + rookCol;
        let rookPiece = this.squares[rookIndex];

        if ((rookPiece & 0b111) !== Piece.Rook) return false;

        if (isWhite) {
            if (col2 === 6 && this.whiteRookMoved.kingSide) return false;
            if (col2 === 2 && this.whiteRookMoved.queenSide) return false;
        } else {
            if (col2 === 6 && this.blackRookMoved.kingSide) return false;
            if (col2 === 2 && this.blackRookMoved.queenSide) return false;
        }

        let step = col2 > col1 ? 1 : -1;
        for (let c = col1 + step; c !== rookCol; c += step) {
            if (this.squares[row1 * 8 + c] !== 0) return false;
        }

        return true;
    }

    return false;
}
```

Pawn:

```
pawnMove(currentPiece, currentIndex, newIndex) {
    let isWhite = this.moveNumber % 2 !== 0;

    let direction = isWhite ? -1 : 1;
    let startRow = isWhite ? 6 : 1;

    if (this.flipped) {
        direction = isWhite ? 1 : -1;
        startRow = isWhite ? 1 : 6;
    }


    let row1 = Math.floor(currentIndex / 8);
    let col1 = currentIndex % 8;
    let row2 = Math.floor(newIndex / 8);
    let col2 = newIndex % 8;

    let distance = row2 - row1;

    if (col1 === col2) {
        if (this.squares[newIndex] !== 0) return false;
        if (distance === direction) return true;
        if (row1 === startRow && distance === 2 * direction) {
            let intermediateIndex = currentIndex + 8 * direction;
            if (this.squares[intermediateIndex] === 0 && this.squares[newIndex] === 0) {
                return true;
            }
        }
    }

    if (this.enPassantCount > 0 && this.enPassantCount < 2) {
        if (Math.abs(col2 - col1) === 1 && distance === direction) {
            if (newIndex === this.enPassantSquare - 8 || newIndex === this.enPassantSquare + 8) {
                return true;
            }
        }
    }

    if (Math.abs(col2 - col1) === 1 && distance === direction) {
        return this.squares[newIndex] !== 0;
    }

    return false;
}
```

Knight:

```
knightMove(currentIndex, newIndex) {
        let currentRow = Math.floor(currentIndex / 8);
        let currentCol = currentIndex % 8;

        let newRow = Math.floor(newIndex / 8);
        let newCol = newIndex % 8;

        let rowDiff = Math.abs(newRow - currentRow);
        let colDiff = Math.abs(newCol - currentCol);

        return (rowDiff === 2 && colDiff === 1) || (rowDiff === 1 && colDiff === 2)
}
```

Bishop:

```
bishopMove(currentIndex, newIndex) {
    let currentRow = Math.floor(currentIndex / 8);
    let currentCol = currentIndex % 8;

    let newRow = Math.floor(newIndex / 8);
    let newCol = newIndex % 8;

    let rowDiff = Math.abs(newRow - currentRow);
    let colDiff = Math.abs(newCol - currentCol);

    if (rowDiff !== colDiff) return false;

    let rowStep = newRow > currentRow ? 1 : -1;
    let colStep = newCol > currentCol ? 1 : -1;

    let row = currentRow + rowStep;
    let col = currentCol + colStep;

    while (row !== newRow && col !== newCol) {
        let index = row * 8 + col;
        if (this.squares[index] !== 0) return false;
        row += rowStep;
        col += colStep;
    }

    return true;
}
```

Rook:

```
rookMove(currentIndex, newIndex) {
    let currentRow = Math.floor(currentIndex / 8);
    let currentCol = currentIndex % 8;
    let newRow = Math.floor(newIndex / 8);
    let newCol = newIndex % 8;

    if (currentRow === newRow) {
        let colStep = newCol > currentCol ? 1 : -1;
        for (let col = currentCol + colStep; col !== newCol; col += colStep) {
            let index = currentRow * 8 + col;
            if (this.squares[index] !== 0) return false;
        }
        return true;
    }

    if (currentCol === newCol) {
        let rowStep = newRow > currentRow ? 1 : -1;
        for (let row = currentRow + rowStep; row !== newRow; row += rowStep) {
            let index = row * 8 + currentCol;
            if (this.squares[index] !== 0) return false;
        }
        return true;
    }

    return false;
}
```

6. Puzzle Integration

Each puzzle is an object with the following properties:

fen: Initial position.

turn: whose turn it is.

solution: an array of correct moves, the player has to follow.

opponentMoves: expected opponent responses to be automated.

Move count to solve the puzzle.

```
{
  id: 7,
  fen: "7k/1p4p1/p1n5/3pb1Bq/7P/2P2r2/PPB3Q1/R5K1",
  turn: "Black",
  opponentMoves: [[9, 17]],
  maxMoves: 3,
  solution: ["Rg3", "Bxg3"],
},
```

Puzzles are loaded using the **loadPuzzle()** function, which initializes the board with the puzzle FEN. Adjusts turn and move count. Sets expected solutions. Loads opponent responses for auto-play.

```
function loadPuzzle(puzzle) {
  c.clearRect(0, 0, canvas.width, canvas.height);

  let newCanvas = canvas.cloneNode(true);
  canvas.replaceWith(newCanvas);
  canvas  = newCanvas;
  c  = canvas.getContext('2d');

  board  = new Board()

  board.initializePieces(puzzle.fen);

  if (puzzle.turn === "Black") {
    flipBoard()
    board.moveNumber = 2;
  }

  board.solution = puzzle.solution;
  board.opponentMoves = puzzle.opponentMoves;
  board.maxMoves = puzzle.maxMoves;
  board.drawBoard();
  board.drawPieces();
  board.handleDragAndDrop();

  window.onresize = () => resizeCanvas(board);
  resizeCanvas(board);

  document.getElementById("moveList").innerHTML = ""
  document.getElementById("label").innerHTML = puzzle.turn + " to move."
}
```

Every player move is checked against the expected solution in **checkSolution()**. If the move is incorrect, visual feedback is given. If correct, the game may execute one or more automated opponent moves to simulate realistic responses.

```
async checkSolution() {
    let j = 0;
    let correct = this.moveHistory.every((move, i) => {
        if (i % 2 === 0) {
            return move === this.solution[j++];
        }
        return true;
    });


    let label = document.getElementById("label")
    this.moveHistory.every((move, i) => console.log(move))

    if (correct) {
        this.highlightTile("#80ff80", this.solutionIndex);

        let isWhite = this.moveNumber % 2 !== 0;

        if (this.controlledSquares.has(this.findKing(isWhite))) {
            this.kingChecked = true;
            this.highlightTile("red", this.findKing(isWhite));
        } else {
            this.kingChecked = false;
        }

        label.innerHTML = this.moveHistory[this.moveHistory.length - 1] + " is Correct!"
        label.style.backgroundColor = "seagreen"

        if (this.moveHistory.length === this.maxMoves) {
            label.innerHTML =  "Success!"
            this.gameOver = true;
        }

        if (this.opponentMoves.length > 0) {
            let moves = this.opponentMoves.shift();

            let index = moves[0]
            let newIndex = moves[1]
            let piece = this.squares[index]

            let captured = this.squares[newIndex] !== 0 || (this.enPassantTarget && (piece & 0b111) === Piece.Pawn &&
newIndex === this.enPassantSquare);

            await this.makeMove(piece, index, newIndex);

            let moveNotation = this.generateMoveNotation(piece, index, newIndex, captured);
            this.moveHistory.push(moveNotation);
            this.updateMoveList();

            let isWhite = this.moveNumber % 2 !== 0;
            if (this.controlledSquares.has(this.findKing(isWhite))) {
                this.kingChecked = true;
                this.highlightTile("red", this.findKing(isWhite));
            } else {
                this.kingChecked = false;
            }
            this.moveNumber++;
        }
```

```
        return true;
    } else {
        document.getElementById("label").innerHTML = "Incorrect!"
        label.style.backgroundColor = "darkred"
        return false;
    }
}
```

On success: Green tile highlight. Label displays “**Correct”** or **“Success!”.**

On failure: **Red tile highlight.** Move is reverted. Label displays “**Incorrect”.**

7. Conclusion

This chess puzzle trainer is a comprehensive JavaScript-based application that demonstrates real time graphics rendering, event-driven interactivity, complex rule implementation, and state management in a single file. The use of FEN for board initialization and algebraic notation for move tracking reflects professional standards. Its modular design and clear separation between game logic and rendering make it suitable for future enhancements.