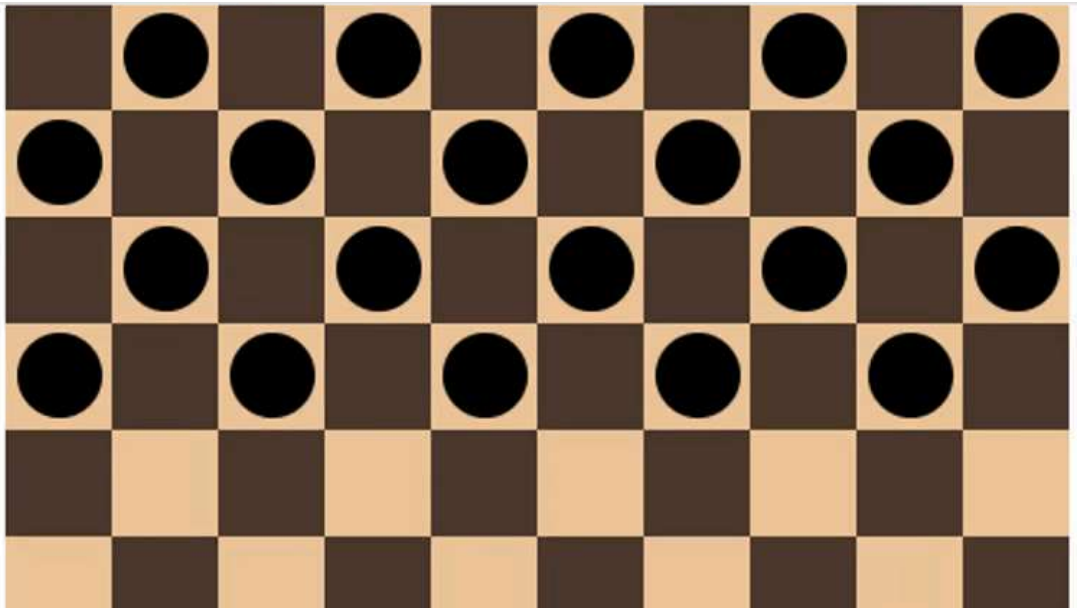


Marius Vincent NIEMET
Posted on 13 de jul. de 2022 • Updated on 14 de jul. de 2022

How I built a checkers game with javaScript

One of my goals for this year is to build more projects with javascript vanilla instead of using Reactjs or another framework every time. I was thinking about the kind of project that I can build and I thought about a checkers game because it's something that I play a lot and making little games is a lot different than what I usually do at work, so I found it to be a fun, achievable challenge.
For people who don't know what a checkers game is, it's a group of strategy board games for two players which involve diagonal moves of uniform game pieces and mandatory captures by jumping over opponent pieces. You have many variants of a checkers game with different rules but in this one, we will refer to the international rules. Let's see how we can build that.

The Board

When I face a problem, my first thought is to split it into multiple small items, and when I split the checkers game the first part was the board, how can I represent it? which data structure should I use? Honestly, I didn't think twice because the 2D array seemed perfect for this use case for me. Secondly, if you check the board below, you will notice that each case have three possible value, either it's occupied by a white piece or a black piece, or it's empty. I decided to use a different value for each case:

- empty case = 0

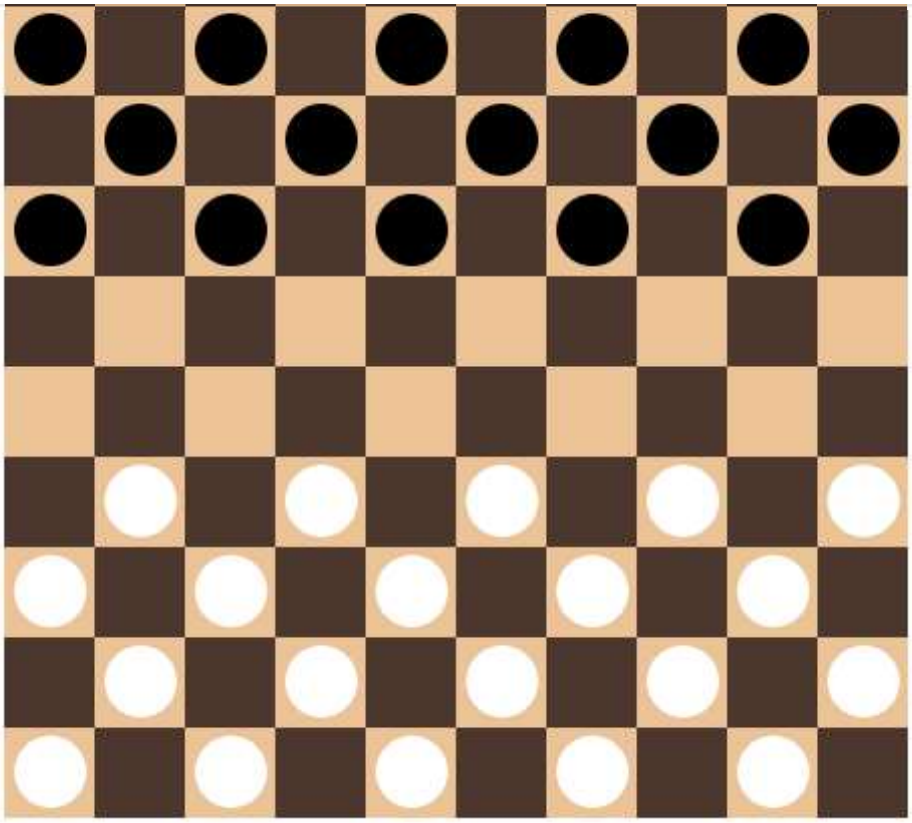
```
let board = [
  [0, -1, 0, -1, 0, -1, 0, -1, 0, -1],
  [-1, 0, -1, 0, -1, 0, -1, 0, -1, 0],
  [0, -1, 0, -1, 0, -1, 0, -1, 0, -1],
  [-1, 0, -1, 0, -1, 0, -1, 0, -1, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
  [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
];
```

Now I had a 2d array that represents the board but we all agree it doesn't look like the image above. I had to create the board game by using the 2d array and make it looks like a real checkers game board function to build the checkers game

```
function builBoard() {
  game.innerHTML = "";

  for (let i = 0; i < board.length; i++) {
    const element = board[i];
    let row = document.createElement("div"); // create div for each row
    row.setAttribute("class", "row");

    for (let j = 0; j < element.length; j++) {
      const elmt = element[j];
      let col = document.createElement("div");
      let piece = document.createElement("div");
      let caseType = "";
      let occupied = "";

      if (i % 2 === 0) {
        if (j % 2 === 0) {
          caseType = "Whitecase";
        } else {
          caseType = "blackCase";
        }
      } else {
        if (j % 2 !== 0) {
          caseType = "Whitecase";
        } else {
```

```
        // add the piece if the case isn't empty
        if (board[i][j] === 1) {
          occupied = "whitePiece";
        } else if (board[i][j] === -1) {
          occupied = "blackPiece";
        } else {
          occupied = "empty";
        }

        piece.setAttribute("class", "occupied " + occupied);

        // set row and colum in the case
        piece.setAttribute("row", i);
        piece.setAttribute("column", j);
        piece.setAttribute("data-position", i + "-" + j);

        //add event listener to each piece
        piece.addEventListener("click", movePiece);

        col.appendChild(piece);

        col.setAttribute("class", "column " + caseType);
        row.appendChild(col);

      }

      game.appendChild(row);
    }
  }
```

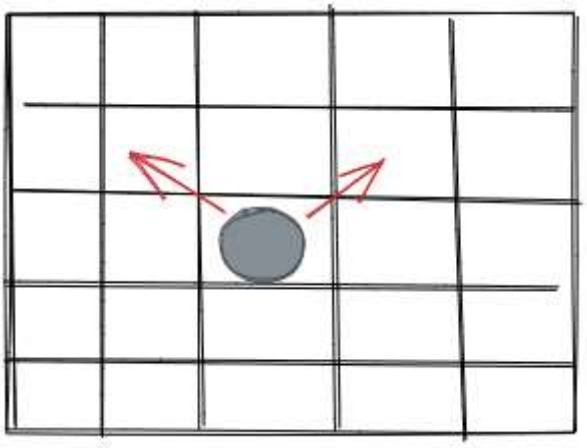
So I added a bit of logic, we have ten lines and since we are in javascript, we start counting from 0, which makes 9 lines. If the number of the line is even the first case will be black, the second white, and so on otherwise the first case will be white, the second black, and so on. Also while traversing the array if the current value was equal to -1 or 1, I added the class "occupied " to mark that this case isn't empty and according to the value, if it's -1 I addedblack pieceotherwisewhite Piece if it's 1, to know which kind of piece occupies this case. Finally, I added a data-position attribute and an event listener to easily determine which case the user clicked and retrieve its position and value.
The result:



Yeah, CSS style made some miracles :)

Can I move? and where?

A move consists of moving a piece diagonally to an adjacent unoccupied square.
Players alternate turns
Basically, a player moves his piece by making two actions, firstly he selects the piece he wants to move, in the meantime, the engine defines where this piece can move, and mark them, then the player selects a new position from those marked by the engine. The piece can only move forward.



listener to each piece to allow me to get the row and the column in which the user clicked.
Let me show the code that handles movements and then explain it.

```
let currentPlayer = 1;

let newPiecesPositions = [];

function movePiece(e) {
  let piece = e.target;
  const row = parseInt(piece.getAttribute("row"));
  const column = parseInt(piece.getAttribute("column"));
  let p = new Piece(row, column);


  if (newPiecesPosition.length > 0) {
    enableToMove(p);
  }

  if (currentPlayer === board[row][column]) {
    findPossibleNewPosition(p, player);
  }
}

function findPossibleNewPosition(piece, player) {
  if (board[piece.row + player][piece.column + 1] === 0) {
    readyToMove = piece;
    markPossiblePosition(piece, player, 1);
  }

  if (board[piece.row + player][piece.column - 1] === 0) {
    readyToMove = piece;
    markPossiblePosition(piece, player, -1);
  }
}

function enableToMove(p) {
  let found = false;
  let newPosition = null;


  posNewPosition.forEach((element) => {
```

```
        return;
      }
    });

    if (found){
       // if the current piece can move on, edit the board and rebuild
      board[newPosition.row][newPosition.column] = currentPlayer;
      board[readyToMove.row][readyToMove.column] = 0;

      // init value
      readyToMove = null;
      posNewPosition = [];
      capturedPosition = [];

      currentPlayer = reverse(currentPlayer);

      displayCurrentPlayer();
      builBoard();
    } else {
      builBoard();
     }
  }
```

I declared the variable currentPlayer to know which player is allowed to move his piece, here the value is 1 which means that it is up to the black player to move his piece.
Since a piece can move towards two positions, the engine has to define them, that's the role of the newPiecesPositions . Basically, it's an array of possible new positions.
When a user clicks on a piece on the board, the movePiecefunction is called. Firstly we retrieve the row and column of the selected piece, we create a new Piece object. Since users alternate turns, at line 16 we check if the piece on which the user clicked is a current player's piece, if so, we call the function by passing it two parameters, the piece we want to move and the current player value.
As his name indicates the aim of the findPossibleNewPositions function is to find the possible new positions of the piece it receives as a parameter and store them in the newPiecesPositionsarray. Since a piece can only move forward and diagonally, the function checks if the two cases in front of it are empty, if so it saves the piece in a variable called readyToMove and calls the markPossiblePosition function. has as aim to add a green background for the new possibles positions cases to show to the user where he can move his piece and save those positions in an array.

Finally moving :)

After the engine has calculated and stored the new positions, the player clicks for the second time on a case, he clicks where the piece should move.

```
let piece = e.target;
const row = parseInt(piece.getAttribute("row"));
const column = parseInt(piece.getAttribute("column"));
let p = new Piece(row, column);

 if (posNewPosition.length > 0) {
    enableToMove(p);
 }

function enableToMove(p) {
  let find = false;
  let newPosition = null;
  // check if the case where the player play the selected piece can move on
  posNewPosition.forEach((element) => {
    if (element.compare(p)) {
      find = true;
      newPosition = element;
      return;
    }
  });

  if (find) moveThePiece(newPosition);
  else builBoard();
}

function moveThePiece(newPosition) {
  // if the current piece can move on, edit the board and rebuild
  board[newPosition.row][newPosition.column] = currentPlayer;
  board[readyToMove.row][readyToMove.column] = 0;

  // init value
  readyToMove = null;
  posNewPosition = [];
  capturedPosition = [];

  currentPlayer = reverse(currentPlayer);

  displayCurrentPlayer();
```

When the player clicks the second time on a case, the engine checks if the posNewPositions array isn't empty, if so that means the user had already selected the piece he wants to move and this case is where he wants to move the old piece. We call the function enableToMove by passing it the current case. Since a piece can only move forward and diagonally, the engine checks if the selected case is part of the posNewPositions then it edits the board. The current case receives the player value to mark it as occupied and the old case receives 0 to mark it as empty. Finally, the board is rebuilt, and all variables are reset.

Capture an opponent piece

If the adjacent square contains an opponent's piece, and the square immediately beyond it is vacant, the piece may be captured (and removed from the game) by jumping over it.

```
function findPieceCaptured(p, player) {
  let found = false;
  if (
    board[p.row - 1][p.column - 1] === player &&
    board[p.row - 2][p.column - 2] === 0
  ) {
    found = true;
    newPosition = new Piece(p.row - 2, p.column - 2);
    readyToMove = p;
    markPossiblePosition(newPosition);
    // save the new position and the opponent's piece position
    capturedPosition.push({
      newPosition: newPosition,
      pieceCaptured: new Piece(p.row - 1, p.column - 1),
    });
  }

  if (
    board[p.row - 1][p.column + 1] === player &&
    board[p.row - 2][p.column + 2] === 0
  ) {
    found = true;
    newPosition = new Piece(p.row - 2, p.column + 2);
    readyToMove = p;
    markPossiblePosition(newPosition);
    // save the new position and the opponent's piece position
```

```
    });
  }

  i


  return found;
}
```

When a player clicks on a case, the engine defines if it can capture an opponent piece. The function findPieceCaptured receives the piece and the player value as parameters then check if the adjacent case contains an opponent's piece, and the case immediately beyond it is vacant, if so two pieces are saved, the new position and the opponent's piece that gonna be captured.

```
function enableToCapture(p) {
  let find = false;
  let pos = null;
  capturedPosition.forEach((element) => {
    if (element.newPosition.compare(p)) {
      find = true;
      pos = element.newPosition;
      old = element.pieceCaptured;
      return;
    }
  });

  if (find) {
    // if the current piece can move on, edit the board and rebuild
    board[pos.row][pos.column] = currentPlayer; // move the piece
    board[readyToMove.row][readyToMove.column] = 0; // delete the old position
    // delete the piece that had been captured
    board[old.row][old.column] = 0;

    // reinit ready to move value

    readyToMove = null;
    capturedPosition = [];
    posNewPosition = [];
    displayCurrentPlayer();
    builBoard();
    // check if there are possibility to capture other piece
```

```
    }
  }
```

After the second player click, the engine checks if the selected square is part of the capturedPosition array, if so the board is edited, the old square receives 0 to mark it as empty, the opponent's square receives 0 because he lost a piece and the new position receives the player value.

Have I still pieces?

Below the board, we have the number of coins for each player displayed. To do this you have to go through the matrix and count the number of pieces. Fortunately, the board is already built by going through the whole matrix. I only had to add a few lines of code to count the number of pieces per player during the board construction, and I created a second function that receives these values as parameters and modifies the dom by displaying them.

```
    if (board[i][j] === -1) {
      blackPiecesCounter++;
    } else if (board[i][j] === 1) {
      whitePiecesCounter++;
    }

  function displayCounter(blackPiecesCounter, whitePiecesCounter) {
      var blackContainer = document.getElementById("black-player-count-pieces");
      var whiteContainer = document.getElementById("white-player-count-pieces");
      blackContainer.innerHTML = blackPiecesCounter;
      whiteContainer.innerHTML = whitePiecesCounter;
  }
```

Is it my turn?

The current player is displayed on the top of the board. I just created a function that is called each time a player moves a piece on the board which means it's the other player's turn to play. The function retrieves the old displayed piece and replaces it with a white piece if it was black and vice versa by switching the class.

```
function displayCurrentPlayer() {
  let container = document.getElementById("next-player");
  if (container.classList.contains("whitePiece")) {
```

```
    }
  }
```

Winning

A player with no valid move remaining loses. This occurs if the player has no pieces left.
The know which player won the game, I had to check the number of pieces for each player every time the board was rebuilt, if one player had 0 pieces that mean the other player won then a modal should be displayed.

```
if (black === 0 || white === 0) {
    modalOpen(black);
  }
}

function modalOpen(black) {
  document.getElementById("winner").innerHTML = black === 0 ? "White" : "Black";
  document.getElementById("loser").innerHTML = black !== 0 ? "White" : "Black";
  modal.classList.add("effect");
}
```

The code above displays the following modal:

Conclusion

I hope you liked my write-up for the checkers game app! Of course, the full code is available on Github
Finally, we have a light version that you can play with your friends. They are some features that can be added to improve the game such as allowing a piece to become a king or adding a server to play online. There are also a few little bugs here and there, such as moving pieces that are at the board extremity. I'm sure if you try to find more bugs you'll be able to.
This app was fun to make, I found it easier to code than to solve :)
Hope you enjoyed reading about this and feel inspired to make your own little games and projects because I think this kind of project can only help you to improve your problem-solving skills.
The Nothing 🔺

Top comments (0)

JOINED

2 de out. de 2020

More from Marius Vincent NIEMET

How to create a local Kubernetes cluster with Kind

#kubernetes #orchestration #kind #infrastructre

Containers Orchestration and Kubernetes

Containers Orchestration and Kubernetes

Got 99 problems but auth deployment ain't one!

Auth0 + Vercel join forces to make your life easier.

Learn more