Developing a simple Tic-Tac-Toe (Noughts & Crosses) game in Javascript

V1

Introduction

A Tic Tac Toe game in which you can play the computer is a nice beginner project.

It is about as simple as a 'deterministic' (no random factors) turn-based two-player game can be but still offers an interesting little challenge when it comes to implementing a strategy for the computer player.

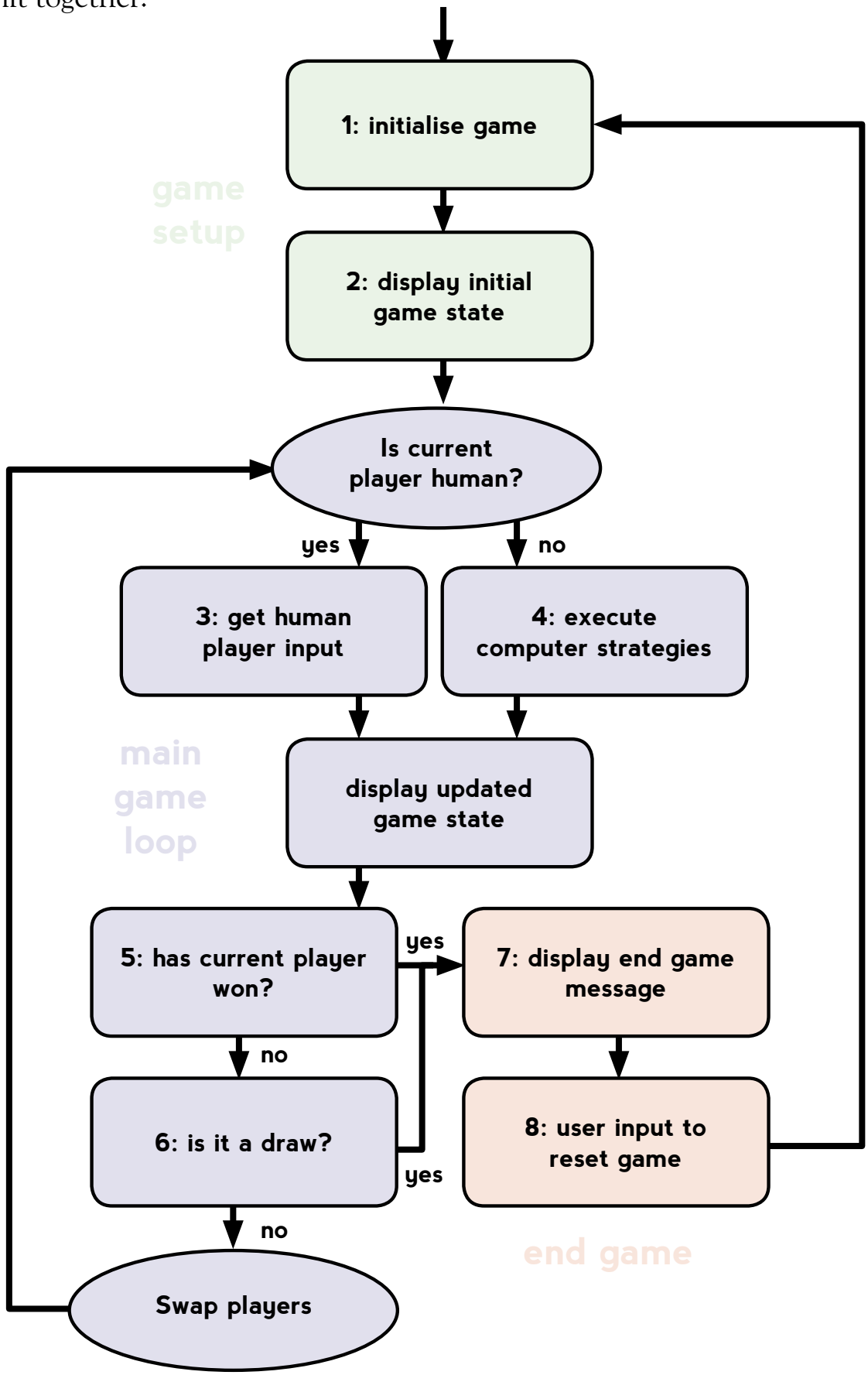
The game can provide a playground for exploring elementary AI decision-making, but a perfect solution does exist and that is what I will investigate the implementation of here. It is impossible to achieve anything better than a draw against this kind of perfect player, so designing such a game is probably more enjoyable than playing it :)

It is also a good exercise in the practice of separating the inner game 'model' that lies 'under the hood' from the interface and visual display that the player will actually interact with.

I will only discuss the conceptual framework we need to develop for such a program rather than any actual details of the Javascript needed to develop it, so hopefully might be a good starting point for people developing their own version.

The anatomy of a Tic Tac Toe game

Here is a diagram showing the basic functions we will need in a game and how they fit together:

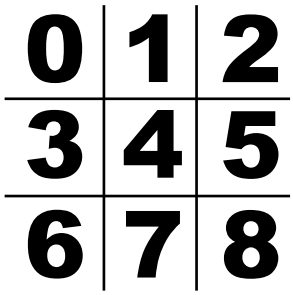


The game model

Ignoring for now the various technicalities and aesthetic choices that come with displaying the game board on screen and allowing the user to interact with it, it would be good if we could settle on a nice simple, easy to manipulate data representation of the game that we can work with as it progresses. This will also allow us to develop the various functions we will need in isolation.

The board state

We can think of the board as nine squares numbered 0-8, eg:



The 3x3 grid of the board immediately suggests a 3x3 array like so: [[0,1,2],[3,4,5],[6,7,8]] in which each of the elements represents a square on the board and the contents of the element represents its current state, empty, 'X' or 'O'.

We could also just use a simple one-dimensional array representing the squares left-to-right and top-to-bottom, eg [0,1,2,3,4,5,6,7,8], but I think the 3x3 array is more intuitive way of thinking about it so will use that throughout this document.

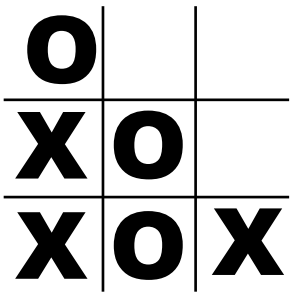
The players

If we wanted to be literal about it we could simply represent each move on the board in the array with strings, eg "X" and "O", and maybe a "" for an empty square.

Instead, representing the player's moves with numbers will make assessing the state of the board quicker and easier.

I suggest a 1 for an X, a -1 for an O and a 0 for an empty square. Then we can, for example, add up a column, row or diagonal in the array and if it equals 3 we know X has won., or if it sums to -3 then O has won.

So, for example, the **BOARD STATE ARRAY** [[-1,0,0],[1,-1,0],[1,-1,1]] represents this board:



I will use the term '**BOARD STATE ARRAY**' to refer to this array object.

Notes on the game functions

A more detailed examination of each function from the previous page.
Inputs and outputs detailed here can either be global variables / user inputs / display instructions OR function arguments and return values depending on each case. They are for indication only.

1: Initialise game

INPUTS: User mouse click
OUTPUTS: board state array, is human X or O?

- Allow user to click on a button to choose to play as X or O.
  X plays first.

- Initialise the game board:
  board state array = [[0,0,0][0,0,0][0,0,0]]

2: Display initial/updated game state

INPUTS: board state array
OUTPUTS: screen rendering of board

- Uses current condition of the board state array to render an on-screen representation in as pretty a fashion as desired.

- Could use basic draw functions – lines, circles, fills etc.

Or go more advanced with image files/animations.

The squares on the board display need to match the positions of the click detection areas used for user input.

3: Get human player input

INPUTS: user mouse click, board state array
OUTPUTS: updated board state array

- If current player is human, we need to wait for the user to make a valid move via a mouse click.

- We need click detection areas corresponding to the remaining empty squares, and to update the board state array when a valid move is made.

4: Execute computer strategies

INPUTS: board state array
OUTPUTS: updated board state array

NOTE: This will likely be by some way the most involved part of the game.

- If the current player is the computer, we will pass the board state array to a computer strategy function and it will return the array with its next move added.

- A perfect strategy exists, which can be found at https://en.wikipedia.org/wiki/Tic-tac-toe#Strategy

- A perfect computer player will always win or draw. For a marginally more interesting game, future versions could explore a variable difficulty with imperfect computer players that make some mistakes.

The computer evaluates a ranked list of preferred moves and executes the first possible move in the list. In brief these moves are:

1: Win

2: Block

3: Fork

4: Block fork

5: Centre

6: Opposite corner

7: Empty corner

8: Empty side

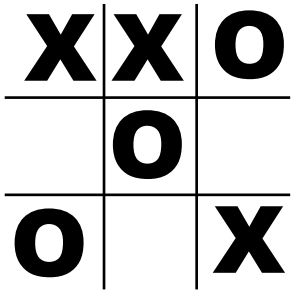
Each of these moves would probably be best developed as an individual function for simplicity of development and testing. I will look at these in more detail later.

5: Has current player won?

INPUTS: board state array, current player value (1/-1)
OUTPUTS: true/false

- After the current player has made their move and it has been displayed on screen we need to find out if they have won.
- We need to look at the sums of **each of the rows, columns and diagonals** and see if any add up to 3 * the current player value – either 3, or -3.
- If so, the current player has won and we can exit the game loop.

For example,
the board state array
[[1,1,-1],[0,-1,0],[-1,0,1]]
represents the winning board:



If we evaluate the sum of the diagonal, eg the indices: [0][2] + [1][1] + [2][0], we can see it adds to -3. So we can deduce that 'O' has won.

6: Is it a draw?

INPUTS: turn counter variable OR board state array
OUTPUTS: true/false

If we have reached a draw we must also exit the game loop.

- Simplest way to find out would be a turn counter variable that increments with each move. If we have reached move 9 we know that there are no empty spaces left on the board and the game must end with a draw.
- Or we could search the BOARD STATE ARRAY for zeros. If there are no remaining zeros then the board is full and we must terminate the game with a draw.
- A truly intelligent approach might be able to detect if a draw has been reached before the board is full – if it is impossible for either player to get three in a row even before the board is full.

7: Display end game message

INPUTS: game result
OUTPUTS: display on screen message

- Display the results of the game, whether win or draw.

- If a human player has won, congratulate them for their statistically impossible victory :)

8: User input to reset game

INPUTS: user input
OUTPUTS: restart game

- Allow user to click a button that resets the game back to the beginning.

Detailed computer strategy functions

It would be a good idea to break up the computer strategies function (5) into individual functions per move type, that can be tested and developed in isolation. Here are some more thoughts on each.

INPUTS ON ALL: board state array
OUTPUTS ON ALL: board state array, flag to indicate if move has been made

1: Win

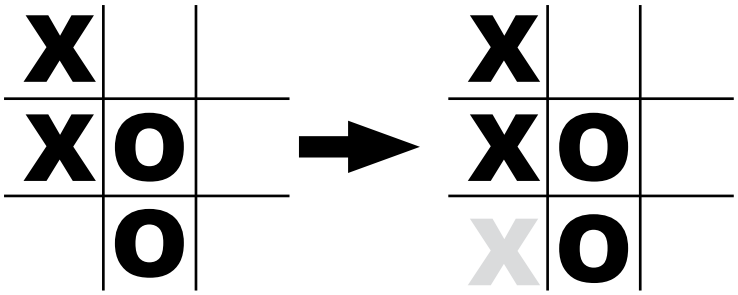
'If player has two in a row, place third for three in a row.'

HOW?

- Look for sums of columns, rows or diagonals that add up to 2 * the current player value (2 for X, -2 for O).

- Add player move to make three in a row and return board state array.

Example:
for BOARD STATE ARRAY
[[1,0,0],[1,-1,0],[0,-1,0]]
which represents board:



the sum of the first column at indices [0][0] + [1][0] + [2][0] is 2. So we need to place a move at [2][0] and return the board state array.

2: Block

'If the opponent has two in a row, the player must play the third themselves to block the opponent.'

HOW?

- Essentially the same as 1: WIN, but looking for columns, rows or diagonals that sum to 2 * the opposing player's value. The search function for block and win could probably be combined into one, with the input being either the current or opposing player's value.

3: Fork

'Create an opportunity where the player has two threats to win (two non-blocked lines of 2).'

HOW?

- See if we can find two rows, columns or diagonals that each sum to 1 * the player value and share an empty square. Make a move into that square for a 'fork'.

4: Block opponent's fork

'Create two in a row to force opponent into defending UNLESS it creates a fork. If opponent can create a fork next turn, block it.'

HOW?

- Look for a column, row or diagonal that adds up to 1* the player value. Go through possible moves to make two in a row. Then use the fork algorithm above with the opponent's player value to test if each move will allow the opponent to create a fork. If not, use this move. If they can fork, block the fork.

5: Centre

'A player marks the center. (If it is the first move of the game, playing on a corner gives "O" more opportunities to make a mistake and may therefore be the better choice; however, it makes no difference between perfect players.)'

HOW?

- If the centre square is empty, place a move there.

6: Opposite corner

'If the opponent is in the corner, the player plays the opposite corner.'

HOW?

- Evaluate each corner square. If contains opponent player value, evaluate opposite corner. If empty, make this move.

3: Empty corner

‘The player plays in a corner square.’

HOW?

- Evaluate each corner square. If empty, make this move.

4: Empty side

‘The player plays in a middle square on any of the 4 sides. ’

HOW?

- Evaluate each side square. If empty, make this move.