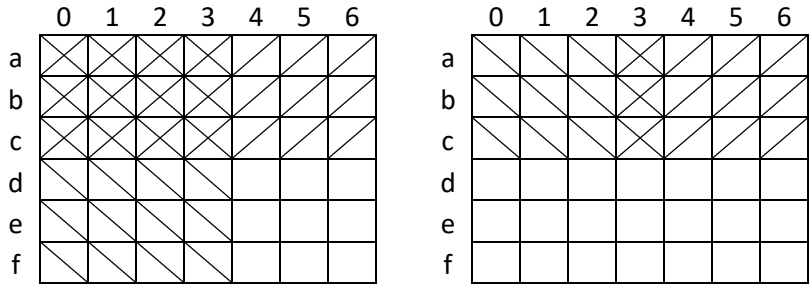
Connect Four with jQuery
by Michael Brothers

Connect Four is a tic-tac-toe like game in which two players drop discs into a 7x6 board. The first player to get four in a row (either vertically, horizontally, or diagonally) wins.

The diagram below shows which squares need to be checked for a win, avoiding duplication:



horizontal check
vertical check

diagonal check right
diagonal check left

For example, checking the squares in rows a, b & c for a vertical win will cover all possibilities. Tests should iterate from the top row downward, since empty squares can be skipped.

So, what's the fastest check algorithm?

Option 1 is to perform each TEST individually, with a dedicated set of loops:

for column 0 to 3, for row a to f, **check horizontal**	for column 0 to 6, for row a to c, **check vertical**	for column 0 to 3, for row a to c, **check diag-right**	for column 3 to 6, for row a to c, **check diag-left**
4x6x1 = 24 tests	7x3x1 = 21 tests	4x3x1 = 12 tests	4x3x1 = 12 tests

Option 2 is to call each SQUARE individually, and perform all necessary tests before moving on

for column 3, **for row a to c,** check horizontal check vertical check diag-right check diag-left	**for column 0 to 2,** **for row a to c,** check horizontal check vertical check diag-right	**for column 4 to 6,** **for row a to c,** check vertical check diag-left	**for column 0 to 3,** **for row d to f,** check horizontal
1x3x4 = 12 tests	3x3x3 = 27 tests	3x3x2 = 18 tests	4x3x1 = 12 tests

Both methods perform a total of 69 tests. In the end, I went with Option 2.

I start by building an array of objects. Each object represents a square on the board, with properties for color, box (eventual position on the board), and the four possible checks (horizontal, vertical, diagonal-down-right and diagonal-down-left).

The first iteration creates the objects and assigns starting values to each property:

```
var tiles = []
for (i=0; i<42; i++) {
  tiles[i] = {}
  tiles[i].color = null;
  tiles[i].box = 7*(i%3)+Math.floor(i/3)+14*Math.floor(i/21);
  tiles[i].horz = null;
  tiles[i].vert = null;
  tiles[i].diagR = null;
  tiles[i].diagL = null;
}
```

The next iterations set the appropriate check properties to "unknown":

```
for (a=0; a<12; a++) {
  tiles[a].horz = "unknown";
  tiles[a].diagR = "unknown";
}
for (a=21; a<33; a++) {
  tiles[a].horz = "unknown";
}
```

```
for (a=0; a<21; a++) {
  tiles[a].vert = "unknown";
}
for (a=9; a<21; a++) {
  tiles[a].diagL = "unknown";
}
```

We need to make a final rearrangement because when "tiles" was set up, squares were arranged in this order, which made it easy to assign check values:

0	3	6	9	12	15	18
1	4	7	10	13	16	19
2	5	8	11	14	17	20
21	24	27	30	33	36	39
22	25	28	31	34	37	40
23	26	29	32	35	38	41

However, for gameplay we want squares in this order:

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41

The last iteration changes the order of squares, casting them into a new array called "game":

```
var game = []
for (b of tiles) {
  game[b.box] = b
}
```

Next, I paint the screen

```
var screen = "";
for (var c=0; c<42; c++) {
  screen += "<div class='tile'></div>"
}
$(".board").html = screen
```

I define a "switchPlayer" function. A "player" variable holds a color value.

```
var player = "red";
function switchPlayer () {
  if (player === "red") {
    player = "blue"
  } else {
    player = "red"
  }
  $("#banner").css("color", player);
  $("#banner").html(player.toUpperCase() + " - It's Your Move");
}
switchPlayer()
```

I assign event handlers to every square, and remove them as squares are played.

```
var chooser = $(".tile");
chooser.on("mouseenter", function() {
  $(this).css("background-color", player);
});
chooser.on("mouseleave", function() {
  $(this).css("background-color", "");
});
chooser.on("click", playMove);
function playMove() {
  // SET BOTTOM-MOST AVAILABLE SQUARE TO PLAYER'S COLOR & REMOVE EVENT HANDLERS
  var t = chooser.index(this);
  for (var v=35; v>=0; v-=7 ) {
    var sum = t + v;
    if(game[sum]) {
      if (!game[sum].color) {
        game[sum].color = player;
        chooser.eq(sum).off("mouseenter");
        chooser.eq(sum).off("mouseleave");
        chooser.eq(sum).off("click");
        chooser.eq(sum).css("background-color", player);
        break;
      }
    }
  }
  if (!testAll()) {  // IF testAll COMES BACK FALSE, KEEP PLAYING
    switchPlayer()
  }
}
```

The last section of code performs the win checks. I iterate through every square starting at the top, then through every kind of check. For each square, a test is only performed if its value is "unknown" – otherwise it's skipped. If a test fails for a fully populated set of squares, the tested square is marked "fail" to avoid retesting.

```
var tests = {horz: [1,2,3], vert: [7,14,21], diagL: [6,12,18], diagR: [8,16,24]}
function testAll() {
  for (var x=0; x<39; x++) { // the last three squares are never checked
    for (var key of Object.keys(tests)) {
      if (game[x][key]) {
        if (game[x][key] != "fail") {
          if (game[x].color && game[x+tests[key][0]].color &&
          game[x+tests[key][1]].color && game[x+tests[key][2]].color) {
            if (game[x].color == game[x+tests[key][0]].color && game[x].color ==
            game[x+tests[key][1]].color && game[x].color ==
            game[x+tests[key][2]].color) {
              $("#banner").html(player.toUpperCase() + " WON!!<br>
              Refresh to replay");
              return true;
            } else {
              game[x][key] = "fail";
            }
          }
        }
      }
    }
  }
}
```

It should be noted that each test grabs a fixed interval of squares – that is, a horizontal test on square **n** grabs the next three squares (**n+1**, **n+2**, **n+3**), a vertical test grabs the squares below (**n+7**, **n+14**, **n+21**), and so forth. For this reason, I never needed to assign row and column values at the start of the game.