Author: Gino Carlo
Date: September 21st, 2023

Mirror Tetris

In this text, I'll be covering the main game logic and how the code executes this logic through its functions.

The Tetris

Let's start by explaining what the original Tetris is and how Mirror Tetris differs from it.

In the original Tetris game, you had a grid composed of numerous blocks (10 columns, 20 rows). A new piece would appear, and that piece would always consist of four blocks arranged together in various forms (which is why it is called Tetris, derived from "Tetra").

This piece would appear on the grid and then start to descend until it reached the last block, after which a new piece would be randomly selected and introduced.

Once you completed a full row of blocks, you would earn a score, and the specific row would be cleared, causing all the rows above it to shift downward.

Ok, now that you understand the main concept of the game, let's take a look at the HTML

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <script src="app.js" charset="utf-8"></script>
    <link rel="stylesheet" href="style.css"></link>
    <title>Basic Tetris</title>
  </head>
  <body>

    <h3>Score:<span id="score">0</span></h3>
    <button id="start-button" >Start/Pause</button>

    <div class="container">
      <div class="grid">
      </div>
      <div class="mini-grid">
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
```

```
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
      </div>
    </div>
  </body>
</html>
```

What matters is that inside the container is the grid, that is where we are going to put our table, but it will not be a convenient table of HTML: we'll add each cell as a div by JS:

```
const width = 12;
const height = 21;

const tetrisGrid = document.querySelector('.grid');

for (let i = 0; i < (height*width); i++) {
  const div = document.createElement('div');
  tetrisGrid.appendChild(div);
}
```

This will create something similar to:

lin 1	0	1	2	3	4	5	6	7	8	9	10	11
lin 2	12	13	14	15	16	17	18	19	20	21	22	23
lin 3	24	25	26	27	28	29	30	31	32	33	34	35
lin 4	36	37	38	39	40	41	42	43	44	45	46	47
lin 5	48	49	50	51	52	53	54	55	56	57	58	59
lin 6	60	61	62	63	64	65	66	67	68	69	70	71
lin 7	72	73	74	75	76	77	78	79	80	81	82	83
lin 8	84	85	86	87	88	89	90	91	92	93	94	95
lin 9	96	97	98	99	100	101	102	103	104	105	106	107
lin 10	108	109	110	111	112	113	114	115	116	117	118	119
lin 11	120	121	122	123	124	125	126	127	128	129	130	131
lin 12	132	133	134	135	136	137	138	139	140	141	142	143

lin 13	144	145	146	147	148	149	150	151	152	153	154	155
lin 14	156	157	158	159	160	161	162	163	164	165	166	167
lin 15	168	169	170	171	172	173	174	175	176	177	178	179
lin 16	180	181	182	183	184	185	186	187	188	189	190	191
lin 17	192	193	194	195	196	197	198	199	200	201	202	203
lin 18	204	205	206	207	208	209	210	211	212	213	214	215
lin 19	216	217	218	219	220	221	222	223	224	225	226	227
lin 20	228	229	230	231	232	233	234	235	236	237	238	239
lin 21	240	241	242	243	244	245	246	247	248	249	250	251
	col 1	col 2	col 3	col 4	col 5	col 6	col 7	col 8	col 9	col 10	col 11	col 12

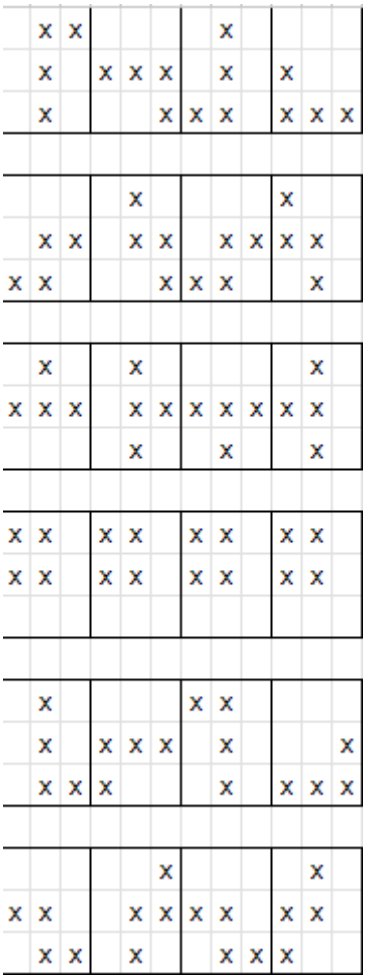
Well, but what are these purple and red borders?

The purple border is used to determine the end of the table, while the red one is used to avoid pieces from rotating in the left or right border of the table (that could cause visual bugs to the game)

To add those borders, we will have to use JS again. Both of them will have the class “taken”:

```
const gridDivs = document.querySelectorAll('.grid div');

for (let i = gridDivs.length - width; i < gridDivs.length; i++) {
  gridDivs[i].classList.add('taken');
}

for (let i = 0; i < (width*(height-1)); i+=width) {
  gridDivs[i].classList.add('taken');
}

for (let i = (width-1); i < (width*height-1); i+=width) {
  gridDivs[i].classList.add('taken');
}
```

Now that we have the table, we will have to talk about the tetrominoes: they are the pieces formed of blocks

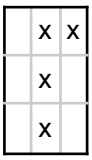


This is how they are projected, each one has 4 forms because of the 4 rotations it can have

We give theirs positions based on this table:

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

So, for example:



This tetromino would have 1,2,11 and 21

With 0 - 9 having the length of ten because of the original tetris table having 10 columns, but that could change according to the table

So the best way to write this tetromino would be: 1,2, length+1 and length*2+1
This is done on the code using the variable width

```
 const lTetromino = [
    [1, width+1, width*2+1, 2],
    [width, width+1, width+2, width*2+2],
    [1, width+1, width*2+1, width*2],
    [width, width*2, width*2+1, width*2+2]
  ]
```

Now that we have the table and tetrominoes, we need to create them

```
  let currentPos = startPos;
  let currentRot = 0;

  let random = chooseRandom();
  let current = theTetrominoes[random][currentRot];

  function drawTetromino(){
    current.forEach(index => {
        squares[currentPos + index].classList.add("tetromino");
        squares[currentPos + index].style.backgroundColor =
colors[random];
    })
  }

  function undrawTetromino(){
    current.forEach(index => {
        squares[currentPos + index].classList.remove("tetromino");
        squares[currentPos + index].style.backgroundColor = "";
    })
  }
```

This means that the initial rotation will always be zero, as well that the positions occupied by by the piece will have the class tetromino

To move the tetromino down, we use the function:

```
function moveDown(){
    depositTetromino(); //check if collided
    undrawTetromino();
    currentPos += width; //goes down a row
    drawTetromino();
}
```

To check if it collided a taken row (being the last or other tetromino), we use the function:

```
function depositTetromino(){
    if(current.some(index => squares[currentPos + index +
width].classList.contains("taken"))) {
        current.forEach(index => squares[currentPos +
index].classList.add("taken"));

        random = nextRandom
        nextRandom = chooseRandom();
        current = theTetrominoes[random][currentRot];
        currentPos = startPos;
        drawTetromino();
        showNextTetromino(); //determina a próxima peça
        checkToCleanRow();
        checkGameOver();
}
}
```

Basically it will check if at least one of the blocks of the next row (determined by summing the width) has the class taken

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23

The next row is nothing less that the current row + the width

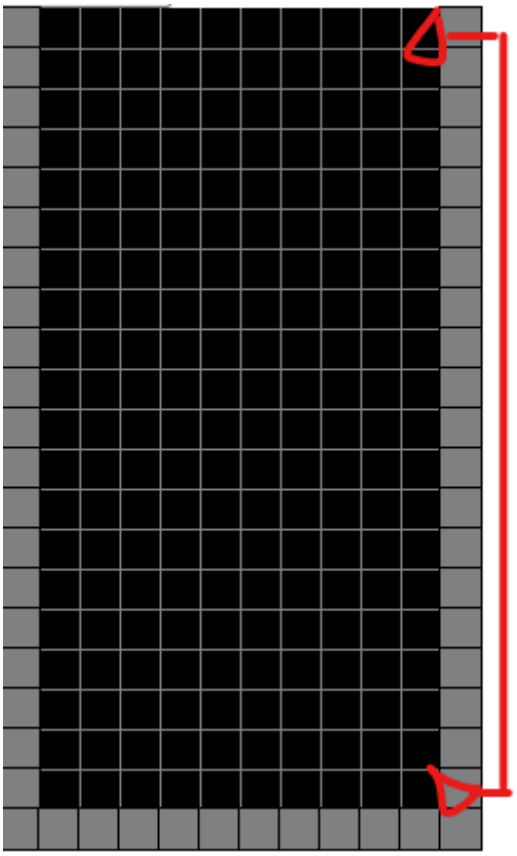
If it has indeed a “taken” class, we will choose the next piece randomly, draw it and show it on the mini grid.

Ok, now we know how the table works, the tetromino, and how he moves down. But how can we check a tetris?

```
function checkToCleanRow(){
  for(let i = 1; i < (width*(height-1)-2); i+=width){
    const row = [i, i+1, i+2, i+3, i+4, i+5, i+6, i+7, i+8, i+9]

    if(row.every(index => squares[index].classList.contains('taken'))) {
      score += upScore; //add score
      scoreDisplay.innerHTML = score; //atualiza ui do score
      const squaresRemoved = squares.splice(i, width)

      if(squaresRemoved.some(index =>
      squaresRemoved.forEach((index) => {
        index.classList = '';
        index.style.backgroundColor = '';
```

```
      })

      squaresRemoved[0].classList.add('taken');
      squaresRemoved[squaresRemoved.length-1].classList.add('taken');

      squares = squaresRemoved.concat(squares)
      squares.forEach(cell => grid.appendChild(cell))
    }
  }
}
```

This will go through every block of every row of the table, checking if everything in that row has the class taken. If this is true, that specific line will be excluded from the main squares and cleaned - that means it will lose the class attribute, except for the borders (left and right). After that, it will be added again to the beginning



Now, what can we check if the game is over?

```
function checkGameOver(){
  if(current.some(index => squares[currentPos +
index].classList.contains('taken'))){
    scoreDisplay.innerText = "Game Over! :( "
    clearInterval(timerId);
```

```
  }
}
```

This means it will do something very similar to the function that check tetris, but it will be seeing only the row where the tetromino is spawned

The interval is also cleared, so that the piece stops falling

At last, one of the more important functions is the rotation one:

```
function rotate(){
  undrawTetromino();

  let newRot = currentRot + 1;

  if (newRot === 4) {
    newRot = 0;
  }

  const newCurrent = theTetrominoes[random][newRot];

  const hasCollision = newCurrent.some(index => squares[currentPos +
index].classList.contains("taken"));

  if (!hasCollision) {
    currentRot = newRot;
    current = newCurrent;
  }

  drawTetromino();
}
```

It is simple: it will just go up on the list of the rotations of the tetromino that is currently chosen. The secret remains in the end, when it checks if the rotation has collided with another cell that has the “taken” class. If that is true, it will not rotate.

The Mirror Tetris

The mirror tetris has an important feature that completely changes the original game. There is another set of tetrominoes, with different colors. When you clear a line that has at least one piece of these “special” tetrominoes, everything will be reversed. If you move right, the game will move left. Not only that, the table will invert as well, as seen below: