Week 16 (Class)

Test Review (Answers)

Question 1

```
If (userIsYoungerThan21 == true) {
userType = "Adult";
}
Else {
userType = "Minor";
}
```

Answer:

```
 userType = (userIsYoungerThan21 == true) ?"Adult" :
"Minor";
```

Question 2

```
serveDrink = (userisYoungerThan21 == true) ? "Grape
Juice":"Wine";
```

Question 3

Given x = 11, y = 13 Computer the value of Z

```
Z = (x <= y) ? ( x + y) : (y - x);
```

Z = (11 <= 13) ? (11 + 13):(13 - 11)
Z = 24

Question 4

Given x = 27, y =13 Computer the value of Z

```
var Z;
Z = (13 >= 27 ) ? (y + x) : (x == y)?  X : (y + 1)  ;
```

Z = (13 + 1)
Z = 14

Question 5

```
var dRow;
if(d == LEFT || d == RIGHT){
dRow = 0; }
else{
   if(d == UP){
dRow = -1; }
   else{
      dRow = 1;
} }
```

dRow = _____

```
var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
```

Keeping on moving tiles

`dRow` is the vertical direction where to move tiles. If we are moving left or right, `dRow` is `0`, because we are moving horizontally and we only move through columns.

If we are moving up, `dRow` is `-1` because we are moving up by one row. If we are moving down, `dRow` is 1 because we are moving down by one row.

The same concept applies to `dCol` which refers to the horizontal direction where to move tiles.

Now that are are about to move, we have to prevent the player to move while the game is already moving tiles:

```
this.canMove = false;
```

Then we need to iterate though the entire board:

```
for(var i = 0; i < gameOptions.boardSize.rows; i++){
    for(var j = 0; j < gameOptions.boardSize.cols; j++){
        // rest of the code
    }
}
```

According to the direction we are about to moving tiles, we will need to iterate through the board starting from different tiles.

If we are moving tiles to the right, we will start moving the rightmost tiles.

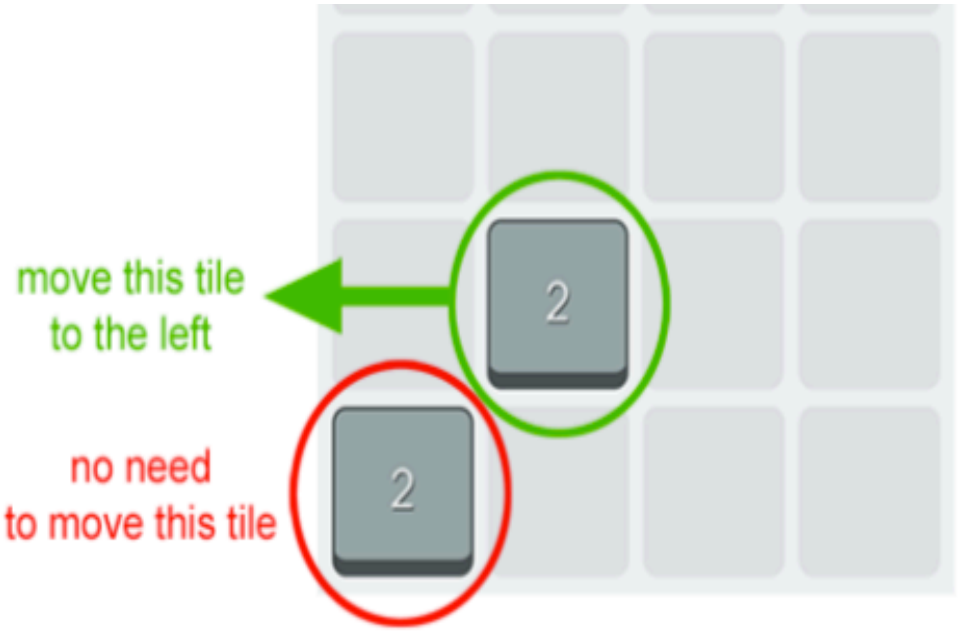
If we are moving tiles to the top, we will start moving the uppermost tiles, and the same concept applies when moving down or right.

Time to place another conditional operator:

```
var curRow = dRow == 1 ? (gameOptions.boardSize.rows - 1) - i : i;
var curCol = dCol == 1 ? (gameOptions.boardSize.cols - 1) - j : j;
```

To better understanding what we are trying to do, let's see what happens when we move tiles to the left.

Have a look at this picture:



Tiles in the red column – column zero – do not need to be moved to the left because they are already in the leftmost position.
We'll start checking and moving tiles in column one, then column two and so on, and the same concept then applies to other directions.
Back to the script, curRow and curCol represent respectively the current row and the current column to be examined at each iteration.
According to dCol and dRow values, curCol and curRow can range from zero to the amount of columns or rows minus one, or from the amount of rows or columns minus one down to zero.
Time for the final recap. we have four directions: left, right, up and down. Each direction generates different dRow and dCol values, which represent the
direction of the movement along the board.
When we loop through the table, curRow and curCol iterates board elements starting from the first row – or column – to the

last one, or from the last row – or column – to the first one.

Direction	dRow	dCol	curRow	curCol
Left	0	-1	* first to last row	first to last column
Right	0	1	* first to last row	last to first column
Up	-1	0	first to last row	* first to last column
Down	1	0	last to first row	* first to last column

Values of `curRow` and `curCol` marked with an asterisk do not really matter as we are moving in a direction which does not affect row or column position of the tiles. Once we know which direction we are moving and which tile on the board we are going to move, let's see if we find a tile:

```
var tileValue = this.boardArray[curRow][curCol].tileValue;
```

Here we saved `tileValue` property of the current `boardArray` item into a variable called `tileValue`.

```
if(tileValue != 0){
    // rest of the code
}
```

The rest of the script will be executed only if `tileValue` is different than zero, that is if there is a tile in the current `boardArray` item.

> `!=` is the **not equal** operator.
>
> Given two values `x` and `y`, `x != y` returns `true` if `x` is not equal to `y`

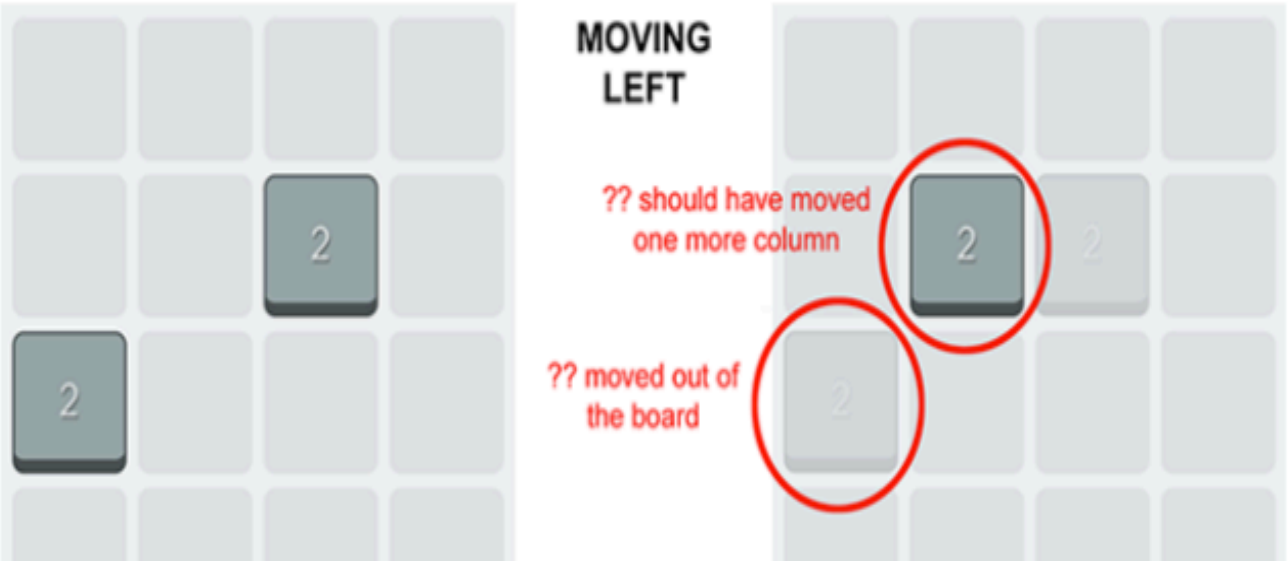
```
var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
```

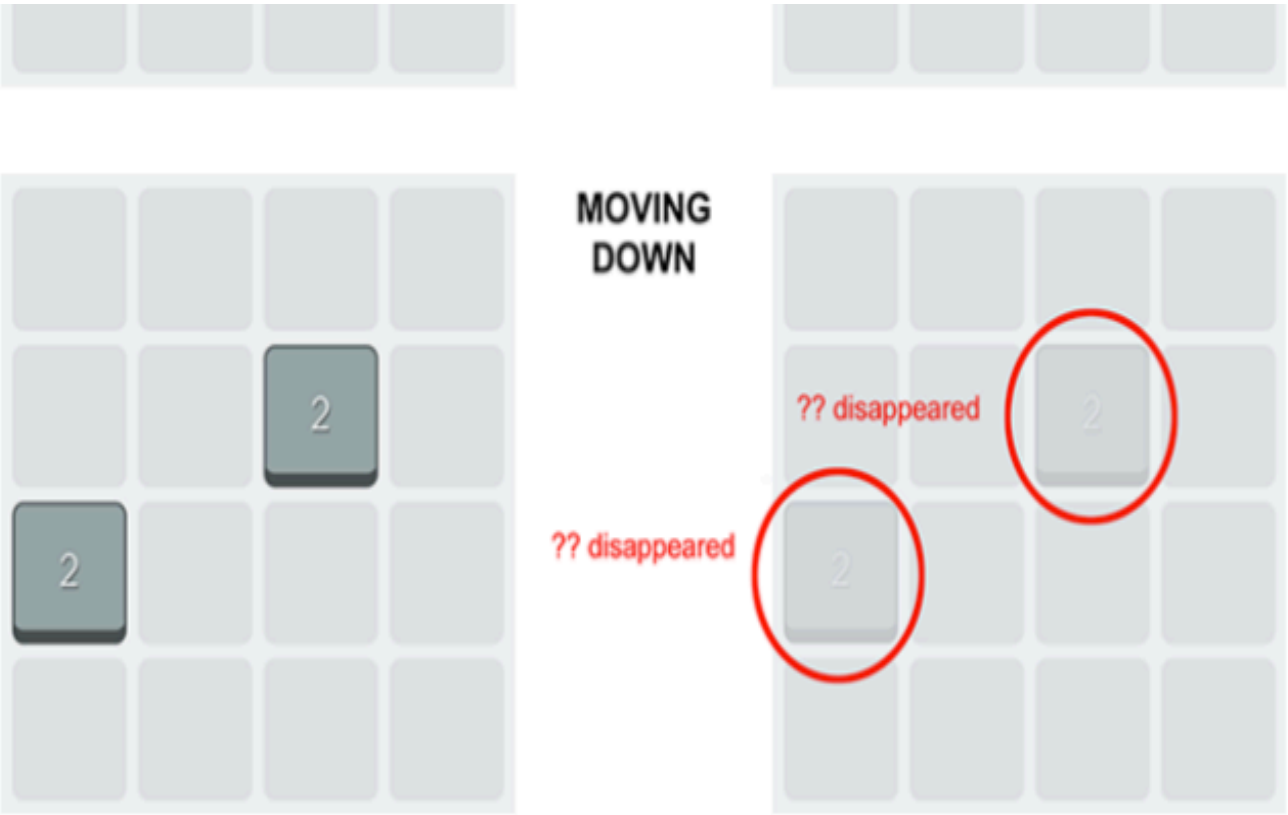
`newPos` is a `Point` object which will contain the new tile position.

We are using `getTilePosition` method to determine it, and its arguments are the current row and column positions – `curRow` and `curCol` – added to respectively the row and column directions where to move – `dRow` and `dCol`.

Then `tileSprite` sprite position is updated to `newPos.x` and `newPos.y`.

Run the game, and make some moves, you'll see some bugs. Can you find them all? Here they are:





Understanding depth or Z-order

Although we are building a 2D game, where we are only allowed to place objects on coordinates based on x and y axis, there's something to say about Z-order.

In 2D environments, objects can overlap, and when there are overlapping objects, there is always one object which covers other objects.

Think about Photoshop layers. Top layers will cover bottom layers. Or think about overlapping windows on your desktop, where top windows cover bottom windows, until you bring them to front.

It's obvious even in 2D environments there is some kind of

depth: the Z-order.

Z-order is an ordering of overlapping two-dimensional objects, such as windows in a stacking window manager, or shapes in a graphics editor.
When objects overlap, objects with a higher Z value hide part or all of objects with a lower Z value.

If you are used to web design, it's the same concept of CSS z-index property, which specifies the stack order of an element.
When we placed the images and sprites representing empty spots and tiles during the creation of the game table, the order we placed the images represents the Z- order used by Phaser to display objects.
This way, moving from left to right and from top to bottom, Z-order increased, and that's why when you move a tile to the right or to the bottom of the canvas, it disappears: it's simply rendered behind other objects which have been placed later, obtaining a higher Z value.
To avoid this issue, we need to group all moving objects and change their Z value – which Phaser calls “depth” – according to the direction they are moving.
This will be useful not only when we are simply moving tiles down and right, but also when we need two tiles with the same value to merge and transform into a
tile with a greater value.
How to handle Z-order with Phaser? In just three lines of code to be added to
makeMove method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var curRow = dRow == 1 ? (gameOptions.boardSize.rows - 1) - i : i;
            var curCol = dCol == 1 ? (gameOptions.boardSize.cols - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

When placed on the stage, all sprites have a `depth` property initially set to zero.

At the beginning of the function, we can declare a new variable called `movedTiles` which will keep count of the tiles we moved.

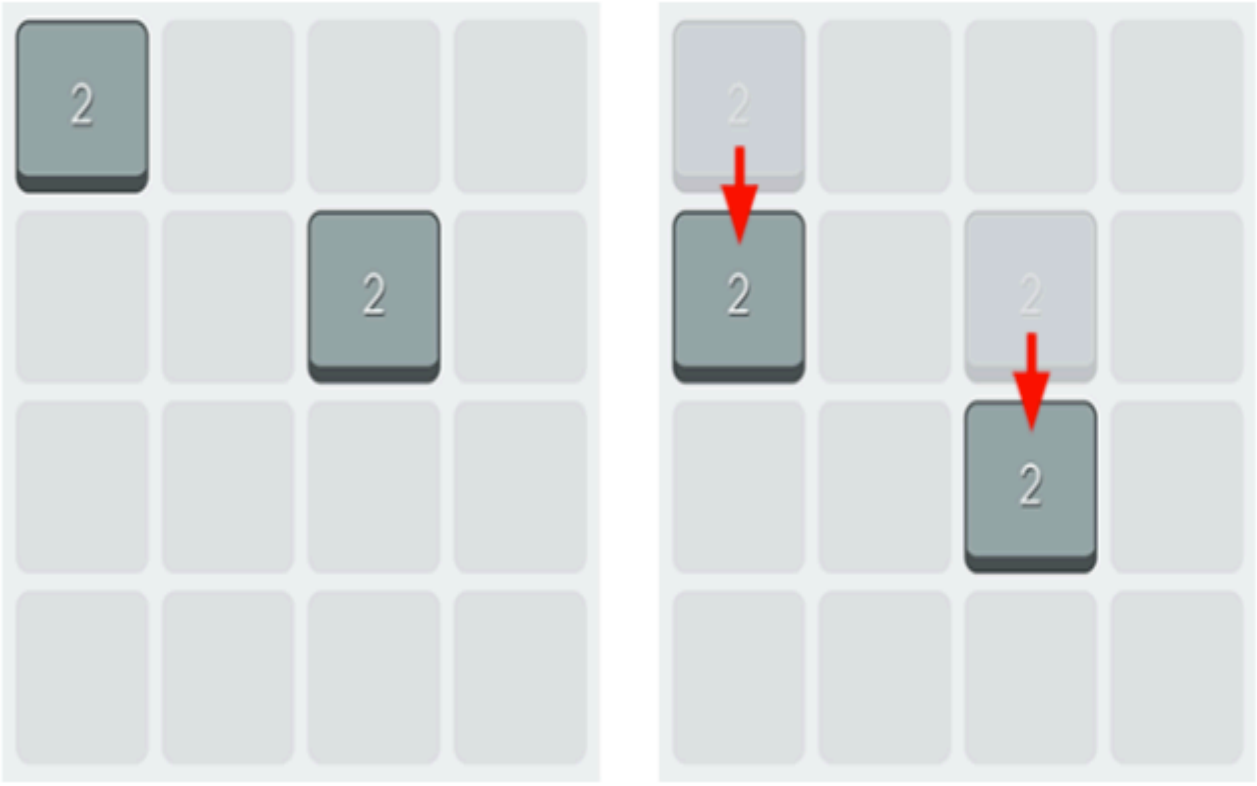
It starts at zero because no tiles have been moved yet.

When we are about to move a tile, first we increment `movedTiles`, then we change `depth` property of the sprite we are about to move to `movedTiles`.

This way, depth increases as we move the tiles so that the last to be moved have a higher value and stay on the top.

> `depth` property of a game object sets its depth within the Scene, allowing to change the rendering order.
>
> It starts from zero and a game object with a higher depth value will always render in front of one with a lower value.

Now test the game and try to move tiles to the right or to the bottom, and you will see they will display correctly.



And the first bug has been solved.

Moving only necessary tiles

Do you know why sometimes tiles disappear from the stage? They disappear because they already are on an edge of the stage, and when we move tiles towards such edge, they simply fly off the stage.

We are moving unnecessary tiles.

When we are moving left, we do not need to move tiles in the leftmost column. They are already in the leftmost position, so why bothering moving them?

Same concept applies when moving up. We don't need to move tiles in the upmost row. And that's the same when you move right, or down.

According to the direction you are moving, there is one row or one column you don't need to move. And you won't move it, thanks to these lines added to makeMove method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
```

```
            }
        }
    }
}
```

Some more conditional operators here, let's split the code line by line and see how we avoid moving unnecessary tiles.

Both in the two `for` loops and when calculating `curRow` and `curCol` values, we do not refer anymore to `boardSize.rows` and `boardSize.cols` stored into `gameOptions` object, because we need to exclude one row or one column, let's see how we are doing it:

```
var firstRow = (d == UP) ? 1 : 0;
```

`firstRow` variable represents the highest row we have to move. Its value is zero – the top row – unless we are moving up, and in this case its value is 1, because we don't need to move tiles in the topmost row.

```
var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
```

`lastRow` variable represents the lowest row we have to move. Its value is `boardSize.rows` unless we are moving down, in this case its value is `boardSize.rows - 1` because we do not need to move tiles in the bottommost row.

The same concept applies to the columns, and this is a comparison table resuming

what happens to `firstRow`, `firstCol`, `lastRow` and `lastCol` variables according to the direction we are moving:

Direction	firstRow	firstCol	lastRow	lastCol
Left	0	**1**	boardSize.rows	boardSize.cols
Right	0	0	boardSize.rows	**boardSize.cols - 1**
Up	**1**	0	boardSize.rows	boardSize.rows
Down	0	0	**boardSize.rows - 1**	boardSize.rows

Highlighted values represent what changed in the two `for` loops and when calculating `curRow` and `curCol` values compared to before, when we simply looped from zero to `boardSize.rows` – or `boardSize.cols` – no matter the direction of movement.

Moving tiles as long as there is room for movement

To make tiles move until there is not any more room for movement rather than for just one step, we need to move them in the selected direction as long as they

just one step, we need to move them in the selected direction as long as they remain inside the board.

This can be easily done in just a few lines to be added to `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newRow = curRow;
                var newCol = curCol;
                while(this.isLegalPosition(newRow + dRow, newCol + dCol)){
                    newRow += dRow;
                    newCol += dCol;
                }
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(newRow, newCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

```
while(condition){

    code block to be executed

}
```

Executes the block of code as long as a condition is true.

condition defines the condition for running the loop. If it returns `true`, the loop will start over again, if it returns `false`, the loop will end.

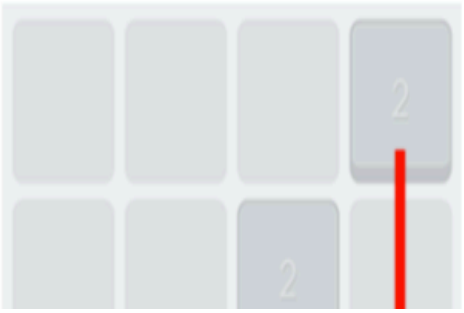
If the condition is always true, the loop will never end, your script will probably freeze and crash your browser.

Basically we are acting as if the player moved several times in the same direction, while next move is in a legal position, determined by `isLegalPosition` method we are about to write:

```
isLegalPosition(row, col){
    var rowInside = row >= 0 && row < gameOptions.boardSize.rows;
    var colInside = col >= 0 && col < gameOptions.boardSize.cols;
    return rowInside && colInside;
}
```

isLegalPosition sees if its arguments – the row and the column we are checking to be in a legal position – are inside the board.

Run the game and make your move, and see how tiles travel along the entire table before stopping on one of its edges, like in this case when the player is moving down.



Another feature has been added.

Merging tiles

Two tiles merge in a bigger tile when two tiles with the same value overlap.

In previous steps we only moved sprites here and there, this time we'll have to work behind the curtains – remember? Where most of the game takes part – reading and changing some boardArray values.

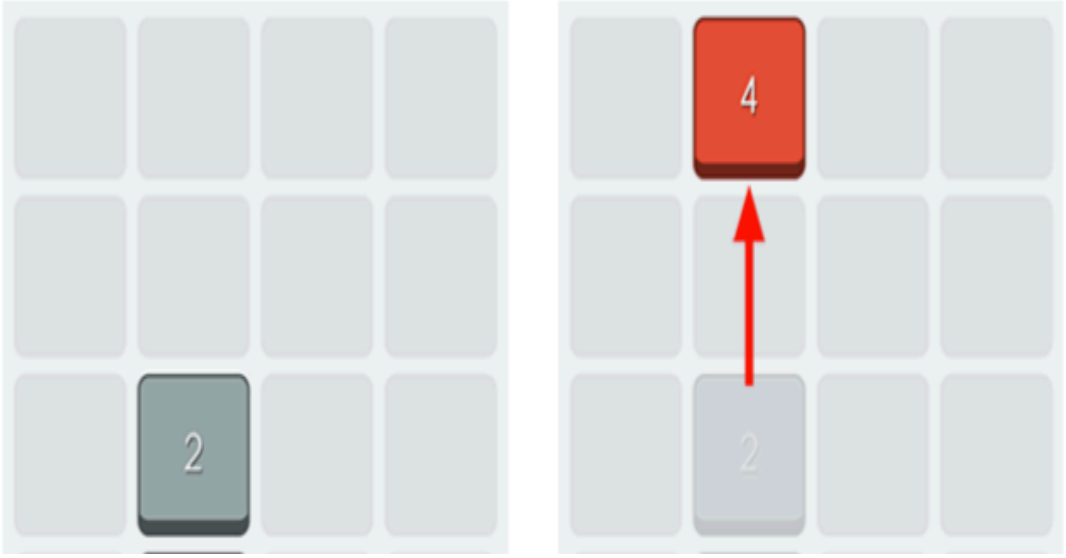
Some more lines to be added to makeMove method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
```

```
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newRow = curRow;
                var newCol = curCol;
                while(this.isLegalPosition(newRow + dRow, newCol + dCol)){
                    newRow += dRow;
                    newCol += dCol;
                }
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(newRow, newCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
                this.boardArray[curRow][curCol].tileValue = 0;
                if(this.boardArray[newRow][newCol].tileValue == tileValue){
                    this.boardArray[newRow][newCol].tileValue ++;
                    this.boardArray[curRow]
                      [curCol].tileSprite.setFrame(tileValue);
                }
                else{
                    this.boardArray[newRow][newCol].tileValue = tileValue;
                }
            }
        }
    }
}
```

If we did not find a destination tile with the same value as the moving tile, we simply set `tileValue` value of `boardArray[newRow][newCol]` item to tileValue.

Test the game and try to make a move to merge two tiles:



The magic happened! Moving up, we managed to turn two “2” tiles into one “4”.