Week 13

Trinity - Introduction to Computer Programming With JavaScript: Game Development

Continuing from Last Week - The game we are going to build: 4096

We are about to create a game called 4096, which will be our take to Gabriele Cirulli's 2048 smashing hit, a puzzle game originally written in Javascript played on a 4x4 board.

Every turn, a new tile will appear in a random empty spot with a value of 2. Using arrow keys, you slide tiles until they are stopped by either another tile or the edge of the grid. If two tiles with the same number collide, they will merge into a tile with the total value of the two tiles that collided. Tiles can merge only once per turn.

We are going to build this game, also adding some more features and room to customization, continuing from the game template we just built.

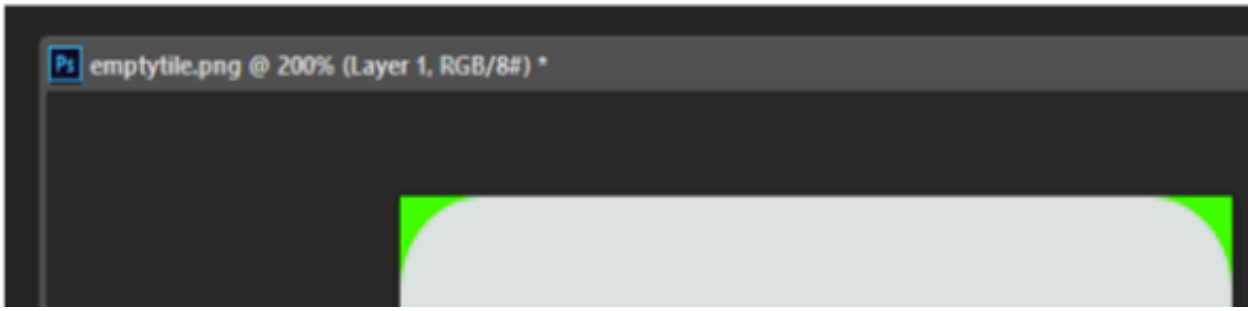
game template we just built.

Preloading and adding images to the game

HTML5 games are a collection of images and sounds which move accordingly to game design and player input, so we need to learn how to preload and add images to the game.

Why do we need to preload images? Because one of the worst things you can do in the making of a game is to handle graphic assets before you actually loaded them. **This can cause a big DELAY, and frustrate users.**

I created a 200x200 pixels image called emptytile.png and will use this image to display the game board.



It's a PNG image with transparency, which I am showing you on a green background to let you see the transparency.

> Always save images as **PNG** as this format has the advantages of being lossless – it does not lose quality when saved – and supports alpha channel (transparency).
>
> Anyway, its lossless compression means larger files than JPEG.

To keep game folder well organized, the image has been saved into a folder called `sprites` which is created in a folder called `assets`.

You can obviously organize your game folder as you prefer, just keep in mind you

should organize it in any way.

Once the image has been saved in its folder, it's time to preload it.

Add these lines to bootGame class:

```
class bootGame extends Phaser.Scene{
    constructor(){
        super("BootGame");
    }
    preload(){
        this.load.image("emptytile", "assets/sprites/emptytile.png");
    }
    create(){
        this.scene.start("PlayGame");
    }
}
```

Just like create, preload is a reserved method of Scene class and is executed when the scene is preloading, and this is when we'll load the assets like the image we just created.

load.image(key, url) loads an imagewants as arguments respectively the unique asset key of the image file and the URL of the image.

Once the execution of preload is over, Phaser executes create method which calls PlayGame scene, and this is where we are going to write the code to place the image somewhere in the canvas.
Add this line to playGame class:

```
class playGame extends Phaser.Scene{
    constructor(){
        super("PlayGame");
    }
    create(){
        this.add.image(100, 100, "emptytile");
    }
}
```

Now launch the game, and this is what you should see:

Our 200x200 pixels image has now been placed in the game

Our 200x200 pixels image has now been placed in the game at coordinates (100, 100) starting from the upper left corner, where the origin (0, 0) coordinate is placed.

add.image(x, y, key) places an image on the stage and wants as arguments the x coordinate of the image, in pixels, the y coordinate of the image, in pixels, and the key of the image used.

From the image above is also easy to see the anchor point of the images added by add.image is the center of the images themselves.
This means the (100, 100) coordinate where we placed the image refers to the center of the image.

> The **anchor point** of an image sets the origin point of the texture. The default anchor point is in the center, this means the texture's origin is centered.

Knowing the anchor point of your images is important because lets you know exactly how your images will be displayed in the game.

Adding a single image is not enough, as we said we are going to play on a 4x4 board. Let's display the entire board with the 16 images needed.

First, we need to make the game size bigger, as we need to display 4 images 200 pixels each, so we are changing `gameConfig` object:

```
var gameConfig = {
    width: 800,
    height: 800,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

Now the game will be 800x800 pixels, and we also change the background color from red to a light grey.

We are about to add 16 images, actually 16 instances of one image, so we don't need to preload anything else at the moment. Once you preloaded an image, you are free to use it as many times as you need.

Change `create` method inside `playGame` class this way:

```
create(){
    for(var i = 0; i < 4; i++){
        for(var j = 0; j < 4; j++){
            this.add.image(100 + j * 200, 100 + i * 200, "emptytile");
        }
    }
}
```

The line which we used to add the image to the stage now is placed inside two `for` loops which will take care of executing `add.image` method 16 times, changing the coordinates according to `i` and `j` values.

The `for` loop continually repeats a block of code until a certain condition is reached.

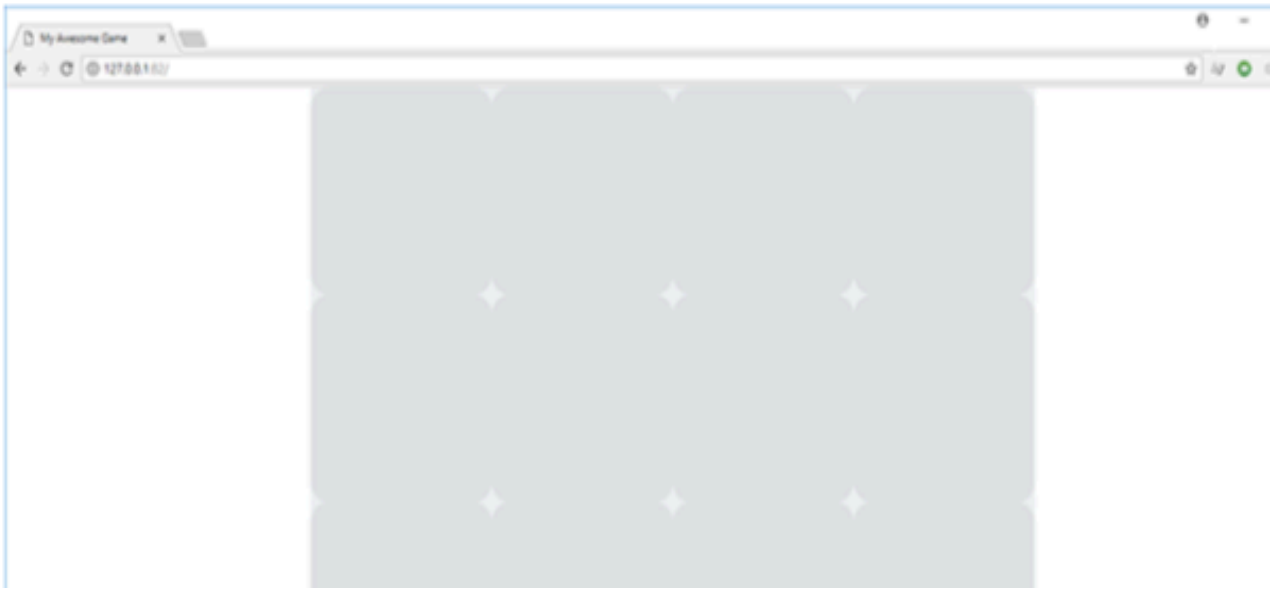
```
for(start action; condition; recurring action){
    code block to be executed
}
```

start action is executed only once before the loop starts.

condition defines the condition for running the loop. The loop will be executed as long as the condition is satisfied.

recurring action is executed each time after the loop has been executed.

Run the game and here's what you'll see:



Here we can see the 16 images placed on the state, but as they are too close to each other, the game will look better if insert some spaces between them.

> `++` is the increment operator which increments (adds one to) its operand.
>
> Both `i = i + 1` and `i++` add `1` to `i` variable

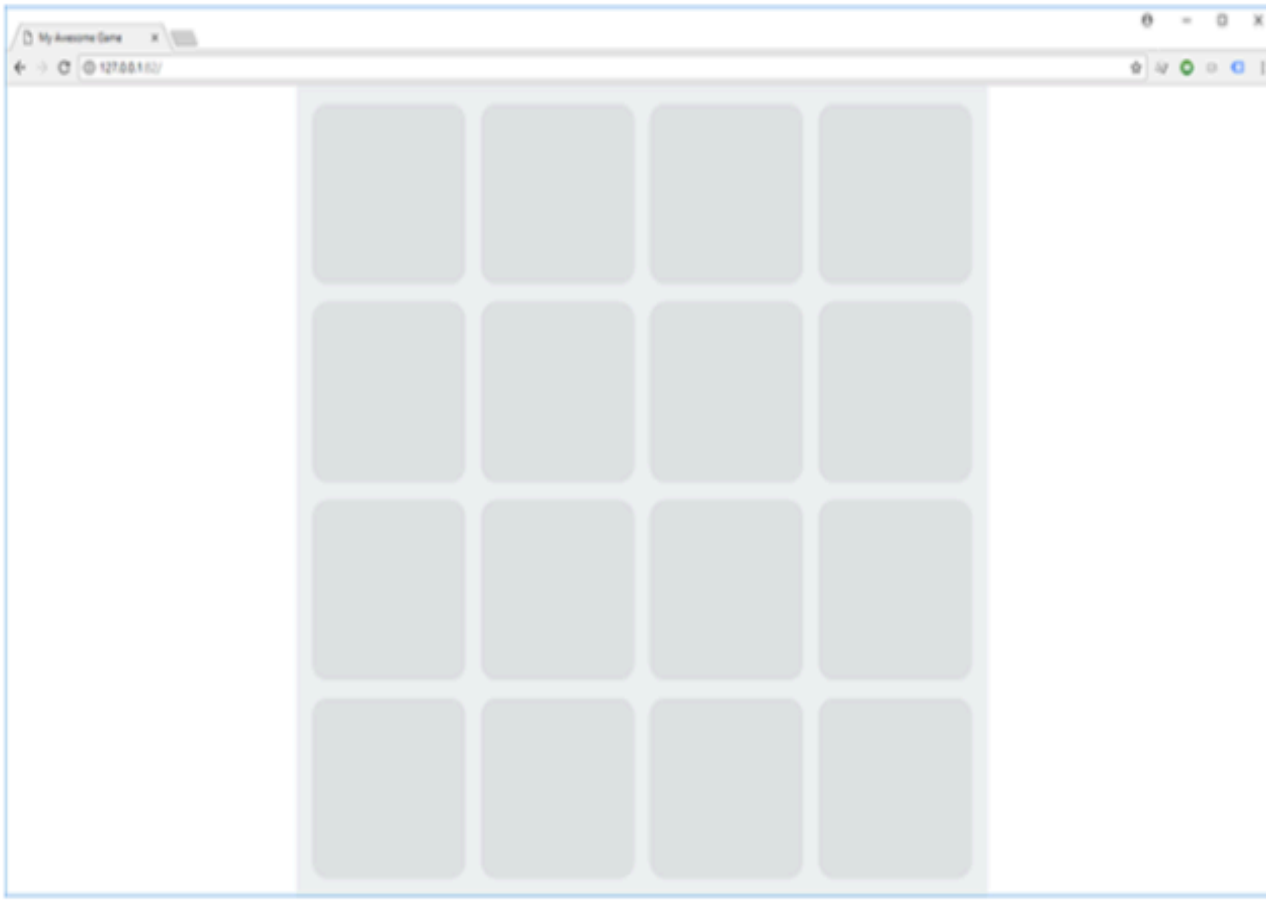
Let's add a 20 pixels spacing between tiles, by changing again `gameConfig` object to increase game size:

```
var gameConfig = {
    width: 900,
    height: 900,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

Then by modifying `create` method inside `playGame` class:

```
create(){
    for(var i = 0; i < 4; i++){
        for(var j = 0; j < 4; j++){
            this.add.image(120 + j * 220, 120 + i * 220, "emptytile");
        }
    }
}
```

Now test the game and this is what you'll see:



Way better now.

We created our 4x4 game board filled with 200x200 pixels tiles with a spacing of

The importance of storing all game customizable variables in a single place

Every game has a series of variables which define the essence of the game itself.

In the making of 4096 game, for instance, we said we will be playing on a 4x4 grid, which is a grid with 4 rows and 4 columns, and each tile on the grid is a 200x200 pixels image with a 20 pixels spacing.

We are going to refer to these numbers a lot of times in the making of the game, each time we will need to know the size of the board, or the size of a tile, and believe me, it will happen quite often.

Rather than filling the source code with a series of “4”, “200”, and so on scattered here and there, it would be better to store these values somewhere safe and easy to access.

Not only our source code will be more readable, but above all your script will be a lot easier to modify should we decide to change the size of the board to, let's say, 5 rows by 3 columns, or use smaller or bigger tiles.

No more “search and replace” operations, but a single value to change.

That's why we are going to add a global object with some values we know we'll need a lot of times.

Add an object called `gameOptions` with these values:

```
var game;
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    }
}
```

Just immediately after the declaration of `game` variable, `gameOptions` object contains:

`tileSize`: the size of each tile, in pixels.

`tileSpacing`: the spacing between two tiles, in pixels.

`boardSize`: another object which contains the amount of rows and columns to be added to the board.

These values are the same ones we used before, injecting them directly inside the code. With this in mind, we have to change the rest of the game accordingly.

Look at this image:

To display two tiles, we need the width of the two tiles themselves – red segments – and the width of three tile spacings – blue segments.

To display n tiles, we can say we need n red segments and (n + 1) blue segments, simplified in:

```
n * (red segment + blue segment) + blue segment
```

Now we know how to set game width and height starting from the number of tiles, tile size and tile spacing:

```
var gameConfig = {
    width: gameOptions.boardSize.cols * (gameOptions.tileSize +
        gameOptions.tileSpacing) + gameOptions.tileSpacing,
    height: gameOptions.boardSize.rows * (gameOptions.tileSize +
        gameOptions.tileSpacing) + gameOptions.tileSpacing,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

By looking at the same picture, we can also determine the coordinate where to place tiles.

Assuming the first tile position is zero, the second tile position is

one and so on, we can say the position of tile zero is given by the blue segment plus half the red segment.

The position of tile one is given by two blue segments plus one and a half red segments, and so on, simplified in:

(n + 1) * blue segment + (n + 0.5) * red segment

We are adding a new method to `playGame` class:

```
class playGame extends Phaser.Scene{
    constructor(){
        super("PlayGame");
    }
    create(){
        // same as before
    }
    getTilePosition(row, col){
        var posX = gameOptions.tileSpacing * (col + 1) + gameOptions.tileSize *
                (col + 0.5);
        var posY = gameOptions.tileSpacing * (row + 1) + gameOptions.tileSize *
                (row + 0.5);
        return new Phaser.Geom.Point(posX, posY);
    }
}
```

`getTilePosition` just applies the concept explained right above, and given a row and a column position, determines tile position in pixels.

> `return` statement stops the execution of a function and returns a value from that function.

The result is stored inside a `Phaser.Geom.Point` object, which is useful to store data with x and y coordinates.

> `Geom.Point` object represents a location in a two-dimensional coordinate system, where x represents the horizontal axis and y represents the vertical axis.

Finally we have to use `getTilePosition` method to place tiles inside `playGame`'s `create` method:

```
create(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytile");
        }
    }
}
```

First, we save into `tilePosition` the result of `getTilePosition` method, then we access to its `x` and `y` properties to get the coordinates where to place the tile.

Test the game and you'll see exactly the same thing as before.

Moreover, having a lot more math, I have to say.

Well, this was the goal of the new code: doing exactly the same stuff as before, just some math starting from a configuration object.

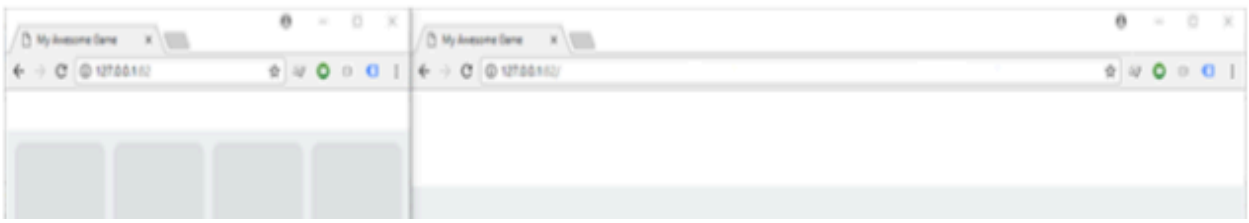
I know it worked like a charm even before these changes, but this is not the point: we worked a little more now to save a lot of time later on.

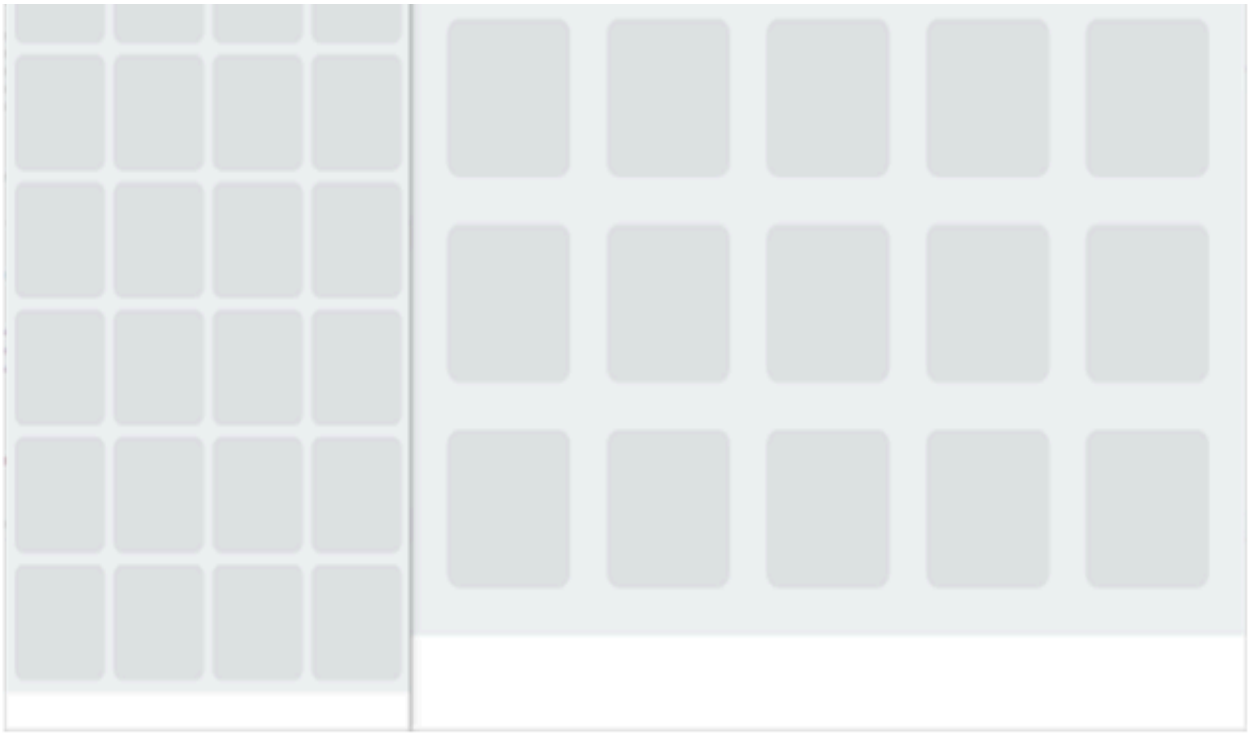
Do you want to play on a 5x4 board rather than a 4x4 board? Just change the proper value in `gameOptions` object.

Did you draw 150x150 pixels tile? Ajdust `tileSize` value in `gameOptions` object.

Need more spacing between tiles? I am sure you know how to do, changing just one value.
If you love TV series, we can say gameOptions object comes into play when it's time to change something in gameplay and you'd “Better call Saul”.
Believe me, it's a real pain when you wrote some thousands lines of code with values scattered here and there and you have to crawl line by line trying to adjust some option.
gameOptions is going to save you a lot of time in the long run.

Look at this picture:





Different rows and columns with different spacing just changing a couple of values.

> You will find the full project in the folder
>
> **001 - Preloading and adding images to the game**

Done with drawing the game board, let's create the tiles with the various numbers.

Creating tile graphics as a sprite sheet and using it in the game

We are going to create the tiles representing the numbers following the power of two, from 2 to 4096.
Since 4096 is 2^12, we need 12 tiles, one for the “2”, one for the “4”, one for the “8”, and so on until the one with “4096” on it. We need to draw 12 images.
At this time, we can save the 12 images in twelve distinct files or group them all into a sprite sheet.

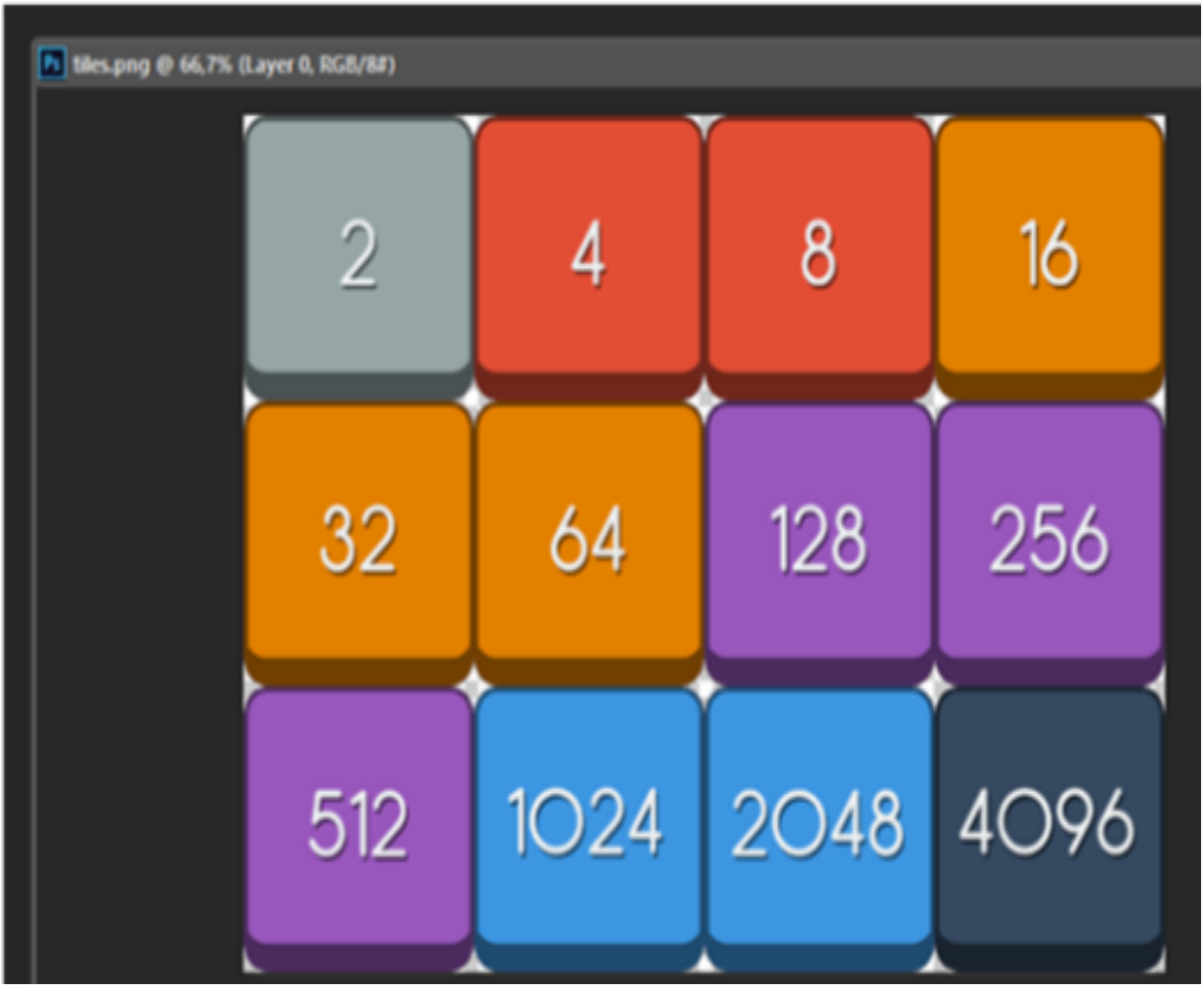
A **sprite sheet** is a series of images combined into a larger image. Usually the images are frames of an animation, thus a single image inside a sprite sheet is called **frame**.

Why using a sprite sheet?
Basically, every game is made by various graphical objects. In a space shooter you will find images representing spaceships, bullets and explosions, while in our 4096 game there will be different tiles. It does not matter the subject of the images. What we know is we are using all of them.
Each image has a width and a height, which represent the amount of pixels building such image, and each pixel requires some memory to hold its color information.
For each image – and more generally for each file – saved anywhere, there is a certain amount of memory that is wasted due to a series of features regarding the way the file system handles the files.
Explaining this concept goes beyond the scope of this book,

just keep in mind the more files you have, the bigger the amount of memory wasted. It's not a problem when you are dealing with a dozen files, but in complex games with a lot of images, packing them into bigger images can save quite an amount of resources.

Moreover simply storing images is not enough. We also have to place them on the screen.
No matter the graphic engine your device will be using to display images, there will be a process which must know which image to paint, get the image from the place where it's stored, then know which part of the image to paint – normally the entire image – and where to paint it, and finally place it on the screen.
Once the first image has been placed on the screen, this process needs to be repeated for each other image, while your game until all images have been placed. Normally you don't notice it because it happens – or at least it should happen – in 1/60 second, but a lot of images to be placed on the screen of a slow device can slow down performance.
Using a sprite sheet, you will have all – or most of – your graphic assets placed into a big image, inside an invisible grid, in order to avoid the “what image should I load” question, keeping only the “which part of the image should I paint”, and speed up the drawing process.

Look at this image:



Following this concept, I made a 800x600 pixels image containing all tiles, and now we need to repeat the process of preloading the image and using it in the game.

This image has also been saved in `assets/sprites` folder, so here's how we preload a sprite sheet in `preload` method of `bootGame` class:

```
preload(){
```

```
    this.load.image("emptytile", "assets/sprites/emptytile.png");
    this.load.spritesheet("tiles", "assets/sprites/tiles.png", {
        frameWidth: gameOptions.tileSize,
        frameHeight: gameOptions.tileSize
    });
}
```

Basically the image is loaded in the same way as before, we just need an extra argument containing an object where to store the width and the height of the individual tile.

We are using the values stored inside `gameOptions` object.

> `load.spritesheet(key, url, config)` loads a sprite sheet and wants as arguments respectively the unique asset key of the file, the URL to load the texture file from and a configuration object with `frameWidth` value representing the frame width of each tile, in pixels, and `frameHeight` value representing the frame height of each tile, in pixels.

Once the sprite sheet has been loaded, it's easy to add the tiles to the game, in `create` method of `playGame` class:

```
create(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytile");
            this.add.sprite(tilePosition.x, tilePosition.y, "tiles", 0);
        }
    }
}
```

The syntax is almost the same, we only have one more argument to specify the frame we want to display, in this case zero = the first frame.
The big difference in with the previous line is we are adding a sprite rather than an image. This has nothing to do with the sprite sheet, although there's the “sprite” word in it you can load a sprite sheet and add its tiles as images too.
The main difference between a sprite and an image is you cannot animate images, and for this reason sprites takes a fraction longer to process.
I have to say, such “fraction” is not relevant unless you have thousands of images on the stage.
While you could to everything just using sprites and forget about images, I still use images to display static content such as the game interface, and sprites for the rest of the visual content, it makes the code look cleaner.

```
this.add.sprite(tilePosition.x, tilePosition.y, "tiles", 3);
```

Changing the last argument will change the frame displayed.

`add.sprite(x, y, key, frame)` places an image on the stage and wants as arguments the x coordinate of the image, in pixels, the y coordinate of the image, in pixels, the key of the image used and optionally the number of the

> frame to display, if a sprite sheet is used. Default value is zero.

Now that we are able to display images on canvas both using single images and sprite sheets, it's time to start defining game mechanics.

Using two-dimensional arrays to store board configuration

Arrays keep track of multiple pieces of information in linear order, a one- dimensional list. However, the data associated with certain environments like this board game lives in two dimensions, as the board has rows and columns.

To store this data, we need a multi-dimensional data structure, in this case a two- dimensional array.

A two-dimensional array is really nothing more than an array of arrays, a three- dimensional array would be an array of arrays of arrays and so on.

Let's create the two-dimensional array capable of storing board information, adding a few new lines to create method of playGame class:

```
create(){
    this.boardArray = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        this.boardArray[i] = [];
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytile");
            var tile = this.add.sprite(tilePosition.x, tilePosition.y, "tiles",
               0);
            tile.visible = false;
            this.boardArray[i][j] = {
                tileValue: 0,
                tileSprite: tile
            }
        }
    }
}
```

```
this.boardArray = [];
```

An empty array called `boardArray` is created. This will contain board information.

> **Empty arrays** are defined with open/close square brackets `[]`.

We said we are dealing with two-dimensional arrays, so each `boardArray` item itself must be declared as an empty array:

```
this.boardArray[i] = [];
```

The i-th element of `boardArray` is an array. Finally it's time to populate the array: each item will be an object, this way:

```
this.boardArray[i][j] = {
    tileValue: 0,
    tileSprite: tile
}
```

Now the board is defined this way:

tileValue is the value assigned to the tile, where zero means “empty tile”. tileSprite is the sprite which will represent the tile. It's the tile sprite we created a couple of lines before.
Now the board has been defined as an array, other than being displayed on the screen.
If you run the game you won't see anything new, because we just defined the main base of data, the hidden but really important part of each game.

Now the strategy is to handle boardArray according to game events, then display the right sprites in the right places to make players see what happens, but the actual game lies inside boardArray.

At this time we can add tiles, or rather show tiles we previously added and made invisible.

Placing “two” tiles on empty spots on the board

At the beginning of the game, two new tiles with a “two” on them are added to the

board on two empty spaces.

Then, a new “two” tile is added each turn.

This simple step needs to know which tiles are free, choose two random tiles among them, and show the “two” tile previously added and hidden.

We are going to add a custom method to `playGame` class which will take care of adding a new tile.

In `create` method, it will be called twice, this way:

```
create(){
    this.boardArray = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        this.boardArray[i] = [];
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytile");
            var tile = this.add.sprite(tilePosition.x, tilePosition.y, "tiles",
               0);
            tile.visible = false;
            this.boardArray[i][j] = {
                tileValue: 0,
                tileSprite: tile
            }
        }
    }
    this.addTile();
    this.addTile();
}
```

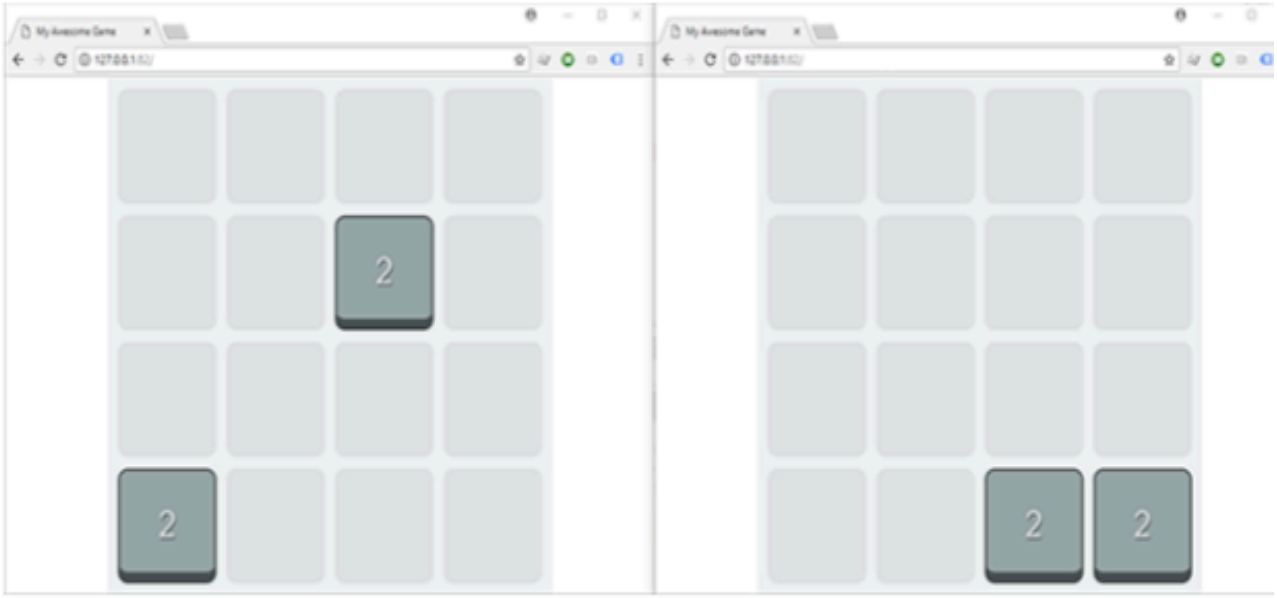
Once we'll code addTile method, we'll have our tiles added to the board. Do you remember boardArray array and the concept of working behind the
curtains?
We can say a tile is empty when the corresponding boardArray

item is an object
with tileValue set to zero.
Looping through boardArray to look for empty tiles is the first thing to do.

Look at addTile method:

```
addTile(){
    var emptyTiles = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            if(this.boardArray[i][j].tileValue == 0){
                emptyTiles.push({
                    row: i,
                    col: j
                })
            }
        }
    }
    if(emptyTiles.length > 0){
        var chosenTile = Phaser.Utils.Array.GetRandom(emptyTiles);
        this.boardArray[chosenTile.row][chosenTile.col].tileValue = 1;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.visible = true;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.setFrame(0);
    }
}
```

Run the game, and each time you should see a couple of "two" tiles appearing in random empty positions:



Let's examine the code line by line:

```
var emptyTiles = [];
```

`emptyTiles` array will store all empty tiles we'll find. It starts as an empty array.

Then it's time to loop through all `boardSize` array, with two `for` loops running through all board rows and columns.

Look, we are using once again `gameOptions` object:

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

The first `for` loops through rows, the second one loops through columns

The first `for` loops through rows, the second one loops through columns.

What are we looking for? For empty tiles, that is `boardArray` elements whose `tileValue` is equal to zero.

```
if(this.boardArray[i][j].tileValue == 0){
    // rest of the code
}
```

At the beginning of the game, this condition will be always true, since the board is empty.

> `==` operator means **equal to**.

What happens when we find an empty tile?

```
emptyTiles.push({
    row: i,
    col: j
})
```

Yes, we add tile's coordinates as an object inside `emptyTiles` array.

> `push` adds new items to the end of an array.

What happens at the end of these two `for` loops? We have `emptyTiles` array filled of objects, each one representing the coordinates of an empty tile.

Now it's time to randomly pick a `emptyTiles` item, but the first thing to do is seeing is there is at least one empty tile, that is if `emptyTiles` array has at least one item.