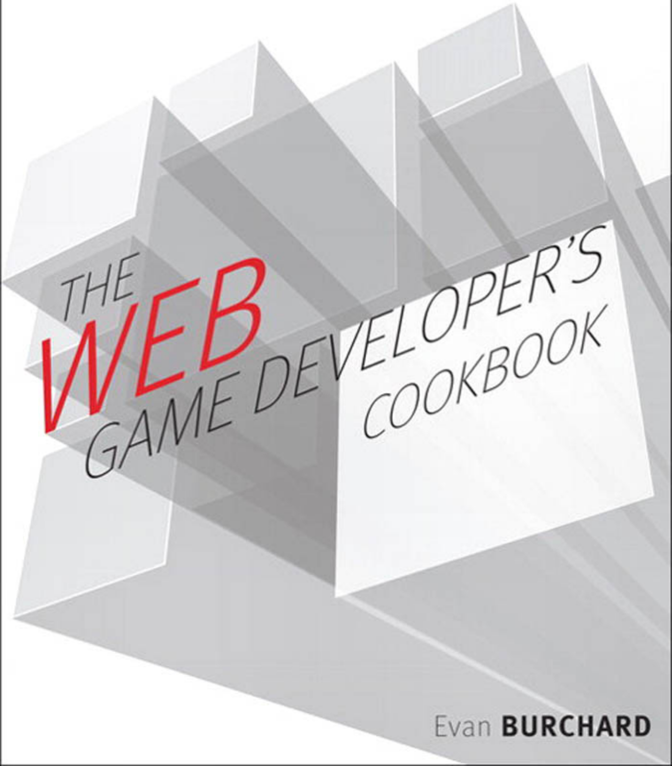
USING JAVASCRIPT AND HTML5 TO DEVELOP GAMES

THE WEB GAME DEVELOPER'S COOKBOOK



Evan BURCHARD

Praise for *The Web Game Developer's Cookbook*

"*The Web Game Developer's Cookbook* is a fun hands-on introduction both to building games and to web technologies. Learning through making is an empowering, exciting first step."

—**Jonathan Beilin**
DIY.org

"It is not only a book about libraries: it teaches how web pages work, how games work, and how to put everything together. Study one, learn three: best deal ever."

—**Francesco "KesieV" Cottone**
Web Alchemist, and Technical Advisor at Vidiemme Consulting

"A wonderful overview of the HTML5 Game Development landscape, covering a wide range of tools and 10 different game genres."

—**Pascal Rettig**
Author of *Professional Mobile HTML5 Game Development*

"With a friendly and reassuring tone, Burchard breaks down some of the most well-known gaming genres into their basic ingredients. *The Web Game Developer's Cookbook* transforms a seemingly daunting task into an approachable crash course even for those who've never written a line of code before."

—**Jason Tocci, Ph.D.**
Writer, Designer, and Researcher

```
      stage.addChild(square);
      stage.update();
    }
  }
```

There is a lot going on here if you haven't dealt much with `for` loops before. In the first line, a variable `i` is created, which keeps track of the loop counter. After the semicolon, the `i<rows*columns` part sets the loop to happen only when `i` is less than the number of squares you want to display. The last part of the first line (after the second semicolon) says to increment the loop counter after each loop iteration.

The x coordinate is set by multiplying the total width of the square `(squareSide+ squarePadding)` by the row position of the square. If you haven't seen the `%` operator before, you can just think of it as the remainder after dividing the first number by the second. For y position, we multiply the total height of the square (also `squareSide+squarePadding`) by the integer portion of the loop counter divided by the number of columns.

If you successfully implemented these changes, you should see a rendering similar to Figure 4.3.

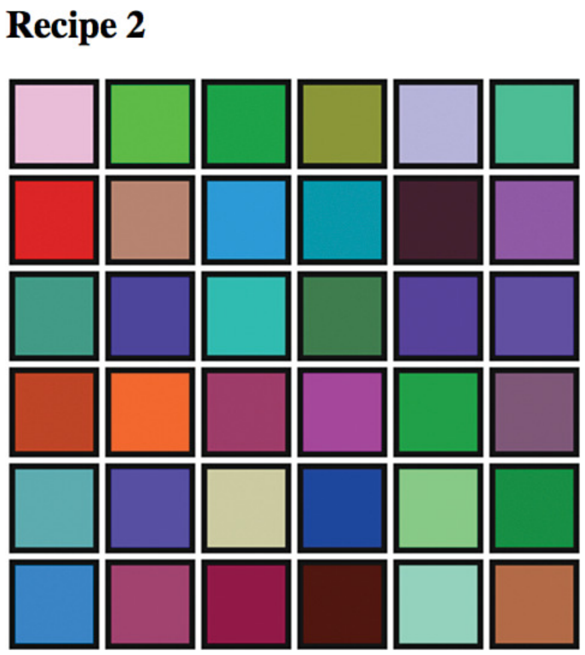


Figure 4.3 Many squares rendered

Recipe: Creating Pairs

So far, every tile has been generated with a random color, and the odds that you get a single pair is actually very improbable. We'll need to ensure that we actually have pairs of tiles to match before adding the logic for click handling.

We need pairs of random colors to appear in random positions. First, let's build a `placement Array` that is the same size as the number of tiles we want to place. Add the following bold lines (shown in Listing 4.13) to your code.

Listing 4.13 Building a placementArray

```
...
var gray = Graphics.getRGB(20, 20, 20);
var placementArray = [];

function init() {
  var rows = 5;
  var columns = 6;
  var squarePadding = 10;
  canvas = document.getElementById('myCanvas');
  stage = new Stage(canvas);

  var numberOfTiles = rows*columns;
  setPlacementArray(numberOfTiles);
...
```

All you're doing here is declaring an empty array and then calling a function that will build the array you want. You also add a `numberOfTiles` variable because with your additions in this recipe, you'll be referencing the value of `rows*columns` more than once. Beyond the benefits mentioned in the last recipe, performing a calculation more than necessary slows down your code.

Next, add the `setPlacementArray()` function that you're calling here (see Listing 4.14). This can go just above your closing `</script>` tag.

Listing 4.14 setPlacementArray Function

```
function setPlacementArray(numberOfTiles){
  for(var i = 0;i< numberOfTiles;i++){
    placementArray.push(i);
  }
}
```

In this function, you create an array that is the size of `numberOfTiles`, with each element of the array equaling its index. The `push()` function simply tacks on each index to the end of the array. This means that you end up with an array that looks like this: `[0, 1, 2, 3, ..., numberOfTiles]`.

With that out of the way, we have some slight adjustments that need to be made to the loop in the `init()` function with the bold lines in Listing 4.15.

Listing 4.15 init() Function Loop Adjustments

```
for(var i=0;i<numberOfTiles;i++){
  var placement = getRandomPlacement(placementArray);
  if (i % 2 === 0){
    var color = randomColor();
  }
  var square = drawSquare(color);
  square.x = (squareSide + squarePadding) * (placement % columns);
  square.y = (squareSide + squarePadding) * Math.floor(placement /
➥columns);
  stage.addChild(square);
  stage.update();
}
```

Other than using `numberOfTiles` rather than `row * column` directly as your termination condition for your loop, there are two more changes to note. First, you are now relying on the `placement`, rather than the index `i` of the loop, to determine the coordinates of the square. Second, you are assigning a new `randomColor` at every even loop index and passing that to the `drawSquare()` function to achieve pairs of colors.

Let's adjust the `drawSquare()` function as shown in Listing 4.16 so that it takes a parameter color and relies on that for `beginFill()` instead of `randomColor()`.

Listing 4.16 The New drawSquare() Function

```
function drawSquare(color) {
  var shape = new Shape();
  var graphics = shape.graphics;
//remove this line: var color = randomColor();
  graphics.setStrokeStyle(squareOutline);
  graphics.beginStroke(gray);
  graphics.beginFill(color);
  graphics.rect(squareOutline, squareOutline, squareSide, squareSide);
  return shape;
}
```

Now you can implement the `getRandomPlacement()` function with the code in Listing 4.17, which can be put just above the closing `</script>` tag.

Listing 4.17 getRandomPlacement

```
function getRandomPlacement(placementArray){
  randomNumber = Math.floor(Math.random()*placementArray.length);
  return placementArray.splice(randomNumber, 1)[0];
}
```

This function does a few things. First, it gets a random element from the `placementArray`. Then, it removes that element from the array and returns it to be set as `placement` in the `init()` function. `getRandomPlacement()` will be executed with each loop, giving one less element to randomly select each time.

Now you have pairs of tiles in random locations, producing something like Figure 4.4.

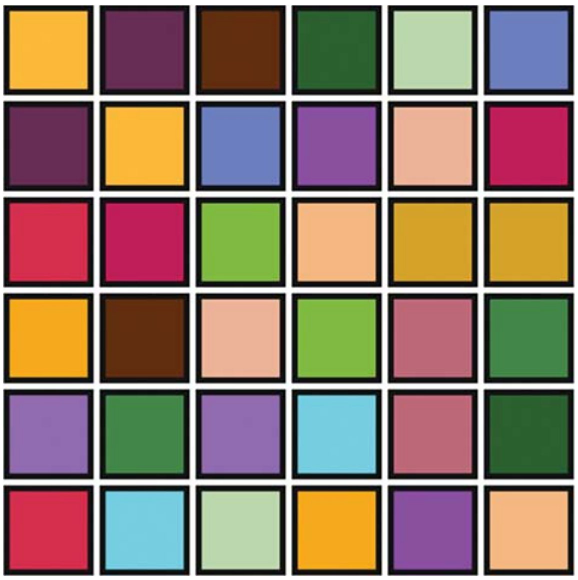


Figure 4.4 Pairs of squares

Recipe: Matching and Removing Pairs

Next, you want to match and remove pairs of tiles. This involves a few tweaks to your setup and a new function for handling click events. Take a look at the simple tweaks first in Listing 4.18.

Listing 4.18 Set Up Tweaks for Clicking, Matching, and Removing

```
var placementArray = [];
var highlight = Graphics.getRGB(255, 255, 0);
var tileClicked;
function init() {
...
for(var i=0;i<numberOfTiles;i++){
  ...
  var square = drawSquare(color);
  square.color = color;
  square.x = (squareSide+squarePadding) * (placement%columns);
  square.y = (squareSide+squarePadding) *
➥Math.floor(placement/columns);
  stage.addChild(square);
  square.onPress = handleOnPress;
  stage.update();
```

First, we create a highlight color and initialize a variable `tileClicked` to store the value of the first tile that is clicked. The last bold line sets a click handler function to each square that is added to the stage. We will define that function in a minute, but first, let's take a closer look at the second bold line, `square.color = color;`.

This is among the author's favorite features of JavaScript. If you have a JavaScript object like `square`, you can define a property of it with this short syntax, and the property can be a number, an object, or even a function. Many languages would require you to define a new method for getting and setting this property. JavaScript does not, and that's awesome. On the not so awesome side, be aware that if square already had a `color` property defined, it would be overwritten by an assignment like this. Also be aware that, although it is namespaced in a fairly safe way because you have to reference it through the object, this property `color` has the same scope as `square`.

Now, to implement the `handleOnPress()` function, put the following code (shown in Listing 4.19) just above the closing `</script>` tag.

Listing 4.19 The handleOnPress Function

```
function handleOnPress(event){
  var tile = event.target;
  if(!!tileClicked === false){
      tile.graphics.setStrokeStyle(squareOutline).beginStroke
➥(highlight).rect(squareOutline, squareOutline, squareSide,
➥ squareSide);
```

```
      tileClicked = tile;
    }else{
      if(tileClicked.color === tile.color && tileClicked !== tile){
        tileClicked.visible = false;
        tile.visible = false;
      }else{
         tileClicked.graphics.setStrokeStyle(squareOutline)
 .beginStroke(gray).rect(squareOutline, squareOutline, squareSide,
➥squareSide);
      }
      tileClicked = null;
    }
    stage.update();
  }
```

In this function, there are two main objects to keep track of. First is the `tile` object, which represents the tile that was just clicked, and the `tileClicked` object, which refers to a tile that was clicked before `tile`. Initially, the previously clicked tile is `undefined` because there is no value set to it. Your conditional check here may seem a little strange. You could have said `if (tileClicked === undefined)`, but this would fail later after setting the variable to `null`. Admittedly, you can set the variable to `undefined` with `tileClicked = undefined`, but defining something as `undefined` is strange and could lead to confusion. Your strategy here is to account for the cases in which the variable is set to either an object (which should evaluate to true) or `undefined/null` (which should evaluate to false). The `!!` operator first says "give me the negative of the 'truthy' value of what comes after the '`!`'," and then it says that again. The effect is that the "truthy" value of what follows `!!` is returned. If you're confused, try experimenting in a console a bit or spending some time with Appendix A. For your purposes, this conditional could be read simply as, "if `tileClicked` is `undefined` or `null`." So what happens in that case?

First, the tile receives a highlighted border. Next, the `tileClicked` variable is set to `tile`. That means that the next time a tile is clicked, it follows the `else` path of this conditional. The conditional checks to see if the two tiles clicked were a match. It does this strictly by checking the color property of the tiles. There is a second condition after the `&&` that ensures that clicking the same tile twice does not count as a match. When a match occurs, the visibility of both tiles is set to `false`, which hides them. If there is no match, the highlight is removed from the tile that was clicked first. In both cases, `tileClicked` is set to `null` so that the next time `if (!!tileClicked === false)` evaluates to `true`, starting the cycle over again.

After opening index.html up, and clicking some of the pairs, you should see something like Figure 4.5.