Canvas from Scratch: Pixel Manipulation

by Rob Hawkes

In the last article, you learned all about transformations, shadows and gradients. Today, I'm going to show you how to manipulate pixels in canvas; from simply accessing color values, to editing images within the canvas just like a photo editor.

This is easily one of the most powerful features built into canvas directly, and once you've learned it, I guarantee that you'll have a whole range of exciting ideas.

Setting Up

You're going to use the same HTML template from the previous articles, so open up your favorite editor and copy in the following code:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Canvas from scratch</title>
        <meta charset="utf-8">

        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>

        <script>
            $(document).ready(function() {
                var canvas = document.getElementById("myCanvas");
                var ctx = canvas.getContext("2d");
            });
        </script>
    </head>

    <body>
        <canvas id="myCanvas" width="500" height="500">
            <!-- Insert fallback content here -->
        </canvas>
    </body>
</html>
```

This is nothing more than a basic HTML page with a `canvas` element and some JavaScript that runs after the DOM has loaded. Nothing crazy.

Placing an Image on the Canvas

You can manipulate pixels with anything drawn onto the canvas, but for the sake of this tutorial, you'll be using images. This is partly because it's important to show you how to load images into the canvas, but also because the ability to perform image manipulation (eg. editing photos) is massive plus-point of this technology.

Before I show you how to access pixel values, let's place an image onto the canvas. Feel free to use any image that you want, but for the sake of this example, I'm going to use one of my own photos from Flickr.

You have permission to use this photo if you wish, which you can download in a variety of sizes.

Loading an image into canvas requires two steps. The first is to load the image into a HTML `image` element, which can be done using HTML or by creating a new DOM element directly within JavaScript. In this example, you're going to create a new DOM element — it's dead simple:

```
var image = new Image();
image.src = "sample.jpg";
$(image).load(function() {
```

```
});
```

All you're doing here is creating a new `Image` DOM element and assigning it to a variable. You then use that variable to load up your image by setting the `src` attribute of the image to the correct path. It's worth noting that you could load in a remote image using this technique, but this raises a few issues for us further down the line so we'll stick with a locally stored image for now. The final step is to listen for the `load` event that will be fired as soon as the image has finished loading and is available for use.

Once the image has loaded, you can then place it on the canvas in one easy step. All you need to do is pass the `image` variable that you just created into a call to the `drawImage` method of the 2d rendering context. Place it inside the `image` load event, like so:

```
$(image).load(function() {
    ctx.drawImage(image, 0, 0);
});
```

In this case, the `drawImage` method takes three arguments; an image element, as well as the *x* and *y* coordinate values to place the image on the canvas. This will draw the image at full size (500px for this image) and at the specified position:

However, `drawImage` can actually take a further two arguments that define the width and height to draw the image, like so:

```
ctx.drawImage(image, 0, 0, 250, 166);
```

This would draw the image at half the original size (250px for this image):

You can even take things a step further and use the full nine arguments for `drawImage` to only draw a small portion of the original image, like so:

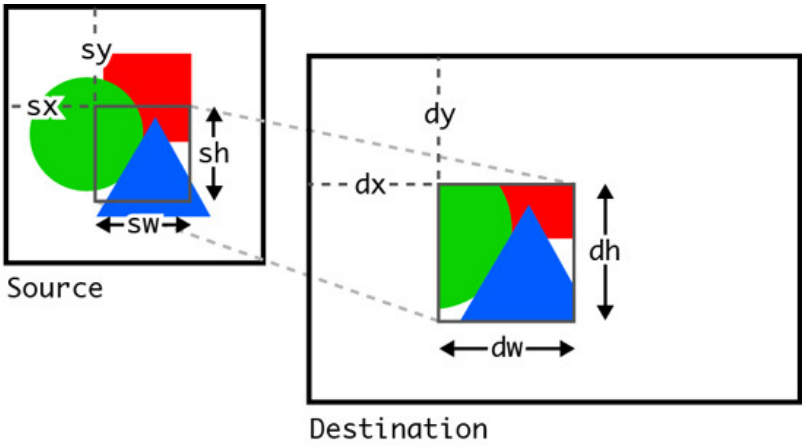
```
ctx.drawImage(image, 0, 0, 200, 200, 0, 0, 500, 500);
```

This would take a 200px square from the top left of the image and draw it on the canvas at 500px square:

In pseudo-code, the full nine `drawImage` arguments can be described like so (s meaning source, and d meaning destination):

```
ctx.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh);
```

And the result is visualized in the following illustration:



Simple, right? In all honesty, nothing in canvas is that complicated once you break it down and look at the pieces individually.

Accessing Pixel Values

Now that you have an image on the canvas it's time to access the pixels so that you can manipulate them. However, let's forget about manipulating them for now and concentrate purely on accessing them as the concept take a little while to get your head around.

Security Issues

If you want to access pixels using canvas you need to be aware of the security limitations that are involved. These limitations only allow you to access the data from images loaded on the same *domain* as the JavaScript. This prevents you from accessing an image from a remote server and then analyzing its pixels, although there is a way to get around it, sort of. Unfortunately not all browsers treat JavaScript and images run locally from the file system (ie, without a domain name) as under the same domain, so you might receive security errors. To get around this you need to either run the rest of this tutorial on a local development environment (like MAMP, WAMP, or XAMPP) or a remote Web server and access the files using a domain name (like example.com).

With that out of the way, let's crack on and get us some pixels!

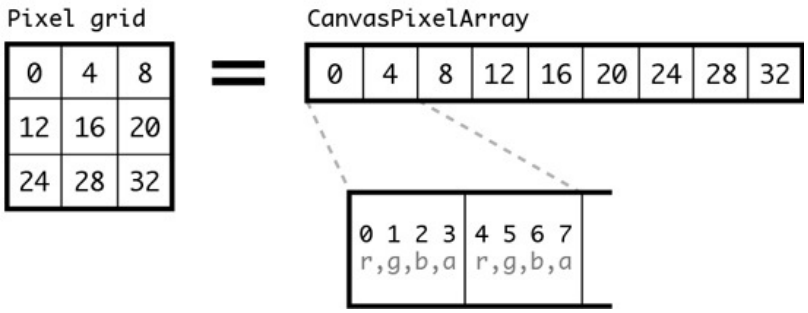
Accessing Pixels is a Little Odd

As I mentioned at the beginning of this section, accessing pixel values in canvas takes a little while to get your head around. This is due to the way that the pixels are stored by canvas; they're not stored as whole pixels at all! Instead, pixels are each broken up into four separate values (red, green, blue, and alpha) and these values are stored in a one-dimensional array with all the color values for the other pixels. Because of this you can't just request the data from a particular pixel, at least not by default. Let me explain.

To access pixels in canvas you need to call the `getImageData` method of the 2d rendering context, like so:

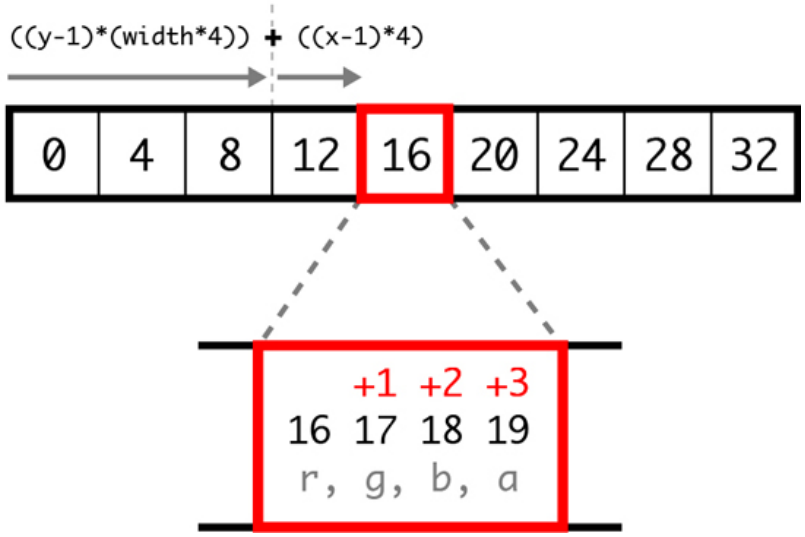
```
var imageData = ctx.getImageData(x, y, width, height);
```

This method takes four arguments that describe a rectangular area of the canvas that you want the pixel data from; an *x* and *y* origin, followed by a *width* and *height*. It returns a `CanvasPixelArray` that contains all the color values for the pixels within the defined area. The first thing to notice with the `CanvasPixelArray` is that each pixel has four color values, so the index of the first color value for each pixel within the array will be a multiple of 4 (0 for the first value of the first pixel, 4 for the first value of the second, etc):



What's interesting about this array (or annoying, depending on how you look at it) is that there is no concept of (x, y) coordinate position, meaning that retrieving color values for a specific pixel is a little harder than accessing a two-dimensional array (eg. using pixelArray[0][3] to access the pixel at (1, 4)). Instead, you need to use a little formula that's actually very easy to understand once it's explained properly:

```
var redValueForPixel = ((y - 1) * (width * 4)) + ((x - 1) * 4);
```



Can you work out what's happening here? Let's break it down and pretend that we want to get the pixel color values for the innermost pixel in a 3×3 pixel grid – the pixel at (2, 2).

If you look at the previous two images you can see that the color values for this pixel will begin at index 16, but to work this out with code you need to do two things; first calculate the index at the beginning of the row that the pixel is on (the *y* position), and then add to that index the number of color values that exist between the pixel and the beginning of the row (the *x* position). It's a bit of a mind-bender, but bear with it.

The first part is easy, you already know that there are four color values per pixel, and you already know the width of the grid (3 pixels). To calculate the index of the pixel at row *y* (2) you pass these values through the first part of the formula, which would look like so:

```
((2 - 1) * (3 * 4))
```

This gives you an index of 12, which you'll see matches up with the first pixel on the second row in the previous images. So far so good.

The next step is to calculate the number of color values that exist before the pixel that you want on this row. To do that you simply multiply the number of pixels before the one you want by four. Simple. In this case the second part of the formula would look like this:

```
((2 - 1) * 4)
```

You can work it all out if you want, but the answer is 4, which when added to the previous value gives you an index of 16. Cool, ey?

I wouldn't worry too much about understanding it fully, just know that this amazing little formula exists so that you can easily get the index of the red color value for any pixel. To get the index of the other color values of a pixel (green, blue, or alpha), you just add 1, 2, or 3 to the calculated index respectively.

Putting this into Practice

Now that you know how to grab any pixel that you want, let's put this into practice and grab color values from an image to change the color of a website background. This kind of technique would work great as a color picker for a photo editing Web application.

The code for this example is fairly straightforward, so let's attack it all in one go:

```
var image = new Image();
image.src = "sample.jpg";
$(image).load(function() {
    ctx.drawImage(image, 0, 0);
});

$(canvas).click(function(e) {
    var canvasOffset = $(canvas).offset();
    var canvasX = Math.floor(e.pageX-canvasOffset.left);
    var canvasY = Math.floor(e.pageY-canvasOffset.top);

    var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    var pixels = imageData.data;
    var pixelRedIndex = ((canvasY - 1) * (imageData.width * 4)) + ((canvasX - 1) * 4);
    var pixelcolor = "rgba("+pixels[pixelRedIndex]+", "+pixels[pixelRedIndex+1]+", "+pixels[pixelRedIndex+2]+", "+pixels[pixelRedIndex+3]+")";

    $("body").css("backgroundColor", pixelcolor);
});
```

You'll recognise the first few lines from the previous examples. All the new stuff is within the click handler on the `canvas` element, which uses a tiny bit of jQuery to tell you when the canvas has been clicked on.

Within the click handler you want to work out the pixel that the mouse has clicked on the canvas. To do this you first need to calculate the offset in pixels of the top left position of the canvas from the top left edge of the browser window, you can use the jQuery `offset` method for this. You can then infer the pixel clicked on the canvas by subtracting the offset from the mouse position of the click event (`pageX` and `pageY`). You should definitely spend some time reading up on the JavaScript click event if you want to understand this further.

The following four lines grab the `CanvasPixelArray` for the canvas (`getImageData`), store it in a variable, find the index of the red color value for the clicked pixel by calculating it using the formula that you saw earlier, and then stores the pixel color values as a CSS `rgba` string. Finally, the last step is to set the background color of the `body` element to that of the clicked pixel.

And with that you're done. Try it out for yourself; click the image on the canvas and watch the background of the website change color. If it doesn't work, make sure that you're running the demo on a server with a domain name, like described in the section on security issues.

It's been a long journey, but you're now able to quickly and easily retrieve the color values of any pixel on the canvas. Did I tell you that you can also change the color values of pixels on the canvas? I didn't? Oops! Let's take a look at that now then, it's dead cool.

Applying Effects to Images

Now that you're able to access the pixel color values of the canvas, changing those values is a breeze. In fact, changing those color values is as simple as changing the values in the `CanvasPixelArray` and then drawing it back onto the canvas. Let's take a look at how to do that.

The first step is to set up the code as you did in the previous section. This code loads an image, draws it onto the canvas, and then grabs the pixel data:

```
var image = new Image();
image.src = "sample.jpg";
$(image).load(function() {
    ctx.drawImage(image, 0, 0);

    var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    var pixels = imageData.data;
    var numPixels = imageData.width * imageData.height;
});
```

So far so good. The next step is to loop through every pixel on the canvas and change its color values. In this example you're going to invert the colors by deducting the current color value (0 to 255) from 255:

```
for (var i = 0; i < numPixels; i++) {
    pixels[i*4] = 255-pixels[i*4]; // Red
    pixels[i*4+1] = 255-pixels[i*4+1]; // Green
    pixels[i*4+2] = 255-pixels[i*4+2]; // Blue
};
```

There's nothing crazy going on here; you're simply multiplying the pixel number (*i*) by 4 to get the index of the red color value for that pixel in the `CanvasPixelArray`. By adding 1 or 2 to that number you can get and change the green and blue color values respectively.

Finally, all you need to do now is to clear the canvas (to get rid of the normal image), and then use the `putImageData` method of the 2d rendering context to draw the saved `CanvasPixelArray` to the canvas:

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
ctx.putImageData(imageData, 0, 0);
```

And that's honestly all there is to it; reload your browser and take a look for yourself. Cool, isn't it?

Wrapping Things Up

There is so much more to pixel manipulation in canvas, but I hope that you've experienced enough in this article to get your juices flowing. I encourage you to explore this area further and see what else you can do with pixels. Why? Because all of the techniques that you leant about pixel manipulation can be used for HTML5 video as well as images. Now that's cool!

In the next article, the final one in this series, we'll be taking a different look at canvas. This time, you'll learn how to animate on the canvas, which will give you the basics required to create cartoons, animations, and games. This is undoubtably my favorite use of canvas.