HTML5 canvas - the basics

DEV.OPERA

By Mihai Sucan

Table of contents

Introduction

The HTML5 specification (http://www.whatwg.org/specs/web-apps/current-work/multipage/) includes lots of new features, one of which is the `canvas` element (http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html). HTML5 `canvas` gives you an easy and powerful way to draw graphics using JavaScript. For each `canvas` element you can use a "context" (think about a page in a drawing pad), into which you can issue JavaScript commands to draw anything you want. Browsers can implement multiple canvas contexts and the different APIs provide the drawing functionality.

Most of the major browsers include the 2D canvas context capabilities - Opera (http://opera.com/), Firefox (http://mozilla.com/), Konqueror (http://konqueror.org/) and Safari (http://apple.com/safari). In addition, there are experimental builds of Opera that include support for a 3D canvas context, and an add-on that allows 3D canvas support in Firefox:

- Download an Opera build featuring 3D canvas, HTML video and File I/O support. (/articles/view/labs-video-3d-canvas-and-file-i-o-repeat/)

- Find more out about using the Opera 3D canvas context. (http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension)
- Find more out about obtaining and using the Firefox 3D canvas context. (http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/)

This article takes you through the basics of implementing a 2D canvas context, and using the basic canvas functions, including lines, shape primitives, images, text, and more. You are assumed to have mastered JavaScript basics already.

Note that you can download all the code examples in a single zip file (canvas-primer.zip), as well as viewing them live using the links below.

The basics of using canvas

Creating a canvas context on your page is as simple as adding the `<canvas>` element to your HTML document like so:

```
<canvas id="myCanvas" width="300" height="150">
Fallback content, in case the browser does not support Canvas.
</canvas>
```

You need to define an element ID so you can find the element later in your JavaScript code, and you also need to define the width and height of the canvas.

That's your drawing pad created, so now let's put pen to paper. To draw inside your canvas you need to use JavaScript. First you find your canvas element using `getElementById`, then you initialize the context you want. Once you do that, you can start drawing into the canvas using the available commands in the context API. The following script (try running the example live (http://www.robodesign.ro/coding/canvas-primer/20081208/example-using-canvas.html)) draws a simple rectangle into the canvas we defined above:

```
// Get a reference to the element.
```

```
var elem = document.getElementById('myCanvas');

// Always check for properties and methods, to make sure your code doesn't brea
k
// in other browsers.
if (elem && elem.getContext) {
  // Get the 2d context.
  // Remember: you can only initialize one context per element.
  var context = elem.getContext('2d');
  if (context) {
    // You are done! Now you can draw your first rectangle.
    // You only need to provide the (x,y) coordinates, followed by the width and
    // height dimensions.
    context.fillRect(0, 0, 150, 100);
  }
}
```

You can choose to include this script inside the `head` of your document, or in an external file - it's up to you.

The 2D context API

Now we have created our first basic canvas image, let's look a bit more deeply into the 2D canvas API, and see what is available for us to make use of.

Basic lines and strokes

You already saw in the example above that it's really easy to draw rectangles colored the way you want.

With the *fillStyle* and *strokeStyle* properties you can easily set the colors used for rendering filled shapes and strokes. The color values you can use are the same as in CSS: hex codes, rgb(), rgba() and even hsla() if the browser supports it (for example this feature is supportd in Opera 10.00 and later).

With `fillRect` you can draw filled rectangles. With `strokeRect` you can draw rectangles only using borders, without filling. If you want to clear some part of the canvas, you can use `clearRect`. These three methods all use the same arguments: *x*, *y*, *width*, *height*. The first two arguments tell the (x,y) coordinates, and the last two arguments tell the width and height dimensions for the rectangle.

To change the thickness of the lines, you can use the *lineWidth* property. Let's look at an example that uses `fillRect`, `strokeRect` `clearRect` and more (http://www.robodesign.ro/coding/canvas-primer/20081208/example-rects.html):

```
context.fillStyle   = '#00f'; // blue
context.strokeStyle = '#f00'; // red
context.lineWidth   = 4;

// Draw some rectangles.
context.fillRect  (0,   0, 150, 50);
context.strokeRect(0,  60, 150, 50);
context.clearRect (30, 25,  90, 60);
context.strokeRect(30, 25,  90, 60);
```

This example gives you an output like that seen in Figure 1.

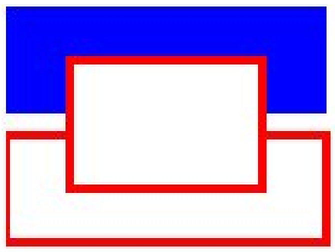
(http://www.robodesign.ro/coding/canvas-primer/20081208/example-rects.html)

Figure 1: Example rendering of fillRect, strokeRect and clearRect.

Paths

The canvas paths allow you to draw custom shapes. You draw the "outline" first, then choose to draw the stroke and fill the shape at the end, if you wish. Creating a custom shape is simple - to start drawing the path, use `beginPath()`, then draw the path that makes up your shape using lines, curves and other primitives. Once you are done, call `fill` and `stroke` if you want to fill your shape or to draw the stroke, then call `closePath()` to finish off your shape.

An example is in order - the following code will draw a triangle (http://www.robodesign.ro/coding/canvas-primer/20081208/example-triangle.html):

```
// Set the style properties.
context.fillStyle   = '#00f';
context.strokeStyle = '#f00';
context.lineWidth   = 4;

context.beginPath();
// Start from the top-left point.
context.moveTo(10, 10); // give the (x,y) coordinates
context.lineTo(100, 10);
context.lineTo(10, 100);
context.lineTo(10, 10);

// Done! Now fill the shape, and draw the stroke.
// Note: your shape will not be visible until you call any of the two methods.
context.fill();
context.stroke();
context.closePath();
```

This will give an output like that shown in Figure 2.

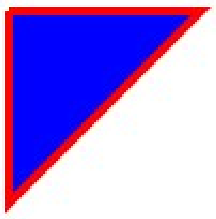

(http://www.robodesign.ro/coding/canvas-primer/20081208/example-triangle.html)

Figure 2: A basic triangle.

I have also prepared a more complex paths example featuring lines, curves and arcs (http://www.robodesign.ro/coding/canvas-primer/20081208/example-paths.html) - check it out.

Inserting images

The `drawImage` method allows you to insert other images (`img` and `canvas` elements) into your canvas context. In Opera you can also draw SVG images inside your canvas. This is quite a complex method, which takes three, five or nine arguments:

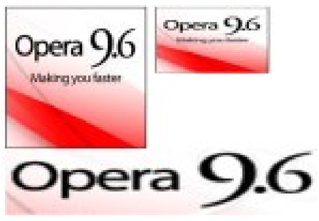
- Three arguments: The basic `drawImage` usage involves one argument to point to the image to be included, and two to specify the destination coordinates inside your canvas context.
- Five arguments: The middle `drawImage` usage includes the above three arguments, plus two to specify the width and height of the inserted image (in cases where you want to resize it).
- Nine arguments: The most advanced `drawImage` usage includes the above five arguments, plus two values for coordinates inside the source images, and two values for width and height inside the source image. These values allow you to dynamically crop the source image before bringing it into your canvas context.

The following example code shows all three types of `drawImage` in action (http://www.robodesign.ro/coding/canvas-primer/20081208/example-drawimage.html):

```
// Three arguments: the element, destination (x,y) coordinates.
context.drawImage(img_elem, dx, dy);

// Five arguments: the element, destination (x,y) coordinates, an
d destination
// width and height (if you want to resize the source image).
context.drawImage(img_elem, dx, dy, dw, dh);

// Nine arguments: the element, source (x,y) coordinates, source
width and
// height (for cropping), destination (x,y) coordinates, and dest
```

```
ination width
// and height (resize).
context.drawImage(img_elem, sx, sy, sw, sh, dx, dy, dw, dh);
```

This should render as shown in Figure 3.



(http://www.robodesign.ro/coding/canvas-primer/20081208/example-drawimage.html)

Figure 3: Example `drawImage` rendering.

Pixel-based manipulation

The 2D Context API provides you with three methods that help you draw pixel-by-pixel: `createImageData`, `getImageData`, and `putImageData`.

Raw pixels are held in objects of type `ImageData`. Each object has three properties: *width*, *height* and *data*. The *data* property is of type CanvasPixelArray, holding a number of elements equal to `width*height*4`; this means that for every pixel you define the red, green, blue and alpha values of all the pixels, in the order you want them to appear (all the values range from 0 to 255, including alpha!). Pixels are ordered left to right, row by row, from top to bottom.

To better understand how all this works take a look at an example which draws a block of red pixels (http://www.robodesign.ro/coding/canvas-primer/20081208/example-imagedata2.html).

```
// Create an ImageData object.
var imgd = context.createImageData(50,50);
var pix = imgd.data;

// Loop over each pixel and set a transparent red.
for (var i = 0; n = pix.length, i < n; i += 4) {
  pix[i  ] = 255; // red channel
  pix[i+3] = 127; // alpha channel
}

// Draw the ImageData object at the given (x,y) coordinates.
context.putImageData(imgd, 0,0);
```

Note: not all browsers implement `createImageData`. On such browsers, you need to obtain your `ImageData` object using the `getImageData` method. Please see the provided example code (http://www.robodesign.ro/coding/canvas-primer/20081208/example-imagedata2.html).

With the `ImageData` capabilities you can do a lot more than that. For example, you can do image filtering, or you can do mathematical visualisations (think fractals and more). The following code shows you how to create a simple color inversion filter (http://www.robodesign.ro/coding/canvas-primer/20081208/example-imagedata.html):

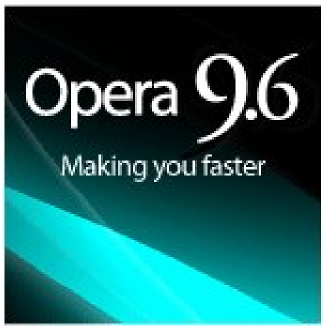
```
// Get the

 CanvasPixelArray

 from the given coordinates and dimensions.
var imgd = context.getImageData(x, y, width, height);
var pix = imgd.data;

// Loop over each pixel and invert the color.
for (var i = 0, n = pix.length; i < n; i += 4) {
  pix[i  ] = 255 - pix[i  ]; // red
  pix[i+1] = 255 - pix[i+1]; // green
  pix[i+2] = 255 - pix[i+2]; // blue
  // i+3 is alpha (the fourth element)
}

// Draw the
```

```
 ImageData
```

at the given (x,y) coordinates.

```
context.putImageData(imgd, x, y);
```

Figure 4 shows the color inversion filter applied to an Opera graphic (compare to Figure 3, which shows the original color scheme of the Opera graphic).



(http://www.robodesign.ro/coding/canvas-primer/20081208/example-imagedata.html)

Figure 4: The color inversion filter in action.

Text

The Text API is only available in recent WebKit builds, and in Firefox 3.1 nightly builds, but I decided to include it here for completeness.

The following text properties are available on the `context` object:

- `font`: Specifies the font of the text, in the same manner as the CSS `font-family` property)
- `textAlign`: Specifies the horizontal alignment of the text. Values: `start`, `end`, `left`, `right`, `center`. Default value: `start`.
- `textBaseline`: Specifies the vertical alignment of the text. Values: `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, `bottom`. Default value: `alphabetic`.

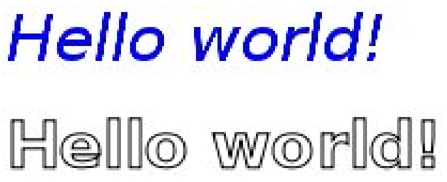
You have two methods for drawing text: `fillText` and `strokeText`. The first one draws the text shape, filled using the current *fillStyle*, while the latter draws the text outline/border using the current *strokeStyle*. Both take three arguments: the text you want to display, and the (x,y) coordinates to define where to render it. There's also an optional fourth argument - maximum width. This causes the browser to shrink the text to fit inside the given width, if that's needed.

The text alignment properties affect the text position relative to the (x,y) coordinates you give to the drawing methods.

At this point, an example is in order - the following code is a simple canvas text "hello world" example (http://www.robodesign.ro/coding/canvas-primer/20081208/example-text.html).

```
context.fillStyle    = '#00f';
context.font         = 'italic 30px sans-serif';
context.textBaseline = 'top';
context.fillText  ('Hello world!', 0, 0);
context.font         = 'bold 30px sans-serif';
context.strokeText('Hello world!', 0, 50);
```

Figure 5 shows the output of this example.



(http://www.robodesign.ro/coding/canvas-primer/20081208/example-text.html)
Figure 5: A simple canvas text rendering.

Shadows

The Shadow API gives you four properties:

- `shadowColor`: Sets the shadow color you want. The value allowed is the same as the CSS color values.
- `shadowBlur`: Sets the amount of blur on the shadow, in pixels. The lower the blur value, the sharper the shadows are. It gives a very similar effect to gaussian blur in Photoshop.
- `shadowOffsetX` and `shadowOffsetY`: Specifies the x and y offset of the shadow,

again in pixels.

Usage is very straightforward, as shown by the following canvas shadow code example (http://www.robodesign.ro/coding/canvas-primer/20081208/example-shadows.html):

```
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur    = 4;
context.shadowColor   = 'rgba(255, 0, 0, 0.5)';
context.fillStyle     = '#00f';
context.fillRect(20, 20, 150, 100);
```

This will render as shown in Figure 6.

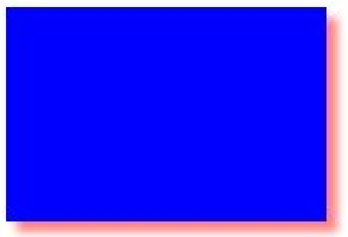

(http://www.robodesign.ro/coding/canvas-primer/20081208/example-shadows.html)

Figure 6: Example canvas shadow - a blue rectangle with a red shadow.

Gradients

The `fillStyle` and `strokeStyle` properties can also have `CanvasGradient` objects assigned to them, instead of CSS color strings - these allow you to use color gradients to color your lines and fills instead of solid colors.

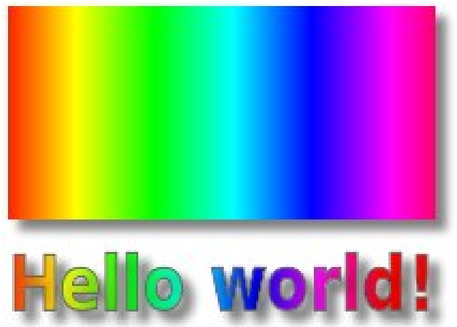
To create `CanvasGradient` objects you can use two methods: `createLinearGradient` and `createRadialGradient`. The former creates a linear gradient - lines of color all going in one direction - while the latter creates a radial gradient - circles of color emanating out from a single point.

Once you have the gradient object you can add color stops along the gradient using the `addColorStop` method of the object.

The following code shows you how to use gradients:

```
// You need to provide the source and destination (x,y) coordinates
// for the gradient (from where it starts and where it ends).
var gradient1 = context.createLinearGradient(sx, sy, dx, dy);

// Now you can add colors in your gradient.
// The first argument tells the position for the color in your gradient. The
// accepted value range is from 0 (gradient start) to 1 (gradient end).
// The second argument tells the color you want, using the CSS color format.
gradient1.addColorStop(0,   '#f00'); // red
gradient1.addColorStop(0.5, '#ff0'); // yellow
gradient1.addColorStop(1,   '#00f'); // blue

// For the radial gradient you also need to provide source
// and destination circle radius.
// The (x,y) coordinates define the circle center points (start and
// destination).
var gradient2 = context.createRadialGradient(sx, sy, sr, dx, dy,
dr);

// Adding colors to a radial gradient is the same as adding colors to linear
// gradients.
```

I have also prepared a more advanced example, which makes use of a linear gradient, shadows and text (http://www.robodesign.ro/coding/canvas-primer/20081208/example-gradients.html). The example produces an output as seen in Figure 7.



(http://www.robodesign.ro/coding/canvas-primer/20081208/example-gradients.html)

Figure 7: Example rendering using a linear gradient.

Online canvas demos

If you want to see what others have done with Canvas, you can take a look at the following projects and demos:

- Newton polynomial (http://www.benjoffe.com/code/demos/interpolate/)
- Canvascape - "3D Walker" (http://www.benjoffe.com/code/demos/canvascape/)
- Paint.Web - painting demo, open-source (http://code.google.com/p/paintweb)
- Star-field flight (http://arapehlivanian.com/wp-content/uploads/2007/02/canvas.html)
- Interactive blob (http://www.blobsallad.se/)

Summary

Canvas is one of the most interesting HTML5 features, and it's ready to be used within most modern Web browsers. It provides all you need to create games, user interface enhancements, and other things besides. The 2D context API includes a wealth of functionality in addition to that discussed in this article - I hope you've gained a good grounding in canvas, and a thirst to know more!

Read more...

- Creating an HTML5 canvas painting application (/articles/view/html5-canvas-painting/)
- SVG or Canvas? Choosing between the two (/articles/view/svg-or-canvas-choosing-between-the-two/)