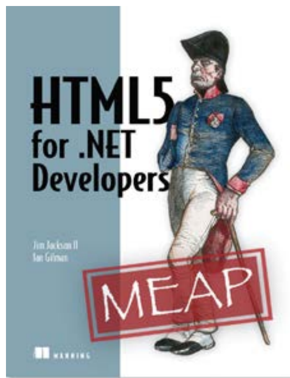


HTML5 for .NET Developers
By Jim Jackson II and Ian Gilman

The speed of graphics manipulation and code execution in modern browsers has opened up new doors to those intrepid souls who have started developing games using the HTML5 canvas. This article based on chapter 5 of HTML5 for .NET Developers explains the basic drawing process in canvas.

To save 35% on your next purchase use Promotional Code **jackson0535** when you check out at http://www.manning.com/.

You may also be interested in…

Introducing Canvas

The HTML canvas element is new in version 5 and provides a means of drawing bitmap-based images inside the browser. Canvas has received a lot of media and developer attention because it is, in fact, a blank canvas upon which anyone can draw practically any visual element. In addition, because the canvas element is an HTML element like any other, it can be addressed in JavaScript as a regular element and participate in page flow and styling.

The canvas element must consist of an opening and a closing tag—it can't be self closing. Therefore, the simplest canvas element on a page must be this:

```
<canvas></canvas>
```

This is done for backward compatibility. If a particular browser does not support canvas, the contents of the canvas tag will be displayed as the fallback presentation. If canvas is supported, then the contents will not be presented. The next important point that we need to know about canvas is how to assign the size. Because it is a normal HTML element, it is possible to assign height and width style settings. You can also use the defined `height` and `width` properties to resize the canvas. Both will change the size of the canvas but with completely different results. The example in figure 1 shows how this works.

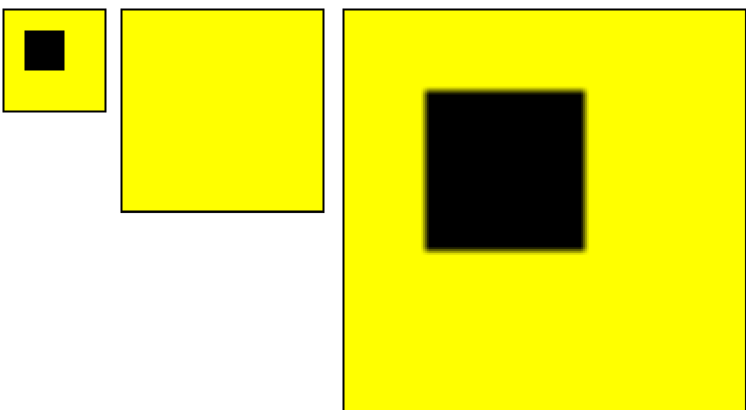


Figure 1 A simple canvas at left is 50 pixels tall and resized by setting height and width properties to 100 pixels. The result is the canvas in the center: resized but cleared of the drawn contents. If the 50 pixel canvas were resized to 200 pixels using CSS style properties, the canvas on the right would be the result: a resized element with contents scaled.

Figure 1 shows how important it is to understand where and when your canvas element is being resized. It will participate in the flow of a page by default as an inline element but editing its size will have an effect on whatever content you have drawn into it, whether by clearing it completely or scaling it to the new size.

Another point to remember about resizing a canvas is that assignment of the `height` and `width` properties will not be honored by the browser if height or width style properties are already applied. The result of that assignment would be a cleared canvas that is sized to the CSS style specification.

The canvas element gives an HTML developer the ability to draw pixel by pixel or to create image content using lines, paths, shapes, images, and text of any color. You can also perform animations. This opens up the areas of gaming, drawing, image processing, and three-dimensional modeling. All these areas were once the province of plugin and desktop developers, but no more. Before you build the next killer app though, you need to know the basics.

The 2d context

In order to draw on a canvas element, you need to get a reference to the `<canvas>` element on your page. With the element variable, you get a reference to the canvas's `context` object. Then the fun really begins! What is a canvas `context`, you ask? It simply is the hook that you use to reference all the canvas-drawing API methods. The only canvas functionality that is not available from the context are the previously mentioned `height` and `width` property settings. All of the rest begins and ends with the `context`.

Pixels drawn with the `context` object do not participate in page layout and they cannot be styled using CSS. They also cannot be displayed outside the boundaries of the canvas itself. So, a square drawn at the top corner of a canvas with CSS `border-radius` set would be partially hidden by the border, as shown in figure 2.

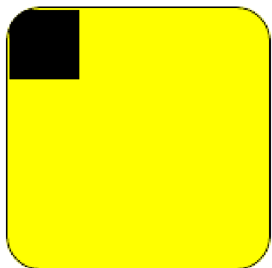


Figure 2 A canvas, regardless of its border-radius properties, will not allow drawn content to appear outside itself.

Getting a reference to the `context` object with JavaScript is a simple two-step process. First, find the element; then, get its context.

```
var canv = document.getElementById("myCanvas");
var ctx = canv.getContext("2d");
```

Notice we are calling for the 2d context. This is for future compatibility with a 3d context that has not yet been specified or implemented by any production browser. Once the HTML5 specification is complete and available across all browsers, expect a 3d canvas context to come soon thereafter.

Getting a reference to the context with jQuery is just as simple as long as you remember that you are working with wrapped sets and not individual elements.

```
var elemCanvas = $("#myCanvas");
var context = elemCanvas[0].getContext("2d");
```

In this code, we get a wrapped set of one canvas element using the id selector. We then call the first element in the wrapped set and get its 2d context.

The following is a complete list of properties and methods of the canvas context object. These should be universally available in any browser that supports HTML5 canvas.

Table 1 Canvas Context(2d) properties

Property	Example Value	Description
Canvas	HTMLCanvasElement	Reference to parent canvas
fillStyle	"#000000"	Background fill for the canvas element
Font	"10px sans-serif"	CSS font value for the canvas
globalAlpha	1	Transparency value for elements on the canvas
globalCompositeOperation	"source-over"	Assigns how elements are drawn on the canvas
lineCap	“round”	Describes shape of line ends
lineJoin	"bevel"	Describes shape for joined lines
lineWidth	1	Default width for line elements
miterLimit	10	Miter limit ratio for line elements
shadowBlur	0	Describes how a shadow effect fades away
shadowColor	"rgba(0, 0, 0, 0)"	Color of a shadow effect
shadowOffsetX	0	How far left or right to offset a blur effect
shadowOffsetY	0	How far up or down to offset a blur effect
strokeStyle	"#000000"	CSS-style stroke value for lines
textAlign	"start"	Align text relative to text starting point defined
textBaseline	"middle"	Vertical text alignment value

Table 2 Canvas Context(2d) methods

Arc(x, y, radius, startAngle, endAngle [, anticlockwise])	Describes an arched path
arcTo(x1, y1, x2, y2, radius)	Describes an arc between two points
beginPath()	New path starter
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)	Bezier curve between two points
clearRect(x, y, w, h)	Clears pixels in a rectangle
clip()	Clip a portion of the canvas with a region
closePath()	Explicitly ends a path
createImageData(sw, sh)	Return ImageData object using dimensions
createImageData(imagedata)	Return ImageData object from an ImageData
createLinearGradient(x0, y0, x1, y1)	Creates linear gradient fill color
createPattern(image, repetition)	Replicates an image with a specified pattern
createRadialGradient(x0, y0, r0, x1, y1, r1)	Creates radial gradient fill color
drawImage(image, dx, dy)	Draw a bitmap on the canvas
drawImage(image, dx, dy, dw, dh)	Draw a bitmap on the canvas
drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)	Draw a bitmap on the canvas

fill()	Fills a shape or path
fillRect(x, y, w, h)	Draws and fills a rectangle
fillText(text, x, y [, maxWidth])	Draws text on the canvas
getImageData(sx, sy, sw, sh)	Gets pixel image data for a section of canvas
isPointInPath(x, y)	Determines if a point is in the current path
lineTo(x, y)	Appends a line between two points
measureText(text)	Returns the length of a text string
moveTo(x, y)	Moves a point and starts a new path
putImageData(imagedata, dx, dy [, dirtyX, dirtyY, dirtyWidth, dirtyHeight])	Draws image pixel data onto a canvas
quadraticCurveTo(cpx, cpy, x, y)	Draws Bezier curve using a path
rect(x, y, w, h)	Draws a rectangle
restore()	Reverts canvas to previously saved state
rotate(angle)	Rotates the canvas
save()	Save the current state
scale(x, y)	Scales the canvas by x and y factors
setTransform(a, b, c, d, e, f)	Transform the canvas based on a matrix
stroke()	Draws the current path
strokeRect(x, y, w, h)	Creates a rectangle outline
strokeText(text, x, y [, maxWidth])	Draws the text in the specified position
transform(a, b, c, d, e, f)	Edits the transform matrix of the canvas
translate(x, y)	Assigns the center point for transforms

As you can see, the canvas element's 2d context API has a rich set of drawing features. There are limits to what you can do with canvas (like 3d drawing) but the base implementation is impressive.

The basic drawing process

Once you have a reference to your canvas `context` object, you can start drawing. Drawing on the canvas always requires at least an x and y coordinate. These numeric values are based on pixels measured from the top left of the canvas, as shown in figure 3.

Figure 3 Canvas x and y coordinates are always based on 0 pixels at the top left corner.

> **TIP** The top/left measurement of pixels in the drawing context will remain the top left corner even if the containing canvas has a radiused border. No CSS margins, borders, padding, or box-model flow is honored inside the context.

The next thing to note about drawing is that the order of your drawing operations matters. For instance, consider the following code:

Listing 1 Drawing on the canvas requires an awareness of the current context settings

```
var canvas=document.getElementById("myCanvas");
var ctx=canvas.getContext("2d");
ctx.rect(20,20,50,50);
ctx.fillStyle = "rgb(0, 255, 0)";
ctx.fill();                                                #A
//
var canvas=document.getElementById("myCanvas");
var ctx=canvas.getContext("2d");
ctx.rect(20,20,50,50);
ctx.fill();                                                #B
ctx.fillStyle = "rgb(0, 255, 0)";
```

#A A green square will be drawn here
#B A black (default color) square will be drawn here because the fill style was set after the fill() method had already finished drawing the square. Future pixels drawn will be green.

In addition to assigning colors and other settings, remember that if a pixel is placed on the drawing surface on top of another pixel, it will completely obfuscate it unless it is clipped or the current context color has an opacity (alpha) value assigned. If clipped, the clipping region will determine which pixels are overwritten. If the current color has opacity, the pixel colors will be blended based on the rendered color below and the transparency of the new pixel.

The next basic operations that we want to look at are the `save()` and `restore()` methods. These are used to save the current state of the drawing context for properties related to lines, fills, text, and shadows. Listing 2 shows how the save and restore methods work.

Listing 2 Context save and restore methods manage context drawing properties

```
var canvas=document.getElementById("myCanvas");
var ctx=canvas.getContext("2d");
ctx.fillRect(10,10,40,40);             #A
ctx.save();                            #B
ctx.fillStyle = "rgb(0, 255, 0)";      #C
```

```
ctx.fillRect(20,20,40,40);
ctx.restore();                            #D
ctx.fillRect(30,30,40,40);
```

#A The first rectangle is drawn using the default fill color of black
#B The save operation saves this value into state for the canvas
#C Changing the fillStyle property will assign green as the fill color for all new pixels drawn
#D The restore operation will revert the canvas back to its original color

The result of listing 2 is displayed in figure 4. The first rectangle is black and overlaid by green then reverted back to a black rectangle.

Figure 4 Save and restore operations can be used to manage all drawing properties at once.

The next step is to create some interesting effects with a canvas. To do that, we will create a small sample page and fill in some basic functionality. This example will be strictly client-side HTML with no dependencies on ASP.NET MVC or WCF Web APIs. Using Visual Studio to build the example files will be helpful though, in giving you a basic level of intellisense. We will be using a simple MVC project for this demonstration application. The canvas application we want to write is a flying lines screensaver.

To start, create a cshtml view (without a master page) with the following markup:

```
<!DOCTYPE html>
<html>
   <title>Canvas Demo</title>
   <script type="text/javascript"
      src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3.js"></script>
   <script type="text/javascript" src="Scripts/main.js"></script>
</head>
<body>
  <canvas id="main" />
</body>
</html>
```

Now, in the Scripts folder, add a JScript file named “main.js.” At the top of this file, we will reference the latest version of jQuery and the vsdoc file that gives us the most up-to-date intellisense in Visual Studio.

```
/// <reference
   path="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3.js" />
/// <reference
   path="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3-vsdoc.js" />
```

Now we start coding. We need a document-ready event handler to get things started after all resources are loaded in the page. Listing 3 shows the outline. All of our code will go in this ready handler.

Listing 3 A beginning ready handler in the main.js file

```
$(document).ready(function () {
   var $canvas = $("#main");                              #A
   var context = $canvas[0].getContext("2d");             #A
   var w;                                                 #B
   var h;                                                 #B
   var maxVelocity = 10;                                  #C
   var points = [];                                       #C
   var radians = 0;                                       #C
   var segments = 3;
   var pointsCount = 15;

   // Image preparation placeholder

   function resize() {
      w = $canvas.width();
      h = $canvas.height();
      $canvas.attr("width", w);
      $canvas.attr("height", h);
```

```
      // Resize function body placeholder                #D
   }
   $(window).resize(resize);                             #E
   resize();                                             #E

   // Randomizer placeholder

   function frame() {                                    #F
      // Frame function body placeholder
   }
   frame();
});
```

#A First, we set up the local variables that reference the canvas element and its 2d context.
#B The height and width elements are used throughout the function for drawing on the surface.
#C Velocity, points and radians are variables used to draw lines and change directions.
#D The resize function will be used to reinitialize the local variables when the window size changes.
#E We bind the document's resize event to the resize function and then immediately call it.
#F The frame function is where all the real work of drawing will be done.

In this function, we have function stubs and placeholder comments for all the work that we will be doing with the `<canvas>` element. The `resize()` function will handle our resets and the `frame()` function will be where the real drawing work happens.

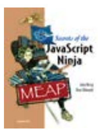
Summary

We have walked through exactly what the HTML5 canvas API is and is not and some of the basic features that you need to know in order to paint pixels on the drawing surface.

Here are some other Manning titles you might be interested in:

Quick & Easy HTML5 and CSS3
Rob Crowther

Sass and Compass in Action
Wynn Netherland, Nathan Weizenbaum, and Chris Eppstein



Secrets of the JavaScript Ninja
John Resig and Bear Bibeault

Last updated: February 20, 2012