JUMP START SERIES

sitepoint

JUMP START Html5 Canvas & SVG

By Kerry Butters

GET UP TO SPEED WITH HTML5 IN A WEEKEND

Summary of Contents

JUMP START HTML5: CANVAS & SVG

BY KERRY BUTTERS

Jump Start HTML5: Canvas & SVG

by Kerry Butters

Product Manager: Simon Mackie
English Editor: Paul Fitzpatrick
Technical Editor: Craig Buckler
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com
Printed and bound in the United States of America

About Kerry Butters

Kerry Butters[1] is a technology writer from the UK. With a background in technology and publishing, Kerry writes across a range of techy subjects including web design and corporate tech. Kerry also heads up markITwrite digital content agency[2], loves to play around with anything tech related and is an all-round geek.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

About Jump Start

Jump Start books provide you with a rapid and practical introduction to web development languages and technologies. Typically around 150 pages in length, they can be read in a weekend, giving you a solid grounding in the topic and the confidence to experiment on your own.

[1] https://plus.google.com/u/0/+KerryButters?rel=author
[2] http://markitwrite.com

To my husband Martin,
(affectionately known to me as my Martian) for being my biggest fan and blowing my trumpet when I won't (which is most of the time!). For being supportive, for helping me to realise my dreams and for being the man I dreamed about for many years before we met. Most of all for being my best friend and my soul mate.

Table of Contents

Preface

HTML5 is an updated version of HTML which introduces new elements to further enrich web pages, and allow designers to create animations and graphical elements in new ways.

SVG and canvas are two such new elements that allow designers to create rich graphics inside the browser with code. In this book, we'll be looking at how to create HTML5 canvas and SVG graphics. Both can make use of scripts to create interactive and animated effects, and while SVG has been around for some time, it can be used alongside HTML5 to even better effect.

HTML5 canvas is a JavaScript API, which can be used to perform complex drawing operations using programming. This is achieved by using the `canvas` element in an HTML document—a blank area inside which you can draw.

This can then be done using a 2D or 3D *(*WebGL*)* drawing context: the former is readily available across all modern browsers, while the latter is more recent and, as such, not yet fully supported. In this book, we will be looking at 2D implementation.

The powerful 2D API enables quick drawing operations. There is no file format, and you can only draw using script. There are no DOM *(*Document Object Module*)* nodes for the shapes you draw—it all happens on the surface, as pixels. This means you can concentrate on drawing without performance penalties as the complexity of the image grows.

Who Should Read This Book

This book assumes that you have a basic working knowledge of HTML and JavaScript, and is aimed at those that are just starting out with HTML5. This means that the book won't go into complex details when it comes to working with SVG and canvas, but will give beginners enough of a grounding to start experimenting.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a ⋮ will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-
➥testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.sitepoint.com/store/jump-start-html5-canvas-svg/

The book's website, containing links, updates, resources, and more.

https://github.com/spbooks/jshtml-canvas-svg1

The downloadable code archive for this book.

http://www.sitepoint.com/forums/

SitePoint's forums, for help on any tricky web problems.

books@sitepoint.com

Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Tools You'll need

You'll need a text editor of some description, such as notepad, and a browser. You may need to use more than one browser depending on support and it's not recommended that you attempt to use a word processing program such as Word. This is because these types of programs add formatting to the document which you can't see and which will mean your code won't work.

Other editors that you may like to try include Sublime Text 2[1]. The beta version of this is available and can be used with:

- OS X (OS X 10.6 or later is required)
- Windows (also available as a portable version)
- Windows 64-bit (also available as a portable version_
- Linux 32-bit
- Linux 64-bit

If you have a Mac and don't mind paying, then MacFlux[2] is worth looking at. A simple editor that's open source is Notepad++[3].

Browsers

Not all browsers support all HTML5 elements, so make sure you download the latest versions of those you will need. Currently, browsers supporting inline SVG and canvas are:

[1] http://www.sublimetext.com/2
[2] http://www.macwareinc.com/products/mac-web-design-software/macflux.html
[3] http://notepad-plus-plus.org

- IE9+
- Firefox
- Chrome
- Opera
- Safari

Enabling Inspection Tools in Chrome

Canvas can be difficult to debug, thanks to its use of calls. With this in mind, it's a good idea to enable inspection tools in Chrome so that you can capture instructions and go through them one at a time.

At the time of writing, the tools are experimental and must be enabled:

1. Open a new tab and type **chrome//:flags** into the address bar.
2. Choose **Enable experimental canvas features** (this feature is available for Mac, Windows, Linux, Chrome OS and Android).
3. Towards the bottom of the page, choose **Relaunch Chrome.**

This is an experimental feature!

Be aware this feature is experimental and may not be stable.

Now, open the Developer Tools and enable **canvas inspection** by clicking the gear icon in the lower right-hand corner to open the settings.

Tick the **canvas inspection** box, then return to Developer Tools. A new canvas profiler tool has appeared that'll enable you to monitor canvas rendering.

Do You Want to Keep Learning?

You can now get unlimited access to courses and all SitePoint books at Learnable[4] for one low price. Enroll now and start learning today! Join Learnable and you'll stay ahead of the newest technology trends: http://www.learnable.com/.

[4] https://learnable.com/

Chapter 1

An Introduction to Canvas

As the Web has evolved and matured, so too has the language used to display web pages effectively. The previous version of HTML, v4.01 has many elements that are now obsolete.

Modern internet users are sophisticated and demanding, and this means they expect web pages to appear in a certain way and to load quickly. HTML5 seeks to address what was lacking in earlier versions of HTML to better handle graphics and meet those expectations.

What Can Canvas Be Used For?

Canvas can be used to draw shapes, such as rectangles, squares and circles, or to embed images or videos in an HTML5 document. You can use multiple instances of it in one document, or just one, depending on your needs.

The basic `canvas` element looks like this:

```
<canvas id="myCanvas" width="300" height="150"></canvas>
```

At this point, it's worth noting that the HTML5 `canvas` element is the **DOM** (Document Object Model) node that's embedded in the page. The context is then created, which is an object that you use in order to render graphics within the container. If you create multiple canvases, you'll need to create `canvas` elements for each context and name them appropriately so that the browser understands to which object you're referring.

The `canvas` element is superficially similar to the `img` element. Both have a height and width, and display in a rectangular block on the page. However, `img` normally loads a pre-prepared graphic, such as a photograph. `canvas` is a programmable image; you use JavaScript drawing methods to directly manipulate the pixels. The technology is fast and permits you to create sophisticated animations and games. Overall, canvas is often compared to technologies such as Flash and Silverlight.

Check out this animated graphic[1] for a good example of what you can do using canvas. You should also check out Canvas Demos[2] for some great working examples, including games and apps that have been created using `canvas`.

Before We Get Started

Some points to think about before we begin playing around with canvas:

- It's usually best to give each canvas a unique `id` attribute so your scripts can reference it directly. No other elements on that page should use the same ID.
- When no styling is applied, the container or the canvas element will be transparent, with no border, so it'll appear as a see-through, rectangular box. The default width is 300 pixels and the default height is 150 pixels.
- In an ideal world, everyone would use the latest browsers. But it isn't, and they don't. This means it's usually necessary to tell the browser how to behave when `canvas` is not supported.

[1] http://raksy.dyndns.org/torus.html
[2] http://www.canvasdemos.com/

- If you're used to working with the `img` element then you'll know that it doesn't require the closing `</img>` tag. `canvas`, on the other hand, does require closing, so you should **always** include `</canvas>` at the end of the container code.

It's also worth mentioning at this point that canvas uses coordinates, paths, and gradients. These can look a little daunting when you first come across them, and often have would-be developers running for the hills screaming **"MATH!**" But there's little need to worry—you'll soon get the hang of it.

Canvas Looks Complex, Why Not Use Flash?

I've come across a lot of questions posted on various forums that all say much the same thing: "Canvas looks far too complicated for creating animations—why shouldn't I just stick to using Flash since I know it already?"

Well, it's true that Flash enables you to create animations using professional tools, which don't necessarily require coding skills. However, canvas is superior to Flash in other ways, including:

- good compatibility on desktop and mobile devices
- it requires no plugins or dependencies outside of the browser
- it's free to use
- once you've learned to use it, canvas can create impressive animations using minimal code

What About WebGL?

WebGL enables 3D graphics to be rendered within the browser window, and for those graphics to be manipulated using JavaScript. If you're interested in using `canvas` then it's likely you'll be interested in investigating WebGL, too, at some point; the idea that you can create 3D graphics without plugins is a very attractive one. It's already supported[3] by most browsers, too. That said, we won't be covering WebGL in this book.

[3] http://caniuse.com/webgl

Chapter 2

Canvas Basics

First of all, let's look at how to create a canvas document. As noted earlier in this book, the `canvas` element itself looks like this:

```
<canvas id="MyCanvas" width="300" height="150"></canvas>
```

HTML5 Canvas Template

Let's start with a basic template that we can use to begin working with. We'll add the `canvas` element to the page and a small self-executing script that gets the context:

```
<html>
  <head>
  <title>Getting started with Canvas</title>
  <style type="text/css">
    canvas { border: 1px solid black; }
  </style>
  </head>
  <body>
    <canvas id="MyCanvas" width="300" height="150"></canvas>
    <script>
```

```
      (function() {
        var canvas = document.getElementById('MyCanvas');
        if (canvas.getContext){
          var ctx = canvas.getContext('2d');
        }
      }
    </script>
  </body>
</html>
```

There are two essential attributes that `canvas` has: `width` and `height`. If the attributes are not specified, then the default of 300px wide by 150px high will be used.

The `getElementById` function simply finds the `canvas` element in the DOM, based on the ID we've assigned the canvas, which, in this case, is `MyCanvas`. The line `var ctx = canvas.getContext('2d');` is the 2D context method, which returns an object that exposes the API for the drawing methods we'll use.

Canvas Element Styling

You can style the `canvas` element just as you would any other image, using borders, colors, backgrounds, and so on. However, the styling will not affect the actual drawing on the canvas. A canvas without any styling will simply appear as a transparent area.

Drawing a Simple Shape Onto the Canvas

Let's have a look at how we can draw some simple shapes. All drawing must be done in our JavaScript function, after the line `var ctx = canvas.getContext('2d');`. Let's draw a rectangle:

```
ctx.fillStyle="#0000FF";
ctx.fillRect(0,0,300,150);
```

This draws a blue rectangle that fills the canvas area. The `fillRect` method requires the top-left x and y coordinates of the rectangle to be drawn, followed by its width and height. The code above creates a 300x150px rectangle that is positioned at with its top-left corner at coordinate 0,0 and filled with the current `fillStyle`, which

in this case is a solid blue (`#0000FF`)—see Figure 2.1. `fillStyle` can be a color, gradient, or pattern.

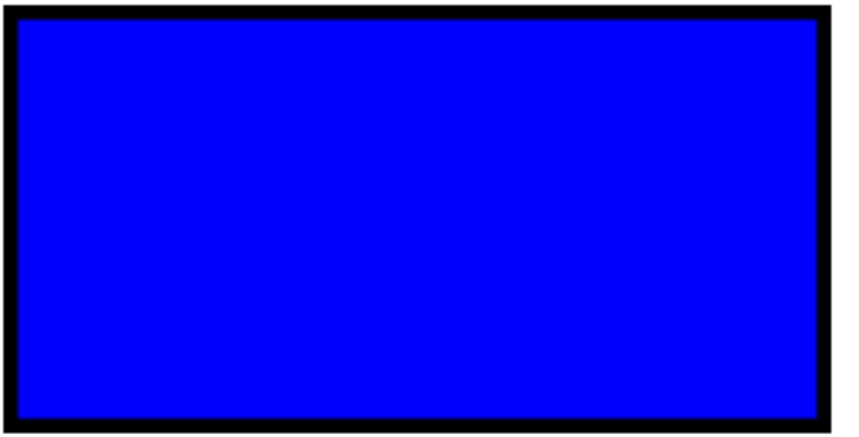

Figure 2.1. Our blue rectangle

The canvas 2D API provides methods for drawing several basic shapes, including:

- Rectangles
- Arcs
- Paths
- Text
- Images

W specified the size of the canvas as being 300x150px. If we reduce the rectangle's size, then you'll see that you have a rectangle within the canvas. For example, if we modify the code as follows:

```
ctx.fillStyle="#0000FF";
ctx.fillRect(0,0,150,75);
```

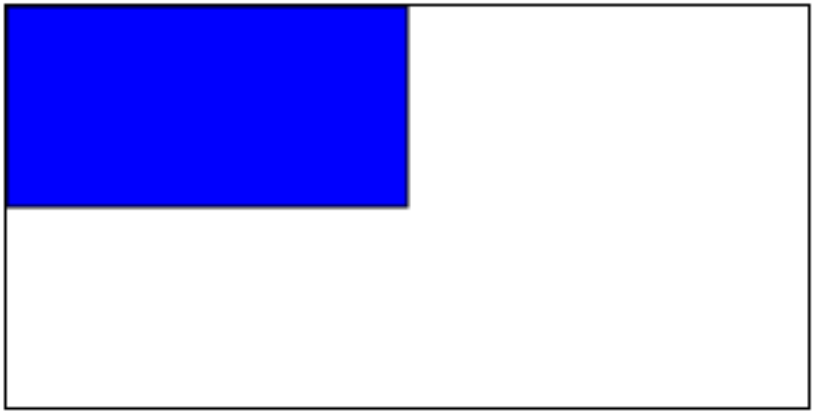

Figure 2.2. A resized rectangle

The canvas itself remains as a transparent box, as you can see. As we've included a black border around the canvas you can see its area. Without the border you'd see nothing but the blue box, but the canvas would still be there.

Remember that the canvas `width` and `height` attributes determine the dimensions of the pixel coordinate system. If you use CSS to specify a different width or height, the canvas image will be squashed or stretched accordingly. For example, if we apply a width of 600px and height of 300px to the canvas in CSS, each canvas 'pixel' would be twice the size of a normal pixel.

Canvas Coordinates and Paths

Canvas uses a two-dimensional coordinates grid. The top-left of the canvas has a coordinate of (0,0). The bottom-right will have a positive x and y coordinate according to the size of the element. In the example above, we used a canvas size of 300x150px, so the bottom-right pixel is at (299,149), because coordinates are zero-based.

Lines are drawn on the canvas using **paths**. You create paths by using the `moveTo()` and `lineTo()` methods, in conjunction with one of the **ink** methods, `stroke()` or

`fill()`. `moveTo()` and `lineTo()` define the start and end points of the line to be drawn. `stroke()` draws a shape by "stroking" its outline, while `fill()` draws a solid shape by filling in the content area of a path.

So, to draw a simple white line through the rectangle we created above:

```
ctx.strokeStyle = "#FFFFFF";
ctx.beginPath();
ctx.moveTo(0,0);
ctx.lineTo(300,150);
ctx.stroke();
```

The `beginPath` method erases any outstanding path drawing operations in preparation for a new path. The `stroke()` method physically draws the path you've defined. In this case, it's a single line.

Drawing Circles

Now let's look at how we'd draw a circle. The simplest way to do this is to use `arc` to effectively create a circular path, which can then be used with ink methods, such as `stroke()` or `fill()`, like this:

```
ctx.beginPath();
ctx.arc(95,50,40,0,2*Math.PI);
ctx.stroke();
```

Here we're using the `arc` method (which can still be part of a path). The parameters specify the x and y coordinates of the arc's center, the arc's radius, the start angle, and end angle (in radians[1]). Therefore, we've created an arc centered on coordinates 95,50, with a radius of 40 pixels, with a start angle of 0 and an end angle of 2*PI (or 360 degrees). We've used the `stroke()` method to draw the path, which gives us a circle, as shown in Figure 2.3:

[1] http://en.wikipedia.org/wiki/Radian

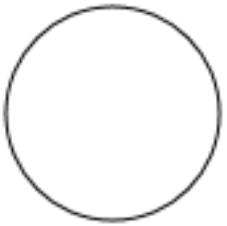

Figure 2.3. Drawing a circle

If you added the `fill()` method, you would end up with a circle that is filled in with the specified color, black by default, as shown in Figure 2.4.

Figure 2.4. A filled circle

Drawing Text

You can draw text onto a canvas using these methods:

- `font`—defines the font properties
- `fillText(text,x,y)`–draws text on the canvas
- `strokeText(text,x,y)`–draws the outline of text on the canvas

Here's an example:

```
ctx.font = "25px Arial";
ctx.fillText("HTML5 Canvas Rocks!",10,50);
```

This will draw the words "HTML5 Canvas Rocks!" using block text, using the font Ariel at 25 pixel size, at the coordinates (10,50), as shown in Figure 2.5.

Figure 2.5. Writing text to the canvas

If you were to replace `fillText()` now with `strokeText()`, you would instead have outlined text, as shown in Figure 2.6.

Figure 2.6. Stroked text

You can also add text effects and colors:

```
ctx.fillStyle= '#0000FF';
ctx.font="Italic 25px Arial";
ctx.fillText("HTML5 Canvas Rocks!",10,50);
```

This will italicize the text and make it blue, as shown in Figure 2.7.

Figure 2.7. Blue italic text

Drawing a Triangle

Let's draw a triangle. As there's no built-in triangle shape for us to draw with, we'll need to construct it using paths. To create a basic triangle we can use the following code:

```
ctx.beginPath();
ctx.moveTo(25,25);
ctx.lineTo(105,25);
ctx.lineTo(25,105);
ctx.fill();
```

This should appear as shown in Figure 2.8:

Figure 2.8. A filled triangle

To create a stroked triangle:

```
ctx.beginPath();
ctx.moveTo(125,125);
ctx.lineTo(125,45);
```

```
ctx.lineTo(45,125);
ctx.closePath();
ctx.stroke();
```

Note the `closePath()` method; this closes the path by drawing a straight line from the current point to the initial point. This will appear as shown in Figure 2.9:

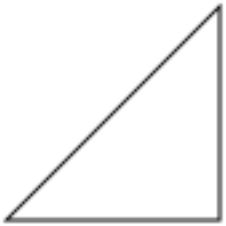

Figure 2.9. A stroked triangle

So that's how to draw basic shapes in HTML5 using the `canvas` element and JavaScript. Now that you've learned the basics, go and practice with different styles, fonts, and shapes to get further accustomed to using the JavaScript code.

Canvas Sizing

Depending on what you're developing using canvas, you can resize to fit the device being used—if you're needing to fill the screen for say, a game. This can be achieved in a number of ways:

- coding in JavaScript
- using CSS
- CSS transforms using JavaScript

Scaling with JavaScript

```
var canvas = document.getElementById('canvas');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
```

This will create a canvas which extends to the current viewport size, but you will need to ensure the element has no margin or is affected by other items on the page. In addition, changing the browser window size will not modify the canvas dimensions.

Scaling with CSS

```
#canvas {
  position: relative;
  left: 0;
  right: 0;
  top: 0;
  bottom: 0;
  margin: auto;
  width: 100%;
  height: 100%;
}
```

This changes the size of the canvas box but not the pixel dimensions; the coordinate system remains the same.

CSS Transforms Using JavaScript

```
var scaleX = canvas.width / window.innerWidth;
var scaleY = canvas.height / window.innerHeight;
var scaleToFit = Math.min(scaleX, scaleY);
canvas.style.transformOrigin = "0 0";
canvas.style.transform = "scale("+scaleToFit+")";
```

Again, this changes the size of the canvas box, but not the pixel dimensions.

Chapter 3

Handling Non-supporting Browsers

In this short chapter, we'll look at creating code that tells the browser how to behave if it doesn't support canvas rendering.

Create Alternative Content

The best way to handle the possibility that a user's browser doesn't support canvas is to place alternative content within the `<canvas>` tag. This does away with confusion for the end user if they can't see what's supposed to be displayed.

You can use an `img` tag, explanatory text, or any other HTML you think necessary for this alternative content. For example:

```
<canvas id="MyCanvas" width="150" height="300">
  <img src="http://yoursite.com/locationofimage.jpg"
  ➥alt="Browser Not Supported" />
</canvas>
```

If the browser supports `canvas`, the `img` tag and any other content between the `<canvas>` and `</canvas>` tags are ignored and won't appear in the document.

How useful you want to make fallback content is up to you; you can offer a download link to the latest version of the user's browser, or you can add a framework that'll allow you to show the content using a different technology, such as SVG or Flash.

You can also use the `getContext` method to check for canvas support in JavaScript, e.g.

```
function supports_canvas() {
  return !!document.createElement('canvas').getContext;
}
```

Alternatively, you can use the `getContext` method on an existing element:

```
var canvas = document.createElement("MyCanvas");
if (!canvas.getContext || !canvas.getContext("2d")) {
    alert("Sorry - canvas is not supported.");
}
else {
    // start drawing
    var ctx = canvas.getContext('2d');
}
```

Chapter 4

Gradients

With HTML5 `canvas`, you're not limited to block colors, but can use gradients to fill shapes such as rectangles and circles. There are two different types of gradient you can use:

```
// create a linear gradient
createLinearGradient(x,y,x1,y1)
```

Figure 4.1. An example of a linear gradient

```
// create a radial gradient
createRadialGradient(x,y,r,x1,y1,r1)
```

Figure 4.2. An example of a radial gradient

Let's start by creating a linear gradient (the canvas context, `ctx`, has already been defined):

```
// create linear gradient
var grd = ctx.createLinearGradient(0,0,400,0);
grd.addColorStop(0,"blue");
grd.addColorStop(1,"yellow");

// fill with gradient
ctx.fillStyle = grd;
ctx.fillRect(40,20,300,160);
```

The result is shown in Figure 4.3.

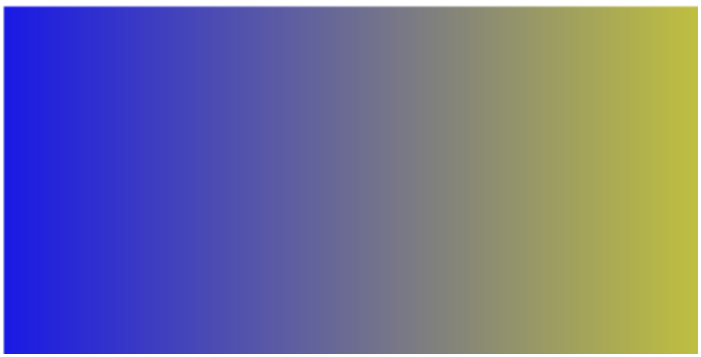

Figure 4.3. Our linear blue-yellow gradient

The first line `var grd = ctx.createLinearGradient(0,0,400,0);` creates a `CanvasGradient` object which defines a gradient between two sets of coordinates

(x1,y1,x2,y2). These determine the size and direction of the gradient. In our example, we use (0,0) to (400,0) which results in a horizontal gradient which is 400 pixels in width. If our box was wider, the last color would extend accordingly.

If we required a 300px vertical gradient, we would use:

```
var grd = ctx.createLinearGradient(0,0,0,300);
```

A 45-degree diagonal gradient in a 100x100px space would be defined as:

```
var grd = ctx.createLinearGradient(0,0,100,100);
```

We can now set the color values at certain **color stop** points within that gradient using the `addColorStop` method. It is passed two values:

- a stop value between `0` (the left-most end of the linear gradient) and `1` (the right-most end of the gradient)
- a color

We have used `"blue"` at stop value 0—or coordinate (0,0)—and `"yellow"` at stop value 1—or coordinate (400,0). The browser uses the values to define a smooth color gradient from blue to yellow.

You can add any number of gradient stops. For example, a `"red"` color stop at stop value 0.5 would create a smooth gradient from blue, to red at the mid-point (200px), to yellow at the end.

Radial Gradients

Now let's look at a radial gradient:

```
// create radial gradient
var grd = ctx.createRadialGradient(150,100,10,180,120,200);
grd.addColorStop(0,"blue");
grd.addColorStop(1,"yellow");
```

```
// fill with gradient
ctx.fillStyle = grd;
ctx.fillRect(0,0,300,150);
```

The `createRadialGradient` parameters are:

- the x and y coordinates of the starting circle
- the radius of the starting circle
- the x and y coordinates of the ending circle
- the radius of the ending circle

Our code produces the output seen in Figure 4.4. You can experiment with different values to create interesting effects.

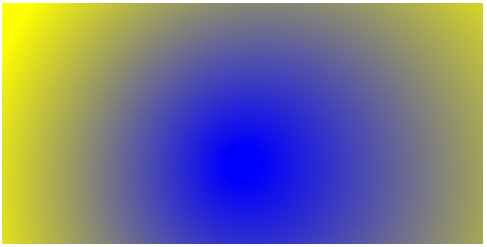

Figure 4.4. A radial gradient

Playing with the Color Stops

Let's modify the linear gradient code we created above and go a little crazy with adding some color stops:

```
var grd = ctx.createLinearGradient(35,25,25,190,105,50);
grd.addColorStop(0,"red");
grd.addColorStop(0.25,"blue");
grd.addColorStop(0.3,"yellow");
grd.addColorStop(0.35,"magenta");
grd.addColorStop(0.4,"green");
grd.addColorStop(0.45,"pink");
grd.addColorStop(0.5,"gray");
grd.addColorStop(1,"white");
```

```
// Fill with gradient
ctx.fillStyle=grd;
ctx.fillRect(20,20,400,400);
```

The results are shown in Figure 4.5.

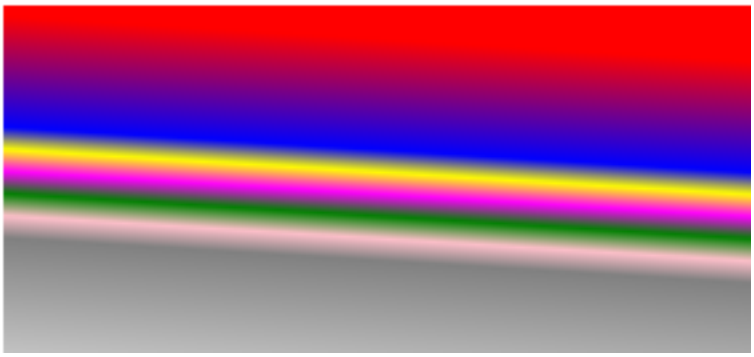

Figure 4.5. A crazy linear gradient

To create a radial gradient using the same colors you could modify one line as follows:

```
var grd=ctx.createRadialGradient(35,25,25,190,105,50);
```

which would display as shown in Figure 4.6:

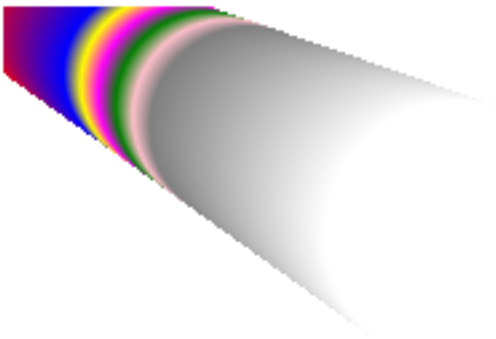

Figure 4.6. A crazy radial fill

Pretty groovy effect on that radial gradient, don't you think?

Chapter 5

Images and Videos

You can use bitmap images and video with canvas. In this chapter, we'll look at how you can copy images and videos onto your canvas.

Images

You can copy a pre-defined bitmap image to your canvas using the `drawImage()` method. The same method can also be used to draw part of an image or alter its size. You can position the image on the `canvas` much in the same way as you would draw a line:

```
var c = document.getElementById("MyCanvas");
var ctx=c.getContext("2d");
var img = document.getElementById("yourimage");
ctx.drawImage(img,10,10);
```

As you can see here, the image (which is on our page with the ID "`yourimage`") is positioned at the x,y coordinates passed in the method: `ctx.drawImage(img,x,y)`.

You can specify the size of the image by adding width and height, like this:

```
ctx.drawImage(img,x,y,width,height);
```

To crop the image and position the cropped part only:

```
ctx.drawImage(img,sx,sy,swidth,sheight,x,y,width,height);
```

In the code above, the `sx` and `sy` coordinates dictate where to begin cropping the image, and `swidth` and `sheight` dictate the dimensions of the image.

Using the `image()` Object

The above example assumes that the image is on the page already. You may find it preferable to load the image dynamically using JavaScript.

```
// canvas set-up
var canvas = document.getElementById('MyCanvas');
var ctx = canvas.getContext('2d');

// load image from a URL
var img = new Image();
img.src = "http://mydomain.com/image1.png";

// is image loaded?
if (img.complete) addToCanvas();
else img.onload = addToCanvas;

// add image to canvas
function addToCanvas() {
    ctx.drawImage(img,10,10);
}
```

Video

The content of an HTML5 `video` element can also be copied to a canvas. You may want to do this so you can overlay additional text or apply processing effects.

```
var video = document.createElement("video");
video.src = "yourvideo.mp4";
video.controls = true;
```

In order to then draw the video to `canvas`, you'll need to add a handler for the video's `onplay` event, which copies the current video frame.

```
var canvas = document.getElementById('MyCanvas')
var ctx = canvas.getContext('2d');

// set canvas dimensions to same as video
video.onplay = function() {
    canvas.width = video.videoWidth;
    canvas.height = video.videoHeight;
    draw();
};

// copy frame to canvas
function draw() {
    if(video.paused || video.ended) return false;
    ctx.drawImage(video, 0, 0);
    setTimeout(draw, 20);
}

// start video playback
video.play();
```

Chapter 6

An Introduction to SVG

SVG stands for **Scalable Vector Graphics**. It allows you to create graphics using the XML markup language. SVG's been around for quite some time and is supported by the majority of browsers. Unlike Canvas, it's not intended for pixel manipulation. It allows you to create scalable graphics and, as it's **resolution independent**, it's ideal for use on projects that are likely to be used on a variety of screen resolutions and sizes. For example, SVG is ideal for sites using Responsive Web Design (RWD).

In fact, the use of SVG in RWD is so obvious, you have to wonder why some websites are redesigned using traditional images. SVG also displays perfectly on retina and other high-resolution screens. As resolutions get better, it's likely they'll be more widely used.

SVG uses an accessible DOM node-based API and is perfect for those with a good understanding of HTML, CSS, and some JavaScript. You can style it using CSS and make it interactive with JavaScript, and for those that aren't overly familiar with JavaScript, there are plenty of libraries around to help.

As with any web technology, SVG is ever changing but many of its features are available for animations, transforms, gradients, filter effects, and much more. It works in all modern browsers—you can check compatibility at caniuse.com[1].

Why Use SVG Instead of JPEG, PNG, or GIF?

There are two types of graphics that can be used in computing: **bitmap** and **vector**. Bitmaps, such as JPEG, PNG and GIF, are also known as **raster graphics** and are composed of individual pixels with differing colors. Vector graphics like SVG, on the other hand, define paths and points; they can be resized and retain their quality. This makes them ideal for web uses such as:

- logos
- banners
- signage
- illustrations
- line art

SVG images have a few inherent advantages over bitmap images:

- Since SVG images are comprised of text, they are often more accessible and search engine-friendly than bitmap images.
- Vectors can also be placed over other objects and made translucent, so the object below remains visible.
- Graphics created using SVG can be edited with relative ease, and SVG can be used in conjunction with CSS in order to style the output. This isn't something that's currently achievable with traditional bitmap images.
- SVG images are normally smaller in terms of file size than bitma

However, while they do have many advantages, like many things in life, vector images are not a perfect solution for every application. For example, it's unlikely that you'd be able to produce realistic-looking photos with vectors.

[1] http://caniuse.com/svg

You can embed SVG in standard HTML documents, and SVG can be created using any text editor. However, you may prefer to use Adobe Illustrator or InkScape[2] (an open source vector graphics editor) to create your SVG images.

Now that you know what SVG is all about, let's get down to the good stuff: learning how to use it.

Getting Started

To get started, you can just use a bare bones HTML5 page and drop inline-SVG code right into it. Let's start with an SVG image of a red circle:

```
<!DOCTYPE HTML>
<html>
<body>
  <h1>A red circle:</h1>

  <!-- inline SVG -->
  <svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
    <circle id="redcircle" cx="100" cy="100" r="100" fill="red" />
  </svg>

</body>
</html>
```

Save the file and open it in your browser and you should see a page with a red circle which is titled "A red circle:".

The SVG section is delimited by the `svg` tag, which defines dimensions of 200x200px for the image on the page.

Try altering some of the code yourself. The `circle` element specifies the shape that we want to draw with various attributes. The `cx` and `cy` attributes define the circle's center in relation to the drawing area; the `r` attribute gives the circle's radius. This means that the diameter (width) of the circle will appear as twice the value you've set as the radius.

You can also add a border around the circle

[2] http://inkscape.org/download/?lang=en

```
<circle id="redcircle" cx="100" cy="100" r="100"
➥stroke="black" stroke-width="1" fill="red"/>
```

Other Shapes

As well as a circle, it's a simple matter to create other shapes by appending appropriate tags within the `svg` block:

- **line**—Creates a simple line

```
<line x1="25" y1="150" x2="300" y2="150"
➥stroke="#F00" stroke-width="5" />
```

Figure 6.1. A line

- **polyline**—Defines shapes built from multiple line definitions

```
<polyline points="0,40 40,40 40,80 80,80 80,120 120,120 120,160"
➥stroke="#F00" stroke-width="5" fill="#FFF" />
```

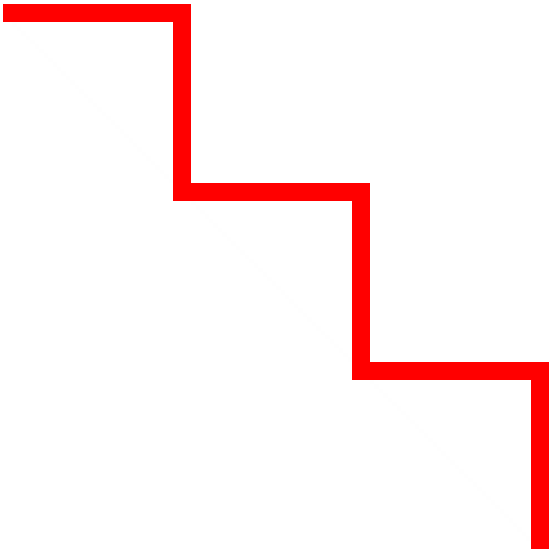

Figure 6.2. A polyline

- **rect**—Creates a rectangle

```
<rect width="300" height="100" fill="#F00" />
```

Figure 6.3. A rectangle

- **ellipse**—Creates an ellipse

```
<ellipse cx="300" cy="80" rx="100" ry="50" fill="#F00"/>
```

Figure 6.4. Ellipse

- **polygon**—Creates a polygon

```
<polygon points="200,10 250,190 160,210"
➥stroke="#000" stroke-width="1" fill="#F00" />
```

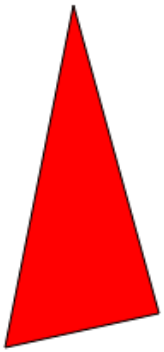

Figure 6.5. A (polygon) triangle

Polygons define a series of x and y co-ordinates in the `points` attribute. This allows you to create complex shapes with any number of sides.

```
<polygon points="100,10 40,180 190,60 10,60 160,180 100,10"
➥stroke="#000" stroke-width="1" fill="pink" />
```

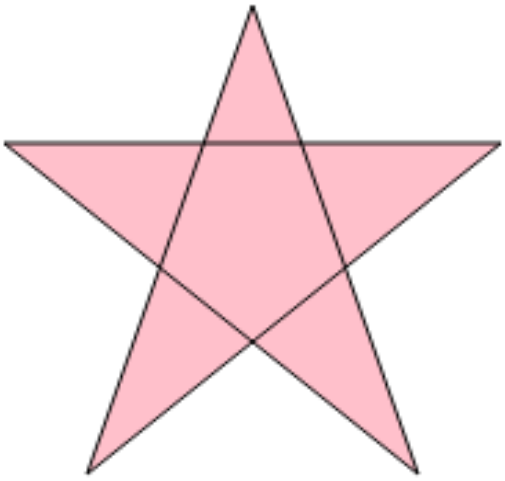

Figure 6.6. A star

- **path**—Allows for the definition of arbitrary paths

The `path` element allows you to created drawings using special commands. These can be upper or lowercase, which apply absolute and relative positioning accordingly. It looks complex and there are many options so please refer to this SitePoint tutorial[3] for more information.

All the above shapes can be made using `paths`. The code below creates a segmented circle using paths and, as you can see in Figure 6.7, this is perfect for creating pie charts and similar graphics.

```
<path d="M300,200 h-150 a150,150 0 1,0 150,-150 z"
➥fill="pink" stroke="red" stroke-width="3"/>
<path d="M275,175 v-150 a150,150 0 0,0 -150,150 z"
➥fill="purple" stroke="red" stroke-width="3"/>
```

[3] http://www.sitepoint.com/svg-path-element/

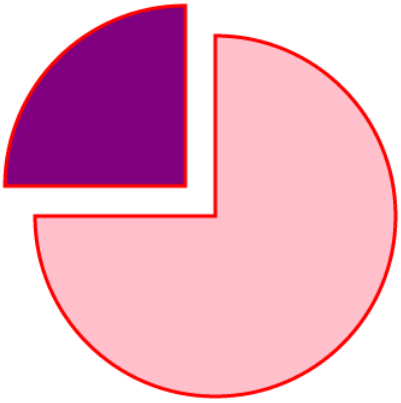

Figure 6.7. Paths

Gradients and Patterns

As with canvas, SVG enables you to paint or stroke shapes using gradients and patterns. This is achieved by creating special gradient tags such as `linearGradient` and `radialGradient` within a `defs` section of the SVG. Other elements can then refer to these by name and reuse them on any shape.

To add a linear gradient to the circle within the `svg`:

```
<!-- define gradient -->
<defs>
  <linearGradient id="MyGradient">
    <stop offset="10%" stop-color="yellow" />
    <stop offset="90%" stop-color="blue" />
  </linearGradient>
</defs>

<!-- use gradient in a circle -->
<circle cx="100" cy="100" r="100" fill="url(#MyGradient)" />
```

Now open it in your browser, you'll see that you now have a blue and yellow circle. To make it a radial gradient, it's then just a case of using the `radialGradient` tag:

```
<!-- define gradient -->
<defs>
  <radialGradient id="MyGradient">
    <stop offset="10%" stop-color="yellow" />
    <stop offset="90%" stop-color="blue" />
  </radialGradient>
</defs>

<!-- use gradient in a circle -->
<circle cx="100" cy="100" r="100" fill="url(#MyGradient)" />
```

Patterns

You can also create repeating designs within a `pattern` tag. This defines a series of SVG elements, which can be used to fill an area:

```
<svg>
<defs>
<pattern id="mypattern" x="0" y="0" width="150" height="100"
➥patternUnits="userSpaceOnUse">
  <circle cx="50" cy="50" r="10" fill="red" stroke="black"
/>
  <rect x="100" y="0" width="50" height="50" fill="cyan"
stroke="red" />
</pattern>
</defs>
  <ellipse fill="url(#mypattern)" stroke="black"
 stroke-width="1" cx="200" cy="200" rx="200" ry="200" />
</svg>

<!-- define pattern -->
<defs>
  <pattern id="mypattern" patternUnits="userSpaceOnUse"
  ➥x="0" y="0" width="50" height="50">
    <circle cx="25" cy="25" r="25" fill="red" stroke="black" />
    <rect x="25" y="25" width="25" height="25"
  ➥fill="cyan" stroke="red" />
  </pattern>
</defs>
```

```
<!-- use pattern in a circle -->
<circle cx="100" cy="100" r="100" fill="url(#MyPattern)"
➥stroke-width="1" stroke="black" />
```

Save the code above and open it in a browser to see the results, shown in Figure 6.8. Now you can experiment to see what other patterns you can make, using various shapes and color gradients.

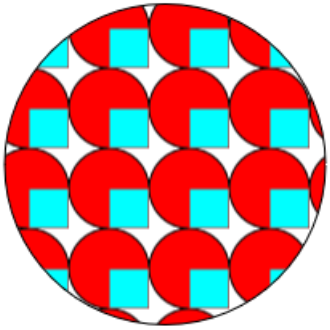

Figure 6.8. Pattern fill

Chapter 7

Using SVG

In modern browsers, SVG can be used anywhere where you would normally use JPGs, GIFs or PNGs. Add to this the ability to add colors and gradients, plus the fact that you get no loss of quality when scaled, and it's something to get excited about for the majority of designers.

Inserting SVG Images on Your Pages

There are several ways to add SVG to your page:

- The `object` tag
- The `embed` tag
- Within an iFrame
- Using a CSS background
- Inline SVG embedded into your HTML5 page
- Using an `img` tag

Use CSS for Repeating Backgrounds

Only CSS can be used for repeating backgrounds. The other methods will just show a single image.

Which Method Should You Use?

That will depend on the project at hand but, generally, `object` or `embed` should be used if you intend to use DOM scripting to manipulate the image in JavaScript. An `iframe` can be used for the same purpose although the code becomes a little more cumbersome. Alternatively, an inline SVG may be appropriate if you need scripting but the image is used on a single page on your website.

If you just need a static SVG, use the `img` tag or a CSS background. These do not permit the SVG to be modified on the client.

SVG MIME type

Your web server should return SVG images with the MIME type `image/svg+xml`. Most servers should do this automatically, but double-check if images do not display correctly.

Let's have a look at how you'd go about it using the `object` method using an SVG file we created using an application such as Illustrator or Inkscape:

```
<object type="image/svg+xml"
➥width="400" height="400" data="image.svg">
</object>
```

An `embed` tag is similar, but `embed` only became standard in HTML5. It's possible some older browsers could ignore it, but most implement the tag:

```
<embed type="image/svg+xml"
➥width="400" height="400" src="image.svg">
</embed>
```

An `iframe` loads the SVG much like any other web page:

```
<iframe src="image.svg">
</iframe>
```

We've already used inline SVG images added directly to the HTML page. This does not incur additional HTTP requests but will only be practical for very small images or those you don't intend using elsewhere:

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <circle id="redcircle" cx="100" cy="100" r="100" fill="red" />
</svg>
```

An `img` tag is identical to any you've used before:

```
<img src="image.svg" />
```

Finally, the CSS `background-image` property can reference an SVG:

```
#myelement {
  background-image: url(image.svg);
}
```

SVG Tools and Libraries

There are many libraries, snippets and useful tools for creating and manipulating SVG images.

Snap SVG[1] from Adobe is a free, open-source tool for generating interactive SVG.

Another great resource, Bonsai[2], provides a JavaScript library with snippets and demonstrations to help you alter SVG images using client-side code.

[1] http://snapsvg.io/
[2] http://bonsaijs.org/

Chapter 8

Bézier Curves

Bézier curves are used extensively in graphics software and are sometimes described as a **polynomial** expression[1], which is basically used to describe a curve. Sometimes, Bézier curves are referred to simply as curves, which can be slightly confusing if you're not familiar with all of the common (or less common) terms when it comes to design.

A Bézier curve is constructed by control points, as shown in Figure 8.1. A quadratic Bézier curve has one control point, whilst a cubic has two.

[1] http://en.wikipedia.org/wiki/Polynomial

Quadratic Bézier Curves

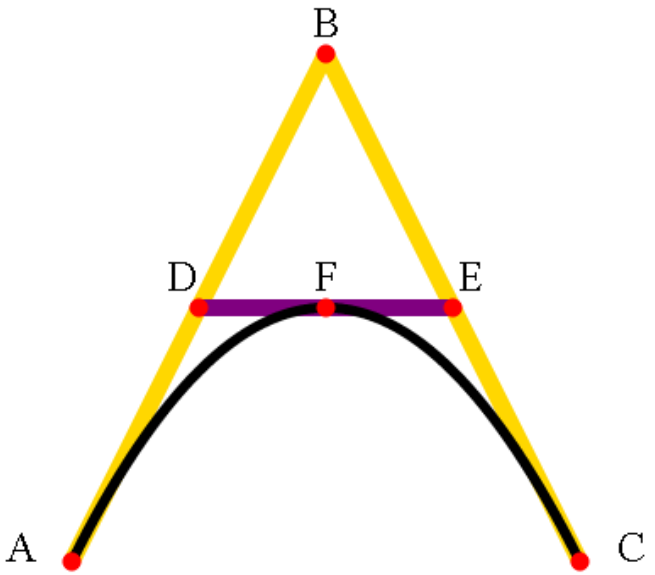

Figure 8.1. A quadratic Bézier curve

Now let's look at creating this whole image in SVG. Save the following code using your text editor and then open it up in your browser and you should see an **A** shape with a curved line reaching to the line that crosses the shape:

```
<!DOCTYPE html>
<html>
<body>
<svg width="500" height="500"
➥xmlns="http://www.w3.org/2000/svg" version="1.1">

  <!-- lines -->
  <path id="lineAB" d="M 100 350 l 150 -300" stroke="gold"
  ➥stroke-width="10" fill="none" />
  <path id="lineBC" d="M 250 50 l 150 300" stroke="gold"
  ➥stroke-width="10" fill="none" />
  <path id="lineDE" d="M 175 200 l 150 0" stroke="purple"
```

```
  ➥stroke-width="10" fill="none" />

  <!-- quadratic bezier curve -->
  <path d="M 100 350 q 150 -300 300 0" stroke="black"
  ➥stroke-width="6" fill="none" />

  <!-- mark points with a red dot -->
  <g stroke="red" stroke-width="5" fill="red">
    <circle id="pointA" cx="100" cy="350" r="3" />
    <circle id="pointB" cx="250" cy="50" r="3" />
    <circle id="pointC" cx="400" cy="350" r="3" />
  </g>

  <!-- Add labels to each point -->
  <g font-size="25" font="sans-serif" fill="black" stroke="none"
  ➥text-anchor="middle">
    <text x="100" y="350" dx="-30">A</text>
    <text x="250" y="50" dy="-10">B</text>
    <text x="400" y="350" dx="30">C</text>
  </g>

</svg>
</body>
</html>
```

The quadratic Bézier curve is defined by the `path` tag:

```
<path d="M 100 350 q 150 -300 300 0" stroke="black"
➥stroke-width="6" fill="none" />
```

The `d` attribute instructs the parser to move to coordinate (100,350). The 'q' defines two further coordinates which are relative to (100,350). The first is the control point (150,-300)—which equates to the absolute position (450,50). The second is the ending point of the curve at (300,0)—which equates to the absolute position (400,350).

Alternatively, we could have used an uppercase 'Q' to use absolute, rather than relative, coordinate references.

Cubic Bézier Curves

While quadratic Bézier curves have one control point, cubic Bézier curves have two. This allows more complex shapes which can reverse direction or wrap back on to themselves.

The following code provides three cubic Bézier examples:

```
<!DOCTYPE html>
<html>
<body>
  <svg width="1200" height="500"
➥xmlns="http://www.w3.org/2000/svg" version="1.1">

  <!-- cubic bezier curves -->
    <path id="cubic1" d="M 100 350 c 150 -300 150 -300 300 0"
  ➥stroke="red" stroke-width="5" fill="none"/>
    <path id="cubic2" d="M 450 350 c 200 -300 100 -300 300 0"
  ➥stroke="red" stroke-width="5" fill="none"/>
    <path id="cubic3" d="M 800 350 c 100 -300 200 -300 300 0"
  ➥stroke="red" stroke-width="5" fill="none"/>

  <!-- show control points -->
    <g stroke="blue" stroke-width="3" fill="blue">

      <!-- left curve -->
      <circle cx="250" cy="50" r="3"/>

      <!-- middle curve control points -->
      <circle cx="650" cy="50" r="3"/>
      <circle cx="550" cy="50" r="3"/>

      <!-- right curve control points -->
      <circle cx="900" cy="50" r="3"/>
      <circle cx="1000" cy="50" r="3"/>

    </g>

  <!-- text -->
    <g font-size="30" font="sans-serif"
  ➥fill="red" stroke="none" text-anchor="middle">

    <text x="250" y="50" dy="-10">
    Both control points
```

```
    </text>

    <text x="650" y="50" dy="-10">
    CP1
    </text>
    <text x="550" y="50" dy="-10">
    CP2
    </text>

      <text x="900" y="50" dy="-10">
    CP2
      </text>
      <text x="1000" y="50" dy="-10">
    CP1
      </text>

    </g>

  </svg>
</body>
</html>
```

This will produce the output shown in Figure 8.2.

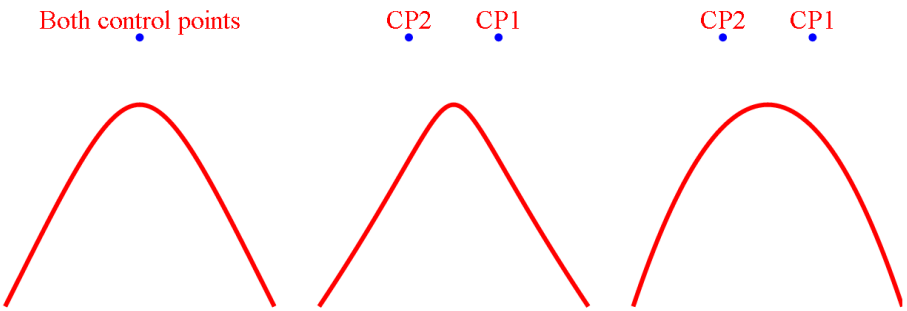

Figure 8.2. A cubic Bézier curve

Let's examine the third curve:

```
<path id="cubic3" d="M 800 350 c 100 -300 200 -300 300 0"
➥stroke="red" stroke-width="5" fill="none"/>
```

The `d` attribute instructs the parser to move to coordinate (800,350). The 'c' defines three further coordinates which are relative to (800,350). The first is the start control point (100,-300)—which equates to the absolute position (900,50). The second is the end control point (200,-300)—which equates to the absolute position (1000,50). The third is the ending point of the curve at (300,0)—which equates to the absolute position (1100,350).

Alternatively, we could have used an uppercase 'C' directive to use absolute, rather than relative, coordinate references.

There are also shorthand 'S' (absolute) and 's' (relative) directives. These accept two coordinates; the end control point and the end point itself. The start control point is assumed to be the same as the end control point.

These commands can be used to change the shape of cubic Bézier curves depending on the position of the control points. Have a play about and don't just view your results in the same browser window either, resize them, look at them on your tablet or smartphone and marvel at how well SVG copes with resizing.

Fortunately, there are tools to help you define curve directives. SitePoint's Craig Buckler has created Quadratic Bézier Curve[2] and Cubic Bézier Curve[3] tools, which allow you to move the control points and copy/paste the resulting SVG code.

In the next chapter, we'll take a look at filters.

[2] http://blogs.sitepointstatic.com/examples/tech/svg-curves/quadratic-curve.html
[3] http://blogs.sitepointstatic.com/examples/tech/svg-curves/cubic-curve.html

Chapter 9

Filter Effects

You can use **filter effects** in SVG. If you use any graphic design or photo manipulation packages, then you're almost certain to have come across filters before. The filters in SVG include:

- `feBlend`
- `feColorMatrix`
- `feComponentTransfer`
- `feComposite`
- `feConvolveMatrix`
- `feDiffuseLighting`
- `feDisplacementMap`
- `feFlood`
- `feGaussianBlur`

- `feImage`
- `feMerge`
- `feMorphology`
- `feOffset`
- `feSpecularLighting`
- `feTile`
- `feTurbulence`
- `feDistantLight`
- `fePointLight`
- `feSpotLight`

If you've used image-editing software you'll probably be familiar with what filters do. Essentially, they apply effects to an image such as bezels, blurring, soft-focus, and so on. In essence, think of filter elements as a form of image processing for the Web. Filter elements can be used in most modern browsers, with the exception of Blackberry Browser. For an example of what filters can do, take a look at Microsoft's hands on: SVG Filter Effects[1].

Using Filter Effects

Especially if you're new to using filters, it's a good idea to begin testing and experimenting with one filter at a time, otherwise you could end up with some pretty weird-looking images. For a comprehensive overview and sample code on filters, take a look at the SVG/Essentials Filters[2] page on O'Reilly Commons.

Below is an example of the code used to create a circle (which you've already learned how to create) with the `feGaussianBlur` filter applied. You can see the output in Figure 9.1.

[1] http://ie.microsoft.com/testdrive/graphics/hands-on-css3/hands-on_svg-filter-effects.htm
[2] http://commons.oreilly.com/wiki/index.php/SVG_Essentials/Filters

```
<!DOCTYPE html>
<html>
<body>
  <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <filter id="f1" x="0" y="0">
      <feGaussianBlur stdDeviation="14"/>
    </filter>
  </defs>

  <circle cx="200" cy="200" r="200" stroke="red"
  ➥stroke-width="5" fill="gold" filter="url(#f1)" />

  </svg>
</body>
</html>
```

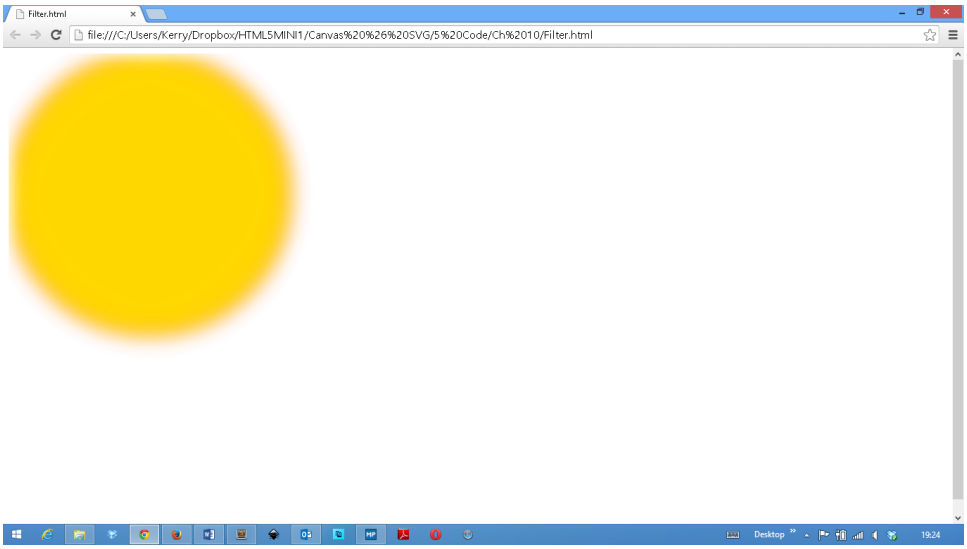

Figure 9.1. A circle with the Gaussian blur filter applied

Playing with Filters

As you can see, you'll have to give the filter an ID so that this can later be specified in the circle: `filter="url(#f1)`. The parameter that's associated with this filter is `stdDeviation`; this controls the amount of blurring. So, for example, if you were to set the `stdDeviation` to a value of `1`, then you would get such a minimal amount of blurring as to be hardly noticeable. However, if you were to set this to, say, 200,

then it creates a blurring effect that's almost transparent. And as we've applied a red stroke to the image (which you can't really see in the example above), this effect fills the SVG canvas with an extremely blurred circle, as shown in Figure 9.2

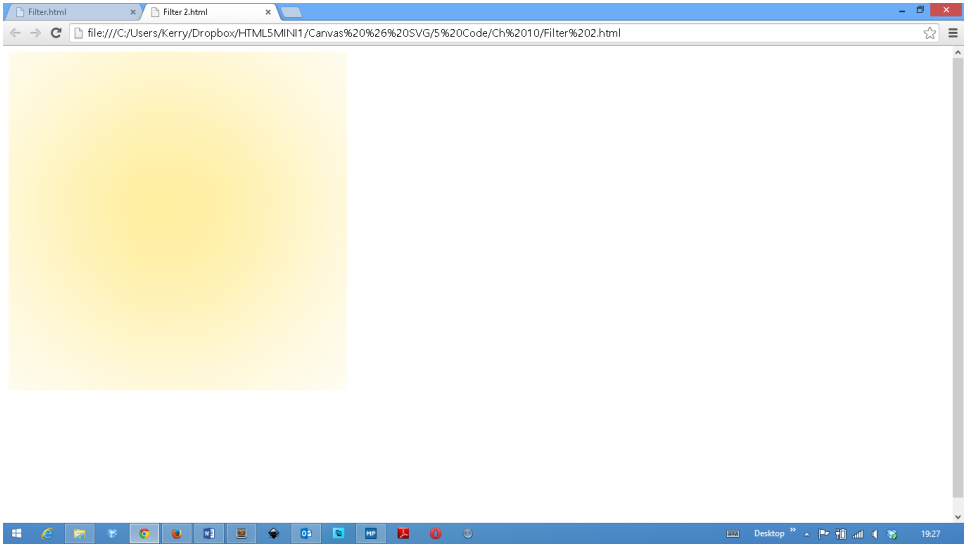

Figure 9.2. A very blurry circle

If you were to apply a lower value, then the blurring wouldn't be so apparent and would allow the stroke to be seen as an orange color with the yellow blurring into the red, as shown in Figure 9.3.

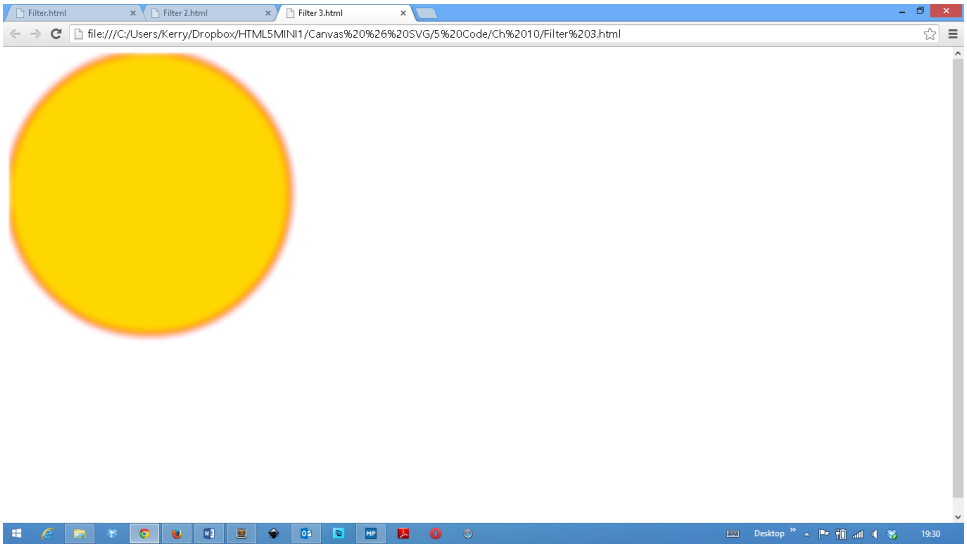

Figure 9.3. A less blurry circle

Let's try combining filter effects. We'll create a drop shadow using the `feOffset` filter; this is achieved by taking the relevant SVG image or element and moving it in the x-y plane. We'll use the `feBlend` and `feOffset` elements, which will create a duplicate image that's slightly offset from the original, to create the effect that one image is sitting behind the other, as shown in Figure 9.4.

```
<!DOCTYPE html>
<html>
<body>
  <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
    <defs>
      <filter id="f1" x="0" y="0" width="200%" height="200%">
        <feOffset result="offOut"  dx="25" dy="20" />
        <feBlend in="SourceGraphic" in2="offOut" mode="normal" />
      </filter>
    </defs>

    <polygon points="220,10 300,210 170,250 123,234" fill="blue"
     ➥stroke="purple" stroke-width="3" filter="url(#f1)" />
```

```
  </svg>
</body>
</html>
```

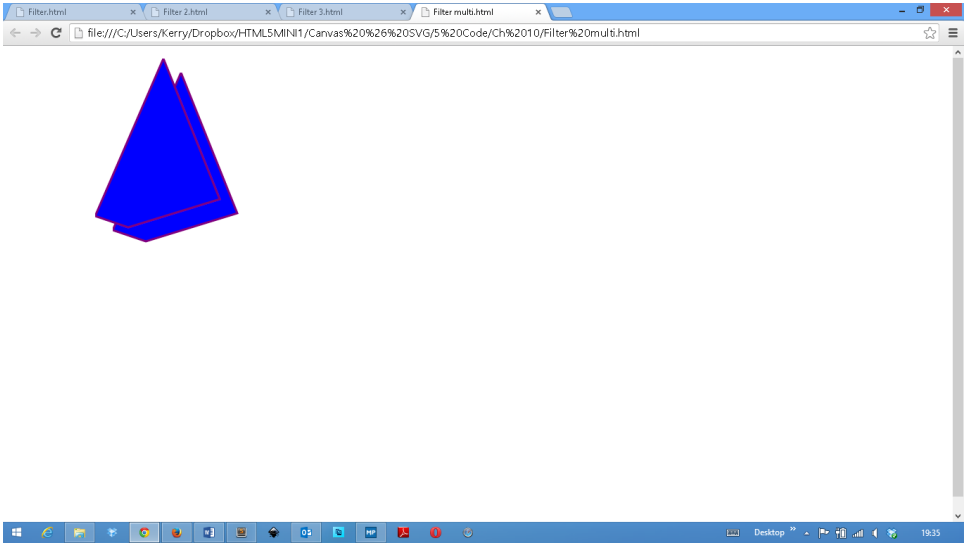

Figure 9.4. Using multiple filters

Filters can help you to create excellent effects. For example, if we were to apply `feGaussianBlur` to the above, then you could blur the rear image to create a kind of 3D effect.

Filter Future

SVG filter technology is coming to CSS3 so you'll be able to apply effects to any HTML element.

Chapter 10

Canvas or SVG

As we've seen, both canvas and SVG can do similar things. So how do you decide which to use?

Firstly, let's look at how each one is defined:

- SVG is short for **Scalable Vector Graphics** and is a language that's used to describe graphics in XML.
- Canvas, on the other hand, is a way of drawing graphics on the fly using JavaScript.

Now that really doesn't tell you a great deal about the differences between them and how they can each be used, so let's look at it in a little more depth. In SVG, drawn shapes are remembered as **objects** and therefore, if the attributes of that object change, the browser can then re-render the shape automatically. However, as canvas elements are drawn pixel-by-pixel, then any changes that are made will require the entire scene to be redrawn.

All elements of SVG are available in the DOM, so you can easily attach JavaScript event handlers to them. For the most part, the project will dictate which element you use, so it's worth giving it some thought at the planning stage.

Bear in mind that SVG is fully scalable, unlike canvas, and so SVG may very well be a better choice if you're designing a responsive site that has graphics that need to scale.

Creation Languages

To create images on a canvas element you have one choice: JavaScript. Those who understand the language will have a head start, but it's still necessary to learn the drawing APIs. However, animating canvas images is incredibly fast; you can draw and animate hundreds of items every second because the element is not constrained by the number of items being shown. This makes it ideal for action games.

SVG files are XML—which is simply structured text. They can be pre-prepared in a vector graphics package such as Illustrator or Inkscape, or you can dynamically create them on the server using any language: Node.js, PHP, Ruby, Python, C#, Java, BASIC, Cobol etc.

You can also create and manipulate SVG on the client using JavaScript with a familiar DOM and event-handling API. It's rarely necessary to re-draw the whole image because objects remain addressable. Unfortunately, this is far slower than moving bitmaps on canvas. SVG may be ideal for an animated bar chart, but not necessarily suitable for fast-moving action games.

Typical Uses

In general, SVG is ideal for:

- static images, especially within responsive and fluid layouts
- images which can be resized to any dimension without losing quality
- projects which benefit from DOM methods to attach events and manipulate objects
- projects which create images server-side

- projects where accessibility and SEO are important

Canvas is ideal for:

- bitmap images, editing photographs or any operation which requires pixel-level manipulation
- images which must be created and animated on-the-fly
- graphically-intense applications, such as games

Sometimes, there will not be an obvious best solution and either technology can be used. Remember neither is mutually exclusive; you can use both canvas and SVG on the same page at the same time, e.g. multiple animated canvas elements over a static SVG background. The only limit is your imagination.

I hope you've enjoyed this book and are ready to get started with some awesome examples of your own.

Thanks for reading, and have fun with canvas and SVG!