A user-friendly reference guide

HTML5 & CSS3

SAMPLE CHAPTER

Rob Crowther

MANNING

Hello! HTML5 & CSS3

by Rob Crowther

Chapter 3

Brief contents

3

Dynamic graphics

This chapter covers

- Using the <canvas> element to draw shapes, text, and images
- Transforming existing images with <canvas>
- Using Scalable Vector Graphics (SVG) in your web pages
- The strengths and weaknesses of <canvas> and SVG
- Cross-browser support

In this chapter, you'll learn about HTML5's facilities for dynamic graphics—graphics that can change in response to user input, data, or simply time passing. This could include charts representing network activity or the location of people on a map.

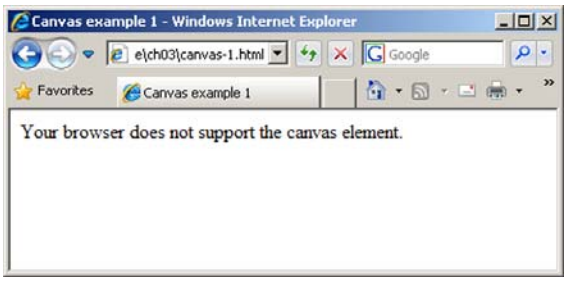
THIS CHAPTER, ESPECIALLY THE PARTS TO DO WITH THE <canvas> ELEMENT, WILL MAKE A LOT OF USE OF JAVASCRIPT. IF YOU'RE NOT FAMILIAR WITH JAVASCRIPT, YOU SHOULD CHECK OUT APPENDIX D BEFORE PROCEEDING.

Getting started with <canvas>: shapes, images, and text

The <canvas> element is an image you can create with JavaScript. The markup for it is similar to an <image> element in that you can specify a width and a height; but it has starting and closing tags that can enclose fallback content, and it doesn't reference an external source:

```
<canvas id="mycanvas" width="320" height="240"
        style="outline: 1px solid #999;">
    Your browser does not support the canvas element.
</canvas>
```

In a browser that doesn't support <canvas> the fallback content is displayed, as in this screenshot.



Browser support quick check: <canvas>

	Canvas 2D context	Canvas text
	4.0	4.0
	2.0	3.5
	9.0	9.0
	9.0	10.5
	3.1	4.0

YOU MIGHT HAVE A STATIC IMAGE AS THE FALLBACK IF IT COULD ADEQUATELY PRESENT SOME OF THE INFORMATION THAT WOULD BE DISPLAYED IN <canvas> IN SUPPORTING BROWSERS. OR, IF YOU WERE PARTICULARLY AMBITIOUS, YOU COULD USE AN ALTERNATIVE RENDERING METHOD SUCH AS FLASH.

You may be more interested to see what the page looks like in a browser that *does* support <canvas>.

If you're wondering where all the whizzy graphics promised in the introduction are, well, they don't appear by magic. To create pictures with <canvas>, there needs to be a JavaScript program that tells the browser what to draw.

Before you get to drawing something, you need to understand a couple of things. You need to know how to get a reference to your canvas object so you can send it drawing commands; and, because you'll be telling the <canvas> element to draw shapes on a grid, you need to know how the grid is defined. First, here's how to get a reference in JavaScript:

```
function draw() {
    var canvas = document.getElementById('mycanvas');
    if (canvas.getContext) {
        var ctx = canvas.getContext('2d');
        //do stuff
    }
}
window.addEventListener("load", draw, false);
```

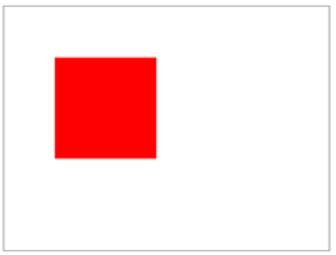
Add this code between <script> tags in the <head> of an HTML document containing a <canvas> element like that shown in the first listing in this section. In the following sections, you'll update the draw() function to create graphics. If you're confused about what this document should look like, please download the code samples from www.manning.com/crowther/ and look at the file ch03/canvas-1.html.

YOU HAVE TO PASS A PARAMETER, 2d, TO THE getContext METHOD. THIS GIVES YOU A TWO-DIMENSIONAL DRAWING CONTEXT. CURRENTLY THIS IS THE ONLY PARAMETER SUPPORTED. SEVERAL BROWSER VENDORS ARE EXPERIMENTING WITH A THREE-DIMENSIONAL DRAWING CONTEXT WITH DIRECT ACCESS TO GRAPHICS HARDWARE, WHICH WILL OPEN UP POSSIBILITIES SUCH AS 3D GAMES, VIRTUAL-REALITY EXPERIENCES, AND MODELING TOOLS.

Drawing shapes

To draw on the canvas, you need to get a drawing context. The context then gives you access to methods that allow the drawing of lines and shapes.

Basic shapes are easy. If you replace the previous `draw()` function, you can draw a rectangle by using the `fillRect` method. The only prerequisite is that you first set the fill color using the `fillStyle` method. You call the `fillRect` method with four arguments: the `x` and `y` values of the upper-left corner and the `width` and `height` to fill:



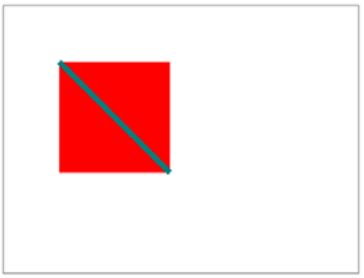
```
function draw() {
 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.fillRect(50,50,100,100);
 }
}
```

IN ADDITION TO `fillRect()`, THERE ARE ALSO METHODS TO CLEAR AN AREA OF PIXELS AND TO DRAW AN EMPTY RECTANGLE: `clearRect()` AND `strokeRect()`, RESPECTIVELY. THEY TAKE THE SAME PARAMETERS AS `fillRect()`.

Let's extend the code to draw a line. Lines are a little more complex. You have to first draw a path, but the path doesn't appear until you apply a stroke. If you've ever used graphics software like Photoshop, this process should be familiar to you.

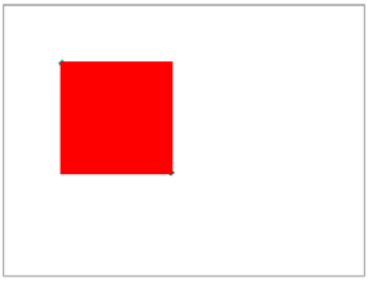
The `moveTo` method moves the "pen" without recording a path, and the `lineTo` method moves the pen and records a path:

```
function draw() {
 var canvas = document
    .getElementById('mycanvas');

 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.fillRect(50,50,100,100);
  ctx.strokeStyle =
          'rgb(0,127,127)';
  ctx.moveTo(50,50);
  ctx.lineTo(150,150);
  ctx.lineWidth = 5;
  ctx.stroke();
 }
}
```



Now for a little experiment. What happens if the line is drawn first and then the box?

```
function draw() {
 var canvas = document
    .getElementById('mycanvas');

 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  ctx.strokeStyle =
          'rgb(0,127,127)';
  ctx.moveTo(50,50);
  ctx.lineTo(150,150);
  ctx.lineWidth = 5;
  ctx.stroke();
  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.fillRect(50,50,100,100);
 }
}
```

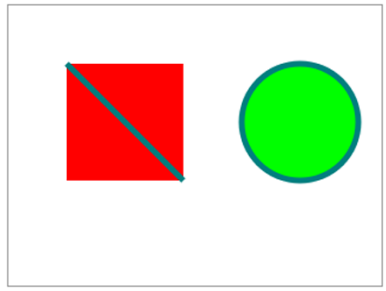


As you can see, the line is mostly obscured by the rectangle. You might think that if you could remove the rectangle the line would still be there underneath; but after you've drawn over it, the line is gone.

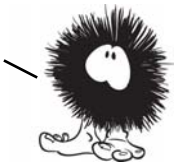
THE ONLY WAY TO GET THE LINE BACK IS TO ERASE BOTH THE RECTANGLE AND THE LINE AND THEN DRAW THE LINE AGAIN. THE <canvas> ELEMENT DOESN'T STORE THE ELEMENTS DRAWN, ONLY THE RESULTING PIXELS.

What about other shapes? The path-then-stroke approach is the way to do it. You can use the `arc` method to draw a circle and then fill it. The `arc` method accepts parameters for the location of the center; the radius; how far around, in radians, the arc should extend; and whether that should be clockwise or counterclockwise:

```
function draw(){
 var canvas = document
    .getElementById('mycanvas');
 if (canvas.getContext) {

  var ctx = canvas.getContext('2d');
  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.fillRect(50,50,100,100);
  ctx.fillStyle = 'rgb(0,255,0)';
  ctx.arc(250, 100,
          50, 0,
          Math.PI*2,
          false);
  ctx.fill();
  ctx.strokeStyle =
          'rgb(0,127,127)';
  ctx.moveTo(50,50);
  ctx.lineTo(150,150);
  ctx.lineWidth = 5;
  ctx.stroke();
 }
}
```

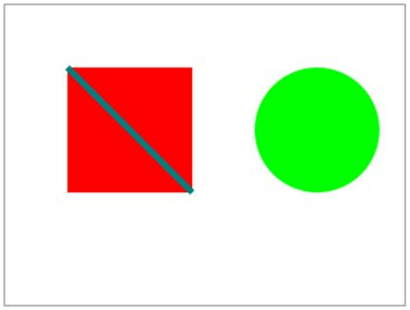


NOTE THAT THE STROKE YOU USE TO DRAW THE LINE AT THE END ALSO GETS APPLIED TO THE CIRCLE, EVEN THOUGH THE `strokeStyle` WAS SET AFTER THE ARC WAS CREATED.



To ensure that the stroke for the line doesn't apply to the circle, you need to explicitly put them on different paths with the `beginPath()` method:

```
function draw(){
 var canvas = document
    .getElementById('mycanvas');
 if (canvas.getContext) {

  ctx.fillStyle = 'rgb(255,0,0)';
  ctx.fillRect(50,50,100,100);
  ctx.beginPath();
  ctx.fillStyle = 'rgb(0,255,0)';
  ctx.arc(250, 100, 50, 0,
          Math.PI*2, false);
  ctx.fill();
  ctx.beginPath();
  ctx.strokeStyle =
          'rgb(0,127,127)';
  ctx.moveTo(50,50);
  ctx.lineTo(150,150);
  ctx.lineWidth = 5;
  ctx.stroke();
 }
}
```



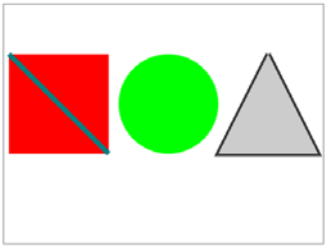
Other shapes are just a matter of creating a path and then stroking or filling, or both. If you move the first two shapes over a little, there's room to add a triangle. First draw the square and the circle again slightly further to the left:

```
ctx.fillStyle = 'rgb(255,0,0)';
ctx.fillRect(5,50,100,100);
ctx.beginPath();
ctx.fillStyle = 'rgb(0,255,0)';
ctx.arc(165, 100, 50, 0, Math.PI*2, false);
```

```
ctx.fill();
ctx.beginPath();
ctx.strokeStyle = 'rgb(0,127,127)';
ctx.moveTo(5,50);
ctx.lineTo(105,150);
ctx.lineWidth = 5;
ctx.stroke();
```

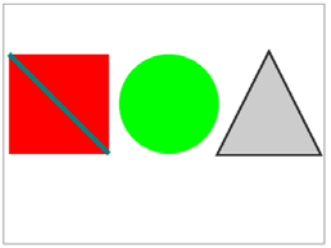
Put this code in your `draw()` function inside the `if (canvas.getContext) {}` block, replacing what you had previously, and then add the code for the triangle to it:

```
ctx.beginPath();
ctx.moveTo(265,50);
ctx.lineTo(315,150);
ctx.lineTo(215,150);
ctx.lineTo(265,50);
ctx.strokeStyle = 'rgb(51,51,51)';
ctx.fillStyle = 'rgb(204,204,204)';
ctx.stroke();
ctx.fill();
```



Notice that, even though the triangle starts and ends at the same point, there's a slight gap at the top.

To prevent this, you need to close the path using the `closePath` method; the additional line required is highlighted bold:



```
ctx.beginPath();
ctx.moveTo(265,50);
ctx.lineTo(315,150);
ctx.lineTo(215,150);
ctx.lineTo(265,50);
ctx.closePath();
ctx.strokeStyle = 'rgb(51,51,51)';
ctx.fillStyle = 'rgb(204,204,204)';
ctx.stroke();
ctx.fill();
```

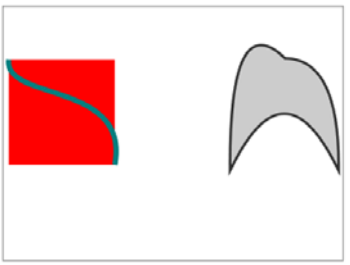
You don't have to restrict yourself to straight lines in paths. Instead of `lineTo`, you can use either of two types of curve: quadratic or Bézier.

Let's replace the first straight line with a Bézier curve. Instead of

```
ctx.moveTo(5,50);
ctx.lineTo(105,150);
```

use the lines

```
ctx.moveTo(5,50);
ctx.bezierCurveTo(0,90, 120,70,
                  105,150)
```



The last pair of numbers is the end point. Preceding that are coordinates for the two control points.

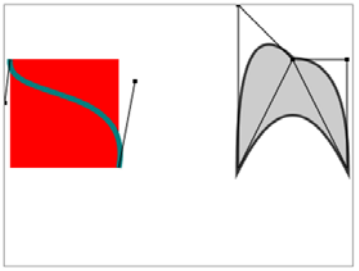
To simplify things the circle has been removed for now, but the sides of the triangle have also been made curvy, this time with a quadratic curve. A quadratic curve is similar, but only needs one control point. This is the code that drew the triangle, using `quadraticCurve` instead of `lineTo`. Here's the original triangle drawing code:

```
ctx.moveTo(265,50);
ctx.lineTo(315,150);
ctx.lineTo(215,150);
ctx.lineTo(265,50);
```

Replace it with this:

```
ctx.moveTo(265,50);
ctx.quadraticCurveTo(315,50, 315,150);
ctx.quadraticCurveTo(265,50, 215,150);
ctx.quadraticCurveTo(215,0, 265,50);
```

In this version of the earlier diagram, the control points have been drawn along with lines connecting the control points to the start and end of the path; this should help you visualize what's going on. The drawn lines are distorted from their direct path so they approach an imaginary line drawn between the start or end point and the control point. Check out the full listings for these examples in ch03/canvas-6.html and ch03/canvas-6-controls.html of the code download at www.manning.com/crowther/.



Placing images

One of the great features of <canvas> is that you can use it to manipulate images and achieve effects that are otherwise difficult to do with HTML and CSS. Here's an example of what can be achieved: a reflection effect.

The <canvas> element can't download images—you can't give it a URL and expect it to fetch the image. Any image you want to use must already be available in your page content. There are various ways to do this, but the easiest is to include the element in the normal way. In this case, it's hidden:

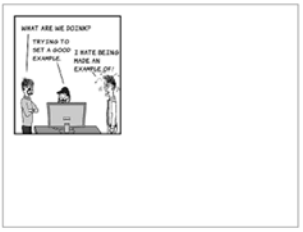
```
<div style="display: none;">
    <img id="myimage" src="example.png" width="236" height="260">
</div>
```

The next few examples take the image at right and import it into the <canvas> element. The simplest example is to import the image and place it on the <canvas>.

You call drawImage with three parameters—the img element and the x and y coordinates:


```
function draw(){
 var canvas = document
    .getElementById('mycanvas');
 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  var img =
document.getElementById('myimage');
  ctx.drawImage(img, 10, 10);
 }
}
```

The example image is too large to fit into the <canvas> frame. You can easily fix this by defining a width and height for the placed image:



```
if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  var img = document
    .getElementById('myimage');
  ctx.drawImage(
    img, 10, 10, 118, 130
  );
}
```

Now you're calling drawImage with five parameters. The additional two are the width and height of the placed image.

You may not even want all of the original image:


```
if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  var img =
document.getElementById('myimage');
  ctx.drawImage(img,
         80, 100, 80,  160,
         10, 10,  160, 200);
}
```

The previous example calls `drawImage` with nine parameters. Let's examine them in more detail:

```
drawImage(img,
          80, 100,#
          80, 160,#
          10, 10,
          160, 200)
```

THE FIRST PARAMETER IS STILL A REFERENCE TO THE `img` ELEMENT.

THE X AND Y OF A POINT IN SOURCE IMAGE.

WIDTH AND HEIGHT FOR SECTION OF SOURCE IMAGE.

THE X AND Y POINT IN THE CANVAS FOR THE PLACED IMAGE.

WIDTH AND HEIGHT FOR THE PLACED IMAGE

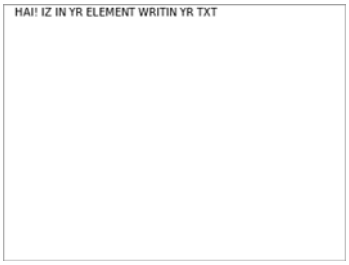
Drawing text

Let's now turn our attention to text. The `<canvas>` element has a limited ability to draw single lines of text on the context. The text-drawing methods are more suitable for drawing labels and titles than for rendering large blocks of text. But the full graphical-processing ability of the `<canvas>` element can be applied to the text that's drawn, allowing for effects like this.

THIS EXAMPLE TAKES ADVANTAGE OF THE TRANSFORMATION AND GRADIENT FILL FEATURES OF THE `<canvas>` ELEMENT. MORE DETAILS ABOUT THEM ARE IN THE SECTION "ADVANCED `<canvas>`: GRADIENTS, SHADOWS, AND ANIMATION."

Drawing text on the <canvas> is easy with the fillText method:

```
function draw(){
 var canvas = document
    .getElementById('mycanvas');
 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  ctx.fillText(
   'HAI! IZ IN YR ELEMENT
    WRITIN YR TXT',
    10,10);
 }
}
```



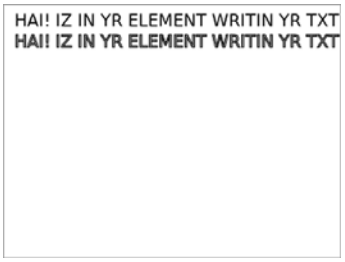
The fillText method has three required parameters: a string that is the text to be drawn, and x and y coordinates to determine where it's to be drawn.

The text is drawn in the current font, which is determined by setting the font property of the drawing context. The <canvas> element's font property behaves like the CSS font property, allowing size and font to be specified simultaneously:

```
ctx.font = "10pt serif";
```

If you set the font size a little larger, you can see an alternative method for drawing text. As with rectangles, you can draw the fill and the stroke separately:

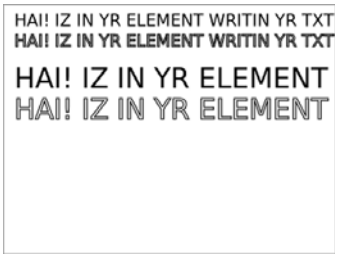
```
function draw(){
 var canvas = document
    .getElementById('mycanvas');
 if (canvas.getContext) {
  var ctx = canvas.getContext('2d');
  ctx.font = "12pt sans-serif";
  ctx.fillText(
    'HAI! IZ IN YR ELEMENT
     WRITIN YR TXT',
     10,20);
  ctx.strokeText(
    'HAI! IZ IN YR ELEMENT
```



```
      WRITIN YR TXT',
      10,40);
 }
}
```

You can of course draw both fill and stroke on a single line of text if you want.

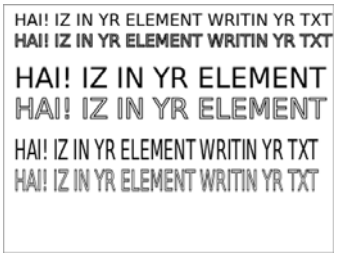
Let's see what happens if you increase the font size a bit more. Remember, the example `<canvas>` element is 320 pixels wide:




```
ctx.font = "20pt sans-serif";
ctx.fillText(
  'HAI! IZ IN YR ELEMENT
   WRITIN YR TXT',
   10,80);
ctx.strokeText(
  'HAI! IZ IN YR ELEMENT
   WRITIN YR TXT',
   10,110);
```

As you can see, the text that doesn't fit flows off the edge of the element without wrapping.

To work around this issue, you can use the fourth, optional, parameter to the `fillText` and `strokeText` methods. This parameter sets a maximum width for the text; if the text will be wider than the value passed, the browser makes the text fit either by narrowing the spacing between the letters or scaling down the font:




```
ctx.fillText(
  'HAI! IZ IN YR ELEMENT
  WRITIN YR TXT',
  10,150,300);
ctx.strokeText(
 'HAI! IZ IN YR ELEMENT
  WRITIN YR TXT',
  10,180,300);
```

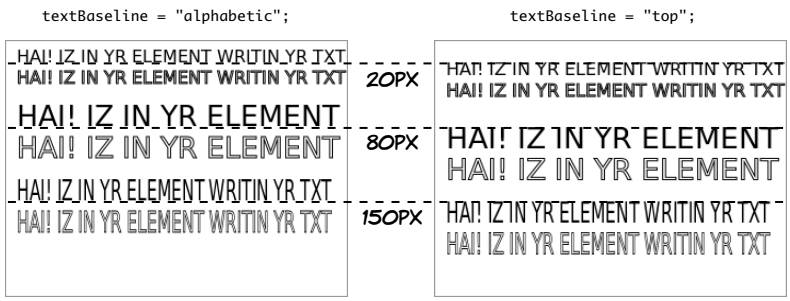
If you've added all three of these examples to the original listing, then you should have something similar to the ch03/canvas-text-4.html file in the code download.

To further control the text position you can set the baseline of the text, which will adjust where it's drawn in relation to the coordinates you provide. This is useful if you're trying to position labels next to things on your canvas because it saves you having to work out exactly how tall the letters will be drawn.

The default value is `alphabetic`, which means the bottom of an uppercase letter is placed at the y coordinate you provide in `fillText`. The following line sets the baseline to `top`, which means the top of an uppercase letter will be placed in line with the provided y coordinate:

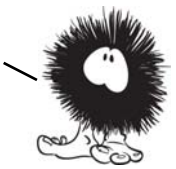
```
ctx.textBaseline = "top";
```

The following figure shows the previous example alongside a similar example, except with `textBaseline` set differently.



Other values for `textBaseline` are `hanging`, `middle`, `ideographic`, and `bottom`.

YOU CAN NOW DRAW IMAGES AND TEXT ON YOUR <canvas> ELEMENT. THE SIMPLE EXAMPLES WE'VE COVERED HERE MAY NOT SEEM MUCH MORE EXCITING THAN WHAT CAN BE ACHIEVED WITH PLAIN HTML AND CSS, BUT WE'VE BARELY SCRATCHED THE SURFACE. IN THE NEXT SECTION, YOU'LL LEARN ABOUT SOME MORE ADVANCED TECHNIQUES: GRADIENTS, DROP SHADOWS, AND TRANSFORMATIONS.



Advanced <canvas>: gradients, shadows, and animation

With the ability to draw a single-pixel shape on any part of the canvas, it's possible for you to create any effect you want by implementing it

yourself in JavaScript. But the `<canvas>` element has some built-in shortcuts for particular effects. This section covers them.

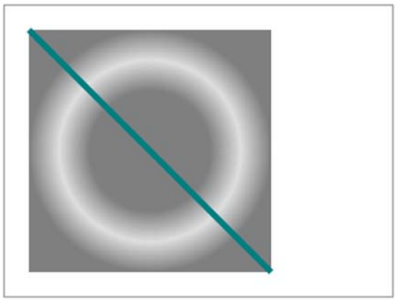
Creating gradients

The `strokeStyle` and `fillStyle` methods you used in "Drawing shapes" to set the color of lines and shapes can also accept a gradient object where the color changes smoothly across a defined space. The `<canvas>` element can create two types of gradient:

- *Linear*—The gradient follows a straight line.
- *Radial*—The gradient is circular.

In this section, you'll create one example of each. There are three steps to creating either gradient type in the `<canvas>` element:

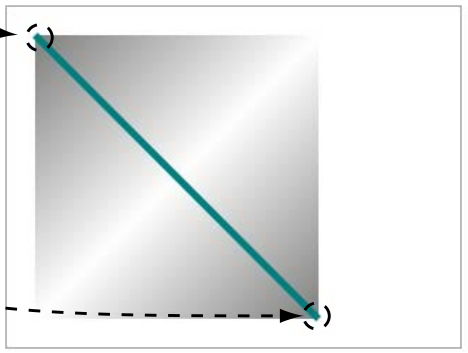
1. Create a gradient.
2. Specify the color stops.
3. Apply the gradient as a fill to a shape.



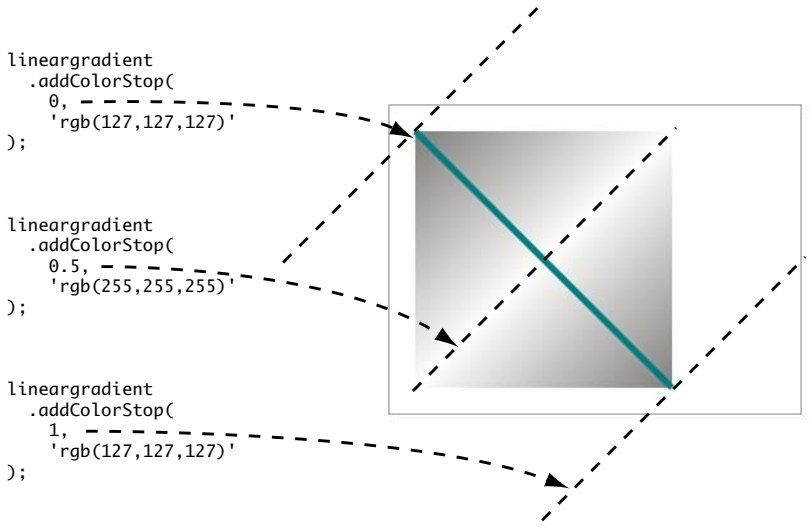
Here's a simple linear gradient in place of the solid fill from the earlier examples.

You define the extents of the gradient with the `createLinearGradient()` method. This method takes four parameters that define the upper-left and lower-right corners. The following diagram contains the code and indicates what the parameters refer to in the screenshot.

```
var lineargradient = ctx.createLinearGradient(
    20,20,
    220,220
);
```



Next you need to add color stops to the `lineargradient` you just created. A *color stop* is a point on the gradient at which you're setting a specific color. The browser interpolates between the color stops to create the gradient. The gradient object has an `addColorStop()` method for this. It accepts two parameters: a position and a color. The position is a number between 0 and 1, where 0 is the start of the gradient and 1 is the end. The code in the next diagram adds three color stops to your gradient.

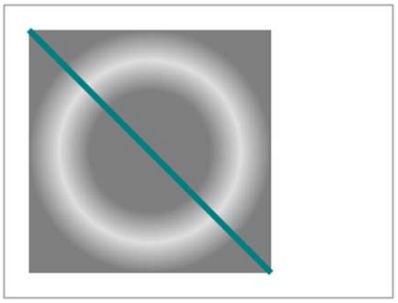


All that remains is to add the gradient to the context as a `fillStyle` and draw a shape. Here's the complete `draw()` function from ch03/canvas-9.html:

```
function draw(){
    var canvas = document.getElementById('mycanvas');
    if (canvas.getContext) {
        var ctx = canvas.getContext('2d');
        var lineargradient = ctx.createLinearGradient(20,20,220,220);
        lineargradient.addColorStop(0,'rgb(127,127,127)');
        lineargradient.addColorStop(0.5,'rgb(255,255,255)');
        lineargradient.addColorStop(1,'rgb(127,127,127)');
        ctx.fillStyle = lineargradient;
        ctx.fillRect(20,20,200,200);
        ctx.strokeStyle = 'rgb(0,127,127)';
```

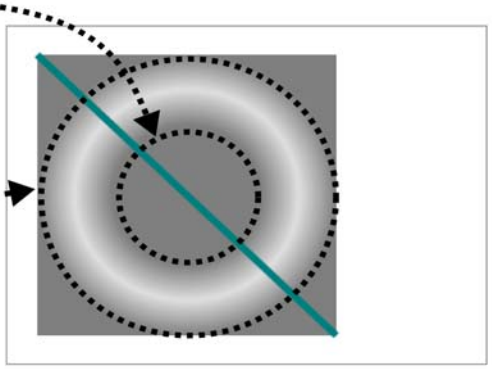
```
        ctx.moveTo(20,20);
        ctx.lineTo(220,220);
        ctx.lineWidth = 5;
        ctx.stroke();
    }
}
```

Now let's create a radial gradient.



For a radial gradient, you use the `createRadialGradient()` method. Six values are required: a center point and a radius for the inner bound, and a center point and a radius for the outer bound. This creates two circles between which the gradient is drawn. The two circles and their corresponding parameters are shown here.

```
var radialgradient = ctx.createRadialGradient(
    120,120,50,
    120,120,100
);
```



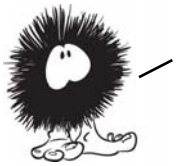
Adding color stops is exactly the same as with the linear gradient, except that now those stops define circles between the two described in the `createRadialGradient` method:

```
radialgradient.addColorStop(0, 'rgb(127,127,127)');
radialgradient.addColorStop(0.5, 'rgba(127,127,127,0.25)');
radialgradient.addColorStop(1, 'rgb(127,127,127)');
```

Finally, the gradient is applied as a `fillStyle` as before:

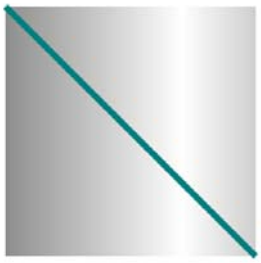
```
ctx.fillStyle = radialgradient;
ctx.fillRect(20,20,200,200);
```

Check out the full listing in the file ch03/canvas-10.html in the code download.

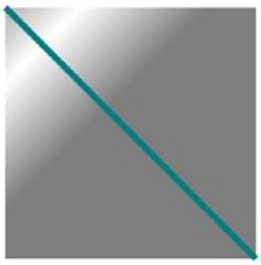


NOTE THAT YOU DEFINE BOTH LINEAR AND RADIAL GRADIENTS WITH COORDINATES RELATIVE TO THE ENTIRE CANVAS CONTEXT, NOT THE SHAPE YOU WANT TO APPLY THEM TO. IF YOU WANT THE GRADIENT TO EXACTLY FILL THE SHAPE, YOU HAVE TO MAKE SURE YOU CHOOSE THE COORDINATES SO THAT THE GRADIENT APPEARS IN THE SHAPE YOU WANT TO FILL IT WITH.

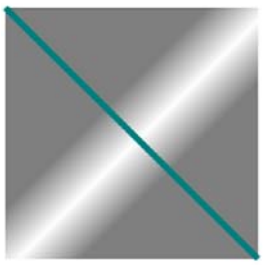
THE GRADIENT ISN'T CONFINED TO THE COORDINATES YOU SPECIFY—IT EXTENDS ACROSS THE CANVAS. THE FOLLOWING EXAMPLES SHOW A LINEAR GRADIENT CREATED WITH THREE DIFFERENT SETS OF COORDINATES.



```
createLinearGradient(
  0,0,320,0
);
```



```
createLinearGradient(
  0,0,100,100
);
```



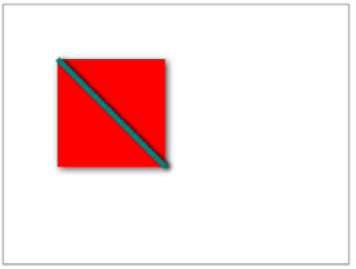
```
createLinearGradient(
  100,100,150,150
);
```

Drawing drop shadows

Drop shadows are an effect much loved by designers, and the `<canvas>` element has built-in support. To create a shadow, define the `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, and `shadowColor` properties on

the context object; the shadow will then be applied to any shape you draw.

This example shows the earlier square with a line through it, now with a shadow in place:



```
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 8;
ctx.shadowColor =
    "rgba(0, 0, 0, 0.75)";
```

Using shadows, you can create effects such as cutout text:


```
ctx.shadowOffsetX = 4;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 5;
ctx.shadowColor =
    "rgba(0, 0, 0, 0.9)";
ctx.fillStyle = 'rgb(0,0,0)';
ctx.fillText('HAI!',170,50);
ctx.fillStyle = 'rgb(255,255,255)';
ctx.fillText('IZ IN YR ELEMENT'
    ,170,70);
ctx.strokeStyle = 'rgb(0,0,0)';
ctx.strokeText('WRITIN YR TXT'
    ,170,90);
```

Transformations

THE <canvas> 2D CONTEXT SUPPORTS A NUMBER OF TRANSFORMATIONS. THESE WORK ON THE CONTEXT ITSELF, SO YOU APPLY THE TRANSFORMATION AND THEN DRAW WHATEVER YOU WANT TO APPEAR SUBJECT TO THAT TRANSFORM.

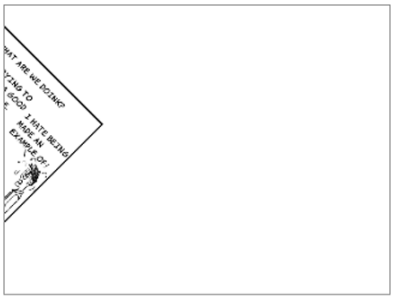
Let's start with a simple translate transformation. This moves the origin of the `<canvas>` element according to the x and y offsets you pass in as arguments:

```
var img = document
    .getElementById('myimage');
ctx.translate(120,20);
ctx.drawImage(
    img, 10, 10, 118, 130
);
```

If you compare this example with the similar one in "Placing images" without the transformation, you'll see you've basically moved the image down and to the right. Not particularly useful when you could have drawn the image there in the first place, but this technique would be useful if you wanted to move a collection of objects around while keeping their relative positions the same.

Next, let's try rotation:

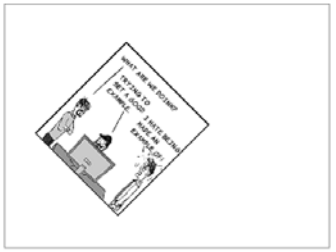
```
var img =
document.getElementById('myimage');
ctx.rotate(Math.PI/4);
ctx.drawImage(img, 10, 10, 118, 130);
```

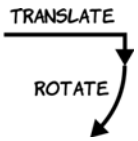


The `rotate()` method takes a value in radians and rotates the drawing context by that angle. As with `translate`, the values you provide to the `drawImage()` method are now relative to the transformation.

You don't want the image off the <canvas> like that, so let's translate it and then rotate it:

```
var img =
document.getElementById('myimage');
ctx.translate(120,20);
ctx.rotate(Math.PI/4);
ctx.drawImage(img, 10, 10, 118, 130);
```

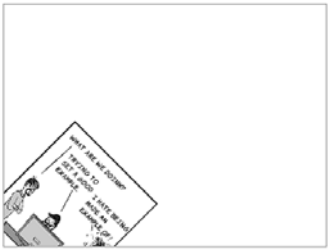


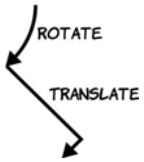


The transformations affect the whole context, so the order in which you apply them is important.

Let's try the opposite order:

```
var img =
document.getElementById('myimage');
ctx.rotate(Math.PI/4);
ctx.translate(120,20);
ctx.drawImage(img, 10, 10, 118, 130);
```

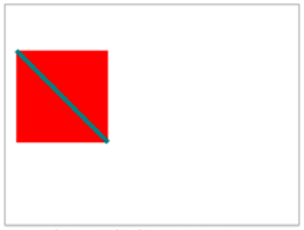




You can see that the rotate now changes the direction the translate goes in.

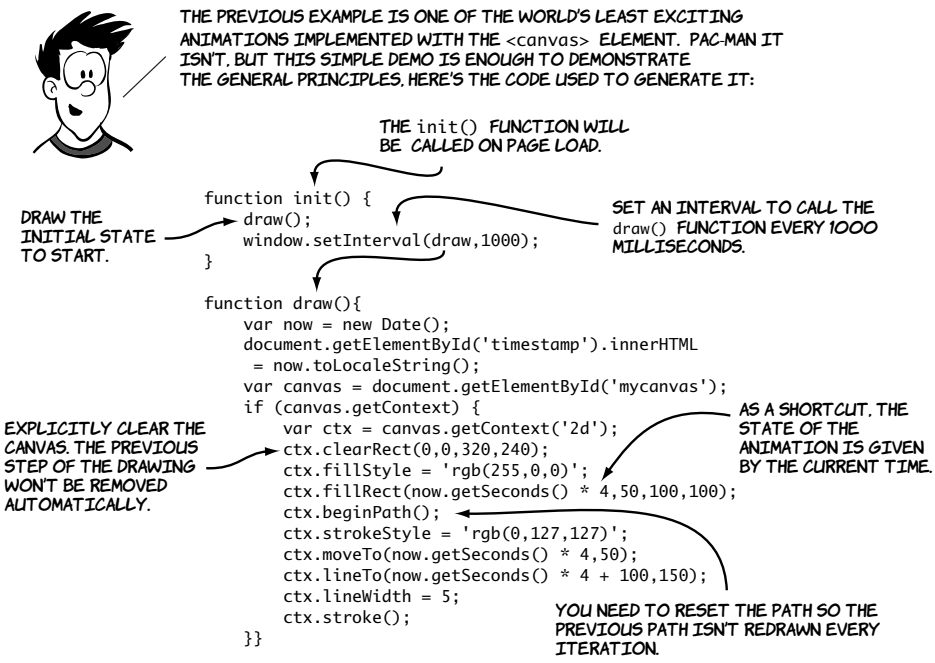
Animation

ONE POTENTIAL USE OF THE <canvas> ELEMENT THAT HAS MANY DEVELOPERS EXCITED IS CREATING GAMES. ALREADY, MANY ARCADE CLASSICS OF THE 1980S AND '90S HAVE BEEN RE-CREATED USING <canvas>. IN ORDER TO CREATE GAMES, YOU NEED TO HAVE ANIMATION. LET'S LOOK AT HOW YOU CAN ANIMATE YOUR CANVAS DRAWINGS.


Mon 21 Jun 2010 15:00:03 BST

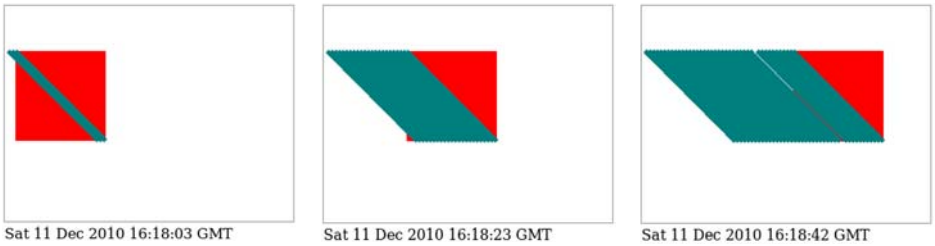

Mon 21 Jun 2010 15:00:24 BST


Mon 21 Jun 2010 15:00:47 BST




```
function init() {
    draw();
    window.setInterval(draw,1000);
}

function draw(){
    var now = new Date();
    document.getElementById('timestamp').innerHTML
     = now.toLocaleString();
    var canvas = document.getElementById('mycanvas');
    if (canvas.getContext) {
        var ctx = canvas.getContext('2d');
        ctx.clearRect(0,0,320,240);
        ctx.fillStyle = 'rgb(255,0,0)';
        ctx.fillRect(now.getSeconds() * 4,50,100,100);
        ctx.beginPath();
        ctx.strokeStyle = 'rgb(0,127,127)';
        ctx.moveTo(now.getSeconds() * 4,50);
        ctx.lineTo(now.getSeconds() * 4 + 100,150);
        ctx.lineWidth = 5;
        ctx.stroke();
    }}
```

Here's what happens if you forget to explicitly start a new path.



If you forget to close any paths you have open, they'll be redrawn as you iterate through your animation steps along with any additions to the path. Clearing the pixels on the context doesn't reset the path.

The <canvas> element allows precise, pixel-level control over what is displayed and is already considered a rival to Flash in the browser game marketplace because it works on iPhones and iPads. Experimental

versions of <canvas> have full 3D support, and several first-person shooters from the 1990s have already been ported to allow play in a browser.

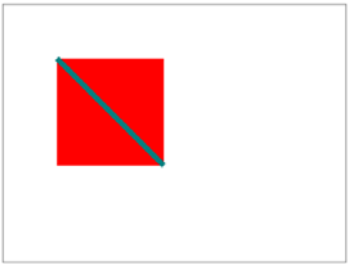
NOW THAT YOU'VE LEARNED ABOUT THE <canvas> ELEMENT, IT'S TIME TO LOOK AT THE SECOND TECHNOLOGY AVAILABLE IN HTML5 FOR DRAWING GRAPHICS: SVG.

Getting started with SVG

Browser support quick check: SVG in HTML

7.0	
4.0	
9.0	
11.6	
5.1	

Scalable Vector Graphics (SVG) is an XML language for displaying vector graphics. It has long been possible to embed SVG within XML-based XHTML documents; but because HTML5 leads you back to HTML-based markup, it adds the useful feature that SVG can be embedded directly.



Let's create a simple SVG drawing.

You're probably thinking it looks familiar, and you're right. Many of the things that can be achieved with `<canvas>` can also be easily achieved with SVG. You'll learn more about the relative strengths and weaknesses of each in the section "SVG vs. <canvas>," but for now all you need to understand is that `<canvas>` and SVG are based on different conceptual models of how to create images. `<canvas>` is what programmers call *imperative*; you provide a detailed list of operations to be performed that will produce a particular result. SVG is *declarative*; you provide a description of the final result and let the browser get on with it. Where `<canvas>` requires JavaScript, SVG requires markup, much like HTML, and it can be included directly in HTML5:

```
<!DOCTYPE html>
<html>
<head>
    <title>SVG example 2</title>
</head>
<body>
    <svg id="mysvg" viewBox="0 0 320 240"
         style="outline: 1px solid #999; width: 320px; height:
  240px;">
        <rect x="50" y="50" width="100" height="100"
              style="fill: rgb(255,0,0)">
        </rect>
        <line x1="50" y1="50" x2="150" y2="150"
              style="stroke: rgb(0,127,127); stroke-width: 5;">
        </line>
    </svg>
</body>
</html>
```

There are several interesting things to be seen in this simple example. First, note that the size of the element on the page is determined by CSS in the `style` attribute, but you also define a `viewBox` with the same values. Because SVG is a vector format, pixels aren't as significant; you can use `viewBox` to define a mapping between the physical dimensions of the element, defined in CSS, and the logical coordinates of everything displayed within.

Look what happens if you use these values: `viewBox="0 0 640 480"`. It's the same SVG graphic as before, but rendered into a larger viewport.

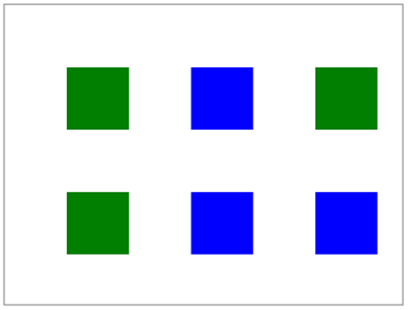
Applying styles to SVG

The previous examples used an inline style to apply colors and stroke thicknesses. Those properties can also be applied directly to the elements in question, like this:

```
<rect x="50" y="50" width="100" height="100"
      fill="rgb(255,0,0)"></rect>
<line x1="50" y1="50" x2="150" y2="150"
      stroke="rgb(0,127,127)" stroke-width="5"></line>
```

But you can alternatively leave off the style and inline attributes and use this in your CSS file, and achieve the same results:

```
rect { fill: rgb(255,0,0); }
line { stroke: rgb(0,127,127); stroke-width: 5; }
```



It looks much like any other CSS, albeit with some unusual properties. As with regular HTML, CSS can make life much easier if you have a lot of similar objects because you can use a class to apply a set of styles to several elements.

In this example there are three green squares (upper left, upper right, lower left) and three blue squares. Rather than specify inline styles on each one, you can declare their commonality with the `class` attribute:

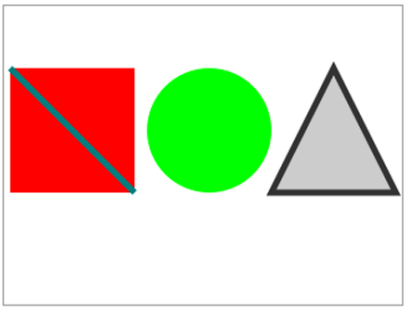
```
<svg id="mysvg" viewBox="0 0 320 240">
    <rect x="50" y="50" width="50" height="50" class="earth"></rect>
    <rect x="150" y="50" width="50" height="50" class="water"></rect>
    <rect x="250" y="50" width="50" height="50" class="earth"></rect>
    <rect x="50" y="150" width="50" height="50" class="earth"></rect>
```

```
    <rect x="150" y="150" width="50" height="50" class="water"></rect>
    <rect x="250" y="150" width="50" height="50" class="water"></rect>
</svg>
```

Then you style the common elements with CSS in the `<head>` of your document in the usual way:

```
<style>
  rect.earth { fill: rgb(0,127,0); }
  rect.water { fill: rgb(0,0,255); }
</style>
```

Drawing common shapes



Let's carry on and re-create the rest of the `<canvas>` example shapes in SVG. In addition to the rectangle and line elements you've seen already, SVG has elements for circles and arbitrary polygons.

For a circle, you need to provide the x and y coordinates of the center and the radius as appropriate attributes. A polygon is slightly more complex; it has an attribute `points` that you use to supply a space-separated list of x,y coordinates. This code, when placed inside the `<svg>` element from the listing in the introduction, generates the previous image:

```
<rect x="5" y="50" width="100" height="100"
      style="fill: rgb(255,0,0);"></rect>
<line x1="5" y1="50" x2="105" y2="150"
      style="stroke: rgb(0,127,127); stroke-width: 5;"></line>
<circle cx="165" cy="100" r="50"
        style="fill: rgb(0,255,0);"></circle>
<polygon points="265,50  315,150  215,150"
         style="stroke: rgb(51,51,51); fill: rgb(204,204,204);
                stroke-width: 5;"></polygon>
```

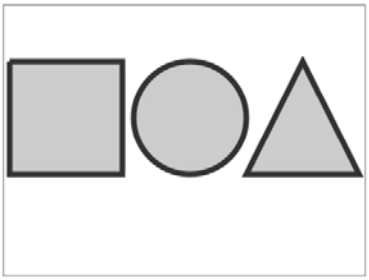
With the `polygon` element you don't have to provide the starting point a second time; it assumes the shape is closed, and the path drawn returns to the first point. If you want to draw an open shape, you can use the

<polyline> element instead; it uses an identical points attribute but doesn't close the path around the shape.

```
<polygon
  points="265,50  315,150  215,150"
  style="stroke: rgb(51,51,51);
         fill: rgb(204,204,204);
         stroke-width: 5;">
</polygon>
```

```
<polyline
  points="265,50  315,150  215,150"
  style="stroke: rgb(51,51,51);
         fill: rgb(204,204,204);
         stroke-width: 5;">
</polyline>
```

YOU'VE SEEN SEVERAL ELEMENTS MAKE DIFFERENT SHAPES IN A SINGLE SVG DRAWING, BUT THERE'S ALSO A WAY TO DRAW SEVERAL DIFFERENT SHAPES IN A SINGLE SVG ELEMENT. FOR THIS YOU USE THE <path> ELEMENT. LET'S LOOK AT AN EXAMPLE.



ALTHOUGH YOU CAN SEE THREE SHAPES IN THE PREVIOUS IMAGE, THEY'RE A SINGLE SVG ELEMENT: A PATH. THE <path> ELEMENT IN SVG IS VERY POWERFUL. HERE'S THE CODE:

THE <path> ELEMENT WORKS AS IF IT WAS AN IMAGINARY PEN. YOU THEN USE THE ATTRIBUTE TO PASS A SERIES OF COMMANDS TO THE PEN TO TELL IT WHAT TO DRAW.

```
<path d="M5,50
```

MOVE TO COORDS 5,50.

```
l0,100 l100,0 l0,-100 l-100,0
```

DRAW A LINE TO -100,0 RELATIVE TO THE CURRENT POSITION.

UPPERCASE LETTERS MEAN ABSOLUTE COORDINATES.

```
M215,100
```

```
a50,50 0 1 1 -100,0 50,50 0 1 1 100,0
```

DRAW TWO ARCS TO MAKE A CIRCLE.

LOWERCASE LETTERS MEAN COORDINATES RELATIVE TO THE CURRENT PEN POSITION.

```
M265,50
```

```
l50,100 l-100,0 l50,-100
```

CLOSE THE PATH.

```
z"
style="stroke: rgb(51,51,51);
       fill: rgb(204,204,204);
       stroke-width: 5;"/>
```

It seems like a path can do anything, so why bother to use anything else? The `<path>` element is difficult to understand and manipulate because of its reliance on a single attribute value. In addition, any style will apply to all shapes on the same path, so all your shapes will have the same border and color. Of course, nothing is stopping you from using more than one path with a different style applied to each.

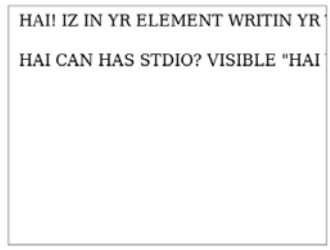
Images, text, and embedded content

Images are easy to embed within your SVG drawing. The syntax is similar to that of HTML, and the only additional information you need to provide over and above the `<image>` element are the coordinates of the upper-left corner:

```
<image x="10" y="10"
       width="236" height="260"
       xlink:href="example.png">
</image>
```

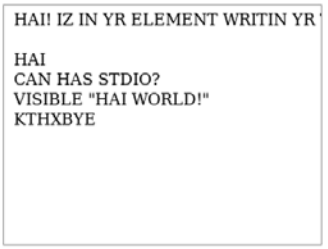
You use an `xlink:href` to link to the image. The `xlink` is a namespace, a legacy of SVG's XML heritage that leaks through to HTML5; more on that shortly. Text is handled a little differently in SVG compared to HTML. In HTML, any text within the body is rendered to the screen—no special wrapping is required. In SVG, text has to be explicitly wrapped within a containing element:

```
<text x="10" y="20">
    HAI! IZ IN YR ELEMENT WRITIN
    YR TXT
</text>
<text x="10" y="60">
    HAI
    CAN HAS STDIO?
    VISIBLE "HAI WORLD!"
    KTHXBYE
</text>
```



The previous example highlights another problem: text that won't fit in the view isn't automatically wrapped. Line breaks also have to be explicitly coded using the <tspan> element:

```
<text x="10" y="20">
    HAI! IZ IN YR ELEMENT WRITIN
    YR TXT
</text>
<text x="10" y="60">
    <tspan x="10">HAI</tspan>
    <tspan x="10" dy="20">
        CAN HAS STDIO?
    </tspan>
    <tspan x="10" dy="20">
        VISIBLE "HAI WORLD!"
    </tspan>
    <tspan x="10" dy="20">
        KTHXBYE
    </tspan>
</text>
```



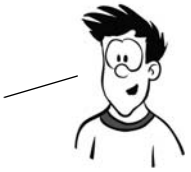
A nice effect you can achieve on short runs of text is to make the text follow a path. If you extract the circle part of the path from the earlier example, you can spread the text along it with the <textpath> element:

```
<defs>
  <path id="myTextPath"
        d="M215,100
           a50,50 0 1 1
          -100,0 50,50 0 1 1
           100,0">
  </path>
</defs>
<text>
  <textPath
      xlink:href="#myTextPath">
      HAI! IZ IN YR ELEMENT
      WRITIN YR TXT
  </textPath>
</text>
```

The path is created in the `<defs>` element, and then you link to it using an `xlink:href` like you used for the image earlier. The link works like other web content, so you could refer to the path in a separate file if you wanted to.

You can also apply gradient fills and any number of other SVG effects to the text. We'll cover this in detail in the next section, but this example shows a gradient from transparent light green to solid dark green applied as a fill to a slightly larger version of the circular text. See ch03/svg-10.html for the full code for this example.

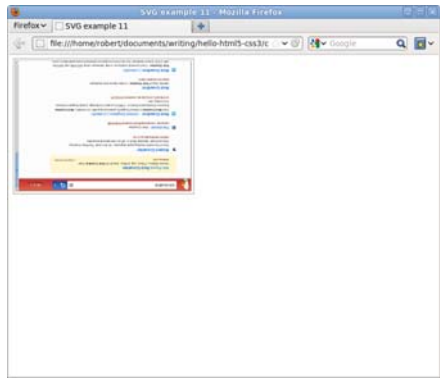
AS WITH THE `<canvas>` ELEMENT, LARGE BLOCKS OF TEXT ARE SOMEWHAT CUMBERSOME IN SVG. THE TEXT ELEMENTS ARE ONLY REALLY USEFUL FOR LABELS AND SHORT DESCRIPTIONS. BUT SVG OFFERS AN ALTERNATIVE—YOU CAN EMBED HTML CONTENT INSIDE ANY ELEMENT WITH `<foreignObject>`.



```
<rect x="5" y="5" width="10" height="160"
      style="stroke-width: 5; stroke: rgb(102,102,102); fill: none;">
</rect>
<foreignObject x="10" y="10" width="100" height="150">
    <body>
      <p>
        <strong>HAI!</strong><br/>
        IZ IN YR ELEMENT
        <em>WRAPPIN</em> YR TXT
      </p>
    </body>
</foreignObject>
```

Everything inside the `<foreignObject>` element is HTML; and unlike the SVG `<text>` element, HTML can cope with wrapping text just fine by itself. It's important to remember that the browser isn't rendering the contents of `<foreignObject>` as if they were HTML; the content is HTML and can be interacted with in the normal way.

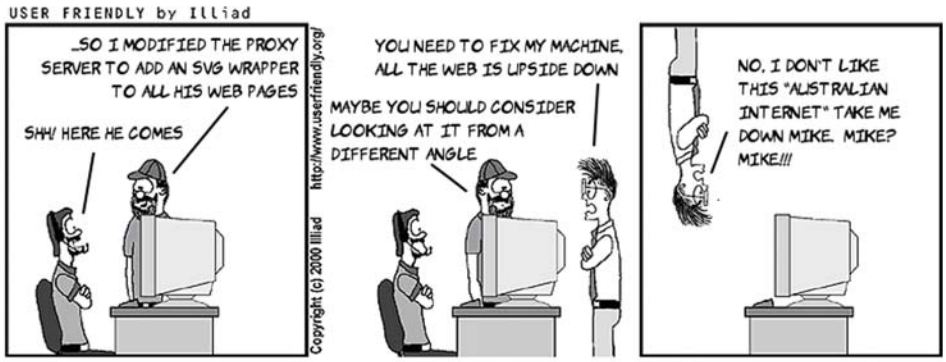
A second example will make this clearer.



On the right is a screenshot of the entire browser window; on the left it's zoomed in to just the content of the `foreignObject` element. The Duck Duck Go home page has been scaled down and rendered upside down inside the browser, but it's still possible to type search terms and see results returned (even if they're too small to read!). This was achieved by wrapping an HTML document inside a `<foreignObject>` element in SVG and then applying some transforms:

```
<svg id="mysvg" viewBox="0 0 800 600">
    <g transform="rotate(180) translate(-800,-600)">
        <foreignObject x="10" y="10" width="800" height="600">
            <body>
                <iframe src="http://duckduckgo.com/"
                    style="width:780px;height:580px">
                </iframe>
            </body>
        </foreignObject>
    </g>
</svg>
```

This example shows a few things you've seen before. The `viewBox` is set to 800 x 600 pixels, even though the element is 320 x 240 pixels; this takes care of the scaling. And an `<iframe>` element is used inside the `<foreignObject>` to fetch the Duck Duck Go page. New in this example are the `<g>` element for grouping SVG content and the `transform` attribute, both of which we'll look at in the next section.



Transforms, gradients, patterns, and declarative animation

SVG is a huge topic, worthy of a book by itself, and we've barely scratched the surface so far. In this section, we'll finish by taking a quick look at some of the more advanced features.

You saw a transform in action in the last example of the previous section. Here it is again:

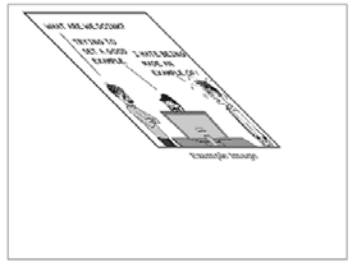
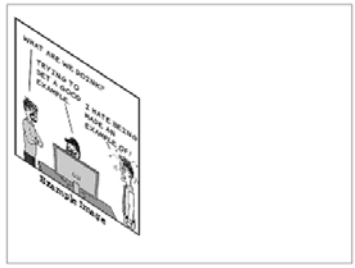
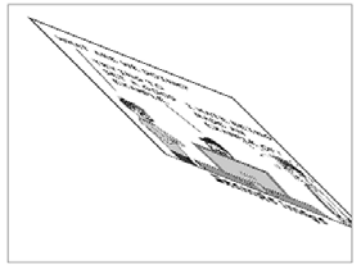
```
<g transform="rotate(180) translate(-800,-600)">
```

The `transform` attribute accepts a space-separated list of commands that are applied in order. The element is rotated 180 degrees and then, because the rotation point by default is the upper-left corner, it's moved back into view with the `translate` transform. You could instead pass a set of coordinates to the `rotate` transform and achieve the same result in a single step:

```
<g transform="rotate(180,400,300)">
```

In addition to `rotate` and `translate`, there are several other transformation commands:

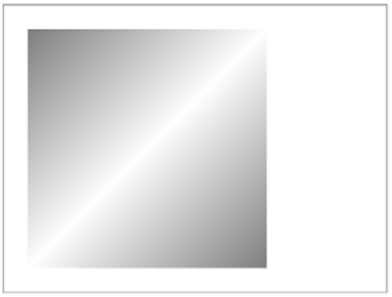
- `scale()`—You've seen examples of scaling already. Earlier examples scaled the entire `viewBox`. This command allows you to control it for specific elements.
- `matrix()`—This is a powerful transformation that allows you to emulate all the others in combination, if you understand the mathematics of matrix transformations. If, like me, you missed that particular part of the curriculum, it's easiest to stick to the other transformations.
- `skewX()` and `skewY()`—See the following table.

	No transform	skewX(45)
No transform		
skewY(33)		

GRADIENTS

As with `<canvas>`, an SVG gradient is defined in a separate object. You can define this object at the top of your SVG file or element inside a `<defs>` element, and then reference the gradient object through CSS:

```
<svg viewBox="0 0 320 240">
 <defs>
  <linearGradient id="grad1"
   x1="0%" y1="0%" x2="100%"
   y2="100%">
    <stop offset="0%" style="
     stop-color:rgb(127,127,127);
     stop-opacity:1"/>
    <stop offset="50%" style="
     stop-color:rgb(255,255,255);
     stop-opacity:1"/>
    <stop offset="100%" style="
     stop-color:rgb(127,127,127);
     stop-opacity:1"/>
  </linearGradient>
 </defs>
 <rect x="20" y="20"
    width="200" height="200"
    style="fill: url(#grad1)">
 </rect>
</svg>
```



The `<rect>` element references the `grad1` gradient through its `fill` style. See the full listing in the ch03/svg-15.html file in the code download.

PATTERNS AND MASKS

You might expect that you could create a repeating background by specifying something like `fill="url(example.png)"`, but that won't work. You have to add the image to a `<pattern>`:

```
<defs>
  <pattern id="img1"
   patternUnits="userSpaceOnUse"
   width="315" height="212">
    <image xlink:href="uf009705.png"
     x="0" y="0"
     width="305" height="212">
  </pattern>
</defs>
```

Then use the pattern to fill:

```
<path d="M5,50
     l0,100 l100,0 l0,-100 l-100,0
     M215,100
     a50,50 0 1 1 -100,
     0 50,50 0 1 1 100,0
     M265,50
     l50,100 l-100,0 l50,-100
     z"
  fill="url(#img1)">
```

The full listing is in ch03/svg-16.html.

You can apply the same pattern to a `<text>` element, although you should pick your image carefully to ensure that things are readable:

```
<text x="0" y="120"
 font-family="sans-serif"
 font-size="80"
 font-weight="bold"
 fill="url(#img1)" >
    <tspan>HTML5</tspan>
    <tspan x="0" y="180"
         font-size="70">
           ROCKS!
    </tspan>
</text>
```

This code is taken from the file ch03/svg-17.html.

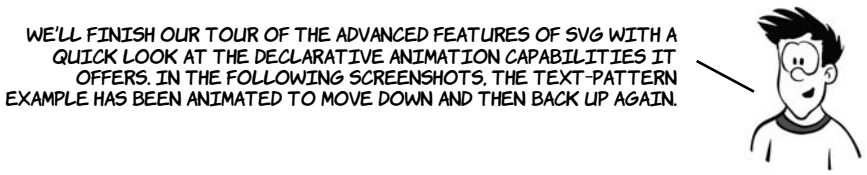
SVG is a large specification, and there's more than one way to achieve this same effect. Instead of applying the image as a background to the text, the text can be used to clip the image. To create a mask, the main change is that the text should be filled with white:

```
<mask id="img1" clipPathUnits="userSpaceOnUse" width="320"
height="200">
    <text x="0" y="120" font-family="sans-serif"
      font-size="80" font-weight="bold" fill="white">
        <tspan>HTML5</tspan>
        <tspan x="0" y="180" font-size="70">ROCKS!</tspan>
    </text>
</mask>
```

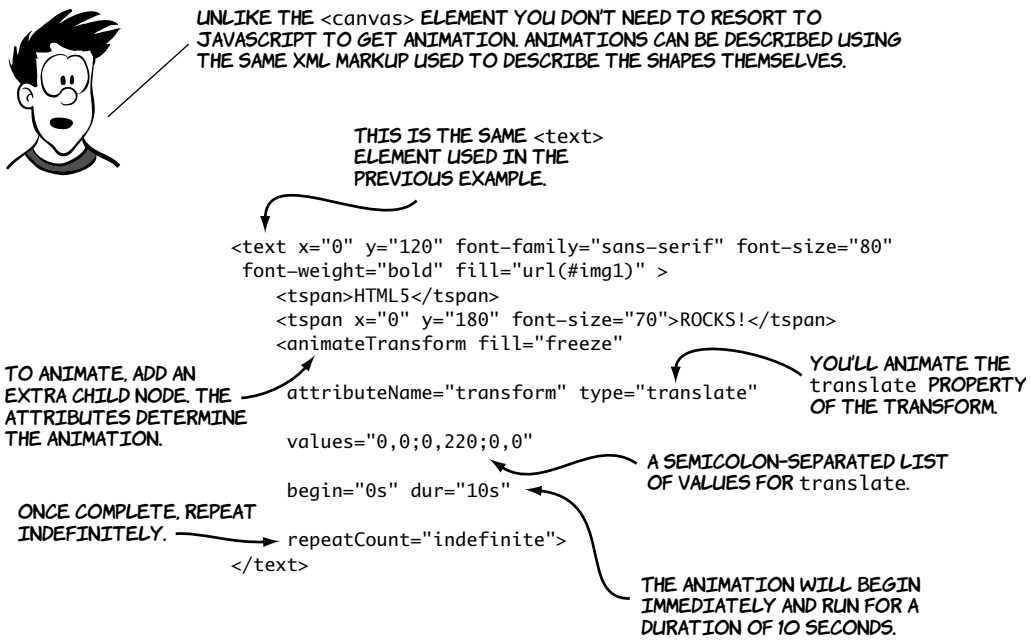
Then attach the mask to the `<image>` element with the `mask` attribute:

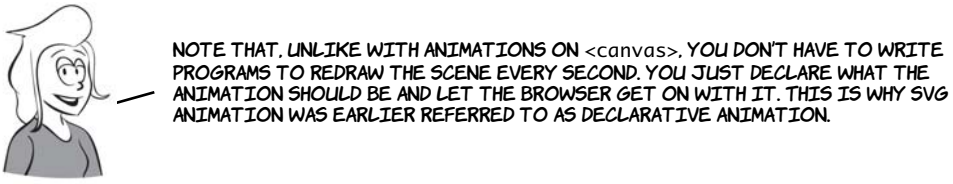
```
<image xlink:href="uf009705.png"
  mask="url(#img1)"
  x="-10" y="-5"
  width="340" height="220" />
```

The image is positioned to approximate the previous example; see the code in ch03/svg-17-clippath.html.



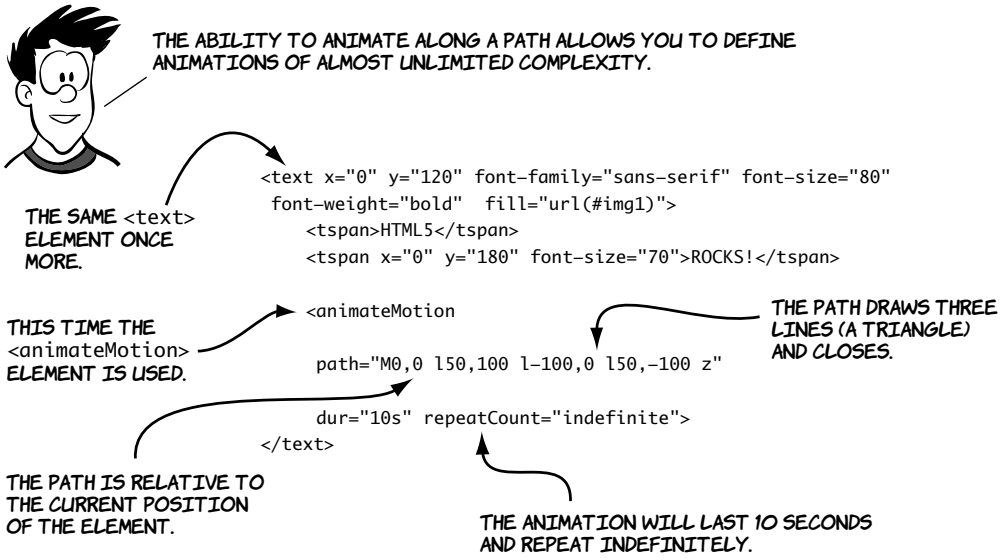
Let's look in detail at how this is done in the listing from ch03/svg-18.html.



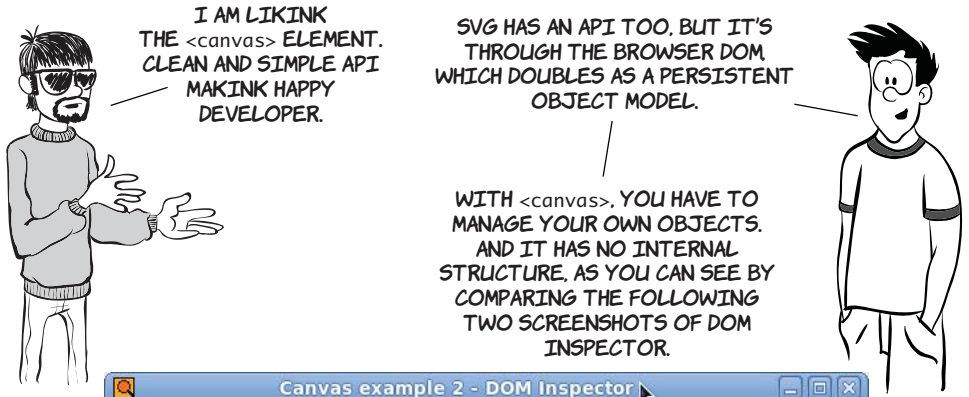


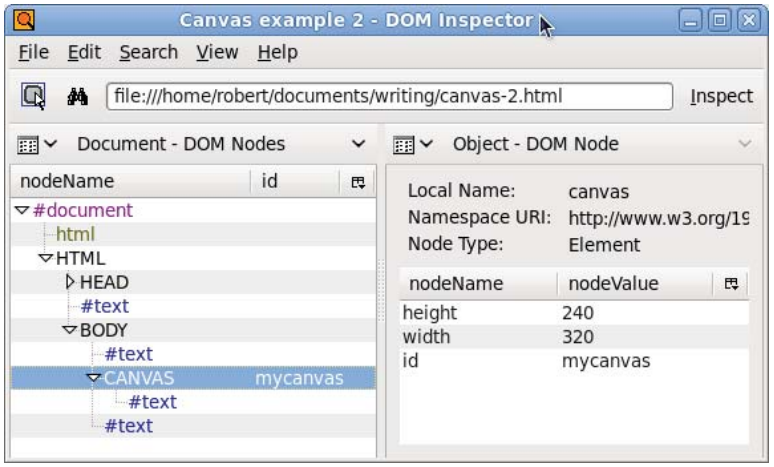
SVG animations aren't limited to simple attribute manipulations. Just as you were able to make text follow a path, it's also possible to make an animation follow a path. Here's an animation around a triangle.

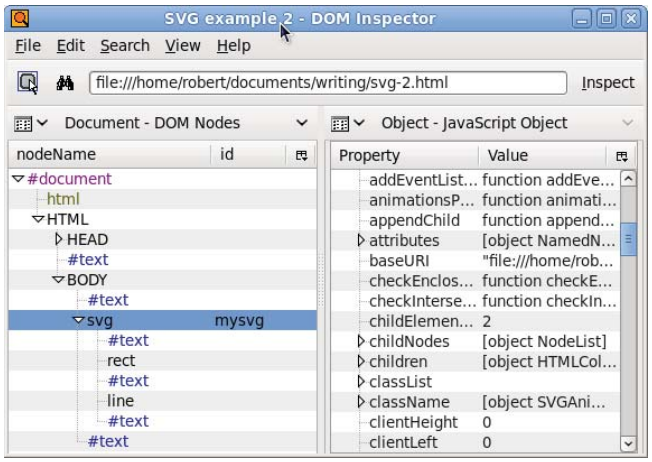
You can see the changes for yourself in ch03/svg-19.html. Here are the key points:

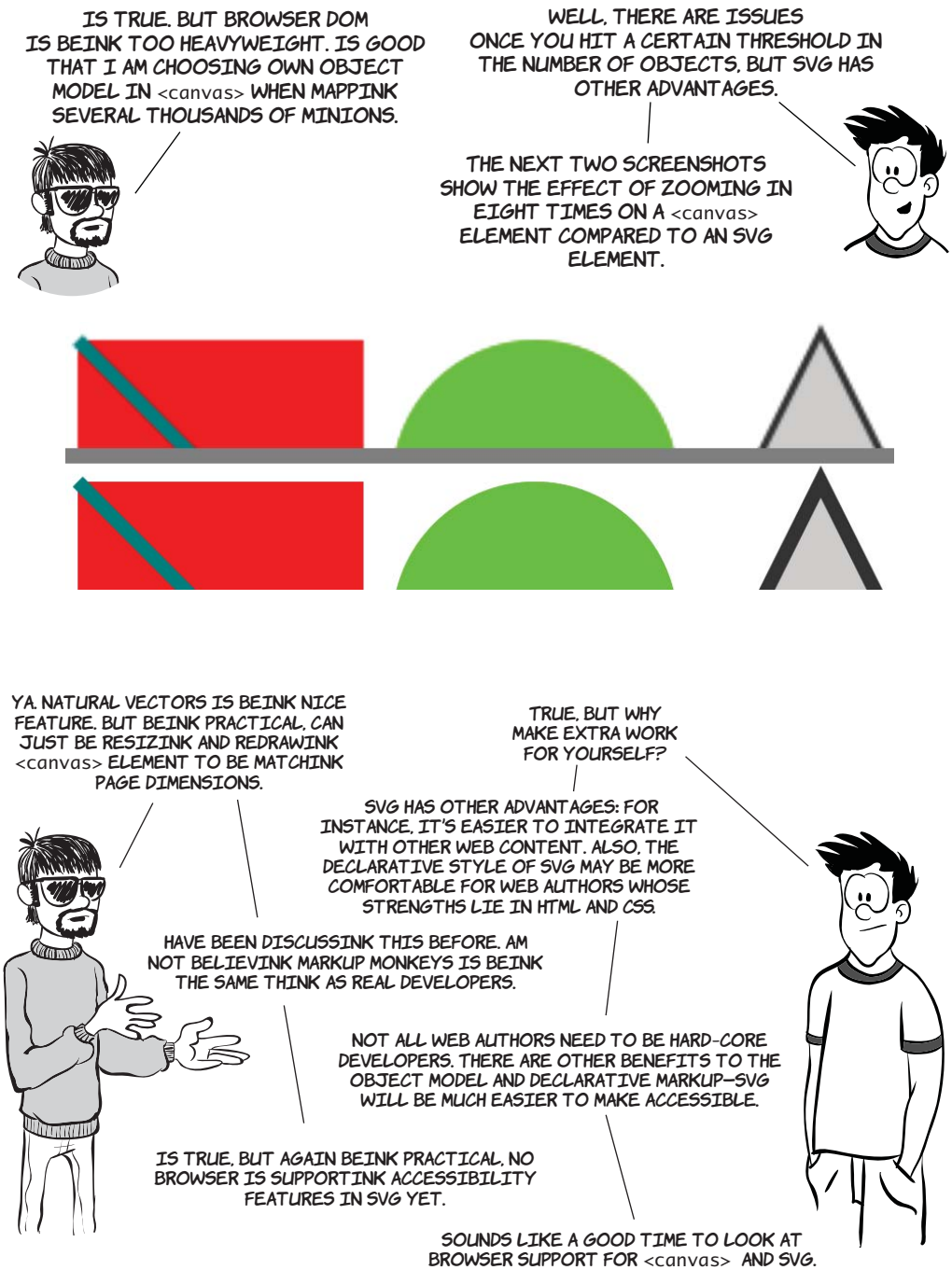


SVG vs. <canvas>








IS TRUE. BUT BROWSER DOM IS BEINK TOO HEAVYWEIGHT. IS GOOD THAT I AM CHOOSING OWN OBJECT MODEL IN <canvas> WHEN MAPPINK SEVERAL THOUSANDS OF MINIONS.
WELL, THERE ARE ISSUES ONCE YOU HIT A CERTAIN THRESHOLD IN THE NUMBER OF OBJECTS, BUT SVG HAS OTHER ADVANTAGES.
THE NEXT TWO SCREENSHOTS SHOW THE EFFECT OF ZOOMING IN EIGHT TIMES ON A <canvas> ELEMENT COMPARED TO AN SVG ELEMENT.
YA. NATURAL VECTORS IS BEINK NICE FEATURE. BUT BEINK PRACTICAL, CAN JUST BE RESIZINK AND REDRAWINK <canvas> ELEMENT TO BE MATCHINK PAGE DIMENSIONS.
TRUE, BUT WHY MAKE EXTRA WORK FOR YOURSELF?
SVG HAS OTHER ADVANTAGES: FOR INSTANCE, IT'S EASIER TO INTEGRATE IT WITH OTHER WEB CONTENT. ALSO, THE DECLARATIVE STYLE OF SVG MAY BE MORE COMFORTABLE FOR WEB AUTHORS WHOSE STRENGTHS LIE IN HTML AND CSS.
HAVE BEEN DISCUSSINK THIS BEFORE. AM NOT BELIEVINK MARKUP MONKEYS IS BEINK THE SAME THINK AS REAL DEVELOPERS.
NOT ALL WEB AUTHORS NEED TO BE HARD-CORE DEVELOPERS. THERE ARE OTHER BENEFITS TO THE OBJECT MODEL AND DECLARATIVE MARKUP—SVG WILL BE MUCH EASIER TO MAKE ACCESSIBLE.
IS TRUE. BUT AGAIN BEINK PRACTICAL, NO BROWSER IS SUPPORTINK ACCESSIBILITY FEATURES IN SVG YET.
SOUNDS LIKE A GOOD TIME TO LOOK AT BROWSER SUPPORT FOR <canvas> AND SVG.

Browser support

Both <canvas> and SVG have wide support in current browsers, with prospects for even better support in the respective next releases. <canvas> support tends to be all or nothing, but the situation with SVG is a lot more complex.

The SVG spec itself is about as complex as the HTML one, and no browser fully supports it, so the following table lists the percentage of the W3C SVG test suite that each browser passes. Figures aren't available for all browser versions, so the results for the most recent test in each browser are shown (thanks to www.codedread.com/svg-support.php for the figures).

											
	12	14	4	6	8	9	10	11.5	12	5	5.1
<canvas>	•	•	•	•		•	•	•	•	•	•
<canvas> text	•	•	•	•		•	•	•	•	•	•
SVG score	89.23%		82.30%		-	59.64%		95.44%		82.48%	
SVG as image	•	•	•	•		•	•	•	•	•	•
SVG in CSS	•	•	•	•		•	•	•	•	•	•
SVG as object	•	•	•	•		•	•	•	•	•	•
SVG in XHTML	•	•	•	•		•	•	•	•	•	•
SVG in HTML	•	•	•	•		•	•		•		•

Key:

- • Complete or nearly complete support
- ○ Incomplete or alternative support

  Little or no support

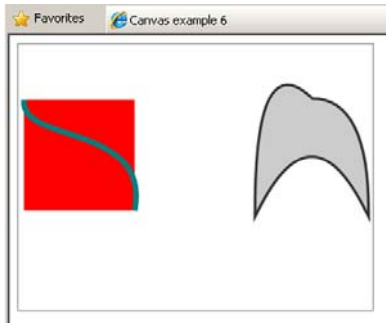
Supporting <canvas> in older versions of IE with explorercanvas

Internet Explorer was the only major browser that had no support for the <canvas> element, although support has been added in IE9. But older versions of IE have support for Vector Markup Language

(VML). VML is a predecessor of SVG, and you've already seen that SVG and `<canvas>` can do a lot of similar things. The `explorercanvas` library implements `<canvas>` in IE8 and earlier using VML. Activating `explorercanvas` is as simple as including a `<script>` element in the head of your HTML document:

```
<head>
<!--[if IE lte 8]><script src="excanvas.js"></script><![endif]-->
</head>
```

If you add that to any of the examples you've seen in this chapter, you should see them rendering in IE8 as the screenshot at right.



SVG in XML vs. SVG in HTML

I mentioned earlier that SVG support isn't as clear-cut as `<canvas>` support. This isn't just because the SVG specification is more complex but also because there are more ways to use SVG from within a web page. This is largely because SVG was originally envisioned as one of a family of XML-based languages that would be used for web content.

In nearly all the major browsers, it has long been possible to embed SVG content in the XML version of HTML/XHTML. Unfortunately, there has been one major obstacle to this happening.

FULLY COMPLIANT XHTML SHOULD BE DELIVERED FROM THE SERVER AS XML CONTENT. THE SERVER TELLS THE BROWSER THE CONTENT TYPE OF THE FILE IN THE HEADER OF THE HTTP RESPONSE. UNFORTUNATELY, IF YOU TRY TO SEND AN XML WEB PAGE TO A VERSION OF INTERNET EXPLORER EARLIER THAN 9, IT REFUSES TO PARSE THE PAGE. BECAUSE DEPLOYING SVG IN XHTML REQUIRES BREAKING IE, FEW PEOPLE HAVE CONSIDERED IT PRACTICAL.

Embedding SVG as an image

SVG can be used in the `<img>` element in the same way as any other image format:

```
<img src="svg-2.svg">
```

When used this way, SVG still has the advantage of being scalable; you can set it to take up half the browser window, and it will remain sharp no matter how high or low your user's screen resolution. But you lose the advantage of being able to manipulate the image from the JavaScript—the elements of the image aren't present in the DOM.

Referencing an SVG image from CSS

In the same way that it can be used as an image in HTML, SVG can be referenced as an image in CSS:

```
div { background: url(svg-2.svg) top right no-repeat; }
```

Embedding SVG as an object

The `<object>` element is a general-purpose method to embed any external content in your web page. To embed SVG with `<object>`, you need to supply two parameters specifying the filename and the file type:

```
<object type="image/svg+xml" data="svg-2.svg"></object>
```

In browsers with native support for SVG, the object-embedding approach has results similar to including the SVG inline: the SVG elements are available in the DOM and can be manipulated. This technique works in every browser that has SVG support; and if you're using the same SVG image on different pages of your site, it's cached the same way a normal image would be, making your site load slightly faster. The corollary of this, of course, is that if you use the image only once it will require a second request to the server, making your site slightly slower to load.

SVG support in older browsers with SVG Web and Raphaël

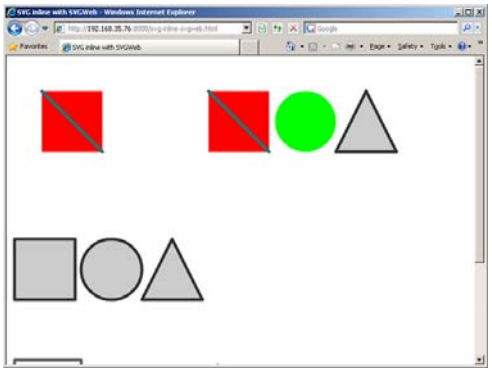
SVG Web is a JavaScript library that, if it detects the browser has no native support for SVG, will replace any SVG graphics it finds with a Flash movie. The Flash movie will then take care of rendering the SVG in the browser. You have to make some slight modifications to your web page in order to enable SVG Web. The first is in the head of the document, where you reference the SVG Web JavaScript library:

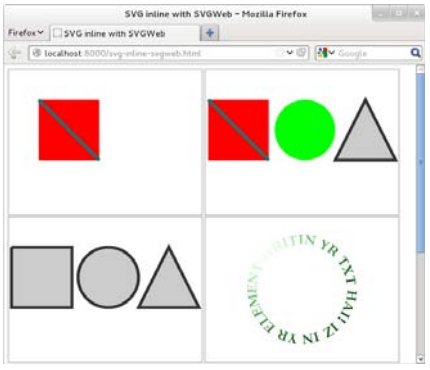
```
<script src="svg.js"></script>
```

Then you have to surround each of your SVG graphics with `<script>` tags:

```
<script type="image/svg+xml">
    <svg viewBox="0 0 320 240">
        <rect x="50" y="50" width="100" height="100"
              style="fill: rgb(255,0,0)"></rect>
        <line x1="50" y1="50" x2="150" y2="150"
              style="stroke: rgb(0,127,127); stroke-width: 5;"></line>
    </svg>
</script>
```

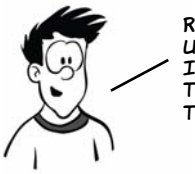
Your SVG graphics will then render as SVG in browsers that support it and as Flash movies in browsers that don't support SVG. In the following examples, at left you can see that SVG Web allows Internet Explorer to render inline SVG, although it doesn't match the native support offered by browsers such as Firefox (shown on the right).





The Raphaël JavaScript library takes a different approach. Instead of making existing SVG work in IE, it presents an API for creating graphics. In Firefox, Chrome, Safari, and Opera it creates SVG; in IE, it creates VML. The interface Raphaël provides looks similar to the <canvas> API:

```
var paper = Raphael(10, 50, 320, 200);
var circle = paper.circle(50, 40, 10);
circle.attr("fill", "#f00");
circle.attr("stroke", "#fff");
```

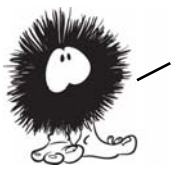


RAPHAËL LOOKS SIMILAR TO <canvas>. BUT IT'S STILL SVG UNDERNEATH. THIS MEANS THAT WHEN YOU CALL THE CIRCLE FUNCTION, IT RETURNS AN OBJECT. THIS OBJECT CAN LATER BE MODIFIED, AND THE DRAWING WILL UPDATE TO REFLECT THE CHANGES; YOU DON'T HAVE TO CLEAR EVERYTHING AND REDRAW IT AS YOU DO WITH <canvas>.

Summary

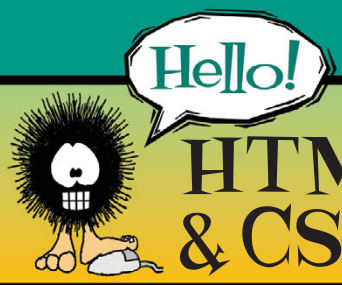
In this chapter you've learned how you can generate graphics in your web page on the fly using two different HTML5 technologies—<canvas> and SVG. Because both can be created and updated dynamically, they don't need the user to reload the page in order to present new information to the user.

You've learned the basic techniques for drawing shapes and lines with both technologies, as well as how to import images and apply effects and transformations. With <canvas> you've looked at how to do simple animation while with SVG you saw how you can import whole web pages and apply transformations to them.



NOW THAT YOU CAN CREATE YOUR OWN GRAPHICS ON THE FLY, IT'S TIME TO COMPLETE YOUR EDUCATION ON THE MULTIMEDIA POSSIBILITIES OF HTML5 WITH A LOOK AT THE NEW AUDIO AND VIDEO ELEMENTS. IN THE NEXT CHAPTER, YOU'LL SEE THAT HTML5 MAKES ADDING AUDIO AND VIDEO TO WEB PAGES AS EASY AS ADDING IMAGES TO WEB PAGES IS IN HTML4.

WEB DEVELOPMENT/HTML



HTML5 & CSS3

Rob Crowther

Free eBook
see insert

"A fast-paced introduction. Recommended to anyone who needs a quick-start resource."
—Jason Kaczor, Microsoft MVP

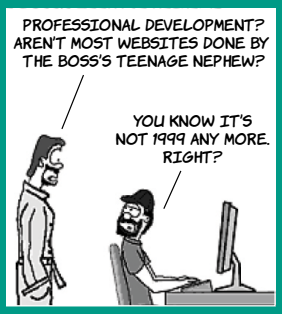
"Everything you need to know explained simply and clearly."
—Mike Greenhalgh, NHS Wales

"It's 2012. You need this book!"
—Greg Donald, CallProof, LLC

"Level up your web skills!"
—Greg Vaughn, LivingSocial

Whether you're building web pages, mobile apps, or desktop apps, you need to learn HTML5 and CSS3. So why wait? **Hello! HTML5 & CSS3** is a smart, snappy, and fun way to get started now.

In this example-rich guide to HTML5 and CSS3, you'll start with a user-friendly introduction to HTML5 markup and then take a quick tour through forms, graphics, drag-and-drop, multimedia, and more. Next, you'll explore CSS3, including new features like drop shadows, borders, colors, gradients, and backgrounds. Every step of the way, you'll find hands-on examples, both large and small, to help you learn by doing.



What's inside

- Easy-to-follow intro to HTML5 and CSS3
- Fully illustrated and loaded with examples
- Designed for low-stress learning
- No prior experience needed!

Don't worry—you aren't alone! The cast of characters from ***User Friendly*** is learning HTML5 and CSS3 along with you as you read.

Rob Crowther is a web developer and blogger from London.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/HelloHTML5andCSS3

ISBN 13: 978-1-935182-89-4
ISBN 10: 1-935182-89-7
53999
9 781935 182894

MANNING US $39.99 / Can $41.99