HOUR 10

Drawing with the HTML5 Canvas Element

What You'll Learn in This Hour:

- How to use the `<canvas>` element to draw on web pages
- How to create lines, rectangles, and circles on canvases
- How to use images as portions of the canvas or as patterns
- What mobile devices support `<canvas>`
- How canvas differs from Flash and SVG

The HTML5 `<canvas>` element lets you use JavaScript to draw shapes, add images, and create animations right in your web page. It is not done with a different language (such as SVG) nor does it use a plug-in (like Flash).

In this hour you will learn what the `<canvas>` element is and how you can use it to create shapes and add images and text right inside the web browser. This hour serves as a jumping-off point for how to use the `<canvas>` element—there is even more out there that you can do with it. You are limited only by your imagination.

Using the Canvas Element

In a nutshell, the HTML5 `<canvas>` element lets you draw whatever you want on your web page using JavaScript. You can use it to add images, create slideshows, display animations, and even build games.

The `<canvas>` element can be scripted. This means that you can create a canvas that changes based on user input. Although most canvas applications have focused on games, you can use it for other things, including:

- Dynamic graphs such as stock tickers
- Photo galleries
- Fancy fonts
- Online visual tools such as mind maps and image editors

Anything you can draw or animate you can do in a `<canvas>` element. However, like every part of HTML5, you should only use it where it's appropriate. For example, you wouldn't use a `<canvas>` element to insert every image on your page, nor should you use it to add your page header, but you can use it to display dynamic graphs or create online games. Instead you would use the `<img>` and `<header>` tags to define those items.

The `<canvas>` element is easy to add to your HTML documents:

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

This line creates a blank canvas in the browser. Because it has no width, height, or content, it doesn't display anything on the screen. Most of the time, you will also want to specify a width and height and give your canvas an ID so you can reference it in your scripts:

```
<canvas width="350" height="450" id="canvas1"></canvas>
```

Of course, if that's all you write, there will simply be a blank 350 x 450-pixel space in your HTML.

By the Way

See Your Empty Canvases

When working with the `<canvas>` element for the first time, setting a border around it so that you know where your canvases are is easiest. Add a line to your style sheet—`canvas { border: solid thin black; }`—to add a thin border around every canvas on your page. When you're done editing, you can delete the style.

The `<canvas>` element is supported by Chrome 3.0+, Firefox 3.0+, Opera 10.0+, and both iOS and Android 1.0 and up. Internet Explorer 9 supports it natively, and IE 7 and 8 need a plug-in such as ExplorerCanvas (http://code.google.com/p/explorercanvas/).

You should include text inside your `<canvas>` element to display if the element isn't supported by the browser:

```
<canvas>
This page requires an HTML5 compliant browser to render correctly.
Other browsers may not see anything.
</canvas>
```

Basic support for the <canvas> element is quite good on both iOS and Android. In fact, on mobile devices, the only one that is iffy is Opera mini.

When you compare <canvas> to Flash support, you can clearly see which you should choose for your mobile applications. Flash is not supported at all on iOS devices, and it seems likely that this will never change.

When you compare <canvas> to SVG support, the results are clear as well. Although Android 3.0 supports SVG, Android 2.3 does not, so unless you are willing to leave out the Android phone devices, <canvas> is still a better choice.

Drawing Shapes on the <Canvas> Element

To have the canvas show anything, you need to script it using JavaScript. To use JavaScript you have to attach the script to an event. An easy event to use is `onclick`. When a user clicks on your canvas, it will draw something. For example:

```
function drawSquare() {
    var canvas = document.getElementById("canvas1");
    var context = canvas.getContext("2d");
    context.fillStyle = "rgb(13, 118, 208)";
    context.fillRect(30, 30, 150, 150);
}
```

Did you Know?

You Can Call Your Script When the Page Loads

Calling your script `onclick` is a way to make the shapes more interactive, but if you need the canvas to be drawn when the page is live, you should draw it when the body loads by calling it in the body tag: `<body onload="drawSquare();">`.

This draws a blue square on the canvas called "canvas1." You call that function on the <canvas> element in your document:

```
<canvas id="canvas1" width="200" height="200"
➥onclick="drawSquare();"></canvas>
```

The script at the beginning of this section ("Drawing Shapes on the <Canvas> Element") does four things:

- **line 1**—Finds the "canvas1" element in your document
- **line 2**—Sets the context to two-dimensions
- **line 3**—Defines the fill color as blue
- **line 4**—Draws a rectangle with four equal sides—a square

Watch Out!

You Must Set the Context

To draw on a <canvas> element, you must pass the string "2d" to the getContext() method. Otherwise, your <canvas> element will not display anything. This way, when other contexts become available, you will be able to take different actions on your canvas elements.

You can then clear the entire canvas by setting the width or height. You don't have to change the width or height, just set it to itself, as described in the following Try It Yourself.

Try It Yourself

Making a Square Appear and Disappear

You can modify the preceding script to make the square appear and disappear.

1. Add the <canvas> element to your document:

```
<canvas id="canvas1" width="200" height="200" style="border: solid 1px
➥black;"
onclick="drawSquare();">
</canvas>
```

2. Put in the JavaScript to draw the square when the <canvas> element is clicked on:

```
function drawSquare() {
    var canvas = document.getElementById("canvas1");
    var context = canvas.getContext("2d");
    context.fillStyle = "rgb(13, 118, 208)";
    context.fillRect(30, 30, 140, 140);
}
```

3. Create an erase function in the JavaScript:

```
function eraseSquare() {
    var canvas = document.getElementById("canvas1");
    canvas.width = canvas.width;
}
```

4. Call the erase function on a double-click in the <canvas> element:

```
ondblclick="eraseSquare();"
```

Drawing Rectangles

Rectangles and squares are the easiest shapes to draw with the <canvas> element because several functions are designed just for building them. The three functions for drawing rectangles are

- **`fillRect`**—For drawing a filled rectangle
- **`strokeRect`**—For drawing a rectangular outline
- **`clearRect`**—For making an empty, transparent rectangular shape

Figure 10.1 shows these three functions used on one canvas. The black boxes are two `fillRect` functions, one first filling the entire canvas and the other one last, filling the inner space. The white box is created with a `clearRect` function to bring the `<canvas>` element back to its default color (white). The thin black line is drawn with the `strokeRect` function.

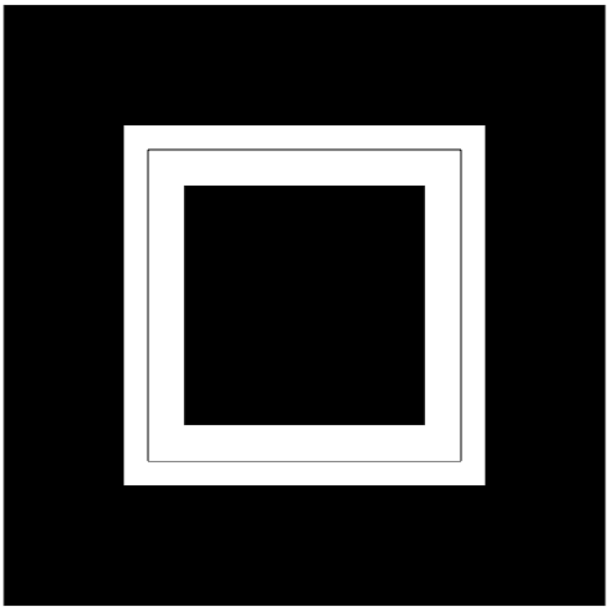


FIGURE 10.1
A canvas with four rectangles drawn on it.

To draw a rectangle, you use any of the three functions with the values `x`, `y`, `width`, and `height`. `x` and `y` specify the position on the canvas, where `0,0` is the upper-left corner. `x` moves the point to the right on the horizontal plane and `y` moves the point down on the vertical plane. The `width` and `height` are the size of the rectangle.

If you draw two shapes on the canvas, they will layer one over the other, with the last one written in the script being on top.

The <canvas> element draws only in black and white unless you set styles. You can set the fill color and the line color with two properties:

- **fillStyle** – The fill color
- **strokeStyle** – The border color

You can use color names, hexadecimal color codes, RGB color values, and RGB with alpha transparency color values. With alpha transparency, you can adjust colors by layering one over another. For example, the following script would create a red box in the upper left of the page, and a faded blue box in the lower right, with the overlap being purple:

```
context.fillStyle = "#ff0000";
context.fillRect(10,10, 300,300);
context.fillStyle = "rgba(0,0,255,0.5)";
context.fillRect(190,190, 300,300);
```

You aren't limited to flat colors with the <canvas> element. You can also create gradients and use them to style your fills and strokes. The two types of gradients are linear and radial. In order to create a gradient you need to use three properties:

- **createLinearGradient**—This takes four arguments defining the x and y coordinates of the starting point and the x and y coordinates of the end point.
- **createRadialGradient**—This takes six arguments. The first three define a circle with x and y coordinates and a radius. The second three define a second circle with x and y coordinates and a radius.
- **addColorStop**—This takes two arguments. The first is the position of the color on the gradient between 0 and 1. 0 is the start of the gradient and 1 is the end. The second argument is the color written in CSS colors, just like the flat colors described earlier.

You can have as many stop colors in your gradient as you like. If you start your fill in the middle of the gradient, that stop color will fill the shape from the start to that stop point. Figure 10.2 shows a linear gradient on top and a radial gradient on bottom.

Here is an example of how to draw a box with a two-colored linear gradient across the diagonal:

```
var linGrad = context.createLinearGradient(0,0, 500,500);
linGrad.addColorStop(0,"red");
linGrad.addColorStop(1,"blue");

// draw gradient box
context.fillStyle = linGrad;
context.fillRect(10,10, 490,490);
```

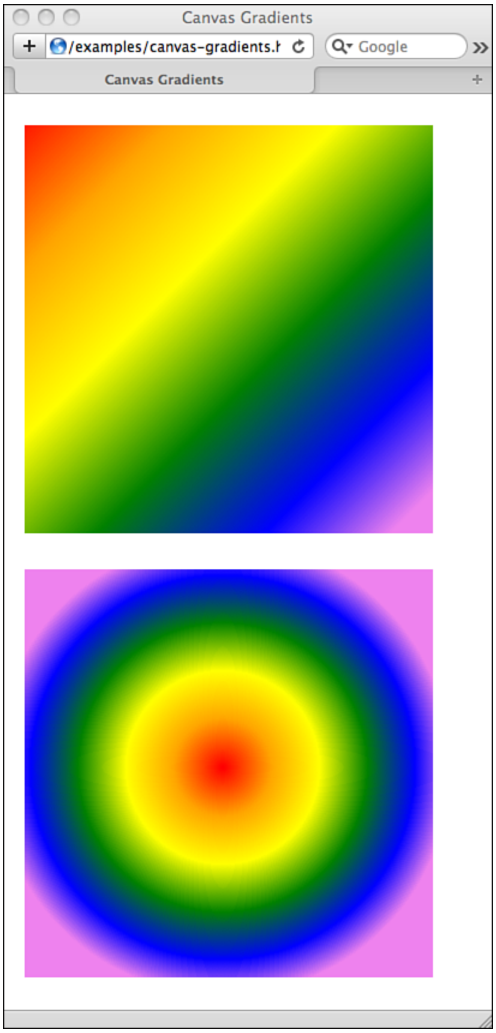


FIGURE 10.2 Two gradients on HTML canvas.

Radial gradients are more complicated than linear gradients because rather than defining two points for the gradient to draw between, you are defining two imaginary circles. The gradient travels in all directions (radially) from the center point of the first circle to the outer edge of the second circle. These two circles define a cone shape that creates the gradient, as shown in Figure 10.3. Circle 1 defines the starting point of the gradient and Circle 2 defines the ending point. The radial gradient travels both along a line and a circle.

This is different than how most graphic programs create radial gradients. In those, the center point of both the first circle and the second circle is the same, so your gradient will always move smoothly out in all directions.

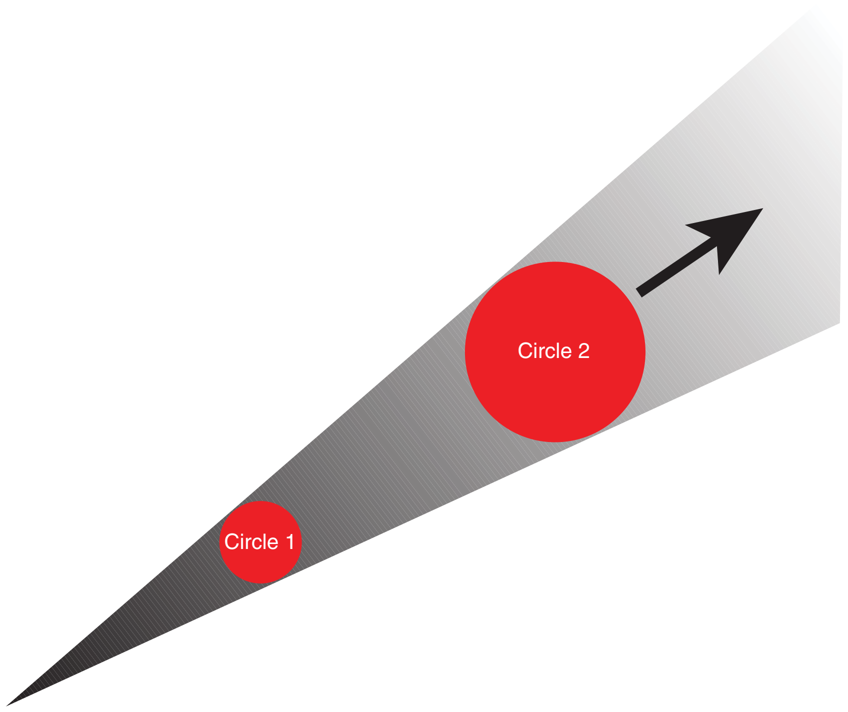


FIGURE 10.3
Diagram of a how a cone-shaped radial gradient works.

To create a radial gradient like the aforementioned one in the `<canvas>` element, position both circles starting at the same x and y coordinates, but make the second one larger than the first.

To define a radial gradient, you use six arguments:

```
createRadialGradient(x1,y1,r1, x2,y2,r2)
```

`x1` and `y1` are the coordinates of the center of the first circle, and `r1` is the radius (in pixels) of that first circle. `x2` and `y2` are the coordinates of the second circle and `r2` is the radius of the second circle.

Because radial gradients are defined as circles, you can use them to draw circles on your canvas. If you want to fill an object that is not a circle, then you need to make sure the second circle completely surrounds the object.

Here is an example of how to draw a circle with a radial gradient:

```
var radGrad = context.createRadialGradient(100,150,0, 100,150,70);
radGrad.addColorStop(0.9, "rgb(105,138,72)");
radGrad.addColorStop(0, "rgba(171,235,108,1)");
radGrad.addColorStop(1, "rgba(105,138,72,0)");
// draw gradient box
context.fillStyle = radGrad;
context.fillRect(10,10, 490,490);
```

You can see this code in action at www.html5in24hours.com/examples/canvas-circles.html.

Drawing Polygons and Lines

To draw other shapes in the `<canvas>` element you use lines or paths. The five methods used to draw and use paths are

- **`beginPath()`**—This method creates the path in the canvas.
- **`closePath()`**—This method draws a straight line from the current point to the start. It won't do anything when paths are already closed or on paths with only one point.
- **`stroke()`**—This draws an outline of your path.
- **`fill()`**—This fills in the shape of your path.
- **`moveTo()`**—This draws nothing, but moves the drawing position to a new location on the canvas.

Always specify your starting position on your paths with a `moveTo()` command first. The `<canvas>` element will treat your first construction that way regardless of what the method actually is, and this will prevent surprising results.

After you have moved your cursor to the starting position, to draw a line you use the `lineTo()` method. The `lineTo()` method takes two arguments: the x and y coordinates where the line should draw. To draw a line, you write:

```
context.beginPath();
context.moveTo(0,0);
context.lineTo(60,60);
context.stroke();
```

By the Way

Fill Closes the Path

If you don't close the path and choose to fill the shape, the shape will close automatically with a straight line from the last point on the path to the first point. You do not need to close the path with the `closePath()` method.

To draw a triangle, you write:

```
context.beginPath();
context.moveTo(20,30);
context.lineTo(500,100);
context.lineTo(250,300);
context.fill();
```

You may need to increase the size of your canvas with the width and height attributes to allow this example to fit.

Try It Yourself

Drawing an Octagon

In this exercise you will use lines to draw the edges of an octagon. This won't draw a perfect octagon but it will create an eight-sided figure.

1. Start your path and move your drawing point to 200,0:

   ```
   context.beginPath();
   context.moveTo(200,0);
   ```

2. Draw a line to 400,0:

   ```
   context.lineTo(400,0);
   ```

3. Draw seven more lines to 600,200; 600,400; 400,600; 200, 600; 0,400; and 0,200:

   ```
   context.lineTo(600,200);
   context.lineTo(600,400);
   context.lineTo(400,600);
   context.lineTo(200,600);
   context.lineTo(0,400);
   context.lineTo(0,200);
   ```

4. Close your path:

   ```
   context.closePath();
   ```

5. Change the fill color to red:

   ```
   context.fillStyle = "#ff0000";
   ```

6. Fill in your octagon:

   ```
   context.fill();
   ```

   If the octagon you build is too big, you may need to increase the size of your canvas to make it fit.

You can also adjust the width of your path lines, the shape of the end of the lines, and how these lines are joined together using the `lineWidth`, `lineCap`, `lineJoin`, and `miterLimit` properties.

The `lineWidth` property changes the width from the default 1 unit (the space between grid marks—essentially a pixel) to whatever positive number you want. The width is centered on the path. To draw a line 2 units wide, you write:

```
context.lineWidth = "2";
```

The `lineCap` property changes how the end point of the lines is drawn. There are three values: `butt`, `round`, and `square`. The default is `butt`. Figure 10.4 shows three lines with the three different caps. The thin line shows the grid line where the path will start. As you can see the `square` and `round` styles extend past that grid line.

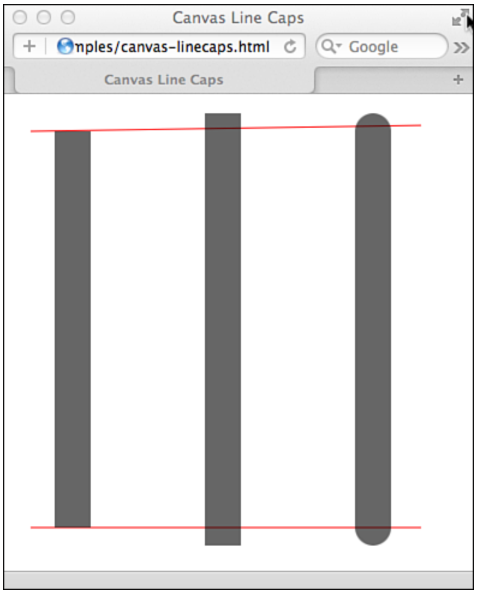


FIGURE 10.4
Three line cap styles: butt, square, and round.

Butt comes right up to the grid line where the path starts. Square adds half the width of the stroke beyond the grid line. Round adds a semicircle with a radius of half the width of the line beyond the grid line.

The `lineJoin` property changes how the lines in a shape are joined together. The three possible values are `round`, `bevel`, and `miter`. The default is `miter`. Figure 10.5 shows the three ways lines can join. The default, `miter`, makes a sharp corner, `bevel` cuts off the tip, and `round` makes it more curved.

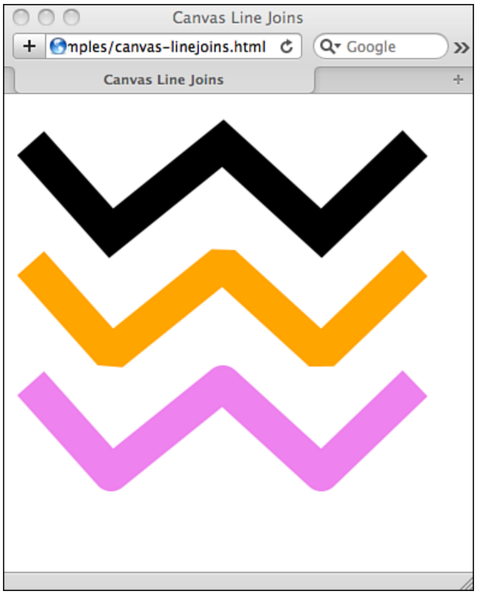


FIGURE 10.5
Three line join styles: miter, bevel, and round.

Try It Yourself

Building a Line with the Three Join Types

You can build a line with one type of join, and then swap it with the other types to see what they look like as well. Follow these steps to create a "W" shape and give the corners different join shapes:

1. Set your line width and begin the path:

```
context.lineWidth = 15;
context.beginPath();
```

2. Move the starting point to slightly inside the canvas and draw a line:

```
context.moveTo(10,20);
context.lineTo(150,200);
```

3. Draw three more lines to create a "W" shape:

```
context.lineTo(290,20);
context.lineTo(430,200);
context.lineTo(570,20);
```

4. Change the line join style:

```
context.lineJoin = "round";
```

5. Stroke the path:

```
context.stroke();
```

Look at your canvas to see the default style of `miter`. Change the line join style to `round` and `bevel` and look at how the corners change. ▲

The `miterLimit` property defines how sharp or dull the miter points are on a join. The higher the miter limit, the sharper your mitered joins can be. For smaller limits, the joins are beveled when they reach that limit.

Drawing Circles

To draw a circle in the `<canvas>` element you use the `arc` method. To understand how to draw the circle, you have to imagine that you are physically drawing the circle with a protractor. You set the point of your protractor in the center of the circle, bend the angle so that the pen is at the radius, start drawing at a point, and lift the pen at a second point. You can draw the circle either clockwise or counterclockwise.

The `<canvas>` element draws a circle in the same way. You set the x and y coordinates for the center of the circle, the radius, the starting point on the circle (in radians), the ending point on the circle (in radians), and finally the direction to draw either clockwise (`true`) or counterclockwise (`false`). The method looks like this:

```
arc(x,y,radius,startAngle,endAngle,true);
```

By the Way

How to Find Start and End Points for Circles and Arcs

Arcs in the `<canvas>` element are measured in radians, not degrees. Radians do not have the same starting point as degrees (in degrees 0° is noon, but in radians it is not). But because most people find thinking in degrees to be easier, having a conversion tool helps. In JavaScript, you can convert degrees to radians with the following expression: `var radians = (Math.PI/180)*degrees`.

To draw a circle, write:

```
var startPoint = (Math.PI/180)*0;
var endPoint = (Math.PI/180)*360;
context.beginPath();
context.arc(200,200,100,startPoint,endPoint,true);
context.fill();
```

Try It Yourself ▼

Drawing Half Circles

Use the arc method of drawing circles, or parts of circles to build a wave pattern with half circles.

1. Decide on the diameter of your circle, and define the radius as half of that:

   ```
   var radius = 125/2;
   ```

2. Define the x and y coordinates of your first circle based on the radius:

   ```
   var y = radius+10;
   var x1 = radius;
   ```

3. Define the coordinates of the subsequent circles as multiples of the radius:

   ```
   var x2 = 3*radius;
   var x3 = 5*radius;
   var x4 = 7*radius;
   ```

4. Draw the first half circle using `Math.PI` as your end point value to get a half circle:

   ```
   context.beginPath();
   context.arc(x1,y,radius,0,Math.PI,true);
   context.stroke();
   ```

5. Draw the second circle, only change the direction to counterclockwise:

   ```
   context.beginPath();
   context.arc(x2,y,radius,0,Math.PI,false);
   context.stroke();
   ```

6. Repeat for as many circles as you want on your canvas:

   ```
   context.beginPath();
   context.arc(x3,y,radius,0,Math.PI,true);
   context.stroke();
   context.beginPath();
   context.arc(x4,y,radius,0,Math.PI,false);
   context.stroke();
   ```

   Figure 10.6 shows how this might look if you added `lineWidth` to make a thicker line.

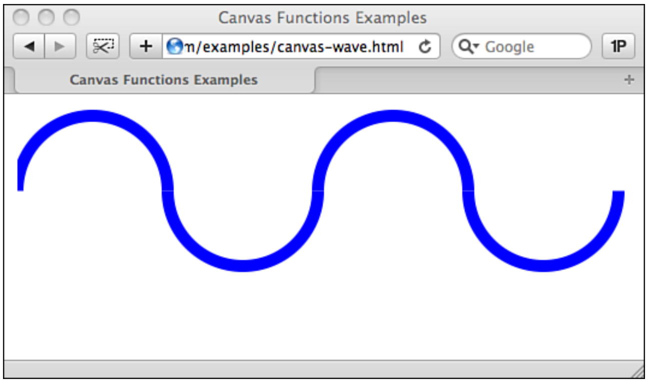

Canvas Functions Examples
m/examples/canvas-wave.html
Google
Canvas Functions Examples

FIGURE 10.6
A wave pattern made with half circles.

You can remove the extra `beginPath()` and `stroke()` methods (all but the first `beginPath()` and last `stroke()`) if you are filling the circles or if you want a line down the middle connecting them all. ▲

Writing Fonts and Text on the Canvas

To draw text on the canvas, use the `fillText()` method to define what and where to draw:

```
context.fillText("Hello World", 250, 100);
```

This will put the text "Hello World" at the x and y coordinates 250,100.

Text on a canvas is drawn on rather than displayed within the CSS box model. You can't define the float, margin, padding, or word wrapping. You just set the font size, family, weight, variant, and line height, and then draw it on your canvas. You can use several attributes to draw text:

- **`font`**—This is anything that you would put in a CSS font property, such as the font family, size, weight, variation, and line height.
- **`textAlign`**—This controls the alignment of your text and is similar to the CSS `text-align` property. It uses the values `start`, `end`, `left`, `right`, and `center`.
- **`textBaseline`**—This controls where the text is drawn relative to the starting position. It takes the values `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, and `bottom`.

Watch Out!

Watch Baselines on Non-English Text

For English text, you can stick with `top`, `middle`, or `bottom` for your `textBaseline` values, but if you are going to be writing in other languages you will need to be aware of where they are anchored so that you use the correct baseline.

You can style your text in the same way you style text in CSS. If you change the font family, you should define multiple families the same way you would in CSS. Separate them with commas and put them in order of preference. You should end with a generic font type such as `serif`, `sans-serif`, or `monospace`. Remember that if your users don't have the font on their device, the `<canvas>` element is not going to provide it for them. You'll learn more about custom fonts in Hour 11, "Fonts and Typography in HTML5."

Here is an example of how to add custom text to a <canvas> element:

```
context.font = "bold 3em/3.5em 'Palatino Linotype', 'Book Antiqua',
➥Palatino, serif";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillStyle = "#f93";
context.fillText("Hello World!", 250, 250);
```

As you can see, you change the color with the `fillStyle` property because you are filling with text with the `fillText` method.

By the Way

Style Canvas Fonts with CSS

Setting the font size in CSS on the <canvas> element will affect the font size of the text. However, other CSS styles may not be applied. Be careful when you apply properties that may be inheritable, because your canvas text may end up looking differently than you expect.

You can add shadows to your drawings with four properties:

- **`shadowOffsetX`**—How far the shadow should extend on the x axis. Negative values move the shadow left.
- **`shadowOffsetY`**—How far the shadow should extend on the y axis. Negative values move the shadow up.
- **`shadowBlur`**—The size of the blurring effect. The default is 0.
- **`shadowColor`**—The color the shadow should be. The default is fully transparent black (`rgba(0,0,0,0)`).

Try It Yourself

Adding a Shadow to Text

Use the shadow properties to add text shadows to a block of text in a `canvas` element.

1. Add some text to your canvas:

```
context.font = "bold 3em/3.5em 'Palatino Linotype', 'Book Antiqua',
➥Palatino, serif";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillStyle = "#f93";
context.fillText("Hello World!", 250, 250);
```

2. Above the text, set the x and y offsets for the shadow:

```
context.shadowOffsetX = -2;
context.shadowOffsetY = 2;
```

3. Define the blur:

   ```
   context.shadowBlur = 2;
   ```

4. Choose a shadow color:

   ```
   context.shadowColor = "rgba(0,0,0,0.5)";
   ```

   Having the `fillText` line last in your code is important, because otherwise the shadow and fill won't show up. You can see an example at www.html5in24hours.com/examples/canvas-text.html.

Displaying Images

To display an image inside a `<canvas>` element you need to reference an image object as a source file, and then draw the image onto the canvas with the `drawImage` function.

You have two choices for the first step. You can access an existing image on the page (in an `<img>` tag), or you can create a new image in the JavaScript, as follows:

```
var img = new Image();
img.src = "images/mydogs.png";
img.onload = function() {
    context.drawImage(img, 10,10);
}
```

As you can see, first you create the image in the JavaScript, and then, after that image has finished loading, draw it on your canvas at the x and y coordinates noted in the `drawImage` method.

Scaling and Clipping Images

You can do more than just display an image with the `<canvas>` element. You can also change the size of the image that is drawn on the canvas (scaling) and the portion of the image that is displayed (clipping).

You use four parameters in the `drawImage` method to change the image size:

```
context.drawImage(x, y, width, height)
```

The x and y coordinates specify where you want the image to be placed on the canvas. The width and height parameters are the new width and height. You can make the image larger or smaller.

Clipping images is a little trickier than scaling them, but it makes sense after you've tried it. Clipping an image adds a mask around the parts of the image that should not display on the canvas, it does not edit the image or crop it. To clip an image you need to indicate the coordinates, width, and height of the area to be clipped and the coordinates, width, and height where it should go on the canvas:

```
context.drawImage(imageID, clipx, clipy, clipwidth, clipheight,
gox, goy, gowidth, goheight)
```

Following is an example of clips used in an image on the canvas:

```
var mydogs = new Image();
mydogs.src = "images/mydogs.png";
mydogs.onload = function() {
    context.drawImage(mydogs, 20,50);
    context.drawImage(mydogs, 148, 14, 92, 120, 20,370, 123,160);
    context.drawImage(mydogs, 235, 122, 65, 85, 298,370, 122,160);
}
```

As you can see in Figure 10.7, some text was added for each image, but this figure displays just one image that was drawn on the canvas in three ways, once as the full image and two other clips of the image.

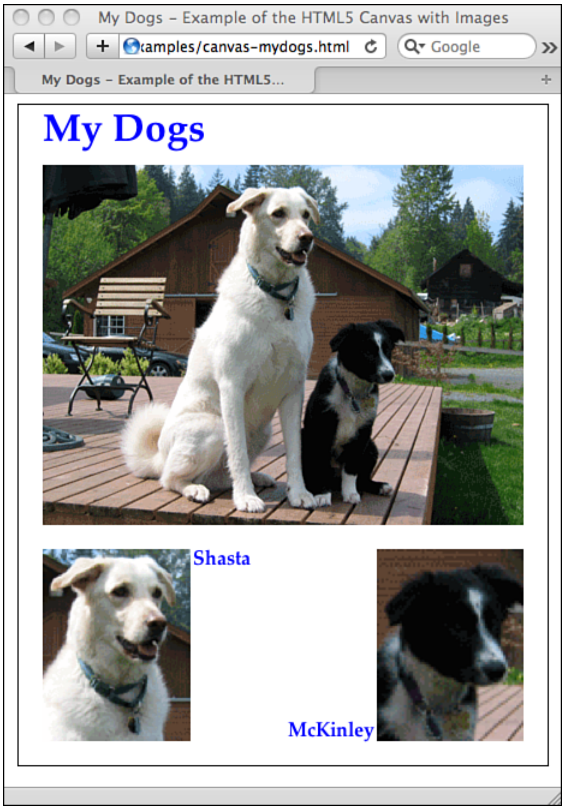

My Dogs – Example of the HTML5 Canvas with Images
xamples/canvas-mydogs.html
Google
My Dogs – Example of the HTML5...
My Dogs
Shasta
McKinley

FIGURE 10.7
Clips used in a <canvas> element.

Remember that scaling is done by the browser, and there is no quality control. Images can become blurry if scaled up too much and grainy if scaled down too much.

Adding Patterns

You can create patterns with your images with the canvas method `createPattern()`. You tell it which image you want to use and how you want it patterned in your canvas:

```
context.createPattern(image, repeat-x)
```

You can set a pattern to `repeat-x` (horizontally), `repeat-y` (vertically), `repeat` (tiled), and `no-repeat`.

Try It Yourself ▼

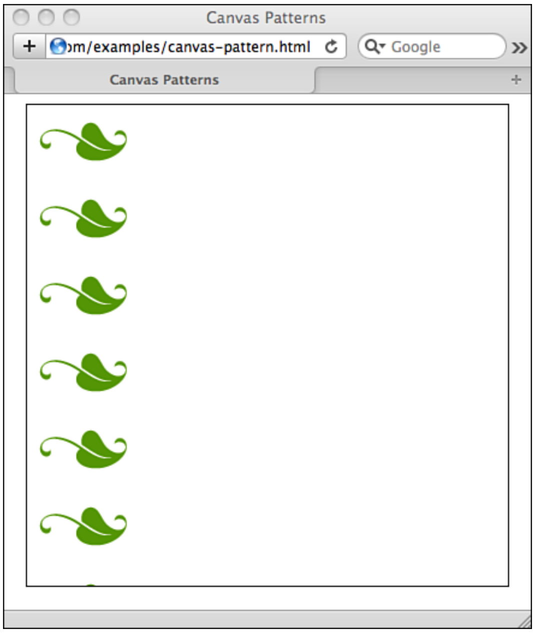
Drawing a Fancy Border on a Canvas

In this exercise you use a single image as a repeated pattern to create a border around the outside of a `canvas`.

1. Create a new image object for your pattern:
   ```
   var pattern = new Image();
   pattern.src = "images/leaf-icon.png";
   ```
2. Open a function when the image loads:
   ```
   pattern.onload = function() {
   ```
3. Create a new pattern:
   ```
   var ptn = context.createPattern(pattern,"repeat-y");
   ```
4. Set the fill style to the pattern:
   ```
   context.fillStyle = ptn;
   ```
5. Build a rectangle and fill with the pattern:
   ```
   context.fillRect(0,0,500,500);
   ```
6. Close the function:
   ```
   }
   ```

Figure 10.8 shows how a border of leaves can be created using just one image and the `canvas` element.

FIGURE 10.8
A border pattern on a canvas.



How Is Canvas Different from SVG or Flash?

The biggest difference between the <canvas> element and SVG or Flash is that <canvas> is an HTML element and as such is built right into the browser and the web page. This makes accessing the <canvas> element contents for scripting and dynamic web applications easy.

SVG is a completely separate language written in XML, and Flash is written in SWF. Although most modern browsers and mobile devices support SVG, Flash does not work on iOS, and Android 2.3 and Internet Explorer 8 do not support SVG. Table 10.1 shows some of the features of Canvas, SVG, and Flash

TABLE 10.1 Different Features of Canvas, SVG, and Flash

	Canvas	**SVG**	**Flash**
Vector graphics	Canvas is bitmap, but can draw vectors.	SVG is vector based, but you can load bitmaps.	Flash is vector based.
Inline HTML	Canvas is a native HTML element.	SVG is XML and must be embedded.	Flash is SWF and must be embedded with a plug-in.
Scripts required	Canvas won't display anything with JavaScript turned off.	SVG can be written completely offline and loaded.	Flash requires a plug-in, but can be written completely offline.
Program support	Few if any commercial programs exist for building canvas graphics.	Many vector graphics programs can write SVG.	Commercial applications are available to write Flash.
Speed of rendering	Canvas renders images very quickly.	SVG renders images slower than canvas.	Flash renders images slower than canvas.
Event handling	Users can only click on the entire canvas.	Users can click on individual elements in SVG.	Users can click on any element in Flash.
User adoption	Canvas is HTML5 and so requires modern browsers.	SVG requires modern browsers.	Flash has been around a long time and has wide-spread support.
Search engine optimization (SEO)	Canvas is text based and so is SEO friendly.	SVG is text based so is SEO friendly.	Flash is an embedded SWF file and is harder for search engines to read.

Summary

The <canvas> element is a powerful tool for building vector graphic images. In this hour you learned how to draw squares, polygons, lines, and circles using paths and rectangles.

You also learned how to add text and images to your canvas as well as put shadows on your drawings, add gradients to your fills, and manipulate images that are drawn on the canvas.

This hour also discussed some of the differences between the `<canvas>` element, Flash, and SVG. You also got a sense of what each one does well so that you can use them all effectively.

The HTML5 `<canvas>` element is a very complex element, and there is still a lot more you can learn about it. This hour provided a good overview, but it only touches on the surface of what you can do with the HTML5 `<canvas>` element. You can also:

- Animate your canvas
- Scale and transform your drawings
- Combine multiple drawings with clipping paths
- Add interactivity
- Build complex games

Canvas is a very fun and exciting part of HTML5. The resources listed in Appendix C, "HTML5 and Mobile Application Resources," can help you learn more about it.

Q&A

Q. ***I see that you only mentioned two-dimensional canvases (2d context). Is there a three-dimensional canvas (3d context)?***

A. Right now there is no accepted standard for a three-dimensional context on the `<canvas>` element. It will eventually, but right now you can only create two-dimensional drawings.

An experimental context has been enabled in Chrome, Firefox 4, and Internet Explorer 9 to enable three-dimensional context: WebGL. You use `"experimental-webgl"` to enable this context.

Q. ***When I draw a straight line, with a path, they are not as sharp as I want them to be, why is this?***

A. This is because of how the grid on the canvas works. The x and y coordinates form lines in a grid, but when these lines are drawn on the screen, they are actually drawn in the spaces *between* the grid lines. When you draw lines with an even thickness, the line is drawn down two whole pixels, but when drawn with an odd thickness, it is drawn down the line with half on one side and half on the other. Because computer monitors cannot split a pixel in half, they compensate by anti-aliasing the pixels to either side.

To fix this problem, you need to move the location of your lines to halfway between the two grid marks. So, instead of drawing a 1-pixel-wide line from 3,0 to 3,6, you should draw it from 3.5,0 to 3.5,6. This will place the line right in between the grid lines and remove the anti-aliasing.

Q. ***Can you add shadows to anything other than text?***

A. Yes, you can add shadows to all the canvas shapes, but if you add a shadow to a drawing with a gradient fill applied to it, the shadow will not be displayed. This is a browser bug that has been marked as fixed, but still seems to be a problem. Adding a shadow to a gradient-filled shape works in Firefox 4 but not Chrome or Safari, as of this writing.

Workshop

The workshop contains quiz questions to help you process what you've learned in this chapter. Try to answer all the questions before you read the answers. See Appendix A, "Answers to Quizzes" for answers.

Quiz

1. When you write a `<canvas>` element in your HTML, what is displayed in the browser?
2. The `clearRect()` method does which of the following:
   - **a.** Creates an empty, transparent rectangle
   - **b.** Deletes the existing rectangle
   - **c.** Removes all rectangles from the canvas
3. True or False: You cannot use transparent colors in the `strokeStyle` property.
4. Which of these is the default cap style for lines:
   - **a.** Butt
   - **b.** Square
   - **c.** Round

5. True or False: Circles are drawn using radians as the start and end points, not degrees.
6. True or False: You cannot set box model styles on canvas text.
7. What two things do you need to do to draw an image file on a canvas?
8. Which of these three are vector based:
    a. Canvas
    b. Flash
    c. SVG

Exercises

1. Build a canvas item with a rainbow linear gradient of at least seven colors across the diagonal of a rectangle.
2. Put together a page that uses as many of the `<canvas>` element features as you can. Be sure to use images, text, shapes, fills, shadows, and gradients.