HTML5 GRAPHICS

The HTML5 specification includes a definition for the Canvas element. This is a native graphical surface that can be placed within a web page and controlled by Javascript code. Special provisions are included in the specification so that the native graphics hardware on a machine can be used transparently – this leads to the availability of high-speed and high-specification graphics elements within web pages; commonly these facilities are used to create games within a web page.

The Canvas element and the drawing Context

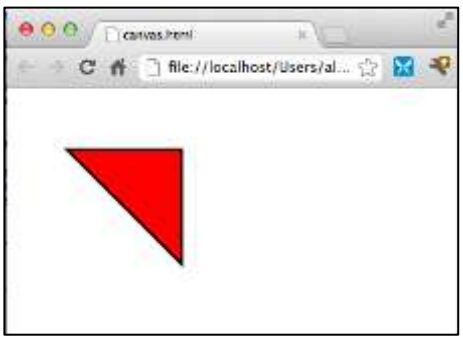
In HTML mark-up, a canvas is simply a blank area of the page. It is placed on the page relative to the tags that appear before and after it and given a size in pixels – its width and height attributes. In addition to this, it needs an ID attribute so that it can be identified by Javascript code that will manipulate it. An html page with a canvas can be as simple as that shown in figure 7.1:

Figure 7.1: The simplest possible page with a canvas, and how it appears in the browser

The trivial page shown in figure 7.1 has one major drawback – since there is no script tag attached to it, there is no way to actually put any graphics on to the canvas. The resulting empty element takes up a 400 x 200 pixel area of blankness.

To make the canvas work for a living, we need to include some Javascript code. This can be quite simple, for drawing a single figure like a circle, rectangle or line, or as complex as you like. The following script could be placed between the closing </body> tag and the closing </html> tag:

```
<script type="text/javascript">
    var canvas = document.getElementById("canvas"),
        context = canvas.getContext("2d"),
        fillColour = "#ff0000", // red
        outlineColour = "#000"; // black
    context.fillStyle = fillColour;
    context.strokeStyle = outlineColour;
    context.beginPath();
    context.moveTo(50, 50);
    context.lineTo(150, 50);
    context.lineTo(150, 150);
    context.closePath();
    context.lineWidth = 5;
    context.stroke();
    context.fill();
</script>
```



Listing 7.1: Script to draw a red triangle with a black outline

The relationship between the web page and the canvas element is made clearer in Listing 7.1 and the

accompanying figure. The canvas element is a device for reserving an area of a web page to draw on. It has a width and height and handles some events, but it delegates most of its work to another object – the Context object (its full name is the HTML Graphics Context). This is more or less the equivalent of a set of drawing and painting tools that can affect the canvas the context belongs to.

Having created a canvas in HTML code, I can ask the document for a reference to it (using **document.getElementById()**) and from it ask for a reference to its ***context*** (the first 2 statements in the listing 7.1 script). From here, all of the manipulations are either to assert a particular setting in the context (its drawing colour, fill style, line thickness etc.) or to perform a drawing operation. In listing 7.1 you can see that some operations can only happen over several function calls – for example, drawing a closed figure by creating a path in multiple steps.

Using the html file from figure 7.1 and the code from listing 7.1 we can produce a web page with a fixed graphic on it. This does not seem to be any advancement on what we could do with a simple image file and an <img> tag in the html, so why bother with a canvas at all?

Interactive Scripting

The point about the canvas element is that it can be updated on the browser according to external stimuli – events to you and I. That means that we could start with a blank canvas and draw images on it over a period of time, or we could display images according to the user's choices, or we could even generate images based on some truly external factor, like the current temperature at some chosen location (obtained via a web link). Using the canvas on a web page gives us the power to do several things that are difficult or impossible without 3rd party plug-ins for the browser. We can:

1. Generate graphs and charts depending on ever-changing data from the user, or a web server
2. Draw figures that represent external information that can be accessed from web sites and data sources
3. Manipulate graphics according to the user's interaction with a page – for example buttons on a form, or the canvas element itself, using a mouse or keyboard
4. Animate graphics to embellish a page in some way (e.g. showing the landing status of a flight into a local airport as an animated pictogram)
5. Create animations for visualization of complex situations, like the number of users on a network, or the flow of gasses in a jet engine
6. Create animated games

The last of these is becoming a big topic in HTML5 development – see http://html5games.com/ for many good examples (and some crappy ones).

For the purposes of this module, the canvas element has one big advantage over many of the other HTML5 elements, in that it provides a very good way to visualize objects in a program. Simple Javascript code to create and manipulate objects that represent customer data or employees in a company may be very useful and informative, but can have an underwhelming impact when you are not writing them for a real-world application. Simple definitions of objects that can represent themselves on a canvas element immediately have more impact – you can see clearly what your code is doing, and this is a major benefit educationally.

Over the rest of this chapter we'll develop some graphical objects and manipulations that will illustrate both the capabilities of the canvas element and the power and flexibility of object-oriented programming.

'Current' settings – colour, styles etc.

Context settings give you a way to select the properties of a drawing instrument – e.g. a pen that draws in a particular colour and width, or a painting colour, or a line style (dotted, dashed, solid etc.), or shadow effects or even the way line segments terminate and join to each other. Common examples of these are listed below:

Setting	Description	Example
fillStyle	Returns the colour, gradient or pattern used to fill a drawing element line a circle, rectangle or path	`ctx.fillStyle="#FF0000";` `ctx.fillStyle = "blue";`
strokeStyle	Sets or returns the colour, pattern or gradient used for strokes (i.e. lines)	`ctx.strokeStyle="#222222";` `ctx.strokeStyle="#00f";//blue`
shadowColor	Sets or returns the colour used for shadows	`ctx.shadowColor="#ccc";` `// a dark grey`
font	Sets or returns the current font properties	`ctx.font="30px Arial";`
globalAlpha	Sets or returns the current alpha (transparency) value for drawing	`ctx.globalAlpha = 0.5;`

Table 7.1: Some HTML canvas context properties

Table 7.1 is a short list of the properties of a context object that you are likely to use often. Colours are given as simple text strings containing red, green and blue components (in hexadecimal). The values start with a '#' and can use 3 digits (e.g. "#000" - black) or 6 digits (e.g. "#ffffff" - white) depending on the number of shades you intend to support.

There are other more exotic properties, e.g. for defining how the end points of a line are made, how text aligns with other drawn figures etc. You can examine the full list at http://www.w3schools.com/html5/html5_ref_canvas.asp. The important thing about property settings is that you apply them before using one of the 'drawing' methods that draw on a canvas.

Drawing simple graphical figures

For drawing shapes, the context only has a few methods, but these are very powerful and flexible. Apart from two exceptions, drawing rectangles and placing bitmapped images on a canvas, it is necessary to create and work with a ***path***, which is a sequence of drawn elements that can include lines, curves and arcs. The sequence can be as complex as you line, and a path can be drawn as an outline or a filled shape.

Although this may at first appear to be an awkward way to manipulate graphics, it is very powerful and means a vast range of shapes can be drawn using only a few methods.

For example, the easiest way to draw a rectangle is…

```
var ctx = document.getElementById("canvas").getContext("2d");
ctx.fillRect(10, 10, 100, 100);
```

Listing 7.2: The quick way to draw a rectangle

..and it is certainly a bit more effort to use a path to do the same thing:

```
ctx.beginPath();
ctx.moveTo(10, 10);
ctx.lineTo(110, 10);
ctx.lineTo(110, 110);
ctx.lineTo(10,110);
```

```
ctx.closePath();
ctx.fill();
```

Listing 7.3: Using a path to draw a rectangle

However, a rectangle is one of the few exceptional shapes supported by the HTML canvas. Paths can be used to to draw single lines, triangles, 40-sided figures, complex curves etc., and the set of methods needed to do any of these remains the same.

Drawing regular shapes (squares, circles, triangles, ellipses etc.) is common in graphics programming. We can build from a few simple shapes into very complex diagrams and charts. One approach to drawing a regular graphical shape is to create a function. So, for example, to draw a line:

```
function drawLine(context, x, y, width, height){
   context.beginPath();
   context.moveTo(x, y);
   context.lineTo(x+width, y+height);
   context.stroke();
}
```

Listing 7.4: A function that draws a line

Note how the graphical context has been passed as the first parameter to our ***drawLine()*** function. The remaining parameters indicate the start and end-points. To draw a circle, we could use:

```
function drawCircle(context, x, y, radius){
   context.beginPath();
   context.arc(x, y, radius, 0, 2*Math.PI);  // centre, radius, start & end angles
   context.stroke();
}
```

Listing 7.5: Another function that draws a circle

We could easily continue in this manner, providing functions for drawing all of the basic shapes we might need. It works well enough and would save you having to look up the list of parameters for doing odd things with the ***arc()*** method, for example.

What if we wanted to draw a red line or a filled circle? Certainly we could add in extra parameters, for colour, whether the shape was filled, different stroke styles etc. The problem with this strategy is that instead of having to remember the odd HTML canvas syntax for drawing a path, we now need to remember long lists of parameters in the order they are needed in.

An altogether better approach is to create objects to represent rectangles, lines, circles etc. That way we can have properties to assert the settings needed (like colour, filled, stroke style etc.), and keep the number of parameters to a minimum. To allow for the possibility of many circles, many rectangles etc., a constructor for each shape would be better, and if we work it carefully, we could make use of inheritance to minimize the amount of work needed.

Inheritance in drawing can be very powerful. For example, if we consider the things that might be needed by **any** shape that we draw (position, colour, line-thickness, whether it is a filled shape or not, does it cast a shadow and so on), then it seems sensible to put as much of the code for this common set of features into a single type – Shape would be the obvious name. This would then provide the basis for a whole family of shapes.

We saw UML class diagrams in chapter 5 of the notes: they provide an ideal format for use to consider for a family of shape types.

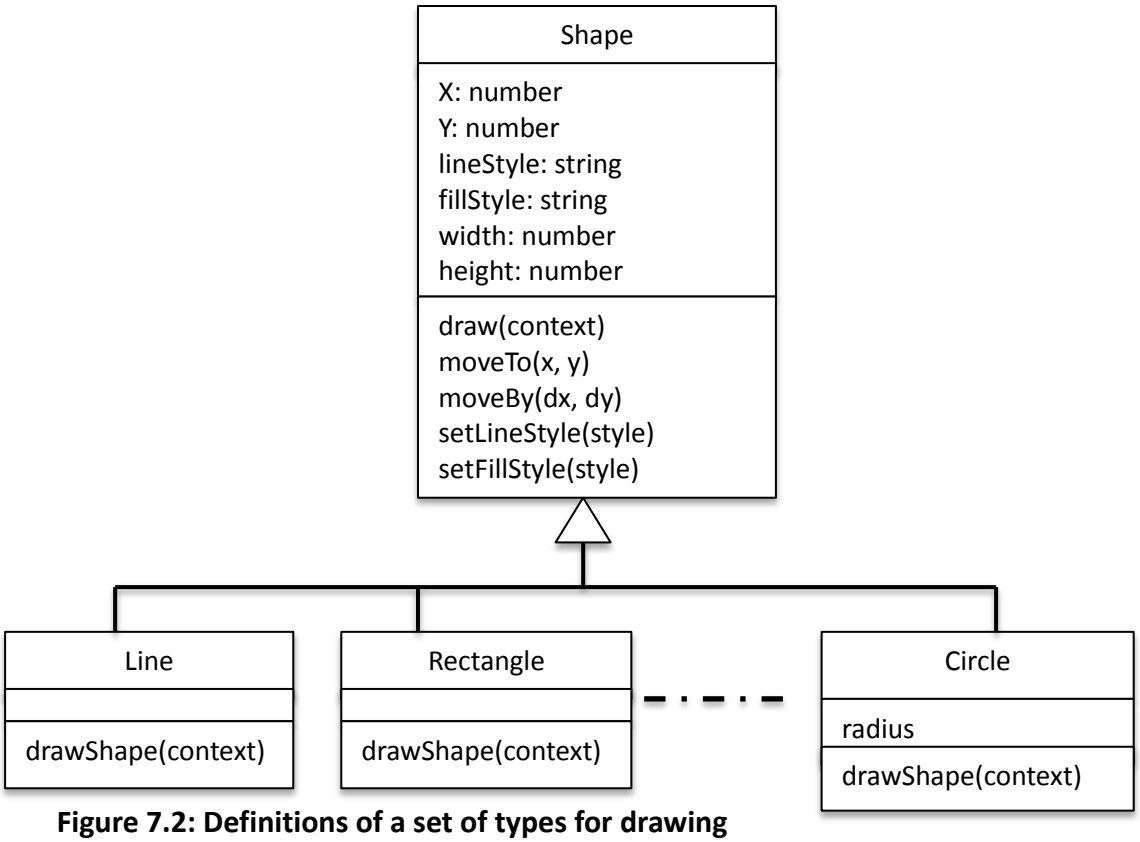


Figure 7.2: Definitions of a set of types for drawing

The Shape type coding

Figure 7.2 shows the general strategy – once we have created a Shape type (i.e. by creating a constructor for shape objects) we can use inheritance to create as many specialised versions of it as we wish. We can do a lot of the groundwork into the Shape constructor and methods and then minimize the amount of work needed to make all the rest. One notable feature of the design in Figure 7.2 is that while the Shape type has a draw() method, the inheriting types all have a drawShape() method.

This isn't a mistake. We could implement a draw() method in every inheriting type, and then the draw() code in all of them would need to set up line styles, fill styles and any other aspects of a particular shape we wanted to set. However, if we leave it so that all of the other types inherit the Shape's draw() method, and code that method so that it calls a drawShape() method, we can put just the code that actually draws out a rectangle, line etc. in drawShape(), and draw will start a path, set up the drawing and fill colours and finish drawing the shape for free. Here's the Shape code:

```
var Shape = function(x, y, width, height){
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.lineStyle = "black";
    this.fillStyle = null;              // Some shapes don't get filled (e.g. lines)
};
```

```
Shape.prototype.draw = function(context){
    context.strokeStyle = this.lineStyle;
    context.fillStyle = this.fillStyle;
    context.beginPath();
    // Note - no actual drawing gets done here.
    // What would we draw?
    // However, we could call a method that all objects that
    // inherit from this one would be required to provide...
    if(this.drawShape){
        this.drawShape(context);
    }
    // The advantage of doing it this way is that we can do all of
    // the line and fill setup around the basic shape drawing.
    context.stroke();
    if(this.fillStyle){
        context.fill();
    }
};

Shape.prototype.moveTo = function(x, y){
    this.x = x;
    this.y = y;
};

Shape.prototype.moveBy = function(dx, dy){
    this.x += dx;
    this.y += dy;
};

Shape.prototype.setLineStyle = function(style){
    this.lineStyle = style;
};

Shape.prototype.setFillStyle = function(style){
    this.fillStyle = style;
};
```

Listing 7.6: A Shape class – made for inheriting from

You ought to realize that we won't get anything useful out of creating a shape object. It doesn't have a drawShape() method, so the central requirement of the draw() method is missing. That's fine – Shape is a deliberately vague name for a type, and based on the premise that there will be lots of different Shape sub-types, but no actual Shape objects. The technical name for a type that is built only for inheriting from is an ***abstract type***.

With Shape in place, creating a new, concrete type of drawable shape is very easy, needing 3 simple

steps:

1. Inherit the Shape member values by applying its constructor in the constructor of a new type:
```
var Line = function(x, y, width, height){
   Shape.apply(this, arguments);
};
```
2. Make sure that the Shape prototype functions are also part of this new type:
```
Line.prototype = new Shape();
```
3. Add a drawShape() method to the new type:
```
Line.prototype.drawShape = function(context){
   context.moveTo(this.x, this.y);          // No need to start a path
   context.lineTo(this.x+this.width,
                  this.y+this.height);      // or to finish drawing.
};
```

Listing7.7: The steps to create a new Shape type by inheritance

That is not very much code considering the amount of functionality a line object now has. Note that the line will be drawn from corner to corner of a rectangle that starts at x, y and is width, height in size. I'll show the rectangle type with no annotation:

```
var Rectangle = function(x, y, width, height){
   Shape.apply(this, arguments);
};

Rectangle.prototype = new Shape();

Rectangle.prototype.drawShape = function(context){
   context.moveTo(this.x, this.y);
   context.lineTo(this.x+this.width, this.y);
   context.lineTo(this.x+this.width, this.y+this.height);
   context.lineTo(this.x, this.y+this.height);
   context.closePath();
};
```

Listing 7.8: A Rectangle type

Now we have an object-oriented way to draw geometric shapes, we can use them in drawing code quite easily:

```
var doDrawing = function(){
   var ctx = document.getElementById("canvas").getContext("2d");
   var l = new Line(100,100,50,-10),
       r = new Rectangle(300, 100, 100, 50),
       c = new Circle(100, 300, 80),
       t = new Triangle(300, 250, 80, 50);
   l.setLineStyle("#f00");
   c.setLineStyle("#000");
   c.setFillStyle("#00f");
   r.setLineStyle("#00f");
```

```
    r.setFillStyle("#0f0");
    t.setLineStyle("#000")
    t.setFillStyle("#00f")
    l.draw(ctx);
    r.draw(ctx);
    c.draw(ctx);
    t.draw(ctx);
};
```

Listing 7.9: Using a set of shape types in a drawing

Note that I've also defined Circle and Triangle types to be used by the code in listing 7.9. The result is shown in figure 7.3:

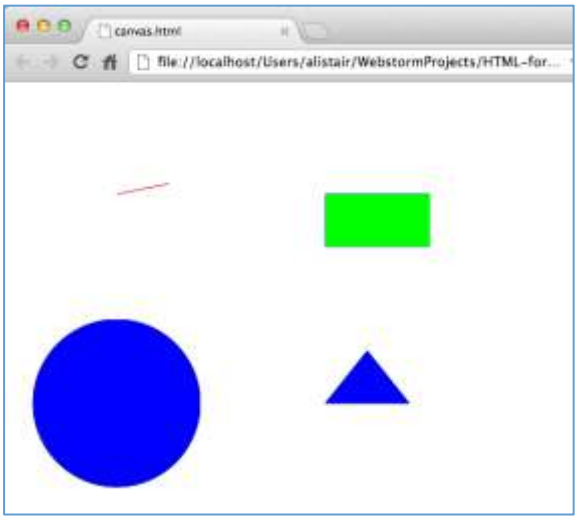


Figure 7.3: Examples of our drawable shape classes

Drawing from events

We still have the problem that our drawing is no more use than a PNG image placed in an <img> tag on the html page instead of a canvas. To make drawing into an interactive operation, we need to let the user define what will be drawn and where.

Let's start with where to draw things: the user can describe a rectangle by clicking the mouse button on the canvas at the top-left, dragging the mouse to the bottom right and then letting go of the mouse button. We can use the canvas's **onmousedown** and **onmouseup** events for this. We could attach these to code in the html **<canvas>** tag, like:

```
<canvas id="canvas" onmousedown="mouseDown();" onmouseup="mouseUp();">
```

However, writing code over two different files (a Javascript one and an html one) is never a great idea. A better approach is to attach the events to the canvas object in our Javascript file. This is quite easy to do in Javascript code directly (see listing 7.10).

```
function setup(){
   // First we need a canvas and a context...
   canvas = document.getElementById("canvas");
   context = canvas.getContext("2d");
   // Now we can hang some event handlers on the canvas...
   canvas.onmousedown = mouseDown;
   canvas.onmouseup = mouseUp;
   canvas.onmousemove = mouseMove;
};

window.onload = setup;
```

Listing 7.10: Code to register event handlers for canvas interactions

Note that the setup function in listing 7.1 has been installed as the function that will execute when the page loads (**window.onload = …**). Because of this, the only code that can execute after the page has loaded will be due to the user's mouse interacting with the canvas, via those event handlers.

To test out the code in listing 7.10, create a simple ***mouseMove()*** function that draws on to the canvas at the mouse's current location:

```
function mouseMove(e){
   var x = e.x - canvas.offsetLeft;   // Work out where the mouse is
   var y = e.y - canvas.offsetTop;          // relative to the canvas
   context.beginPath();                     // Then draw something…
   context.arc(x, y, 5, 0, 2*Math.PI);      //…a circle…
   context.fill();                          // …and complete the draw op.
}
```

Listing 7.11: Code to test the mouseMove() event handler

Figure 7.4 shows the result of running the mouse over the canvas using code in listing 7.11. You can clearly see where the edges of the canvas element are in this.

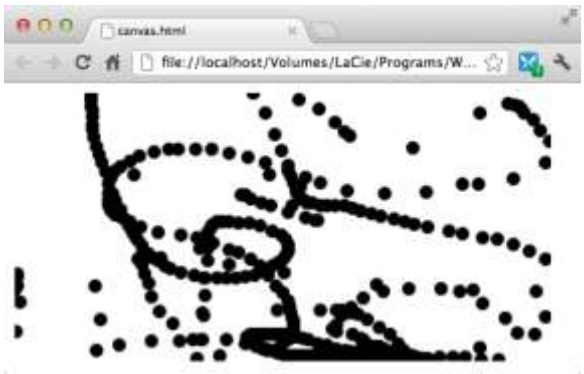


Figure 7.1: The result of running the code in listing 7.11

Of course we will need to handle events a bit more cleverly to create a drawing application. Firstly, the mouse should not draw on the canvas all the time, but only when a mouse button is pressed. Secondly, we don't want to simply draw small circles everywhere, but instead draw the shape elements the user chooses.

To do this we're going to need more of a user interface:

```
<body>
   <canvas id="canvas" width="640" height="480"></canvas>
   <p/>
   <form style="position: absolute; left:10px; top:500px;">
      Line Colour:
      <select id="line" onchange="changeLine();">
         <option value="#fff">None</option>
         <option value="#000" selected>Black</option>
         <option value="#00f">Blue</option>
         <option value="#f00">Red</option>
         <option value="#0f0">Green</option>
         <option value="#ff0">Yellow</option>
         <option value="#fc0">Orange</option>
         <option value="#f0f">Purple</option>
         <option value="#0ff">Cyan</option>
      </select>
        Fill Colour:
      <select id="fill" onchange="changeFill();">
         <option value="#fff" selected>None</option>
         <option value="#000">Black</option>
         <option value="#00f">Blue</option>
         <option value="#f00">Red</option>
         <option value="#0f0">Green</option>
         <option value="#ff0">Yellow</option>
         <option value="#fc0">Orange</option>
         <option value="#f0f">Purple</option>
         <option value="#0ff">Cyan</option>
      </select>
        Shape:
      <select id="shape" onchange="changeShape();">
         <option selected>None</option>
         <option>Line</option>
         <option>Rectangle</option>
         <option>Circle</option>
         <option>Triangle</option>
      </select>
   </form>
</body>
```

Listing 7.12: An html form containing drawing controls

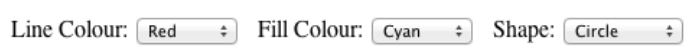


Figure 7.2: The resulting user-interface

The UI coded in listing 7.12 and shown in figure 7.2 seems minimal, but since pointing at the canvas will do all of the actual drawing, it is enough. The user will select a Line Colour, possibly a Fill Colour and a Shape. Then, pressing the mouse button when the cursor is at one corner of the shape, and releasing it after dragging the mouse to the diagonally opposite corner gives the shape its position and size. As the mouse is dragged, we can arrange the shape to be drawn lightly at intermediate positions until the final size is decided. For this we need to use the event handlers for mousedown, mousemove and mouseup. Although the code for this can get quite detailed it is conceptually straightforward. The full listing is at the end of this chapter, and available as a download from the Moodle website (URL********):

```
The mouse button is pressed:
    shape = a new shape based on type (line, circle etc.) fillstyle and strokestyle
                its position is the current mouse position, its size is 1pixel x 1pixel
    drawing = true
    stash the current graphical content of the canvas for later use (use
        context.getImageData() for this
    draw the shape at its current size


The mouse is moved:
    if (drawing)
        shape.width & shape.height are the x & y distances the mouse has moved
        restore the initial state of the canvas (to erase the current shape – use
            context.putImageData)
        draw the shape outline

The mouse button is released:
    drawing = false
    delete the stashed copy of the canvas image
    add shape to the list of shape objects in the drawing
    clear the canvas
    draw the list of shapes
```

Listing 7.13: Pseudo-code description of drawing with a mouse

Drawing from a timer

Using a timer to draw on a canvas can produce a moving image. This could be used in a game to move game elements around the screen, or to create an animated graphic. By using a series of bitmap images (or segments of one big bitmap image), you can do very convincing cell animation at a decent rate (most desktop browsers will handle a lot of images at 60 frames per second – with smartphone browsers this typically drops to 30 frames per second: still well above the persistence of vision threshold).

A simple animation need be no more than repeatedly drawing a simple shape, but updating its position each time it is drawn. The repetition can be handled by Javascript's in-built window.setInterval() function, which will call another function at set intervals:

```
var canvas, context, x=10, y=10;

var animate = function(){
    // Moving the thing we draw amounts to animation…
    x += 1;
    y += 1;
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.fillRect(x, y, 10, 10);
};

var setup = function(){
    // First we need a canvas and a context...
    canvas = document.getElementById("canvas");
    context = canvas.getContext("2d");
    // We'll fill a rectangle in this colour…
    context.fillStyle = "#ff0000";
    // This kick's the animation off…
    window.setInterval(animate, 100);
};

window.onload = setup;
```

The animation – note, I've removed the call to clearRect() so that a few frames can be seen

Listing 7.14: A very simple animation

This code in listing 7.14 does only the most basic of animations. It is also flawed in one serious way – the animation function will be called at 100mS intervals, whether or not the canvas is visible. i.e. even if there is a window in front of the current canvas, or a different browser tab is being used, the drawing will continue to happen.

The browser developers have thought of this, and so provide a ***requestAnimationFrame()*** function whose job is to schedule a function call (expected to be one that draws on the screen) in the most efficient way possible. If the canvas is not visible, the function won't be called at all. Because animation frames are requested at the optimal frequency for the browser, we can even do without the timer function. We only need to update the code a little to make this work:

```
// This is needed because all of the browsers use different function names
// to request an animation frame…
var requestAnimationFrame = window.mozRequestAnimationFrame    ||
    window.webkitRequestAnimationFrame ||
    window.msRequestAnimationFrame     ||
    window.oRequestAnimationFrame;

var doAnimate = function(){
    animate();
    requestAnimationFrame(doAnimate);
};
```

Listing 7.15: Simple animation code

The first statement in listing 7.15 may seem a bit odd. We want to call a function called ***requestAnimationFrame()***. However, that function goes by different names in different browsers; ***window.mozRequestAnimationFrame()*** in Firefox, ***window.webkitRequestAnimationFrame()*** in Chrome and Safari etc. (the other two browsers are Internet Explorer and Opera).

We can get around that by referring to one of these functions through a variable (***requestAnimationFrame***). The **||** operator in Javascript (**||** meaning 'or') combines truthy and falsey values, and as we know already, anything that is not ***null***, ***undefined***, ***0*** or ***""*** is truthy. Javascript is lazy, and when combining several logical values, it will give up as soon as the outcome is known. Combine ***false*** and ***something-else*** with a **||** operation and the result will depend on the ***something-else***. Combine ***true*** with ***anything-else***, and anything-else will not even be evaluated (since ***true*** is the result immediately).

Using this logic, what gets assigned to **requestAnimationFrame** will be the first function that does not evaluate to ***null***. In Firefox, this will be the first one, in a webkit browser, the second etc. What all this means is that we can call ***requestAnimationFrame()*** and the browser-specific version will be called. After this, the next bit of the code will seem trivial. When ***doAnimate()*** is called, the first thing it will do is execute our ***animate()*** function. The next thing it will do is call ***requestAnimationFrame()***, passing ***doAnimate()*** as the function to call when the browser is ready to display another frame. ***requestAnimationFrame()*** has an inbuilt timer that waits for a browser-specific period of time (typically 1/60 second in a desktop browser, 1/30 second in a smartphone browser). The result is that the function displays a frame and then schedules the next call to itself.

Using graphics files

The html5 canvas includes support for image files in gif, Jpeg or PNG format. You can load an image into Javascript code by simply creating an Image() object and setting its src property:

```
var image = new Image();
image.src = "myImage.png";
context.drawImage(image, 0, 0);
```

Listing 7.16: Code to pick up an image file and draw it on the canvas

The code in listing 7 picks up an image file and draws it. The file must be in the same directory as the script file that loads it, or it must have a name that is a relative path to the image. For example:

```
image.src = "/myimage.png";   // This file is in the root folder of the site
image.src = "../myimage.png"; // This is in the directory above the script
image.src = "img/myimage.src";        // In the /img subdirectory from the script
image.src = "../img/myimage.src";     // The img folder that is at the same level
                                      // as the one the script is in
```

Listing 7.17: Example image file paths

The image drawn in listing 7.16 would be placed at the top-left of the canvas. By specifying x & y coordinates we can place an image at any position on the canvas, so using this strategy we can animate images. We could create a cell-based animation by having a set of images, which are all a character at different stages (e.g. walking or running) and displaying them in successive frames. However, this approach would be tedious (having to load a group of files, one per cell). The canvas gives us a slicker way of doing this. If we have an image that is made up of a number of separate cells, we can select

which part of the image to draw on the canvas.

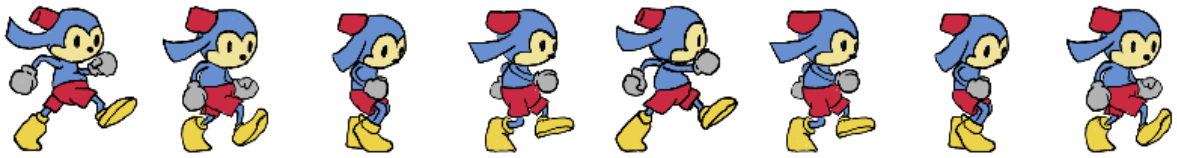


Figure 7.3: A single image containing 8 cells

The image in figure 7.3 has eight separate cells in it. Because they are evenly distributed, it is easy to work out the coordinates of a single cell. If this image is 800x100 pixels, then the first cell is from 0, 0 to 99, 99; the second is from 100, 0 to 199, 99 etc.

Assume we wanted to draw the second cell on a canvas. Assuming our image object was called ***image***, the ***drawImage()*** function can be called with additional parameters to address this:

```
context.drawImage(image, 100, 0, 100, 100, 50, 50);
```

The parameters to this function are (in order), the image object, the x & y ordinates of the top-left of the cell we want to draw, the width and height of the section we want to draw (100 across by 100 down), and finally, the x & y of where we want to draw it on the canvas. If we wanted we could also scale the drawn segment by indicating the width and height we wanted it to appear at as optional extra parameters - for example…

```
context.drawImage(image, 100, 0, 100, 100, 50, 50, 50, 50);
```

…will draw the cell at half-scale (50x50 instead of 100x100).

Since we can work out the overall size of an image when it is loaded (more about this later), we could write a general function to draw one cell of an animation strip (more commonly known as a sprite-sheet):

```
var drawCell = function(image, num_cells, cell_number, at_x, at_y){
    var cell_width = image.width / num_cells;  // How wide is a cell?
    var left = (cell_number-1) * cell_width;
    var right = left + cell_width – 1;
    context.drawImage(image, left, 0, cell_width, image.height, at_x, at_y);
};
```

Listing 7.18: A function to draw one cell from an image

We could now create an animation with a simple loop:

```
var cell_no = 0;
while(animating){
    drawCell(image, 8, cell_no, x, y);
    cell_no = (cell_no+1) %8;
    // Now pause for a while…

}
```

Listing 7.19: Drawing a cell-animation

Of course we would probably want to do this using ***requestAnimationFrame()*** as discussed previously.

Loading Images

It was mentioned earlier that loading an image is simply a matter of creating an ***Image*** object and setting its ***src*** property to the URL of the image file. There is one slight problem with this; an image can be a very large file, and it can be on a server you only have access to over a slow Internet connection. If the image is not fully loaded when you try to draw it, the program will fail.

This is something the designers of Javascript (and web protocols in general) have put an appropriate amount of thought to. The Image object will fire an event when an image file has completed loading. We can attach a function to handle this event, and draw the image in this function:

```
var img;
var get_image = function(url){
    img = new Image();
    img.onload = show_image;
    img.src = url;
}
var show_image() {
    // the image has now loaded – draw it here…
    context.drawImage(img, 0, 0);
};
```

Listing 7.20: Using an Image() object's onload event

Questions

1. How can you find the width and height of the canvas element?
2. Drawing operations using a path can draw either outlines or filled-in figures. What two methods of the context object can be used to decide which?
3. How would I set the context to draw a path that was filled in green with a black outline?
4. Explain the advantage of using Javascript objects to define drawable elements.
5. Which element in drawing code provides the events that could be used to follow the user's interactions?
6. Which context method is the most useful for creating a cell-based animation using an image file?

Exercises

1. Write code for the other two shape types shown in figure 7.3 – Circle (code to draw a circle can be seen in listing 7.5) and Triangle (simply move half-way along the top before drawing to the two bottom corners).

2. Write a simple animation program, using a timer, so that a moving circle is drawn on the screen. The circle's y ordinate should stay constant, but its x ordinate should change when the timer function executes. Arrange the timer code so that if the circle goes off the left of right hand of the canvas, its direction reverses. The result should be a circle that bounces back and forth across the canvas.

Code for the Canvas Drawing Application

```
var Shape = function(x, y, width, height){
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.lineStyle = "black";
    this.fillStyle = null;
};

Shape.prototype.draw = function(context){
    context.strokeStyle = this.lineStyle;
    context.fillStyle = this.fillStyle;
    context.beginPath();
    // Note - no actual drawing gets done here.
    // What would we draw?
    if(this.drawShape){
        this.drawShape(context);
    }
    context.stroke();
    if(this.fillStyle){
        context.fill();
    }
};

Shape.prototype.resize = function(xx, yy){
    this.width = xx - this.x;
    this.height = yy - this.y;
};

Shape.prototype.moveTo = function(x, y){
    this.x = x;
    this.y = y;
};

Shape.prototype.moveBy = function(dx, dy){
    this.x += dx;
    this.y += dy;
};

Shape.prototype.setLineStyle = function(style){
    this.lineStyle = style;
};

Shape.prototype.setFillStyle = function(style){
    this.fillStyle = style;
};
```

```
var Line = function(x, y, width, height){
    Shape.apply(this, arguments);
};

Line.prototype = new Shape();

Line.prototype.drawShape = function(context){
    context.moveTo(this.x, this.y);
    context.lineTo(this.x+this.width,
                   this.y+this.height);
};

var Rectangle = function(x, y, width, height){
    Shape.apply(this, arguments);
};

Rectangle.prototype = new Shape();

Rectangle.prototype.drawShape = function(context){
    context.moveTo(this.x, this.y);
    context.lineTo(this.x+this.width, this.y);
    context.lineTo(this.x+this.width,
                   this.y+this.height);
    context.lineTo(this.x, this.y+this.height);
    context.closePath();
};

var pythagoras = function(w, h){
    return Math.sqrt(w*w + h*h);
};

var Circle = function(x, y, width, height){
    Shape.apply(this, arguments);
};

Circle.prototype = new Shape();

Circle.prototype.drawShape = function(context){
    // Work out centre and radius from the bounding
rectangle...
    var x = this.x + this.width/2;
    var y = this.y + this.height/2;
    var radius = pythagoras(x - this.x, y - this.y);
    context.arc(x, y, radius, 0, 2*Math.PI);
};
```

```
var shapeFactory = function(type, x, y){
    var shape;
    switch(type){
        case "Line":
            shape = new Line(x, y, 1, 1);
            break;
        case "Rectangle":
            shape = new Rectangle(x, y, 1, 1);
            break;
        case "Circle":
            shape = new Circle(x, y, 1, 1);
            break;
        case "Triangle":
            shape = new Triangle(x, y, 1, 1);
            break;
        default:
            shape = null;
    };
    return shape;
};

var doDrawing = function(){
    for(var i= 0, j=shapeList.length; i<j; i++){
        shapeList[i].draw(context);
    }
};

var canvas, context, lineStyle, fillStyle, shapeType,
        drawing=false, shapeList = [], x, y, x2, y2,
        currentState, currentShape;

var mouseUp = function(e){
    // Just completed a drawing operation.  Add the new shape
to the
    // display list and draw the whole thing...
    drawing = false;
    //context.globalCompositeOperation = "source-over";
    context.putImageData(currentState, 0, 0);
    delete currentState;
    currentShape.setLineStyle(lineStyle);
    currentShape.setFillStyle(fillStyle);
    shapeList.push(currentShape);
    doDrawing();
};
var mouseDown = function(e){
    drawing = true;
    // Stash the current screen state, so we can restore it every
    // time the mouse moves...
    currentState = context.getImageData(0, 0, canvas.width,
canvas.height);
    x = e.x - canvas.offsetLeft;
    y = e.y - canvas.offsetTop;
    // What are we drawing...
    context.strokeStyle = "#000";
    currentShape = shapeFactory(shapeType, x, y);
    context.beginPath();
    currentShape.drawShape(context);
    context.stroke();
};
```

```
var mouseMove = function(e){
    if(drawing){
        // We need to drag out a bounding box to get the
        // shapes position and size.
        // Start by restoring the screen to its state when the
mouse
        // button was pressed...
        context.putImageData(currentState, 0, 0);

        // Now work the that the shape should appear at now...
        x2 = e.x - canvas.offsetLeft;
        y2 = e.y - canvas.offsetTop;
        currentShape.resize(x2, y2);
        // And draw it.
        context.beginPath();
        currentShape.drawShape(context);
        context.stroke();
    }
};

function changeLine(){
    lineStyle = document.getElementById("line").value;
};

function changeFill(){
    fillStyle = document.getElementById("fill").value;
};

function changeShape(){
    shapeType = document.getElementById("shape").value;
};

var setup = function(){
    // First we need a canvas and a context...
    canvas = document.getElementById("canvas");
    context = canvas.getContext("2d");
    context.fillStyle = null;
    context.lineStyle = "#fff";
    //context.fillRect(0, 0, canvas.width, canvas.height);
    // Now we can hang some event handlers on the canvas...
    canvas.onmousedown = mouseDown;
    canvas.onmouseup = mouseUp;
    canvas.onmousemove = mouseMove;
};

window.onload = setup;
```

Listing 7.21: A complete listing of the drawing app