OXFORD BROOKES UNIVERSITY

SVG Primer

Contents

Appendices

1. Introduction

1.1 Scalable Vector Graphics

Until recently, the way to add schematic drawings to a web page was to define the drawing as an image (in GIF, PNG, JPEG or some other format) and insert the image into the web page using the <IMG> element. This has the following major drawbacks:

1. **Image size:** The size of an image is defined by the width and height of the image (in pixels) and the number of bits allocated to each pixel in the image. For example, a 100 by 100 pixel image with 8 bits defining the Red, Green and Blue components of each pixel results in an image that takes up over 30 Kbytes before compression. For simple line drawings this is a large amount of information that needs to be moved across the internet for possibly very little content. Also, it is not possible to interact with the image without generating and sending the new image.
2. **Fixed resolution:** Once the image has been defined at a specific resolution, that is the only resolution available. Zooming in on the image just makes the pixels bigger. To get higher resolution, the original schematic drawing has to be reconverted to an image with, say, 500 pixels in each direction.
3. **Binary format:** Image formats store the image data in some binary format which makes it difficult to embed rich metadata about the graphic to help search engines. Also, specialized applications are needed to make even the slightest changes to the image.
4. **Minimal animation:** The GIF format allows several images to be defined in one image file ("animated gifs"), but each image is essentially static. More lively presentations require a video format such as MPEG and this is large, requires a separate plugin and is even more difficult to edit.
5. **No inherent hyperlinking:** Web pages depend on hyperlinking. To do this is with images requires the use of image maps defined as part of the enclosing HTML page. They are difficult to generate and only allow linkage from a region of the image and not from a specific element in the image.

Scalable Vector Graphics, or SVG, is the World Wide Web Consortium's Recommendation for defining 2-dimensional schematic drawings such that the size is more directly dependent on the content in the drawing and the resolution is whatever the user requires. Zooming in on an SVG drawing allows greater and greater detail to be seen if the drawing is complex.

1.2 An XML Application

SVG is an XML application. That means SVG is defined using a set of elements, rather like HTML, and the elements can have attributes associated with them. For example:

```
<text x="20" y="20">Abracadabra</text>
```

The SVG **text** element has a start and end tag written as **<text>** and **</text>** and the content of the element is the string Abracadabra. The text element has two attributes, **x** and **y**, that define the position of the text in the drawing. These are defined as part of the start tag. Being an XML application, several rules have to be obeyed:

- There can only be one outer element that encloses the complete drawing definition and that is **<svg>**
- Every start tag must have a correctly nested end tag
- In SVG, tag names are predominantly lower case with no spaces (multi-word names like **clipPath** use camel case
- Attributes must be enclosed in quotes (either single or double)

If the content of the element is null, a shorthand can be used:

```
<rect x="10" y="10" width="50" height="30"></rect>
<rect x="10" y="10" width="50" height="30" />
```

The slash before the closing **>** in the second line indicates that the element does not have any content. Effectively, all the content is encapsulated in the name of the element and its attributes. The two examples of the **rect** element given above are equivalent.

1.3 Using SVG

The simplest way to use SVG is as part of a web page defined in HTML. You define the SVG document and store it in a file with '.svg' as the file extension. To add it to the web page requires, for example:

```
<p>This can be shown in the following diagram:</p>
<object width="620" height="420" data="myfirstsvg.svg" alt="SVG Drawing"
type="image/svg+xml">
Please download Adobe Plug-in to see SVG diagram </object>
```

The **object** element in HTML 4.0 is similar to the **img** element in that it allows you to insert an external object (myfirstsvg.svg in this case) into a web page. It differs in that it allows you to insert applets and other HTML pages as well as graphics and images. You can also specify a number of alternatives. Here we have given a text message to indicate that the SVG could not be rendered but we could have had an <img> element that defines a png image of the diagram as an alternative. Providing some alternative is useful at the moment as not everybody has an SVG plug-in installed in their browser. The recommended SVG plug-in at the moment is the one from Adobe which can be installed in most of the modern browsers. It is free. You should add it to your favourite browser before you start using SVG. Visit the Adobe site and follow the instructions:

```
http://www.adobe.com/svg/main.html
```

There are a number of stand-alone viewers for SVG that can be used. You just open the SVG file and it will be displayed in the viewer's window. There are also support tools for constructing constructing SVG diagrams just as there are tools for constructing web pages. Some of these also have the ability to view a previously defined SVG file. A complete list of the tools and viewers available is maintained on the W3C web site:

```
http://www.w3.org/Graphics/SVG/Overview.html
```

2. Coordinates and Rendering

2.1 Rectangles and Text

It is difficult to talk about either coordinates or rendering in a vacuum so we first need to specify a couple of SVG drawing elements so that we can illustrate the points being made. The two we will use for the moment are text and rect. We will come back and talk about the drawing primitives in more detail later.

The **rect** element has a large number of attributes but we shall consider just a few for the moment:

```
<rect x="20" y="30" width="300" height="200" rx="10" ry="10"
style="fill:yellow;stroke:black" />
<text x="40" y="130" style="fill:black;stroke:none">Abracadabra</text>
```

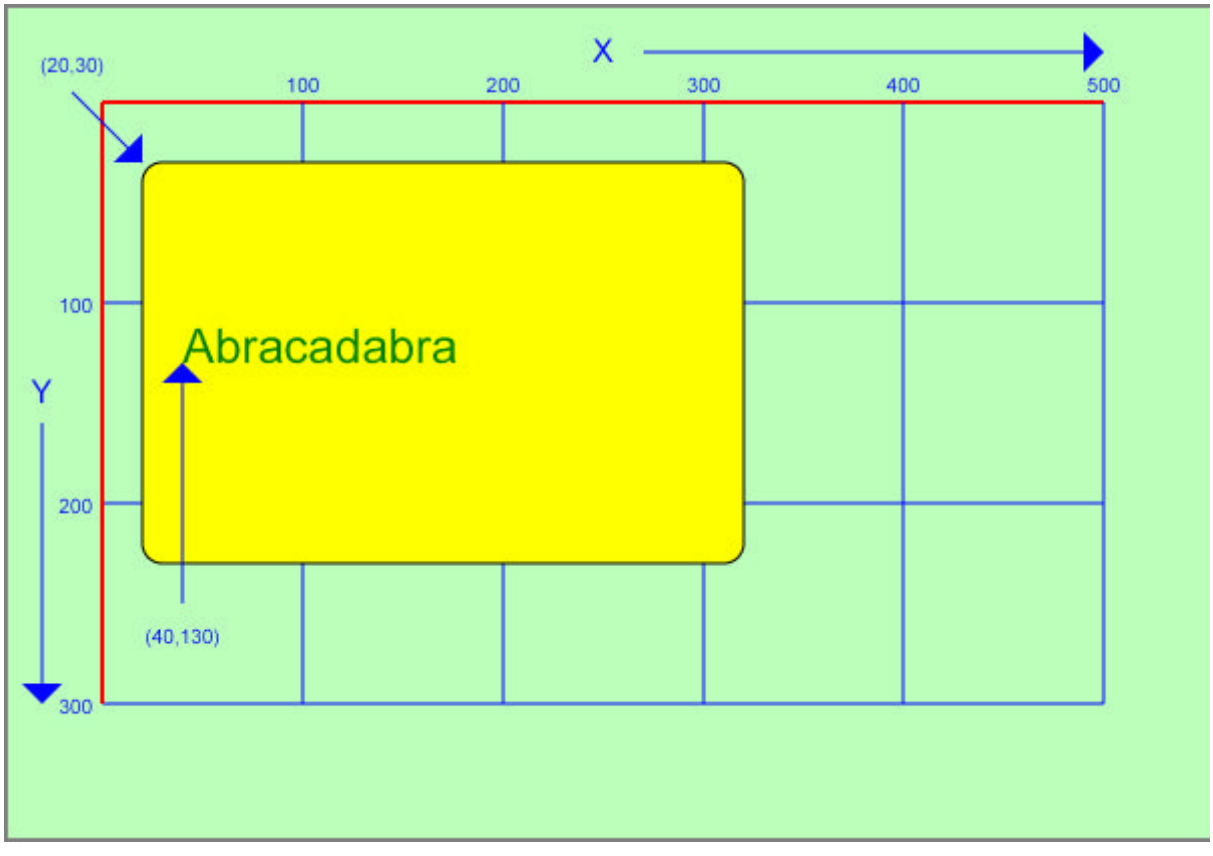

Figure 2.1: SVG Coordinates

The first two attributes, **x** and **y**, of the **rect** element define the origin of the rectangle. The second two define its width and height. The **rx** and **ry** attributes define the radius to be used in rounding the corners. Finally, the **style** attribute defines its rendering. For the **text** element, the first two attributes, **x** and **y**, define the origin of the text string while the third attribute defines the rendering.

The first thing to notice is that the Y-axis in SVG points downwards. This can be a source of error when defining SVG diagrams so take extra care to remember this fact! The X-axis does go from left to right. The origin of the text by default is at the left-hand end of the text on the baseline. By convention the height of the text when used in an HTML page is the same as the medium size text in the HTML page. The font used is at the choice of the browser and plug-in.

2.2 Coordinates

The coordinate system used by the SVG diagram as a whole, when displayed as part of a web page, is a negotiation between the SVG plug-in, what the user would like and the real estate available from the browser.

A complete SVG document containing the drawing defined above could be:

```
<svg viewBox="0 0 500 300">
<rect x="20" y="30" width="300" height="200" rx="10" ry="10"
style="fill:yellow;stroke:black" />
<text x="40" y="130" style="fill:black;stroke:none">Abracadabra</text>
</svg>
```

This could be embedded in an HTML page by the **object** element:

```
<object width="500" height="300" data="figure.svg" alt="SVG Drawing"
type="image/svg-xml">
<img src="figure.png" alt="Alternative PNG image" width="500" height="300"/>
</object>
```

This situation is reasonably straightforward. The **svg** element has a **viewBox** attribute that requests that the area from (0,0) to (500,300) in user coordinates is visible in the browser window. The **object** element requests an area 500 wide and 300 high to put the diagram in. As no units are specified, the assumption is that the units requested are the browser's view of what a pixel width is. Assuming this area is available, the diagram will appear 500 pixels wide and 300 pixels high. A unit in the diagram will be equivalent to a pixel as specified by the browser.

In this Primer, we shall assume that the size of the SVG diagram is defined by the **viewBox** attribute and that the **object** element achieves a mapping of this into an equivalent area on the web page. There are other ways of defining the size of the SVG diagram and it can be specified in units other than pixels. The negotiation can be quite complex if the area required is unavailable or the units are real world ones (centimetres, say) and if the aspect ratio of the requested area is different from the area used by the SVG document. This is outside the scope of a short introduction.

2.3 Rendering Model

Most of the drawing elements in SVG define an area to be rendered. Both **rect** and **text** elements define areas. In the case of **rect** it is the area inside the defined rectangle while for **text** it is the area inside the glyphs making up the individual characters.

The rendering model used by SVG is the one called the **painter's model** which is similar to the way an artist would paint an oil painting. In a simple SVG diagram, the painter starts at the first element to be rendered and paints an area defined by the element. The artist then paints the second element and so on. If the second element is painted in the area occupied by the first element than it will obscure the first element unless the paint being applied is semi-transparent. Both the interior and the edge have to be painted. In SVG, the interior is painted followed by the edge. Consequently, the edge is visible and not partly obscured by the interior. In our example diagram, if the **rect** element had been after the **text** element, nothing would have been seen of the **text** element as the **rect** element would have been painted completely over it.

2.4 Rendering Attributes and Styling Properties

Recall that HTML is a markup language for marking up the content of a textual document. The styling of that document is achieved by defining the style to be applied to each of the markup elements. For example, the **<p>** element produces justified text, the **<h1>** element is bold and in red etc. Similarly, SVG defines the content of a diagram which may be styled in different ways. However, in graphics it is less clear what is style and what is content. For example, a pie chart might use colours to differentiate between individual segments. As long as it provides that differentiation, the specific colour chosen is normally not very relevant. On the other hand, if the diagram depicts a traffic light, interchanging the area to be drawn in green with the one in red would not be a good idea. This applies to most of the rendering attributes in SVG. Consequently the decision was made to allow all the rendering attributes to either be regarded as styling or as an integral part of the content of the diagram.

The use of styling is an extension of the use of styling in HTML. Styling can be achieved by adding a style element to the SVG file:

```
<svg viewbox= "0 0 500 300" >
<style type="text/css">
<![CDATA[
rect {stroke:black;fill:yellow}
rect.different {stroke:red; fill:none}
]]>
</style>
<rect x="20" y="30" width="300" height="200" rx="10" ry="10" />
<rect class="different" x="20" y="330" width="300" height="200" rx="10" ry="10" />
</svg>
```

In this example, the first rectangle will be drawn in yellow with a black boundary whereas the second will be drawn with a red boundary and no internal fill as it belongs to the class different which has a more precise styling than rectangles in general. The stylesheet is enclosed within a CDATA construct to ensure that XML does not do any processing on the style rules. The same effect could be achieved by defining an external sheet in the file mystyle.css as:

```
rect {stroke:black;fill:yellow}
rect.different {stroke:red; fill:none}
```

and attaching it to the SVG document by:

```
<?xml-stylesheet type="text/css" href="mystyle.css" ?>
<svg viewbox= "0 0 500 300" >
<rect x="20" y="30" width="300" height="200" rx="10" ry="10" />
<rect class="different" x="20" y="330" width="300" height="200" rx="10" ry="10" />
</svg>
```

Finally, each element may use the **style** attribute directly:

```
<rect style="stroke:black;fill:yellow" x="20" y="30" width="300" height="200" rx="10"
ry="10" />
<rect style="stroke:red; fill:none" x="20" y="330" width="300" height="200" rx="10"
ry="10" />
</svg>
```

The rules of precedence between linking to an external style sheet, embedding and importing style sheets, attaching styling to an element and user defined style sheets are the same as for CSS when used with HTML.

The alternative method of controlling the rendering of an element is to use the rendering attributes directly:

```
<rect x="20" y="30" width="300" height="200" rx="10" ry="10" fill="yellow"
stroke="black" />
<rect stroke="red" fill="none" x="20" y="330" width="300" height="200" rx="10"
ry="10" />
```

Each property that can be defined as part of the **style** attribute associated with the element can also be defined as a separate attribute. The local effect is the same in both cases. Rather than switch between the two approaches, in this Primer we will define all the local and global rendering via styling. Readers should be aware that they have the choice. A good basis for making a global choice is to use styling when the rendering is not content and use the individual attributes when the rendering is part of the content. Mixing the two does not give the effect that a graphics programmer might anticipate. If you use a rendering attribute, it has lower precedence than any styling introduced by a style sheet. In consequence, if you use rendering attributes do not use style sheets at all.

2.5 Following Examples

To avoid a great deal of duplication, all the following examples are assumed to have an outer **svg** element and associated stylesheet as follows:

```
<svg viewbox= "0 0 600 400" >
<title>Title of Drawing</title>
<desc>This is a long description about what this drawing is about<desc>
<style type="text/css">
<![CDATA[
rect {stroke:black;fill:white;}
line {stroke:firebrick;stroke-width:2}
path {fill:firebrick;stroke:none}
text {font-family:Verdana;font-size:14;fill:darkblue;font-weight:bold}
]]>
</style>
<rect x="1" y="1" fill="#bbffbb" width="598" height="398"/>
<!-- ********** Coloured Screen Area 600 by 400 ********** -->
<!-- ********** Examples added here ********** -->
</svg>
```

The **title** element is normally added straight after the **svg** element and it may be made available to the user by the browser. Similarly, the **desc** element can be used to provide comments throughout a document. Normally it is the first element after a **g** element.

This produces the background for a set of diagrams defined on the (0,0) to (600,400) space as follows:

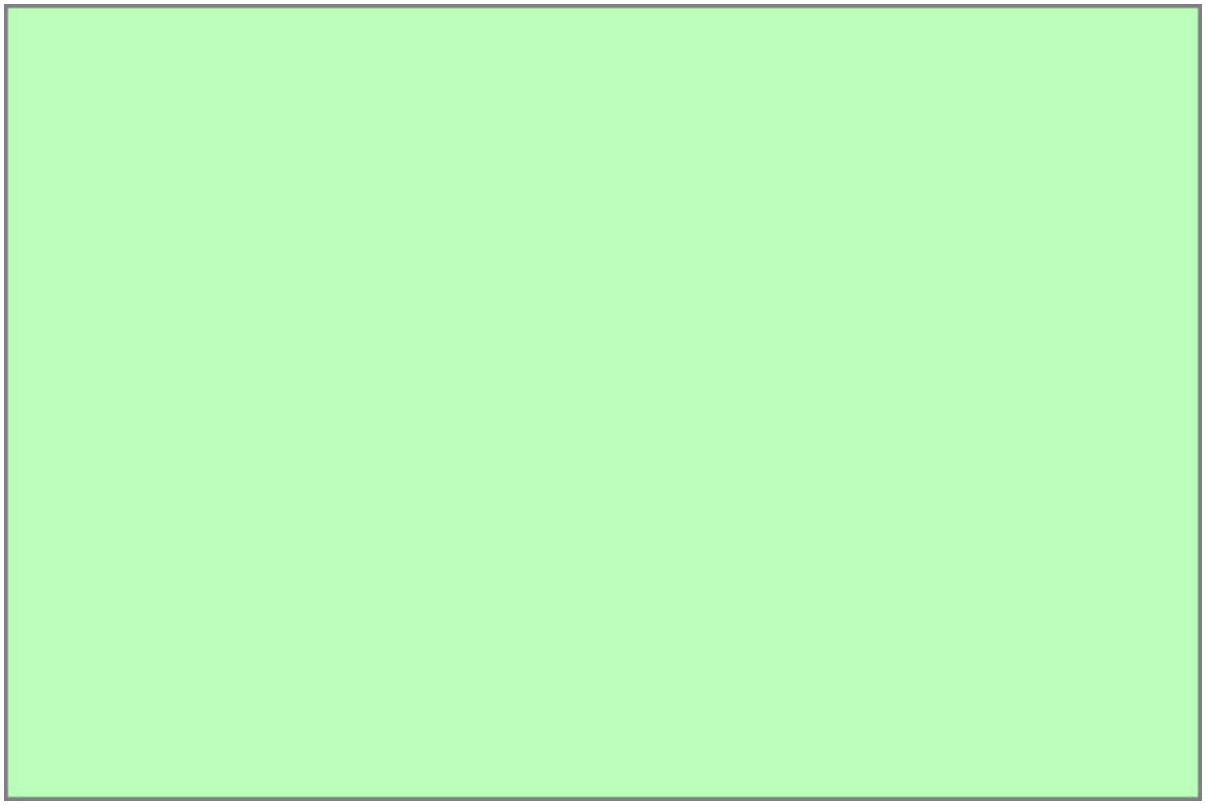

Figure 2.2: Slide background, 600 by 400

The rectangle is set one pixel in from the edge to make sure all the border is visible.

3: SVG Drawing Elements

3.1 Path and Text

The two main drawing elements in SVG are **path** and **text**. There is a set of basic shape drawing elements like **rect** that are essentially shorthand forms for the path element. We will discuss these later. SVG is designed as a transmission format for schematic diagrams in the widest sense. Thus it should be applicable to simple graphs and flow diagrams but also be efficient for CAD diagrams, maps, etc. This means that the main drawing elements must be efficient in quite a wide set of areas. Attention needs to be paid to efficient transmission of complex paths and demanding text.

3.2 Path

The **path** element defines a shape that can be open or closed. A path consists of a sequence of path segments and in many cases this is a single path segment in which case path and path segment are synonymous. Each path segment consists of a sequence of commands where the first defines a new current position and the remainder define a line or curve from the current position to some new position which becomes the current position for the next part of the curve and so on. The form of the **path** element is as follows:

```
<path d="M 0 0 L 100 100">
```

The **d** attribute defines the path. In the example above it defines a path that consists of establishing a current position at the origin (**M**ove to 0,0) and the path goes from there to the point (100,100) as a straight **L**ine. This would be the new current position if there were subsequent commands in the sequence. The following path is a triangle:

```
<path d="M 0 0 L 100 0 L50 100 Z">
```

Here the first line is horizontal from the origin to the point (100,0) and then a straight line goes to the point (50,100). The command Z closes the path with a straight line from (50,100) back to (0,0), the starting position for the path segment.

A path with two path segments would have the form:

```
<path d="M 0 0 L 100 0 L50 100 Z M300,300 L400,300 L350,400 Z">
```

White space has been used to separate the coordinates in the first path segment. Commas can also be used as is shown in the second path segment. For transmission efficiency, surplus separation can be removed. Some of the condensing rules are:

- The coordinate follows the command letter with no intervening space
- Negative coordinates have no separation from the previous coordinate
- Numbers starting with a decimal point need no white space if it is unambiguous
- If the next command is the same as the previous one, the command letter can be omitted

For example:

```
<path d="M0,0L.5.5.8.2Z">
```

This is equivalent to:

```
<path d="M 0, 0 L 0.5, 0.5 L 0.8, 0.2 Z">
```

The basic commands are:

Command	Meaning	Parameters
M	Establish origin at point specified	Two parameters giving absolute (x,y) current position
L	Straight line path from current position to point specified	Two parameters giving absolute (x,y) position of the line end point which becomes the current position.
H	Horizontal line path from current position to point specified	Single parameter giving absolute X-coordinate of the line end point. The Y-coordinate is the same as that of the previous current position. The new point becomes the current position.
V	Vertical line path from current position to point specified	Single parameter giving absolute Y-coordinate of the line end point. The X-coordinate is the same as that of the previous current position. The new point becomes the current position.
Z	Straight line back to original Move origin	No parameters.

If the path being specified consists of many short paths, it may well be more efficient to define the path as relative positions from the previous current position. If the command uses a **lower case letter**, this indicates that the coordinates defined for this command are relative to the previous current position. Figure 3.2 shows some more complex examples.

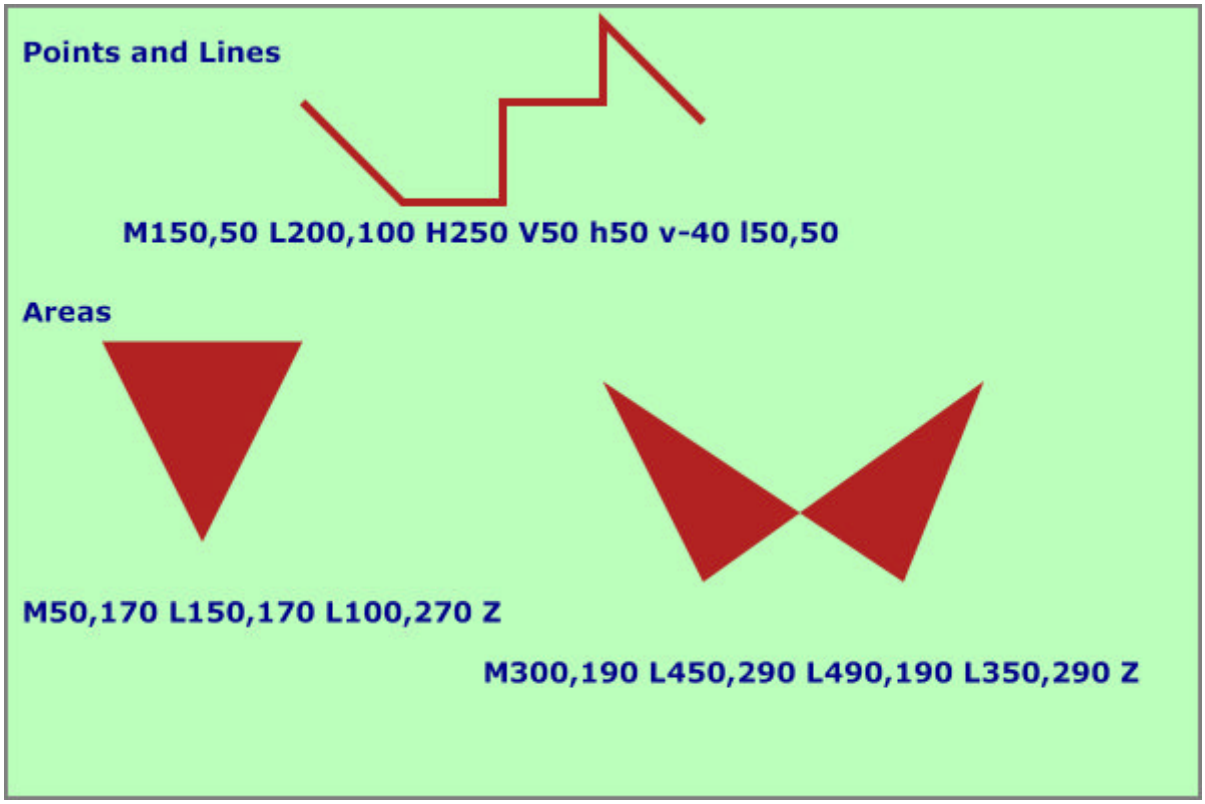

Figure 3.2: Path line commands

The path depicted at the top of the diagram could have been written:

```
<path d="M 150, 50 L 200, 100 L 250, 100 L 250, 50 L 300, 50 L 300, 10 L350, 60">
```

Paths can also be defined as curves (quadratic and cubic bezier, and elliptical arcs). Probably the most useful is the cubic bezier. This has the initial letter **C** and has three coordinates as its parameters. A curved path is defined from the current position (either established by a Move command or a previous line or curve command) to the third point defined in the cubic bezier. The first two points define the bezier control points that give the shape of the curve (Figure 3.2). The positioning of the control points change the shape of the curve under the user's control as can be seen in Figure 3.3. The coordinates used position the curves as they appear on the diagram.

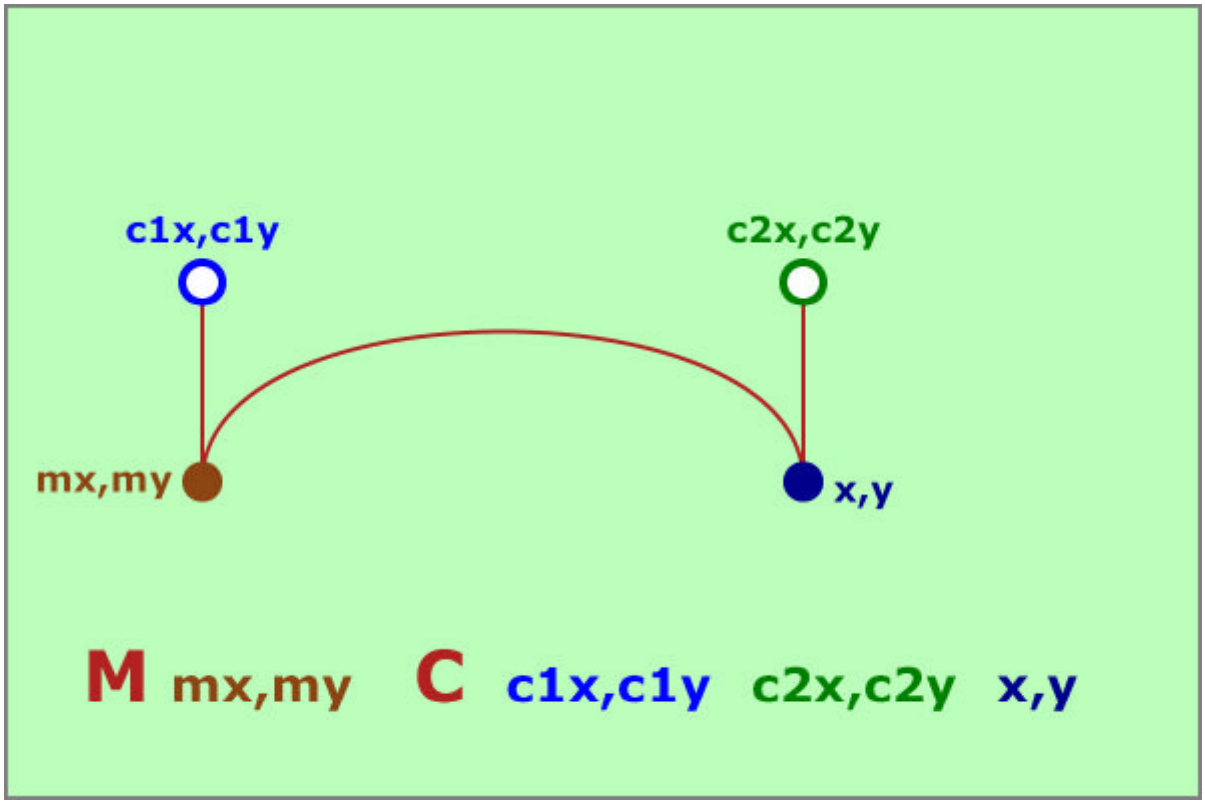

Figure 3.2: Path cubic bezier command

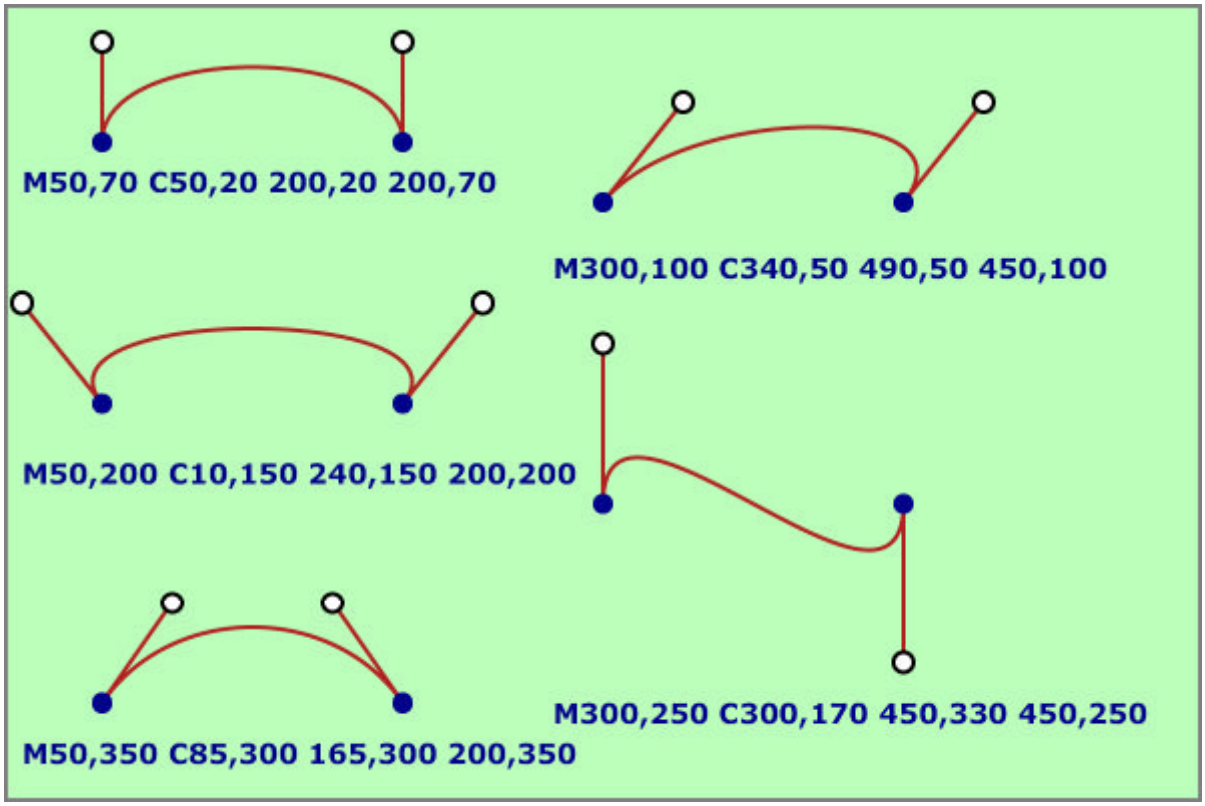

Figure 3.3: Path cubic bezier examples

A real world example is the creation of a duck as shown in Figure 3.4. In the top left the duck has been defined by a set of points and the path is a sequence of straight lines between those points (the points are marked by circles):

```
<path d="M 0 112
L 20 124 L 40 129 L 60 126 L 80 120 L 100 111 L 120 104 L 140 101 L 164 106 L 170
103 L 173 80 L 178 60 L 185 39
L 200 30 L 220 30 L 240 40 L 260 61 L 280 69 L 290 68 L 288 77 L 272 85 L 250 85 L
230 85 L 215 88 L 211 95
L 215 110 L 228 120 L 241 130 L 251 149 L 252 164 L 242 181 L 221 189 L 200 191 L
180 193 L 160 192 L 140 190 L 120 190
L 100 188 L 80 182 L 61 179 L 42 171 L 30 159 L 13 140 Z"/>
```

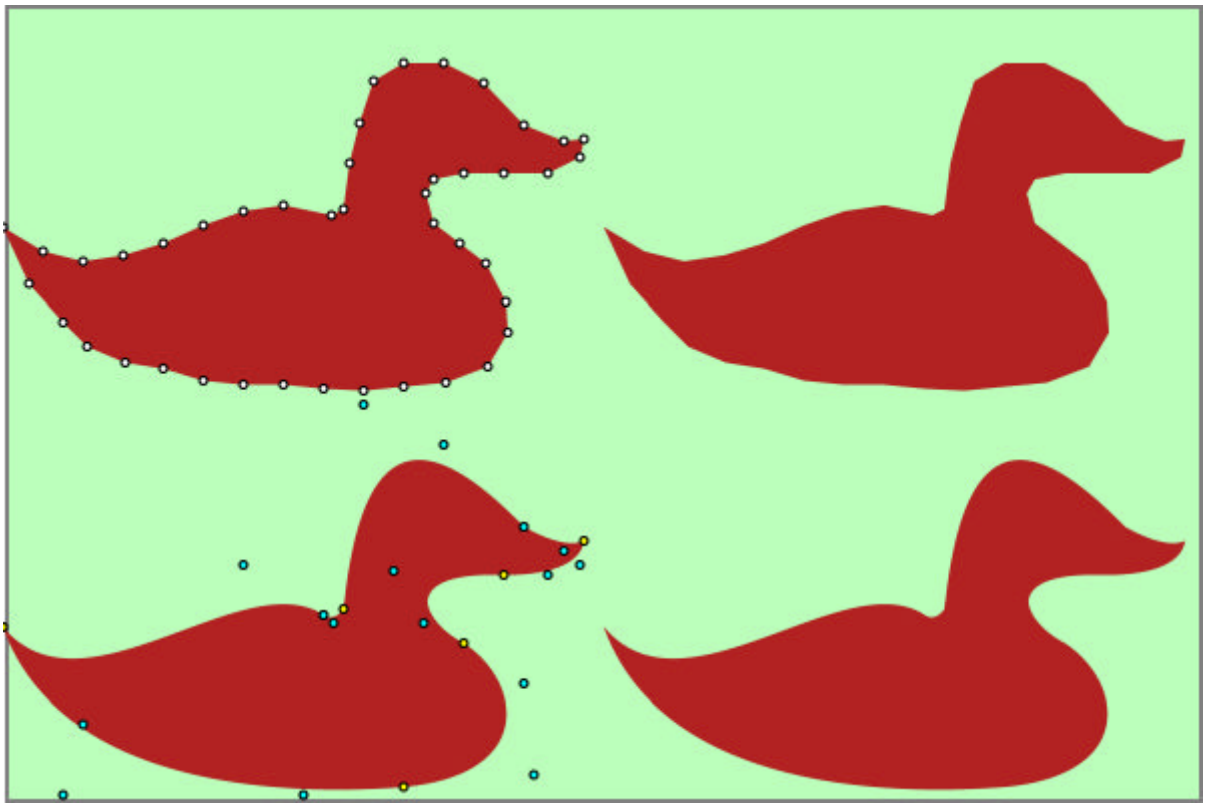

Figure 3.4: Path defined by lines and cubic beziers

The duck without point markers is shown in the top right. In the bottom left the duck has been defined by a set of cubic bezier curves (the control points are marked by aqua circles and the end points by yellow circles) and the duck without the point markers is shown bottom right. The duck defined by bezier curves is:

```
<path d="M 0 312
C 40 360 120 280 160 306 C 160 306 165 310 170 303
C 180 200 220 220 260 261 C 260 261 280 273 290 268
C 288 280 272 285 250 285 C 195 283 210 310 230 320
C 260 340 265 385 200 391 C 150 395 30 395 0 312 Z"/>
```

The number of points in the path defined by lines is 43 while the bezier definition uses 25. The path could also be defined using relative coordinates in which case it would be:

```
<path d="M 0 312
c 40 48 120 -32 160 -6
c 0 0 5 4 10 -3 c 10 -103 50 -83 90 -42
c 0 0 20 12 30 7 c -2 12 -18 17 -40 17
c -55 -2 -40 25 -20 35 c 30 20 35 65 -30 71
c -50 4 -170 4 -200 -79 z"/>
```

Note that it does not really make any difference whether you complete the closed curve with upper or lowercase Z as the effect is identical. Removing unnecessary spaces reduces the path definition to 160 characters compared with the 443 characters in the initial line path representation:

```
<path d="M 0 312c40 48 120-32 160-6c0 0 5 4 10-3c10-103 50-83 90-42c0 0 20 12 30
7c-2 12-18 17-40 17c-55-2-40 25-20 35c30 20 35 65-30 71c-50 4-170 4-200-79 z"/>
```

3.3 Text

The second most important drawing element is text. It has a large number of styling properties that we will discuss later. Here, we will just define the three main elements. Figure 3.5 shows the three main types of text that can be generated:

- Text defined just using the **text** element
- Text that uses the **tspan** element to vary the properties and attributes being used in the text presentation
- Text where the path is defined by the **textPath** element

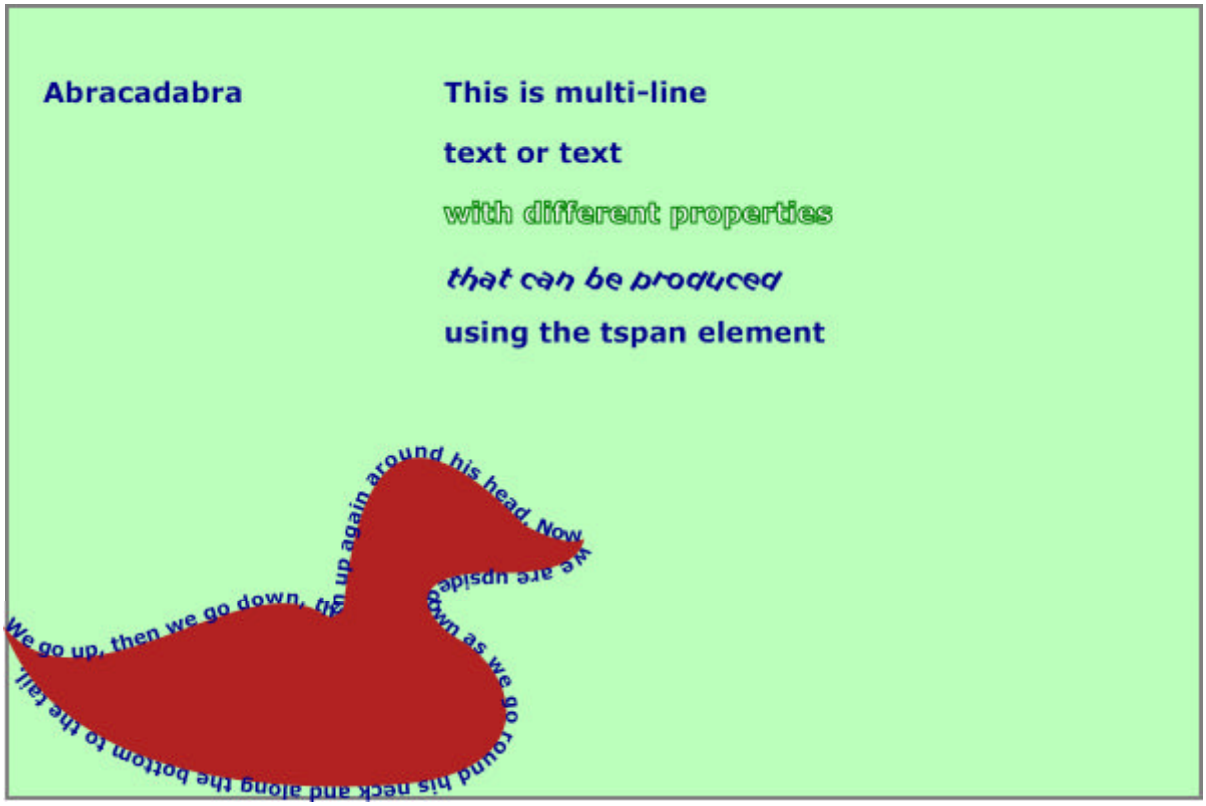

Figure 3.5: Different text elements

```
<text x="20" y="50">Abracadabra</text>

<text x="220" y="20">
<tspan x="220" dy="30">This is multi-line</tspan>
<tspan x="220" dy="30">text or text</tspan>
<tspan x="220" dy="30" style="fill:white;stroke:green">with different properties</tspan>
<tspan x="220" dy="30" rotate="30">that can be produced</tspan>
<tspan x="220" dy="30">using the tspan element</tspan>
</text>

<path id="duck" d="M 0 312 C 40 360 120 280 160 306 C 160 306 165 310 170 303
C 180 200 220 220 260 261 C 260 261 280 273 290 268 C 288 280 272 285 250 285
C 195 283 210 310 230 320 C 260 340 265 385 200 391 C 150 395 30 395 0 312 Z"/>

<text style="font-size:10">
<textPath xlink:href="#duck">
We go up, then we go down, then up again around his head. Now we are upside down
as we go round his neck and along the bottom to the tail.
</textPath>
</text>
```

The use of the **text** element by itself has attributes **x** and **y** that define the origin for the text. The origin is by default at the bottom left of the first character and the characters are displayed from left to right. Attributes associated with the text can change the start position, the characteristics of the text and the drawing direction. We will discuss these later.

If the position of parts of the text or the text's attributes need to change from that which is available using the **text** element, these can be adjusted by including within the **text** element a **tspan** element. The text within a tspan may have its origin specified either by absolute **x** and **y** attributes or relative **dx** and **dy** attributes. The current text position is incremented by the amount specified in the case of the relative attribute. For both **dx** and **dy**, the attribute can be a list in which case the first number defines the increment for the first character, the second defines the increment from that character for the second character and so on. The characters in the text string within the **tspan** element can each be rotated by a defined number of degrees degrees by using the **rotate** attribute. Again, a list of numbers can be provided to define the orientation of each character in the text sequence. Some further examples of tspan usage are shown in Figure 3.6.

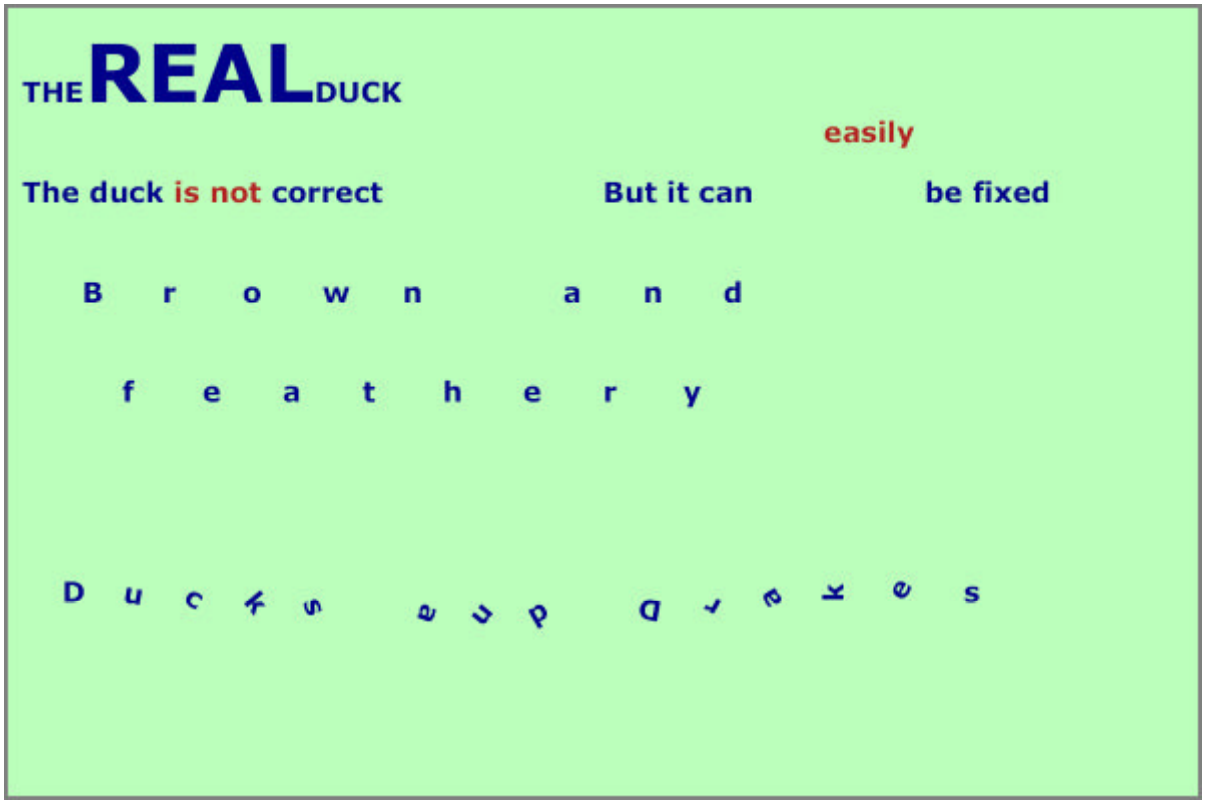

Figure 3.6: Uses for the tspan element

```
<text x="10" y="50" >THE<tspan style="font-size:40">REAL</tspan>DUCK</text>

<text x="10" y="100" >The duck <tspan style="font-weight:bold; fill:firebrick">is
not</tspan> correct</text>

<text x="300" y="100" >But it can <tspan dx="30" dy="-30" style="font-weight:bold;
fill:firebrick">easily </tspan><tspan dy="30">be fixed</tspan></text>

<text>
<tspan x="40 80 120 160 200 240 280 320 360" y="150">Brown and</tspan>
<tspan x="60 100 140 180 220 260 300 340" y="200">feathery</tspan>
</text>

<text>
<tspan x="30 60 90 120 150 180 210 240 270 300 330 360 390 420 450 480 510"
rotate="0 10 30 50 70 90 110 130 150 170 190 210 240 270 300" y="300">Ducks and
Drakes</tspan>
</text>
```

3.4 Basic Shapes

The six basic shape elements in SVG are shorthands for the **path** element. They are **line**, **polyline**, **polygon**, **rect**, **circle** and **ellipse**. The main attributes of each are given in this example (see Figure 3.7) and the meaning of the attributes in the following table.

```
<circle cx="70" cy="100" r="50" />
<rect x="150" y="50" rx="20" ry="20" width="135" height="100" />
<line x1="325" y1="150" x2="375" y2="50" />
<polyline points="50, 250 75, 350 100, 250 125, 350 150, 250 175, 350" />
<polygon points=" 250, 250 297, 284 279, 340 220, 340 202, 284" />
<ellipse cx="400" cy="300" rx="72" ry="50" />
```

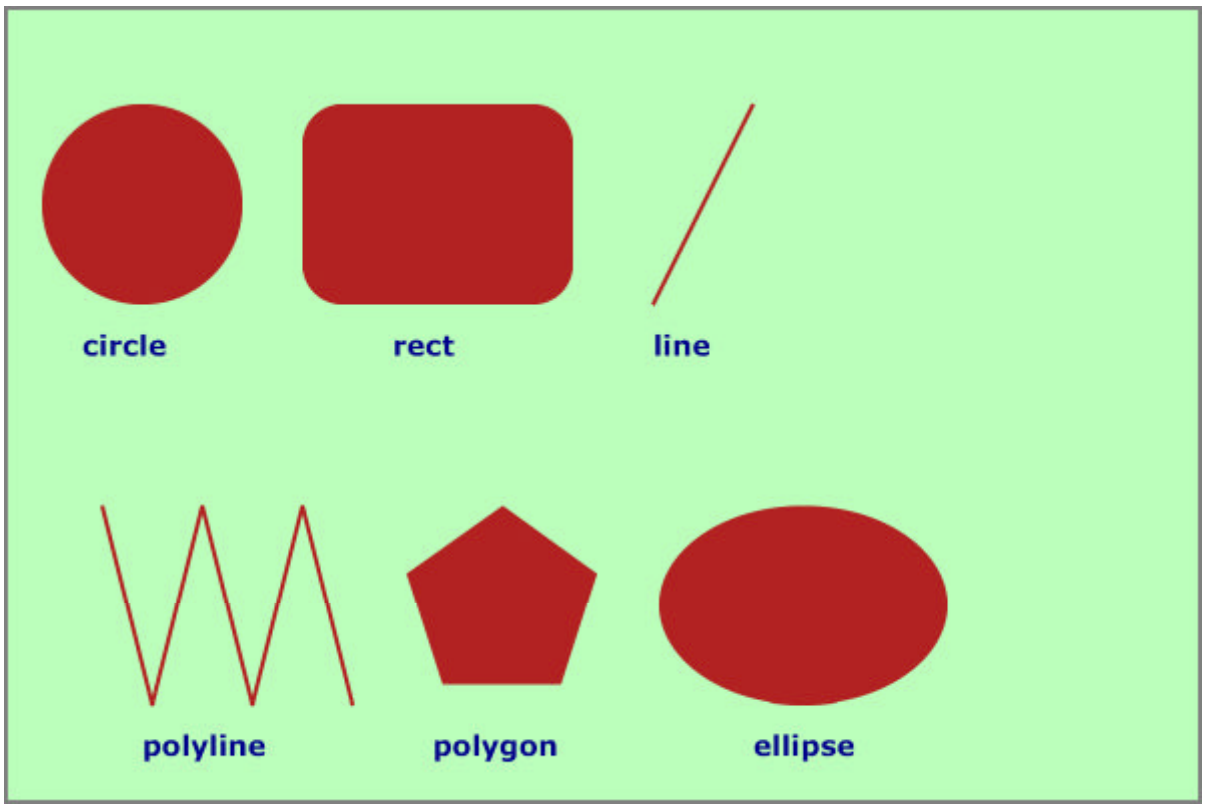

Figure 3.7: Basic elements

Command	Meaning	Parameters
line	Renders a line between two points	x1 and y1 define first point x2 and y2 define second point
polyline	Renders a sequence of lines between points	points defines a sequence of x,y coordinates
polygon	Renders an area defined by a sequence of lines	points defines a sequence of x,y coordinates
rect	Renders a rectangular area	x and y define top left corner width and height define size of rectangle rx and ry define the radii of the elliptic arc that rounds the corners
circle	Renders a circle	cx and cy define the centre r defines the radius
ellipse	Renders an ellipse	cx and cy define the centre rx and ry define the two radii

4. Grouping

4.1 Introduction

Frequently there is a need to group drawing elements together for one reason or another. One reason is if a set of elements share the same attribute. However, probably the major use is to define a new coordinate system for a set of elements by applying a transformation to each coordinate specified in a set of elements. Grouping is also useful as the source or destination of a reference.

Grouping in SVG is achieved by the **g** element. A set of elements can be defined as a group by enclosing them within a **g** element. For example:

```
<g style="fill:red;stroke:black">
<circle cx="70" cy="100" r="50" />
<rect x="150" y="50" rx="20" ry="20" width="135" height="100" />
<line x1="325" y1="150" x2="375" y2="50" />
<polyline points="50, 250 75, 350 100, 250 125, 350 150, 250 175, 350" />
<polygon points=" 250, 250 297, 284 279, 340 220, 340 202, 284" />
<ellipse cx="400" cy="300" rx="72" ry="50" />
</g>
```

The **g** element can have any of the attributes or style properties defined for it that are generally applicable to individual drawing elements. In the example above, all the basic shapes will be rendered with the interior red and the border black.

4.2 Coordinate Transformations

The **transform** attribute applied to a **g** element defines a transformation to be applied to all the coordinates in the group. For example:

```
<g transform="translate(100,0)">
<circle cx="70" cy="100" r="50" />
<rect x="150" y="50" rx="20" ry="20" width="135" height="100" />
</g>
```

Instead of the circle being drawn centred on the point (70,100) it will now be drawn centred on the point (170,100). The rectangle will have a top left corner of (250,50) instead of (150,50). Consequently, a useful method of defining a composition made up of a number of graphical objects is to define each object as a group using the most appropriate coordinate system and then use the transformations applied to the group to construct the graphic as a whole. Groups can be nested to any depth and transformations applied to each. In consequence, a diagram can be constructed out of sub-assemblies that come together to produce objects that are then composed to produce the diagram.

The possible transformations are:

Transformation	Meaning	Parameters
translate	Defines a translation of the coordinates	x and y defining the x and y translation
scale	Defines a scaling of the X and Y coordinates	sx and sy defining the scaling in the X and Y directions s defining the same scaling in the X and Y directions
rotate	Defines a rotation about a point	angle, x and y defining a clock-wise rotation of angle degrees about the point (x,y) angle defining a clock-wise rotation of angle degrees about the origin
skewX	Defines a skew along the X axis	angle degrees defining a skew of the X position by Y*tan(angle)
skewY	Defines a skew along the Y axis	angle degrees defining a skew of the Y position by X*tan(angle)

It is also possible to define a matrix that performs a composite set of transformations.

The **transform** attribute can consist of a sequence of individual transformations in which case they are performed in the order right to left. The same effect can be achieved in a much more readable way by nesting several **g** elements, each with a single transformation. It is recommended that the nested approach is the one taken.

Figure 4.1 gives a montage of various transformations where the text defining the transformation is also transformed.

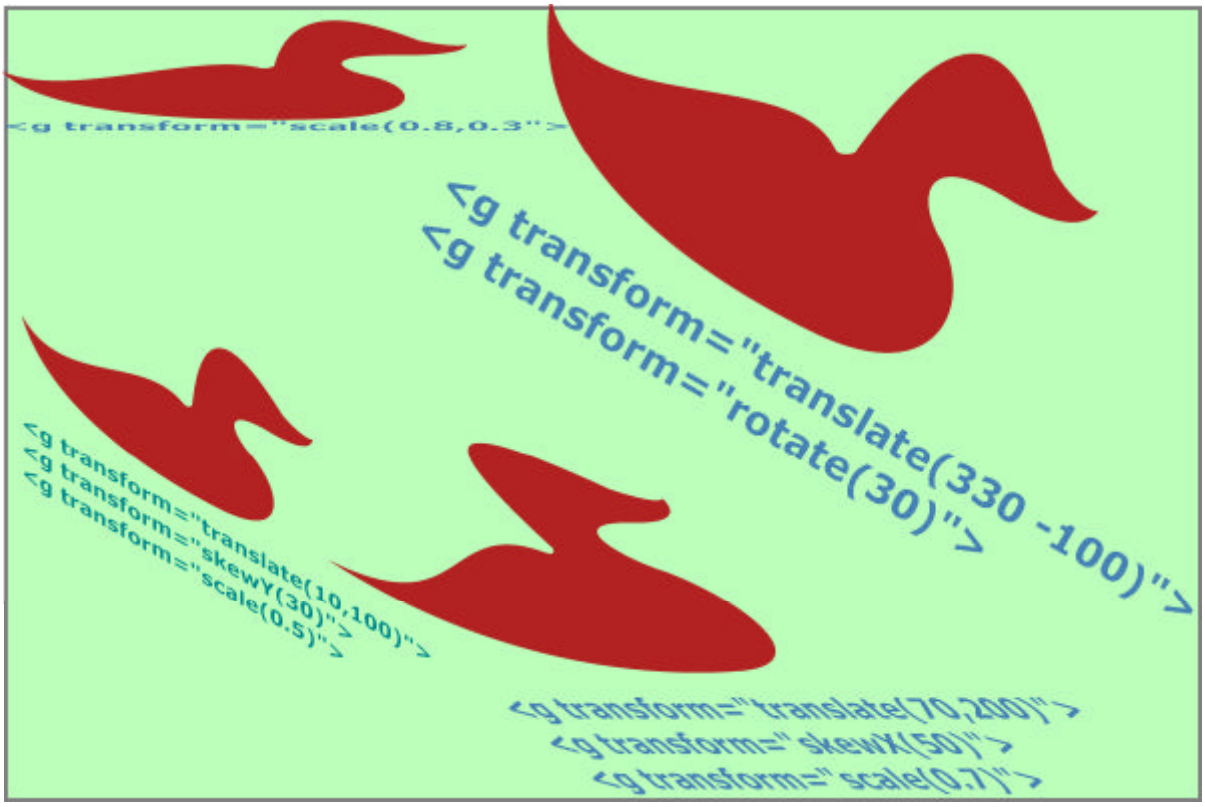

Figure 4.1: Transformations

The **transform** attribute can also be applied to the various drawing elements directly but it tends to be most useful when applied to a group.

4.3 Clipping

A group of elements can be clipped against a clip path which is defined by a **clipPath** element: element:

```
<clipPath id="myClip">
<circle cx="350" cy="100" r="50"/>
</clipPath>

<g style="stroke:none;clip -path:url(#myClip)">
<rect style="fill:red" x="0" y="0" width="500" height="20" />
<rect style="fill:white" x="0" y="20" width="500" height="20" />
<rect style="fill:blue" x="0" y="40" width="500" height="20" />
<rect style="fill:red" x="0" y="60" width="500" height="20" />
<rect style="fill:white" x="0" y="80" width="500" height="20" />
<rect style="fill:blue" x="0" y="100" width="500" height="20" />
<rect style="fill:white" x="0" y="120" width="500" height="20" />
<rect style="fill:blue" x="0" y="160" width="500" height="20" />
<rect style="fill:red" x="0" y="180" width="500" height="20" />
<rect style="fill:white" x="0" y="200" width="500" height="20" />
<rect style="fill:blue" x="0" y="220" width="500" height="20" />
<rect style="fill:red" x="0" y="240" width="500" height="20" />
<rect style="fill:white" x="0" y="260" width="500" height="20" />
<rect style="fill:blue" x="0" y="280" width="500" height="20" />
<rect style="fill:red" x="0" y="300" width="500" height="20" />
<rect style="fill:white" x="0" y="320" width="500" height="20" />
</g>
```

The group of rectangles are clipped against the **circle** basic shape. The **clipPath** element has an **id** attribute and the **g** element has a style or attribute **clip-path** that specifies the path to be used for clipping. It is also possible to clip against a path or even text:

```
<clipPath id="myClip">
<path d="M 0 112 C 40 160 120 80 160 106 C 160 106 165 110 170 103 C 180 0 220
20 260 61 C 260 61 280 73 290 68 C 288 80 272 85 250 85 C 195 83 210 110 230 120
C 260 140 265 185 200 191 C 150 195 30 195 0 112 Z"/> </clipPath>
```

```
<clipPath id="myClip">
<text x="10" y="310" style="font-size:150">DUCK</text>
</clipPath>
```

For referenced items, such as clip paths, it is considered good practice to surround them with a **defs** element to emphasise that they are not rendered directly. The **defs** element acts rather like a **g** element that has the **visibility** attribute set to **hidden**.

Figure 4.2 shows the results of the three clipping paths defined above.

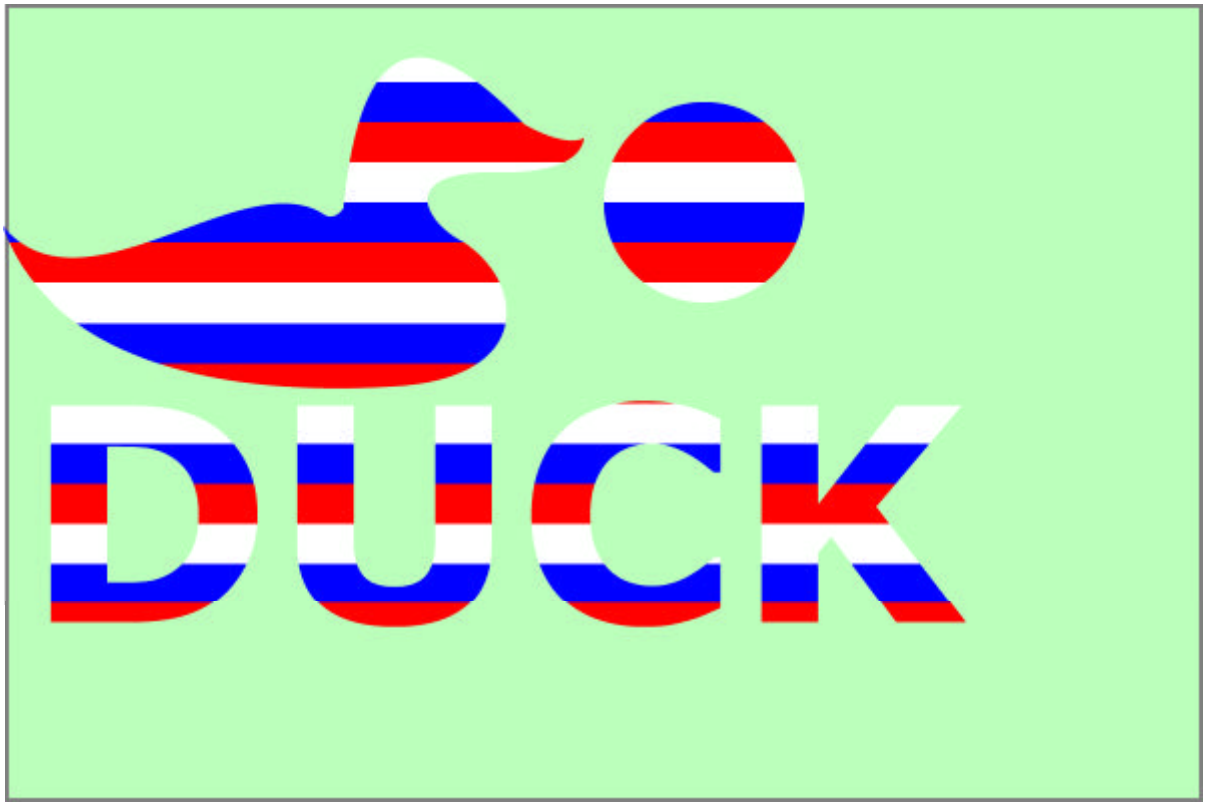

Figure 4.2: Clipping

5. Filling

5.1 Fill Properties

The main fill properties that can be defined as either attributes or properties of SVG basic shapes, paths, text or groups are:

- **fill**: the method of filling the area with a solid colour or gradient. The value **none** indicates that the area is not to be filled.
- **opacity**: how opaque the paint is that fills the area
- **fill-rule**: for complex paths, the definition of the area to be filled

An example setting all three might be:

```
<path style="fill:red;opacity:0.5;fill-rule:evenodd" d="M10,20h100v50h-80v-70h-20v20z" />
```

The **fill** property can either define a colour to be used to paint the area or it can define a colour gradient. We will discuss how colours are specified first and leave the specification of gradients until later.

5.2 Colour

Colour values are used for various operations in SVG including filling and stroking. Colour can be defined in the same set of ways that it can be defined in CSS:

- A colour name such as red, blue, green etc.
- A numerical RGB specification defining the red, green and blue components of the colour in one of three ways:
  - rgb(r,g,b) where r, g and b are values in the range 0 to 255
  - #rgb where r, g and b are hexadecimal values (for example #f00)
  - #rrggbb where rr, gg and bb define a value in the range 0 to 255 as two hexadecimal values

The four rectangles defined below will all be filled with approximately the same colour (the short hexadecimal form does not quite have the required accuracy).

```
<rect width="10" height="10" style="fill:coral" />
<rect width="10" height="10" style="fill:rgb(255,127,80)" />
<rect width="10" height="10" style="fill:#f75" />
<rect width="10" height="10" style="fill:#ff7f50" />
```

There are over 140 colour names defined in SVG and these are given in Appendix A. Figure 5.1 shows a sample of the colours available.

Figure 5.1: Some SVG Colours

5.3 Fill Rule

Filling an area defined by a path, basic shape or text requires there to be a clear definition of what is inside the path and should be filled and what is outside. For simple paths that do not cross, the inside is fairly obvious. However, for a path that intersects itself or is made up of a number of segments (such as a donut shape), the definition of inside and outside is less clear. SVG defines two different methods of defining inside and the user may use either:

- **evenodd**: the number of intersections that a line between a point and infinity makes with the path are counted. If the number is odd, the point is inside the area and should be filled.
- **nonzero**: the number of times the path winds around a point is counted. If the number is non-zero, the point is inside.

Figure 5.2 shows the different results obtained for two paths. Note that it is necessary to know the order in which the two triangles are drawn in order to define the area. If the second triangle had been drawn in the order 5, 4, 6 the area inside for both the evenodd and nonzero methods would have been the same. For simple shapes, staying with the evenodd rule is a good bet.

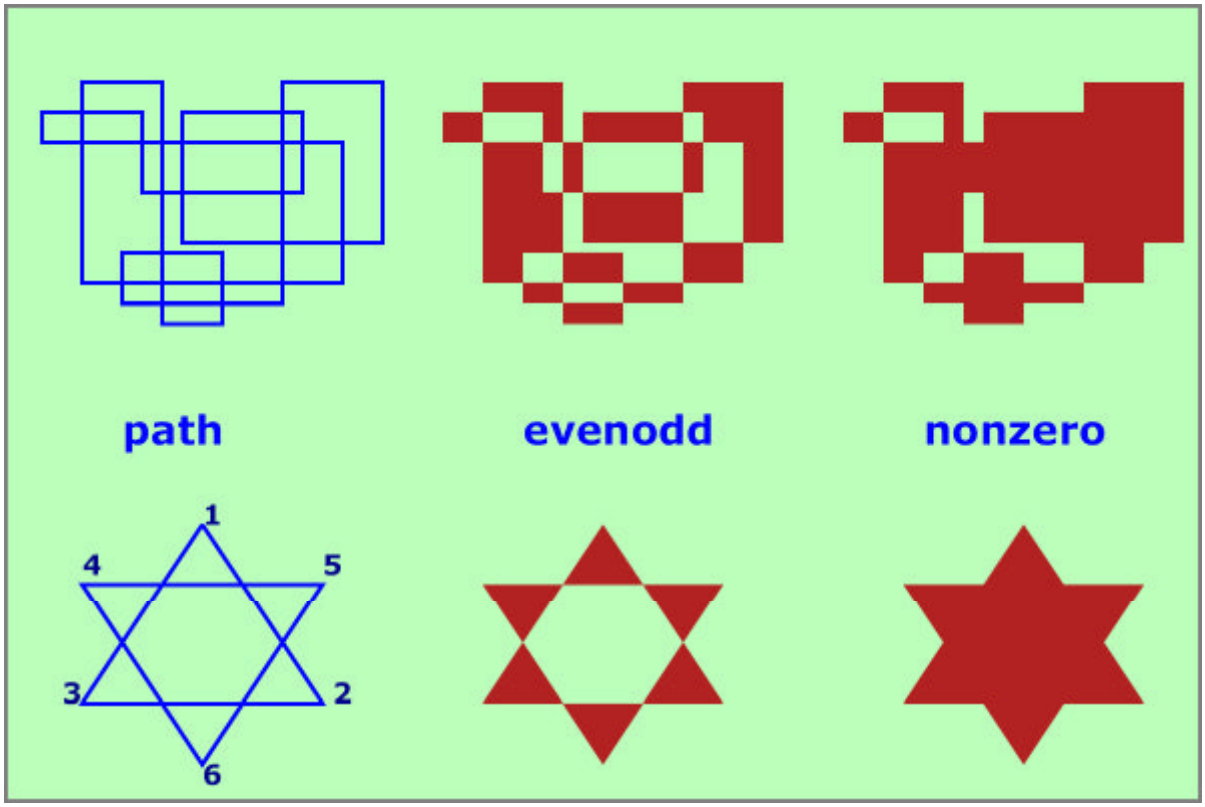

Figure 5.2: Fill Rules

5.4 Opacity

Graphics in SVG are not restricted to being invisble or opaque. It is possible for any area to be filled at an opacity that varies between 1.0 (opaque) and 0.0 (transparent). Properties are available for specifying the opacity of filling and stroking separately but for simple examples it is sufficient to use the **opacity** to control both stroke and fill opacity together. Rules exist for combining two semi-transparent objects that overlap each other. Figure 5.3 shows various objects of different levels of transparency overlapping. In the case of the ducks, they are drawn top to bottom with decreasing opacity. The more opaque duck behind the more transparent one often looks as though it is in front rather than behind where they overlap due to this combination.

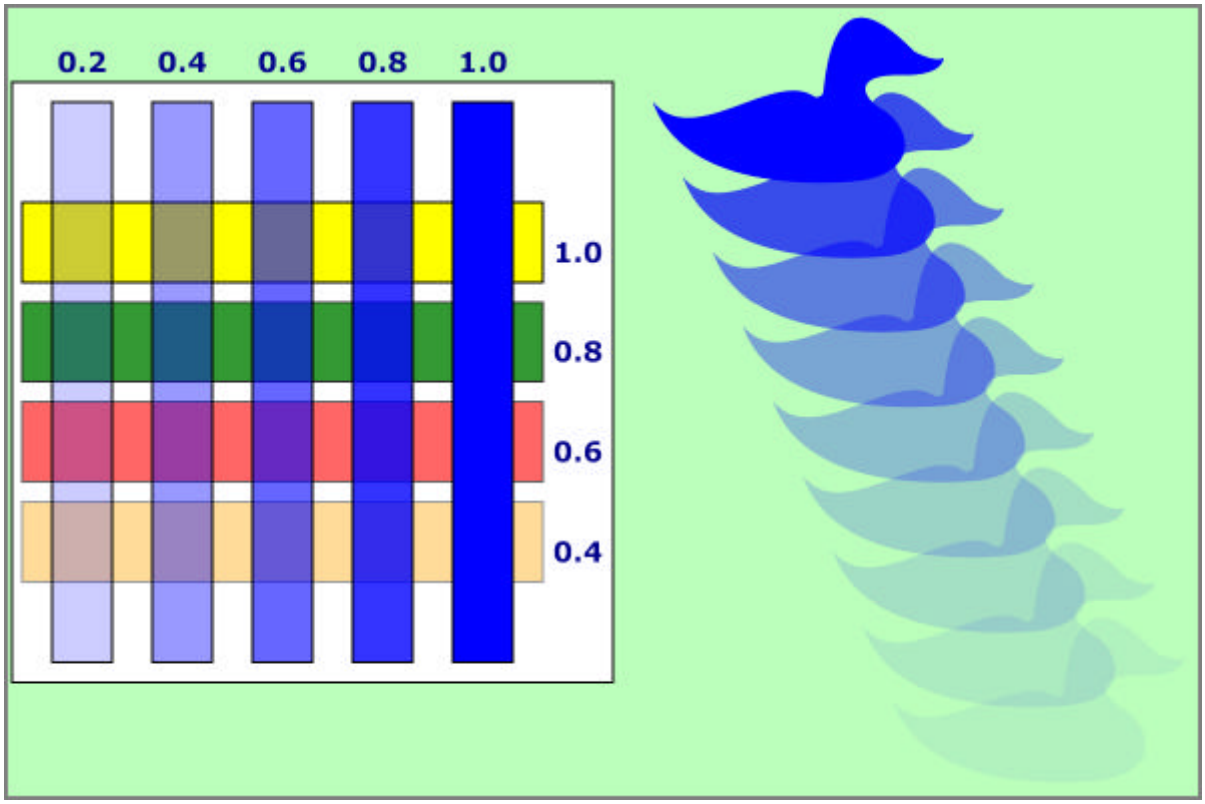

Figure 5.3: Opacity

5.5 Colour Gradients

As mentioned earlier, the **fill** property can have more exotic values than a simple colour specification. One of these is to specify a colour gradient that defines a gradation of colour across the area to be filled and that gradient can change from one colour to another or range across a whole gamut of colours. It is possible also to specify whether the gradient is a linear transformation from one point to another or radiates from some origin.

The colour specification in the **fill** property points to a URL where the gradient is defined:

```
<rect x="20" y="20" width="290" height="120" style="fill:url(#MyGradient)"/>
```

Here the **fill** property is defined by pointing at the definition MyGradient. The gradient specification has the form:

```
<linearGradient id="MyGradient" gradientUnits="userSpaceOnUse" x1="80" y1="44"
x2="260" y2="116">
<stop offset="0" style="stop-color:blue"/>
<stop offset="0.5" style="stop-color:white"/>
<stop offset="1" style="stop-color:green"/>
</linearGradient>
```

This is the one used at the top right hand side of Figure 5.4. The element is either a **linearGradient** or a **radialGradient**. Note the use of Camel case with each word separating the previous one by capitalising the first character. This is used throughout SVG. The main element defines the major parameters of the gradient and the **offset** element defines the way the gradient is rendered in more detail.

In this particular example, the main attributes of the linear gradient are the **id** used to associate it with its use, the point (x1,y1) that defines the start of the gradation and the point (x2,y2) that defines where the gradation ends. Outside this range, the first and last values are retained. This allows the user to define a middle part of the fill as being graded while the remainder has the solid colours defined at the start and end. In the top left part of the Figure, the start, middle and end offset positions are identified by circles.

In the example above, positions are defined between (x1,y1) and (x2,y2) where certain colours will appear. In this example, the colour at (x1,y1) (offset=0.0, the starting position) will be blue and at (x2,y2) (offset=1.0, the finishing position) it will be green. Half way between, the colour will be white (offset=0.5). The number of offsets can be as many as you like as can be seen in the top right where the duck has a large number of offsets specified.

Defining radial gradients is slightly more complex:

```
<radialGradient id="MyGradient2" gradientUnits="userSpaceOnUse" cx="130"
cy="270" r="100" fx="70" fy="270">
<stop offset="0" style="stop-color:blue"/>
<stop offset="0.5" style="stop-color:white"/>
<stop offset="1" style="stop-color:green"/>
</radialGradient>
<rect x="20" y="160" width="290" height="220" style="fill:url(#MyGradient2)"/>
```

The **radialGradient** specifies a circumference where the offset=1.0 value is defined by defining its centre **(cx,cy)** and the radius **r**. The easy option would have been to define the centre (cx,cy) as the offset=0.0 position. instead, a separate offset=0.0 position is defined separately as (fx,fy). The offset=1.0 position is shown in the diagram by the yellow circle and the circle shows the position of the focus. Again, the offset elements define the colours at inbetween positions.

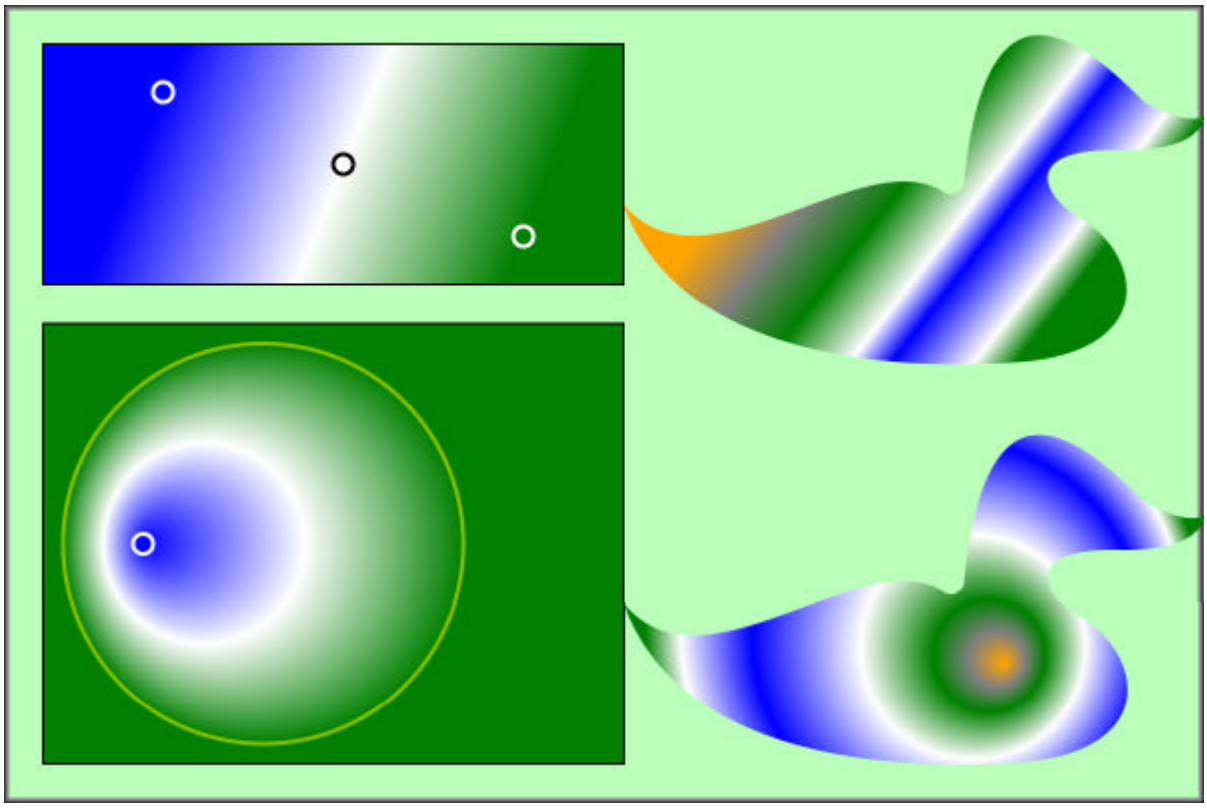

Figure 5.4: Colour Gradients

Once more, a simple example is shown on the lower left and a more complex radial gradient is shown on the lower right.

6. Stroking

6.1 Stroke Properties

A subset of the complete set of stroke properties is:

- **stroke**: the method of rendering the outline with a solid colour or gradient. The possible values are the same as for the **fill** property. A value of **none** indicates that no outline is to be drawn.
- **stroke-width**: defines the width of the outline.
- **stroke-dasharray**: defines the style of the line (dotted, solid, dashed etc).
- **stroke-dashoffset**: for a dashed line, indicates where the dash pattern should start.
- **stroke-linecap**: defines the way the end of a line is rendered.
- **stroke-linejoin**: defines how the join between two lines is rendered.
- **stroke-linemiterlimit**: places some constraints on mitered line joins.

The set of stroke properties are illustrated in Figure 6.1.

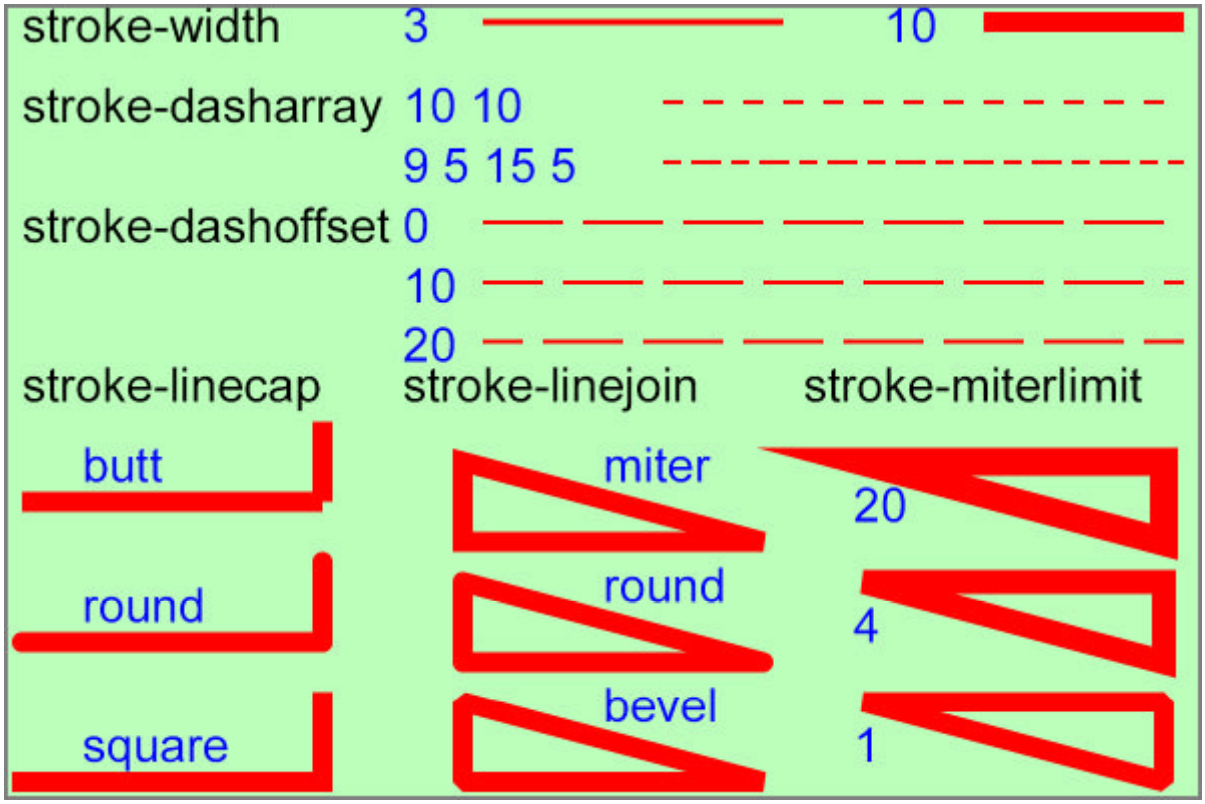

Figure 6.1: Stroke Properties

6.2 Width and Style

The property **stroke-width** property defines the width of the line in the units specified. All the transformations that apply to the graphic object also apply to the stroke-width. So scaling an object by a factor 2 will also double the stroke width. A value of zero is equivalent to setting the value of the property **stroke** to **none**.

Outlines are normally rendered as solid lines. To render them dashed or dotted, the **stroke-dasharray** property has to be set. Its value is either **none** to indicate a solid line or is a set of numbers that specify the length of a line segment followed by the length of the space before the next segment followed by the next line segment and so on. Figure 6.1 shows two examples. The first (stroke-dasharray:10 10) defines a dashed line where the dashes and spaces between are 10 units long. The second example (stroke-dasharray:9 5 15 5) defines a line consisting of short and long dashes with a 5-unit space between each. If an odd number of values is given, these are repeated to give an even number. So **9 5 15** is equivalent to **9 5 15 9 5 15**. Commas rather than spaces can be used to separate the values.

Normally the rendering of the outline will start with the first value in the **stroke-dasharray** list. If this is not what is required, the **stroke-dashoffset** property specifies how far into the dash pattern to start the rendering. For example, a value of **16** in the example **9 5 15 9 5 15** above would mean the stroke rendering would start **13 9 5 15** etc, that is the first dash and space plus the first 2 units of the second dash.

6.3 Line Termination and Joining

When a line or path is terminated, the normal result is to **butt** the end of the line (the line finishes at the end point and the end of the line is perpendicular to the direction of the line).

In Figure 6.1, the poor rendering this achieves when two lines are drawn from the same point is shown. To combat this, two other values can be specified by the **stroke-linecap** property. If set to the value **round**, a semi-circle is added to the end of the line while the value **square** extends the line by the width of the line. In both cases the rendering of two lines or paths coincident at a point will be improved.

A similar problem occurs at intermediate points in a path made up of straight line segments. The normal result is to **miter** the two lines (the outer edges are extended until they meet). This is not always the most pleasing effect. Two other values can be specified by the **stroke-linejoin** property. A value of **round** rounds off the join and **bevel** squares off the join of the two lines.

The **miter** line join looks particularly unattractive when the two line segments are at a small angle to each other (see Figure 6.1). For the **miter** value of **stroke-linejoin**, it is possible to control the extent that the miter extends beyond the end of the line. The property **stroke-miterlimit** defines a value greater than 1 which is the maximum ratio allowed between the miter length and the stroke width. If this ratio is exceeded, the line join has a bevel applied to it. In Figure 6.1, the value of 4 bevels off the worst of the three joins while the value of 1 bevels all of the three joins.

7. Text

7.1 Rendering Text

There are more properties associated with the **text** element than any other. Many are still to be fully implemented in the products currently on the market. Many are concerned with achieving good results when the text is non-European requiring a different writing direction from left-to-right and even bi-directional text (in Hebrew, for example, the writing direction is normally right-to-left but embedded European words are written left-to-right).

The properties are a superset of the ones defined in CSS.

7.2 Font Properties

Figure 7.1 shows some of the properties that are primarily concerned with how individual characters are rendered.

The **font-family** property defines the font to be used for the text. The **font-size** property defines the size of the characters using one of the SVG unit measures.

The **font-style** property has the values **normal**, **italic** and **oblique**.

The **font-weight** property defines the boldness of the rendering and has the same set of possible values as those used in CSS. Similarly, the **text-decoration** property has the same possible values as those used in CSS.

Text is rendered in a similar way to paths and both the interior fill of the characters and the stroke to be used for the outline can be specified by the **fill** and **stroke** properties.

Figure 7.1: Font Properties

7.3 Text Properties

One of the most useful properties associated with the whole text string is the **text-anchor** property (Figure 7.2) which specifies where in the text string the text origin is located. This is particularly useful when trying to centre text, say, within a rectangle. In this case defining the origin at the middle position in the x-direction and defining the value as **middle** will achieve the desired result.

Simple formulae can be rendered using the **baseline-shift** property. The example in Figure 7.2 requires the following:

```
<text x="10" y="240" style="fill:blue" >x
<tspan style="baseline-shift:super">super</tspan>
+y
<tspan style="baseline-shift:sub">sub</tspan>
+1
</text>
```

The **writing-mode** property defines the direction that the text is drawn. The possible values are **lr**, **tb**, and **rl**.

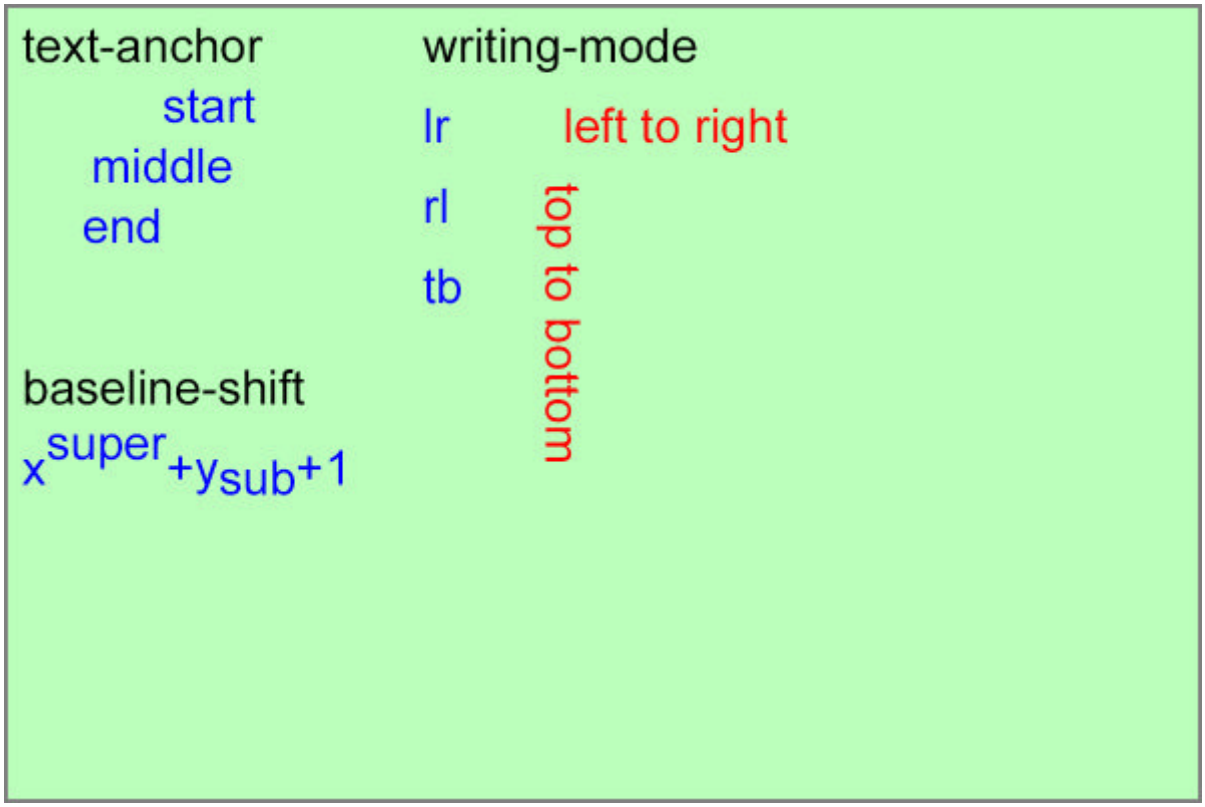

Figure 7.2: Text Properties

8. Animation

8.1 Simple Animation

SVG provides some quite exciting animation facilities that can brighten up your web pages and and are useful in a variety of training and teaching applications. Four elements are provided that define simple animations over attributes and properties:

- animate
- set
- animateTransform
- animateColor

Here is a very simple example to get started with:

```
<rect x="20" y="10" width="120" height="40" >
<animate attributeName="width" from="120" to="40" begin="0s" dur="8s" fill="freeze" />
<animate attributeName="height" from="40" to="82" begin="6s" dur="7s" fill="freeze" />
</rect>
```

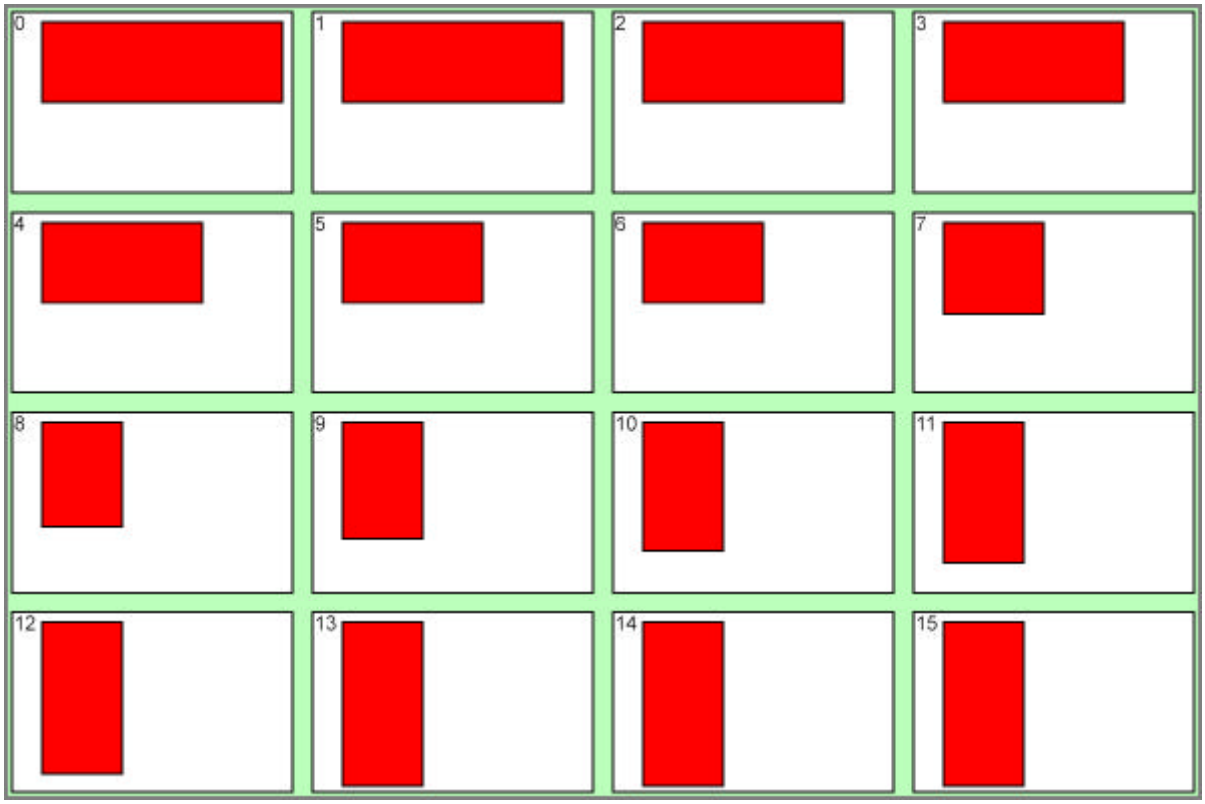

Figure 8.1: Simple Animation

The **animate** element defines animation to be applied to any of the attributes of the **rect** element. In this example, two animations are performed on the element. The first starts the animation at the time the SVG is drawn (0s) and the width of the rectangle is changed from 120 to 40 over the next 8 seconds. Independently, the second animation waits until 6 seconds have elapsed and then increases the height of the rectangle from 40 to 82 over the next 7 seconds. At 8 seconds into the animation, the width stops increasing and stays at the final value (that is what the **freeze** value indicates). After 13 seconds the height stops increasing, and from then on there is a static rectangle displayed with height 81 and width 40. This is illustrated in Figure 8.1 where the rectangle is displayed at times 0 to 15 seconds.

The **animate** element has a slightly different format when the aim is to animate a property defined as part of the **style** attribute. The element then has the form:

```
<circle cx="50" cy="50" r="20" style="fill:red;opacity:1">
<animate attributeType="CSS" attributeName="opacity" from="1" to="0" dur="4s"
repeatCount="indefinite" end="15s fill="freeze""/>
<set attributeType="CSS" attributeName="fill" to="blue" begin="8s" />
<animate attributeName="r" from="20" to="46" dur="13s" />
</circle>
```

The **attributeType** is given the value **CSS** and the CSS name is defined by the **attributeName** attribute. For values that do not have continuous ranges, these can be changed by the **set** element. The results of this animation are shown in Figure 8.2.

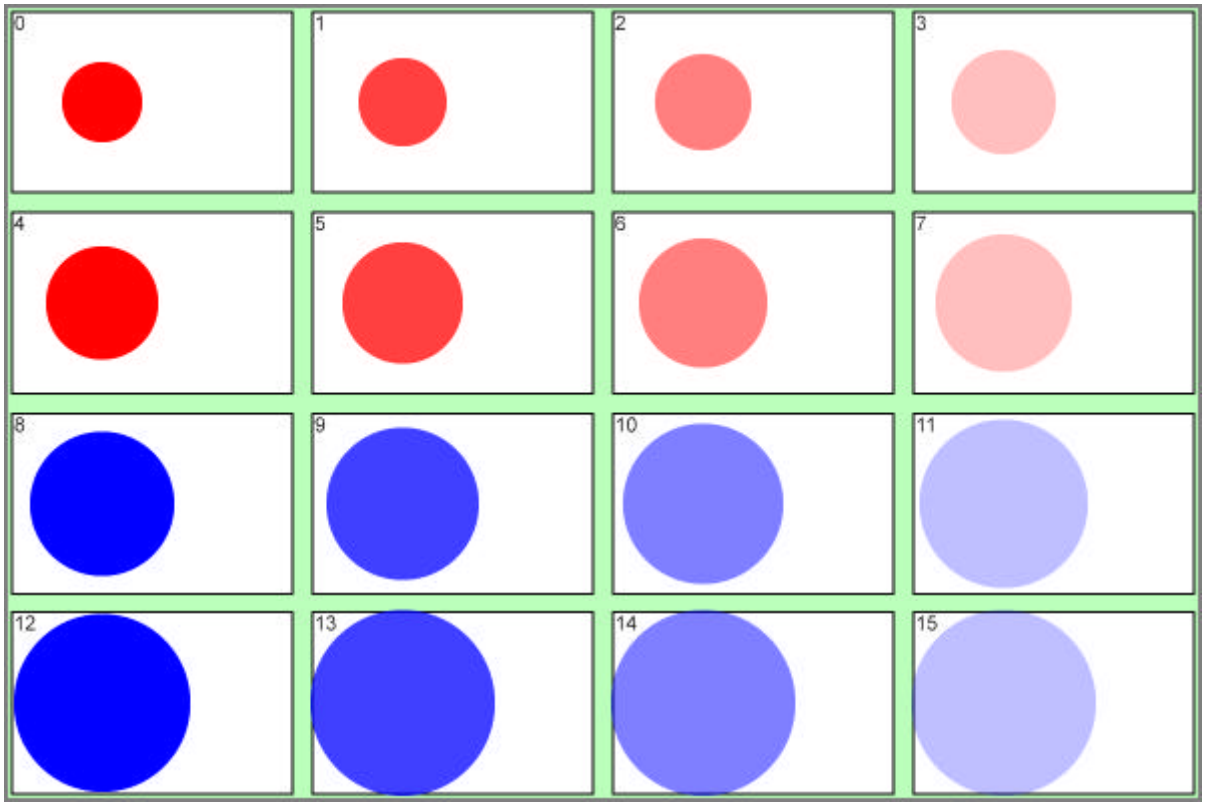

Figure 8.2: Animating Style properties

The **animateTransform** element animates the transformation to be applied to a graphical object. In the example below, the scaling, rotation and translation of the duck are animated. Note that only a single transformation can be animated per element so to achieve this compound effect the **path** element is enclosed within two grouping elements and one transformation animation is applied to each. The result is shown in Figure 8.3.

```
<g>
<g>
<path d="M 20 100 c 40 48 120 -32 160 -6 c 0 0 5 4 10 -3 c 10 -103 50 -83 90 -42 c 0 0
20 12 30 7 c -2 12 -18 17 -40 17 c -55 -2 -40 25 -20 35 c 30 20 35 65 -30 71 c -50 4 -
170 4 -200 -79 z">
<animateTransform attributeName="transform" attributeType="XML" type="scale"
from="0.4" to="0.3" begin="0s" dur="4s" fill="freeze" />
</path>
<animateTransform attributeName="transform" attributeType="XML" type="rotate"
from="0" to="21" begin="4s" dur="7s" fill="freeze" />
</g>
<animateTransform attributeName="transform" attributeType="XML" type="translate"
from="0,0" to="40,20" begin="11s" dur="4s" fill="freeze" />
</g>
```

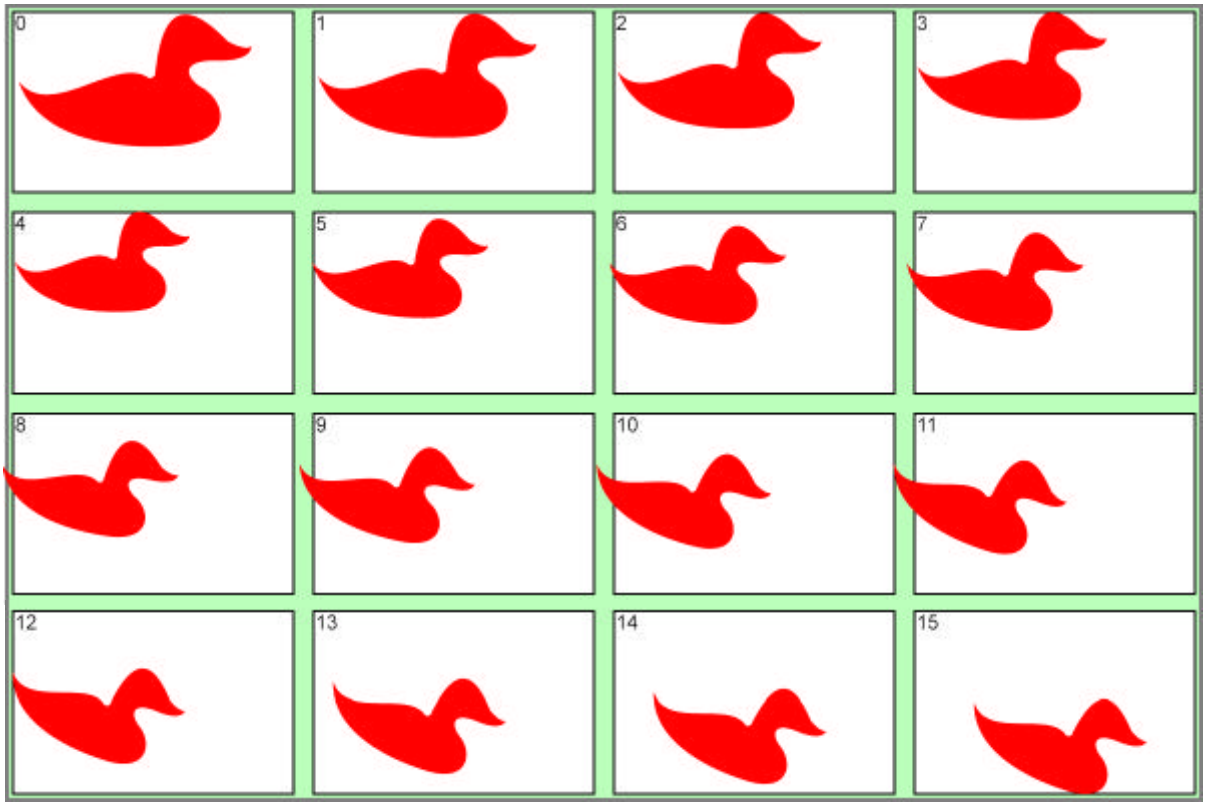

Figure 8.3: Animating Transforms

To animate a colour, the animateColor element is used. For example:

```
<animateColor attributeType="CSS" attributeName="fill" from="aqua" to="crimson"
begin="0s" dur="10s" fill="freeze"/>
```

The **to** and **from** attributes can have the colour specified in any of the usual ways.

8.2 Animation Control

Objects that have been animated so far have had **linear** movement in terms of parameter changes over the duration of the animation. Objects start and stop abruptly. For the animation elements described so far it is possible to define an attribute **calcMode** that specifies how the animation proceeds over time. One of its possible values is **linear** which is the default. A more interesting value is **spline**. In this case, a **values** attribute defines a list of values and a spline function which defines the intermediate value to be used at a specific point in time. The spline function to be used is defined by the **keySplines** attribute. For example:

```
<circle cx="10" cy="90" r="5" style="fill:black">
<animate attributeName="cy" values="90;10" calcMode="spline" keySplines="1 0 0 1"
dur="10s"/>
<animate attributeName="cx" values="10;140" calcMode="spline"
keySplines="0 .75 .25 1" dur="10s"/>
<animate attributeName="cy" values="10;90" calcMode="spline" keySplines="1 0 0 1"
begin="10s" dur="6s"/>
<animate attributeName="cx" values="140;10" calcMode="spline"
keySplines="0 .75 .25 1" begin="10s" dur="6s"/>
</circle>
```

The first **animate** element animates the **cy** value from 90 to 10 over 10 seconds but with the intermediate positions defined by a cubic bezier which goes from (0,0) to (1,1) with control points (1,0) and (0,1). The four coordinates of the two control points are the four values defined by the **keySplines** attribute. The X-axis defines the fraction of the duration passed while the Y-axis gives the fraction of the distance travelled.

In Figure 8.4, the shape of the change for various values of **keySplines** is shown. The top left shows that if the first control point (in grey) coincides with the start point and the second control point in green coincides with the second control point then the result is a linear change.

In the example, the (0 .75 .25 1) value defines a curve where there rapid change earlier on followed by very little change near the end. The value (1 0 0 1) has little change early and late but very rapid change in the middle period.

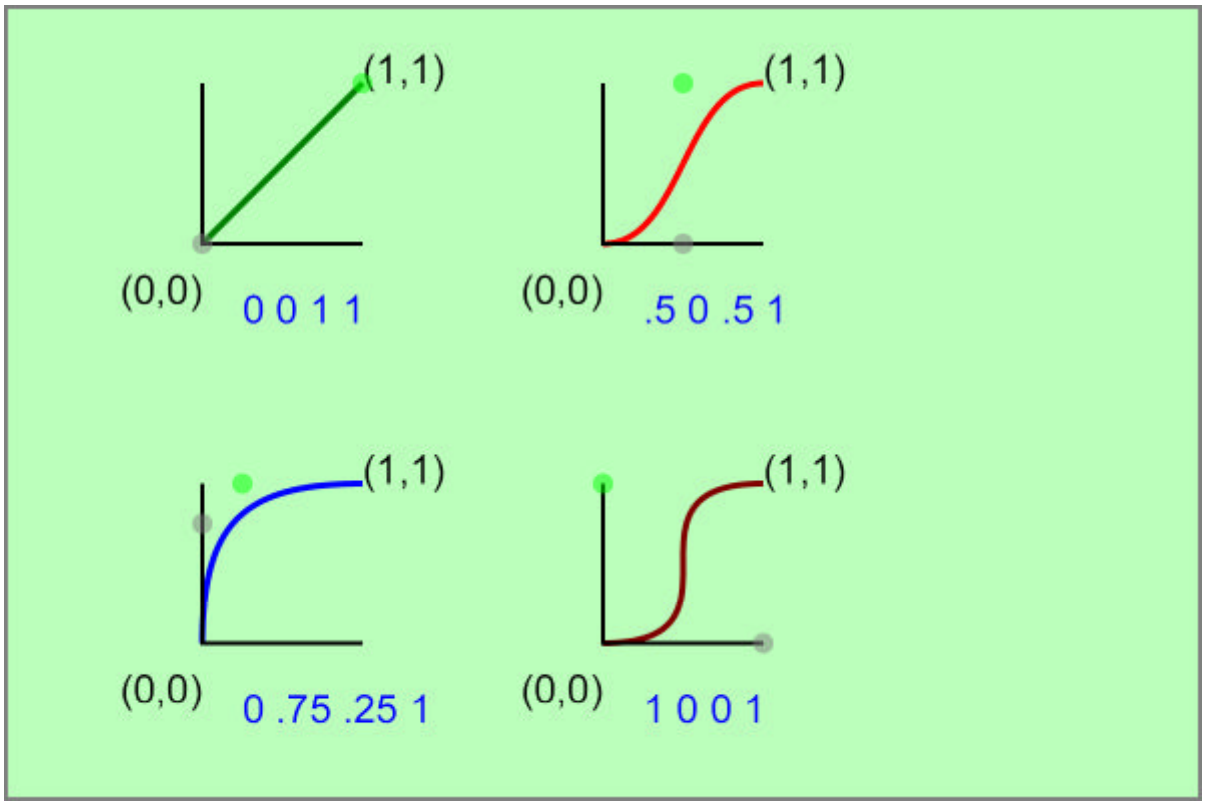

Figure 8.4: Spline Control

The animation of the circle defined above is shown in Figure 8.5 with the last image showing all the intermediate positions of the animated circle. The set for the first 10 seconds are in blue and the remainder in green.

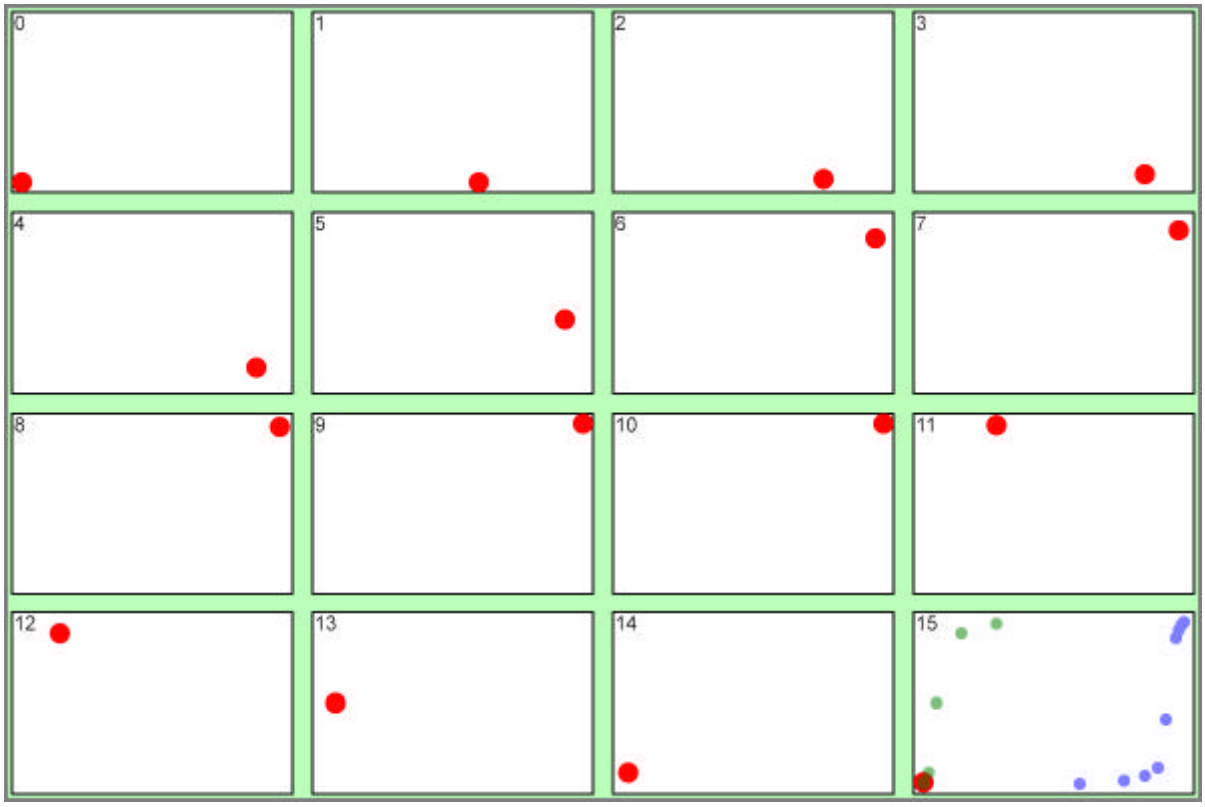

Figure 8.5: Animation Control

The data associated with a path element can also be animated although there is a constraint that the various path segments making up the path must be the same in structure in both the start and finish positions. Effectively each individual value is interpolated between the start and end value. For example:

```
<path>
<animate attributeName="d" from="M 20 100 c 40 48 120 -32 160 -6 c 0 0 5 4 10 -3 c
10 -103 50 -83 90 -42 c 0 0 20 12 30 7 c -2 12 -18 17 -40 17 c -55 -2 -40 25 -20 35 c 30
20 35 65 -30 71 c -50 4 -170 4 -200 -79"
to="M 80 100 c 40 48 120 -2 160 -36 c 0 0 5 -11 10 -18 c 10 -73 50 -23 90 -12 c 0 0 20
-48 30 22 c -2 12 -18 47 -40 17 c 5 -2 20 -5 -20 35 c 30 20 35 95 -30 86 c -80 -116 -
260 94 -200 -94"
fill="freeze" dur="1s"/>
</path>
```

The overall result is shown in Figure 8.6.

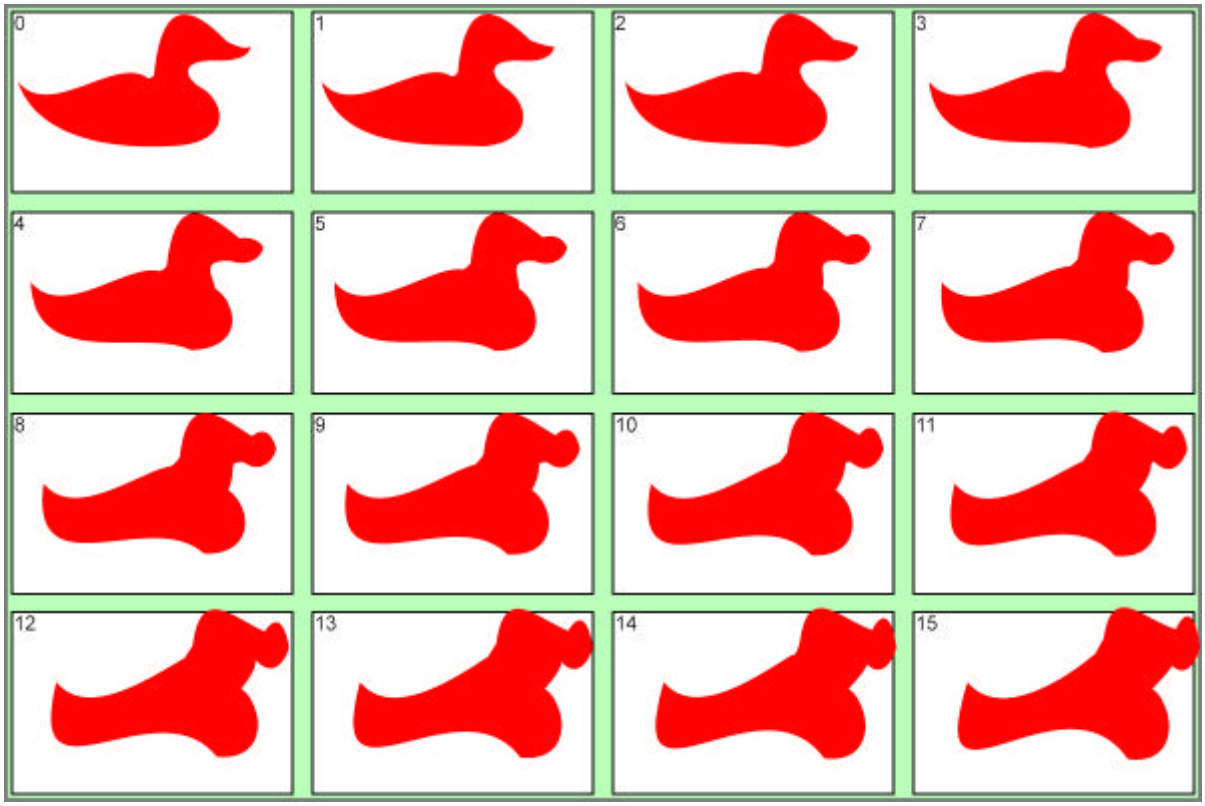

Figure 8.6: Animate path data

8.3 Animation along a Path

An object can be animated so that it proceeds along a specified path using the **animateMotion** element. For example:

```
<path d="M0 0 v -2.5 h10 v-5 l5 7.5 l-5 7.5 v-5 h-10 v-2.5" style="fill:red">
<animateMotion dur="6s" repeatCount="indefinite" path="M 100 150 c 0 -40 120 -80
120 -40 c 0 40 120 80 120 40 c 0 -60 -120 -100 -120 -40 c 0 60 -120 100 -120 40"
rotate="auto" />
</path>
```

The object consists of an arrow and the **animateMotion** element animates along a figure of eight path defined by the cubic beziers and starting on the left side. The **rotate** attribute defines the orientation of the arrow as it proceeds along the path. The value **auto** keeps the orientation of the arrow so that it always points along the path. A value specified as a number indicates that the arrow should stay at that constant rotation from its initial position irrespective of where it is on the curve. The value **auto-reverse** positions the arrow so that it always points away from the direction of motion. Figure 8.7 shows four examples of the **rotate** attribute with the positions of the arrow as the animation takes place in each case. The latest position is opaque and the earlier positions are displayed with decreasing opacity.

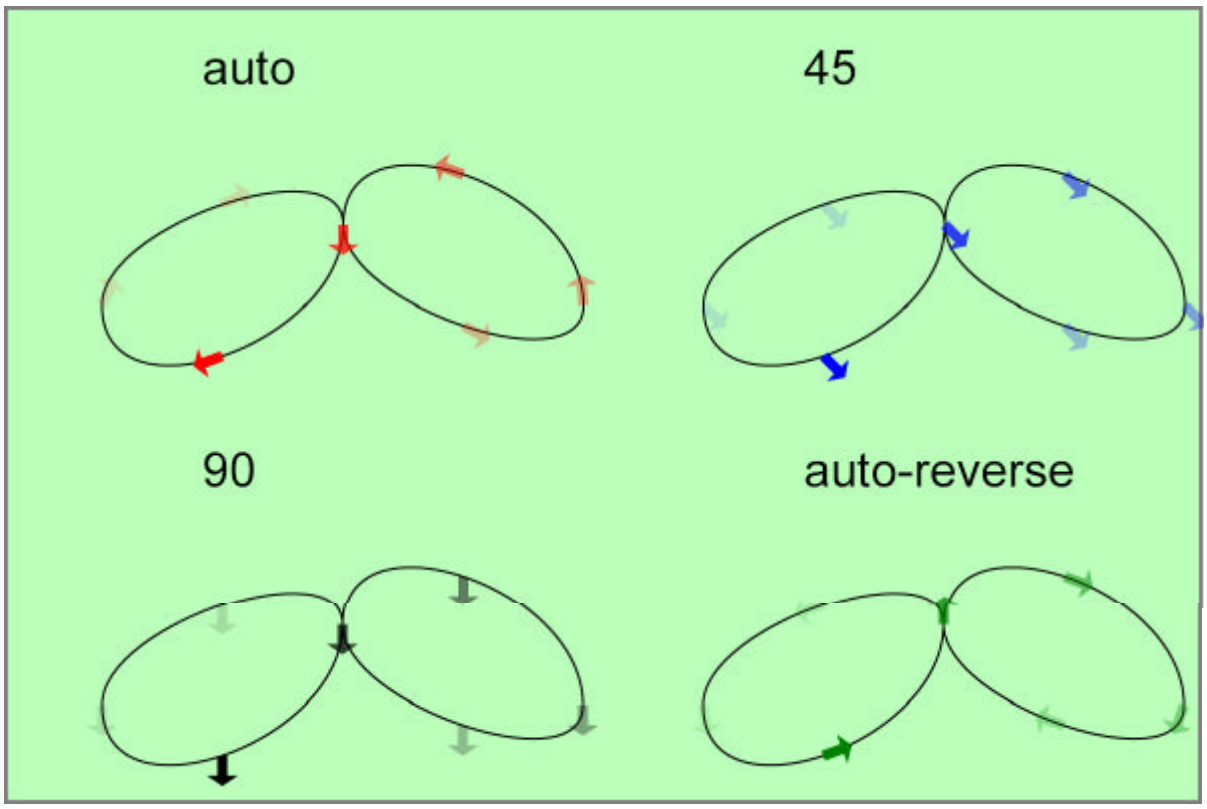

Figure 8.7: Animate along a path

9. Linking and Templates

9.1 Linking

Linking in SVG is much the same as in HTML. SVG has an **a** element that indicates the hyperlink and defines where the link goes. For example:

```
<a xlink:href="http://www.w3.org">
<rect width="200" height="40" /> <text x=100" y="30" style="text-anchor:middle">My
button</text>
</a>
```

This example consists of a rectangle with the word **My button** in the middle. Clicking on any part of the rectangle causes the browser to link to the W3C home page. Note that the URL is defined by xlink:href rather than href. This is because the aim is to use all the functionality of XLink when it is finalised. At the moment this acts just the same as the **href** attribute in HTML. The user should be careful to enclose both the rectangle and the text within the **a** element. Otherwise, clicking on the part of the rectangle where the text is would not cause the link to take place. The text appears later than the rectangle and so **sits on top** of the rectangle.

9.2 Symbols and their Use

Many drawings consist of the same object appearing a number of times in different places with possible minor variations. An example would be symbols on a map. SVG provides a rather simple minded symbol facility that is useful on occasions. However, by providing no parameterisation of the symbol, the times it is useful are limited.

A **symbol** can contain any of the usual drawing elements. For example:

```
<symbol id="duck">
<path d="M 10 90
c 40 48 120 -32 160 -6
c 0 0 5 4 10 -3 c 10 -103 50 -83 90 -42
c 0 0 20 12 30 7 c -2 12 -18 17 -40 17
c -55 -2 -40 25 -20 35 c 30 20 35 65 -30 71
c -50 4 -170 4 -200 -79 z"/>
<text x="150" y="120" style="text-anchor:middle">The Duck</text>
</symbol>
```

The symbol consists of the path defining the duck and the text **The Duck** positioned in its centre. An instance of the symbol is created by the **use** element as follows:

```
<use x="0" y="0" xlink:href="#duck" style="stroke:black;stroke-width:2;fill:none;font-
size:48" />
```

The **use** element is effectively replaced by a **g** element with any attributes associated with the **use** element being transfered to the **g** element except that the origin specified by the attributes **x** and **y** become a **transform** attribute appended to the end of any transformations defined on the **g** element. This is a rather bizarre way of doing it and requires some careful thought before understanding what the result is likely to be. Figure 9.1 shows various examples of the **use** element:

```
<use x="0" y="0" xlink:href="#duck" style="stroke:black;stroke-width:2;fill:none;font-size:48" />
<use x="300" y="0" xlink:href="#duck" style="stroke:black;fill:red;font-size:40;font-family:Verdana" />
<use x="0" y="400" xlink:href="#duck" transform="scale(0.5)" style="stroke:none;fill:red;font-size:64" />
<use x="0" y="0" xlink:href="#duck" transform="translate(0,300) scale(0.5)" style="stroke:white;stroke-width:3;fill:blue;font-size:40" />
<use x="300" y="200" xlink:href="#duck" style="stroke:black;fill:none;font-size:16;writing-mode:tb;" />
```

The first use is quite straightforward. The duck and associated text are drawn in outline (top left) and the font size is specified by the **font-size** property to be 40. The second use in the top right sets the **fill** property to be red and changes the font to Verdana. Notice that if the text is filled so is the path. It would have been better if the two could have been defined separately either by having separate fill properties for text and path or being able to parameterise the symbol.

The third use illustrates the problem as no outline is drawn and it is only by making the text overflow the duck that it can be seen at all. This small duck also illustrates the problem with the way the **use** is executed. The transform to be applied is changed (by the **x**, **y** positioning attributes) to:

```
transform="scale(0.5) translate(0,400)"
```

Multiple transforms are applied right to left. In this case this results in the object being scaled but the translate is also scaled. So the translation applied is only (0,200) which is why the red duck appears where it does.

In the fourth use, the **x** and **y** values are set to zero resulting in no additional transformation being generated. The scaling is done first followed by the translation in this case. A good rule is therefore **if you are going to transform the symbol, do all the transformation using the transform property**.

The final example shows how the writing direction can be changed.

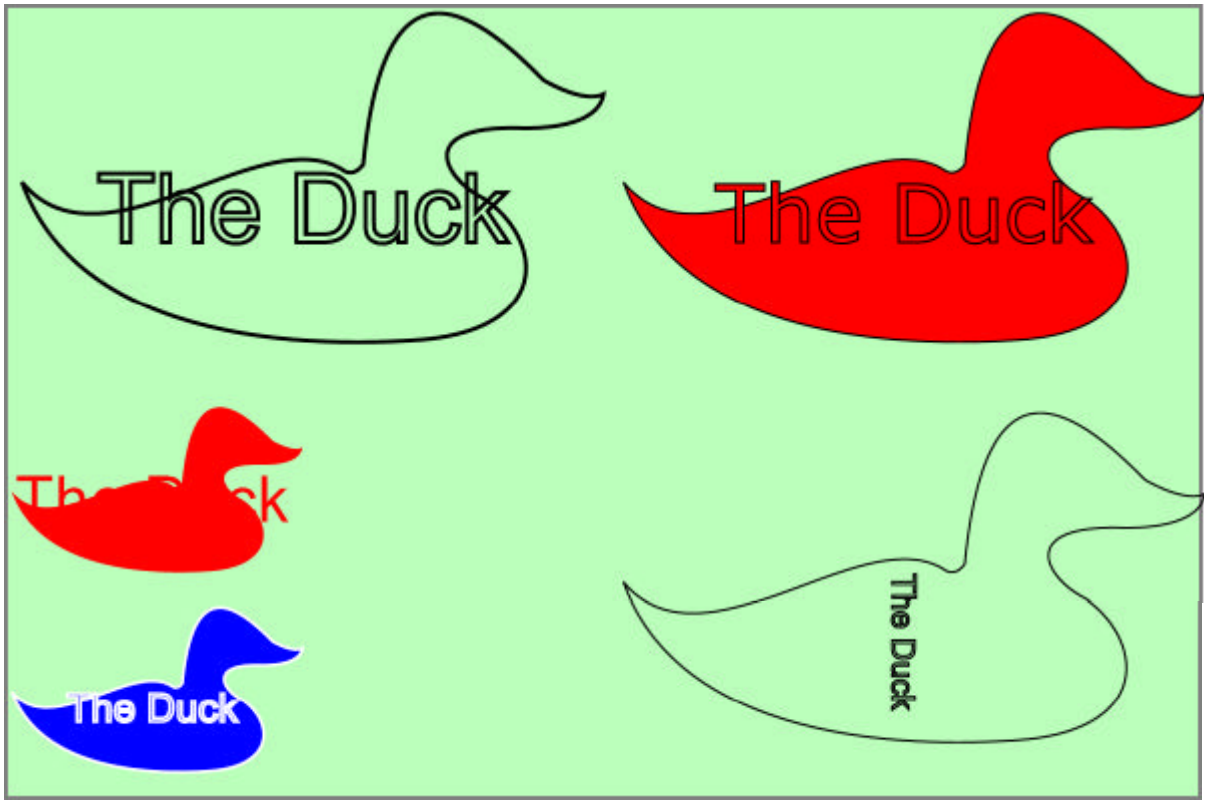

Figure 9.1: Symbols and their Use

9.3 Images

Sometimes it is useful to include bitmap images in an SVG document. These can be PNG, GIF and JPG images and are included in much the same way that images are included in an HTML document:

```
<image> x="20" y="40" width="100" height="200" xlink:href="mypicture.gif">
```

The image is positioned with the top left corner at the position specified and fitted into the width and height given.

9.4 Masking

SVG also provides a facility called masking. The mask object effectively defines the transparency value to be applied to the drawing at each position defined by the mask. For example:

```
<linearGradient id="Gradient" gradientUnits="userSpaceOnUse" x1="0" y1="0"
x2="500" y2="0">
<stop offset="0" style="stop-color:white; stop-opacity:0"/>
<stop offset="1" style="stop-color:white; stop-opacity:1"/>
</linearGradient>

<rect x="0" y="0" width="500" height="60" style="fill:#FF8080"/>

<mask maskContentUnits="userSpaceOnUse" id="Mask">
<rect x="0" y="0" width="500" height="60" style="fill:url(#Gradient)" />
</mask>
<text x="250" y="50" style="font-family:Verdana; font-size:60; text-
anchor:middle;fill:blue; mask:url(#Mask)">
MASKED TEXT
</text>
<text x="250" y="50" style="font-family:Verdana; font-size:60; text-
anchor:middle;fill:none; stroke:black; stroke-width:2">
MASKED TEXT
</text>
```

First a pink rectangle is drawn and then the text is drawn twice. The first one fills the text in blue but the transparency value of the text comes from the mask which is a gradient that is fully transparent on the left and fully opaque on the right. The second draws the black outline.

Figure 9.2 shows the masked text at the top followed by a blue rectangle masked in a similar way and finally the duck masked by a circle with transparency varying from opaque on the right to transparent on the left.

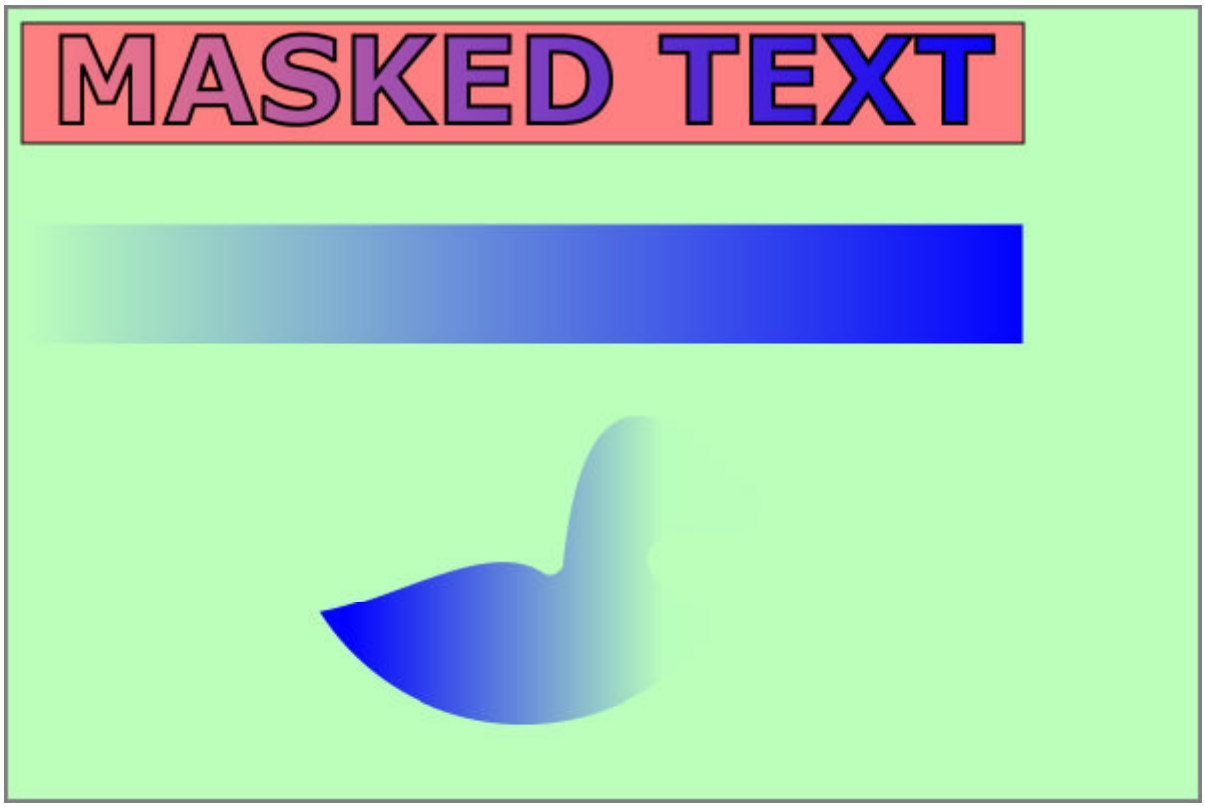

Figure 9.2: Masking

10. Interaction

10.1 Interaction and the DOM

SVG provides support for interacting with an SVG document using the facilities in the W3C Document Object Model 2.0 (DOM) Recommendation. The DOM provides a standard way of accessing the attributes and properties defined in an SVG document and changing them via a scripting language. For example:

```
<svg viewbox= "0 0 600 400" >
<script type="text/ecmascript"><![CDATA[
function changerect(evt)
{
var svgobj=evt.target;
svgstyle = svgobj.getStyle();
svgstyle.setProperty ('opacity', 0.3);
svgobj.setAttribute ('x', 300);
}
]]>
</script>
<rect onclick="changerect(evt)" style="fill:blue;opacity:1" x="10" y="30" width="100"
height="100" />
</svg>
```

This defines a diagram consisting of a single opaque blue rectangle close to the left hand edge. When the mouse or pointing device is clicked over it, the rectangle is repositioned further to the right and becomes semi-transparent.

As scripting languages vary in their capabilities and browsers vary in their support of them, the user may need some trial and error to get started. A general point is that the SVG **script** element behaves in much the same way as the one in HTML. In consequence, following the style used for HTML scripting will usually work for SVG as well.

In the example, the **onclick** attribute calls the script function changerect when the mouse click occurs. The variable evt passed as parameter to the changerect function enables the object over which the mouse was clicked to be identified. The property **target** of evt gives a reference to the object clicked. The variable svgobj is set to the object that was clicked. The ECMAScript method getStyle gives a reference to the object's **style** attribute, setProperty sets the value of a **style** property. The method setAttribute sets an attribute value.

Some of the events that can be handled by SVG are:

- onclick
- onactivate
- onmousedown
- onmouseover
- onmousemove
- onmouseout
- onload

The event onactivate is more general than onclick and will work with devices other than mouse-like devices. The onload event gives a general method of invoking a script when an SVG document is loaded. For example:

```
<svg viewbox= "0 0 600 400" onload="changerect(evt)">
<script type="text/ecmascript"><![CDATA[
function changerect(evt)
{
var svgdoc = evt.getCurrentNode().getOwnerDocument();
svgobj = svgdoc.getElementByID ('MyRect')
svgstyle = svgobj.getStyle();
svgstyle.setProperty ('opacity', 0.3);
svgobj.setAttribute ('x', 300);
}
]]>
</script>
<rect id="MyRect" style="fill:blue;opacity:1" x="10" y="30" width="100" height="100" />
</svg>
```

In this example, the variable svgdoc is set to point to the SVG document as a whole and svgobj is set to the rect object with id equal to MyRect. In this case, the rectangle will appear semi-transparent and on the right as soon as the SVG document is loaded.

It can be seen from these examples that the starting point for any interaction with an SVG document is obtaining a reference to the object tree (at an appropriate node) that represents the document. There are several ways to do this and different browsers may support different approaches. In designing an interactive SVG application, it is wise to start by thinking carefully about where modification will be required, and design the SVG document to facilitate this (for example, by including **id** attributes on appropriate elements).

10.2 Interaction Methods

The most useful methods for modifying an SVG document are:

- getElementById
- getStyle
- setProperty
- setAttribute
- getAttribute
- cloneNode

To create new elements, a useful method is cloneNode. For example:

```
<svg viewbox= "0 0 600 400" >
<script type="text/ecmascript"><![CDATA[
function addrect(evt)
{ var svgobj=evt.target;
var svgdoc = svgobj.getOwnerDocument();
var newnode = svgobj.cloneNode(false);
svgstyle = newnode.getStyle();
var colors = new Array('red', 'blue', 'yellow', 'cyan', 'green', 'lime', 'magenta', 'brown',
'azure', 'burlywood', 'blueviolet', 'crimson');
var x = 10+480*Math.random();
var y = 10+330*Math.random();
var width = 10+100*Math.random();
var height = 10+50*Math.random();
var fill = Math.floor(colors.length*Math.random());
if (fill == colors.length) fill = colors.length-1;
fill = colors[fill];
svgstyle.setProperty ('opacity', 0.3+0.7*Math.random());
svgstyle.setProperty ('fill', fill);
newnode.setAttribute ('x', x);
newnode.setAttribute ('y', y);
newnode.setAttribute ('width', width);
newnode.setAttribute ('height', height);
var contents = svgdoc.getElementById ('contents');
newnode = contents.appendChild (newnode);
} ]]></script>
<rect x="1" y="1" style="fill:#bbffbb" width="598" height="398"/>
<g id="contents">
<rect onclick="addrect(evt)" style="fill:blue;opacity:1" x="250" y="100" width="20"
height="20" />
</g>
</svg>
```

Hitting the single blue square rectangle in the middle of the diagram causes the function addrect to be invoked. This sets svgobj to point at the rectangle and svgdoc to point at the SVG document as a whole. The variable newnode is a new rectangle object (initially a copy of the element hit) that has its fill colour, position, size and opacity defined by resetting the attrributes and properties. The enclosing group with **id** set to **contents** has this new element appended to it as a new child. So after the first click on the blue rectangle the diagram will consist of two rectangles where the second has its position, size and properties randomly defined. After many clicks, the diagram might be as shown in Figure 10.1. This example illustrates a general style, namely creating new elements within an SVG document and then incorporating them into the SVG structure at the appropriate places.

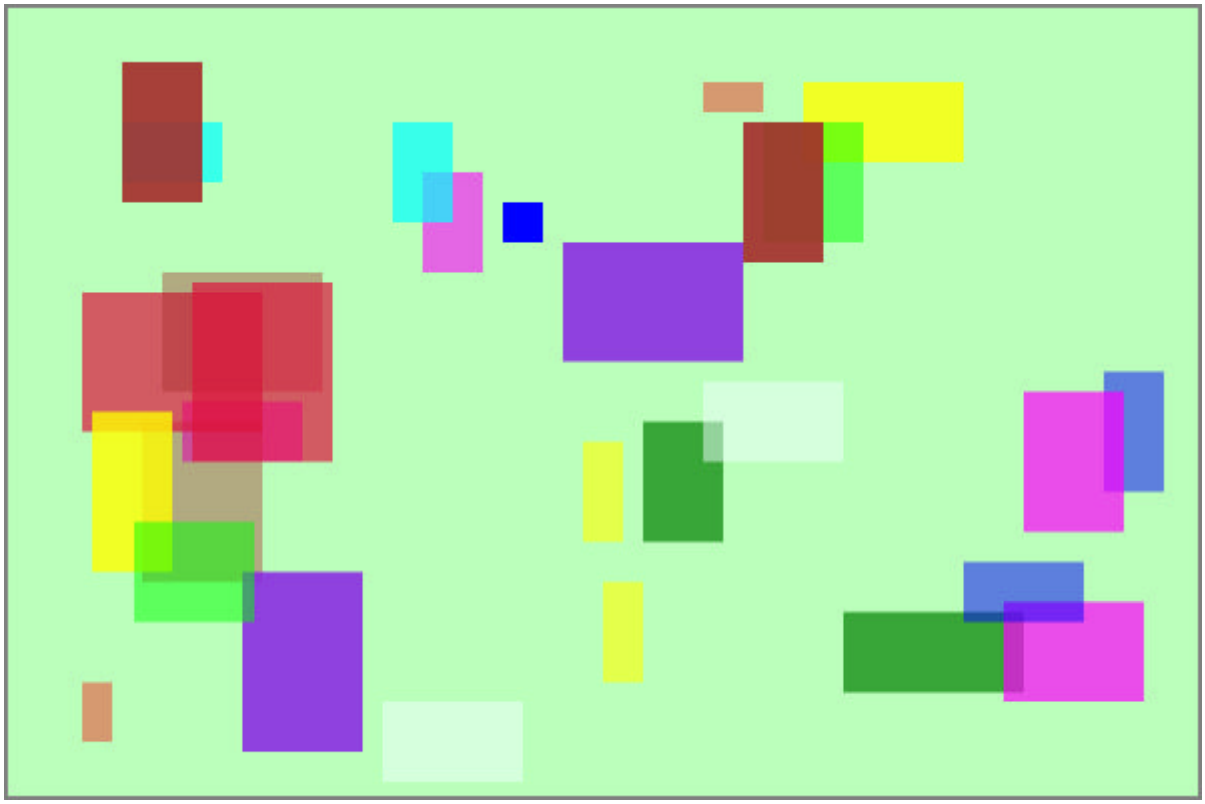

Figure 10.1: Cloning Rectangles

A. SVG Colours

Colour Name	RGB Value	Colour Name	RGB Value
aliceblue	(240, 248, 255)	darkslategrey	(47, 79, 79)
antiquewhite	(250, 235, 215)	darkturquoise	(0, 206, 209)
aqua	(0, 255, 255)	darkviolet	(148, 0, 211)
aquamarine	(127, 255, 212)	deeppink	(255, 20, 147)
azure	(240, 255, 255)	deepskyblue	(0, 191, 255)
beige	(245, 245, 220)	dimgray	(105, 105, 105)
bisque	(255, 228, 196)	dimgrey	(105, 105, 105)
black	(0, 0, 0)	dodgerblue	(30, 144, 255)
blanchedalmond	(255, 235, 205)	firebrick	(178, 34, 34)
blue	(0, 0, 255)	floralwhite	(255, 250, 240)
blueviolet	(138, 43, 226)	forestgreen	(34, 139, 34)
brown	(165, 42, 42)	fuchsia	(255, 0, 255)
burlywood	(222, 184, 135)	gainsboro	(220, 220, 220)
cadetblue	(95, 158, 160)	ghostwhite	(248, 248, 255)
chartreuse	(127, 255, 0)	gold	(255, 215, 0)
chocolate	(210, 105, 30)	goldenrod	(218, 165, 32)
coral	(255, 127, 80)	gray	(128, 128, 128)
cornflowerblue	(100, 149, 237)	grey	(128, 128, 128)
cornsilk	(255, 248, 220)	green	(0, 128, 0)
crimson	(220, 20, 60)	greenyellow	(173, 255, 47)
cyan	(0, 255, 255)	honeydew	(240, 255, 240)
darkblue	(0, 0, 139)	hotpink	(255, 105, 180)
darkcyan	(0, 139, 139)	indianred	(205, 92, 92)
darkgoldenrod	(184, 134, 11)	indigo	(75, 0, 130)
darkgray	(169, 169, 169)	ivory	(255, 255, 240)
darkgreen	(0, 100, 0)	khaki	(240, 230, 140)
darkgrey	(169, 169, 169)	lavender	(230, 230, 250)
darkkhaki	(189, 183, 107)	lavenderblush	(255, 240, 245)
darkmagenta	(139, 0, 139)	lawngreen	(124, 252, 0)
darkolivegreen	(85, 107, 47)	lemonchiffon	(255, 250, 205)
darkorange	(255, 140, 0)	lightblue	(173, 216, 230)
darkorchid	(153, 50, 204)	lightcoral	(240, 128, 128)
darkred	(139, 0, 0)	lightcyan	(224, 255, 255)
darksalmon	(233, 150, 122)	lightgoldenrodyellow	(250, 250, 210)
darkseagreen	(143, 188, 143)	lightgray	(211, 211, 211)
darkslateblue	(72, 61, 139)	lightgreen	(144, 238, 144)
darkslategray	(47, 79, 79)	lightgrey	(211, 211, 211)

Colour Name	RGB Value	Colour Name	RGB Value
lightpink	(255, 182, 193)	paleturquoise	(175, 238, 238)
lightsalmon	(255, 160, 122)	palevioletred	(219, 112, 147)
lightseagreen	(32, 178, 170)	papayawhip	(255, 239, 213)
lightskyblue	(135, 206, 250)	peachpuff	(255, 218, 185)
lightslategray	(119, 136, 153)	peru	(205, 133, 63)
lightslategrey	(119, 136, 153)	pink	(255, 192, 203)
lightsteelblue	(176, 196, 222)	plum	(221, 160, 221)
lightyellow	(255, 255, 224)	powderblue	(176, 224, 230)
lime	(0, 255, 0)	purple	(128, 0, 128)
limegreen	(50, 205, 50)	red	(255, 0, 0)
linen	(250, 240, 230)	rosybrown	(188, 143, 143)
magenta	(255, 0, 255)	royalblue	(65, 105, 225)
maroon	(128, 0, 0)	saddlebrown	(139, 69, 19)
mediumaquamarine	(102, 205, 170)	salmon	(250, 128, 114)
mediumblue	(0, 0, 205)	sandybrown	(244, 164, 96)
mediumorchid	(186, 85, 211)	seagreen	(46, 139, 87)
mediumpurple	(147, 112, 219)	seashell	(255, 245, 238)
mediumseagreen	(60, 179, 113)	sienna	(160, 82, 45)
mediumslateblue	(123, 104, 238)	silver	(192, 192, 192)
mediumspringgreen	(0, 250, 154)	skyblue	(135, 206, 235)
mediumturquoise	(72, 209, 204)	slateblue	(106, 90, 205)
mediumvioletred	(199, 21, 133)	slategray	(112, 128, 144)
midnightblue	(25, 25, 112)	slategrey	(112, 128, 144)
mintcream	(245, 255, 250)	snow	(255, 250, 250)
mistyrose	(255, 228, 225)	springgreen	(0, 255, 127)
moccasin	(255, 228, 181)	steelblue	(70, 130, 180)
navajowhite	(255, 222, 173)	tan	(210, 180, 140)
navy	(0, 0, 128)	teal	(0, 128, 128)
oldlace	(253, 245, 230)	thistle	(216, 191, 216)
olive	(128, 128, 0)	tomato	(255, 99, 71)
olivedrab	(107, 142, 35)	turquoise	(64, 224, 208)
orange	(255, 165, 0)	violet	(238, 130, 238)
orangered	(255, 69, 0)	wheat	(245, 222, 179)
orchid	(218, 112, 214)	white	(255, 255, 255)
palegoldenrod	(238, 232, 170)	whitesmoke	(245, 245, 245)
palegreen	(152, 251, 152)	yellow	(255, 255, 0)
		yellowgreen	(154, 205, 50)

B. SVG Elements and their Attributes

B.1 Attribute Value Types

The types of the attribute values in the following element tables are either listed as a set of possible alternatives or the type of the value. The default value is in maroon.

Type	Value
align	Possible values are: none xMinYMin xMidYMin xMaxYMin xMinYMid **xMidYMid** xMaxYMid xMinYMax xMidYMax xMaxYMax
bzlist	A list of four fraction values between 0 and 1, each set of four (x1, y1, x2, y2) defines a pair of cubic Bezier control points for one interval.
clock	Clock value. Some examples are: 3s 4min 2.5h 100ms 6:45:33.2 45:33.2. If no units are specified, seconds are assumed.
color	A CSS colour value (for example, red, #F00, #FF0000, rgb(255,0,0)).
colorlist	A list of colour values
coord	Coordinate position in the current coordinate system. It will be transformed.
coordfr	Value is either a coordinate (useSpaceOnUse) or a fraction of the bounding box(objectBoundingBox) of the object to which the element is applied.
coordlist	A list of coordinate positions possibly only one.
degree	A rotation value in the clock-wise direction in degrees.
deglist	A list of rotation values in the clock-wise direction in degrees.
evencoordlist	A list of coordinate pairs.
fr	A fraction between 0 and 1.
frlist	A list of fraction values between 0 and 1.
idref	Reference to an **id** attribute such as **xyz.begin** in a time definition where **xyz** is an **id** of another element.
legal	Legal values for the attribute specified.
legallist	List of legal values for the attribute specified.
length	Length in the current coordinate system. It will be transformed.
meetOrSlice	Possible values are: **meet** and slice.
mediatype	media type as in RFC2045.
name	Any legal identifier as in CSS.
num	Any number, does not have a metric.
percent	A per centage value between 0% and 100%.
text	Any text string.
time	Some possible values are: +[clock] -[clock] [idref].begin + [clock] [idref].begin - [clock] [idref].end + [clock] [idref].end - [clock]
timelist	List of [time].
transformlist	List of transformations (for example: scale(2) translate(100,100)).
url	A legal URL.

B.2 SVG Elements Described in this Document

The table below gives a list of the elements in SVG described in this document. For each element, both the attributes that have been described and those omitted are listed. The style attributes are not listed here but have a separate table, see Section B.4. The attributes in bold are the main or unique ones for the element. The complete set of **xlink** attributes are allowed for simple links. For attributes that can have a set of values, the default value is shown in red and bold

Element	Attributes	Comment
a	**xmlns:xlink target** xlink:href etc	The **a** element acts like a **g** element so most of those attributes are also allowed.
animate	**attributeName=[legal] attributeType=[legal]** begin=[timelist] end=[timelist] dur=[[clock] | indefinite] min=[clock] max=[clock] restart=[**always** | never | whenNotActive] repeatCount=[[clock] | indefinite] repeatDur=[[clock] | indefinite] fill=[**remove** | freeze] calcMode=[**linear** | discrete | paced | spline] keyTimes=[frlist] keySplines=[bzlist] from=[legal] to=[legal] by=[legal] additive=[**replace** | sum] accumulate=[**none** | sum] onbegin onend onrepeat	keySplines list is one less than the keyTimes list.
animateColor	begin=[timelist] end=[timelist] dur=[[clock] | indefinite] repeatCount=[[clock] | indefinite] repeatDur=[[clock] | indefinite] fill=[freeze | remove] from=[legal] to=[legal] by=[legal] **values=[colorlist]**	
animateMotion	begin=[timelist] end=[timelist] dur=[[clock] | indefinite] min=[clock] max=[clock] restart=[**always** | never | whenNotActive] repeatCount=[[clock] | indefinite] repeatDur=[[clock] | indefinite] calcMode=[**linear** | discrete | paced | spline] keyTimes=[frlist] keySplines=[bzlist] aditive=[replace | sum] from=[[coord],[coord]] to=[[coord],[coord]] by=[[coord],[coord]] **keyPoints=[frlist] path=[pathdata]** **rotate=[[degree] | auto | auto-reverse]** values=[coordlist]	keySplines list is one less than the keyTimes list.
animateTransform	begin=[timelist] end=[timelist] dur=[[clock] | indefinite] min=[clock] max=[clock] restart=[**always** | never | whenNotActive] repeatCount=[[clock] | indefinite] repeatDur=[[clock] | indefinite] calcMode=[**linear** | discrete | paced | spline] keyTimes=[frlist] keySplines=[bzlist] additive=[replace | sum] from=[legal] to=[legal] by=[legal] **type=[translate | scale | rotate | skewX | skewY]** values=[legallist]	keySplines list is one less than the keyTimes list.

Element	Attributes	Comment
circle	**cx=[coord] cy=[coord] r=[length]**	Draws circle with centre and radius specified. r="0" stops rendering.
clipPath	clipPathUnits=[objectBoundingBox | userSpaceOnUse]	
defs		Encloses elements not to be displayed such as style sheets and symbol definitions. It can have all the attributes of a **g** element.
desc	xmlns	Description of the drawing. May have class and style attributes. Could contain XML fragment.
ellipse	**cx=[coord] cy=[coord] rx=[length] ry=[length]**	Draws ellipse defined by centre and two axes. Either rx="0" or ry="0" stops the rendering
g	All the styling attributes plus id requiredFeatures requiredExtensions systemLanguage xml:lang xml:space externalResourcesRequired class style enable-background flood-color flood-opacity clip overflow transform onfocusin etc	The **g** element can take almost any attribute that an element inside it can have.
image	preserveAspectRatio=[align] [meetOrSlice] **x=[coord] y=[coord] width=[length] height=[length]** xlink:href=[url]	
line	**x1=[coord] y1=[coord] x2=[coord] y2=[coord]**	Defines line between two points. Default for all four values is 0
linearGradient	**x1=[coordfr] y1=[coordfr] x2=[coordfr] y2=[coordfr]** gradientTransform=[transformlist] gradientUnits=[objectBoundingBox | userSpaceOnUse] **spreadMethod=[pad | reflect | repeat]** xlink:href=[url]	Defines a gradient to be applied between (x1,y1) and (x2,y2). If object is larger than this line, pad continues the end values of the gradient outwards, reflect reflects the gradient and repeat repeats it. The linked url can be another gradient whose values are inherited by this one.
mask	**height=[length] width=[length]** **maskContentUnits=[objectBoundingBox | userSpaceOnUse]** **maskUnits=[objectBoundingBox | userSpaceOnUse]** **x=[coord] y=[coord]**	

Element	Attributes	Comment
path	**d=[pathdata] pathLength=[length]**	Defines a path where author gives estimate of pathLength. Values dependent on path length are scaled up to the actual length, for example offset of text on a path.
polygon	**points=[evencoordlist]**	Equivalent to a path that does moveto to first point and absolute lineto to the other points in sequence finishing with a closepath command.
polyline	**points=[evencoordlist]**	Equivalent to a path that does moveto to first point and absolute lineto to the other points in sequence.
radialGradient	**cx=[coordfr] cy=[coordfr]** **fx=[coordfr] fy=[coordfr]** r= gradientTransform=[transformlist] gradientUnits=[objectBoundingBox | userSpaceOnUse] **spreadMethod=[pad | reflect | repeat]**	
rect	**x=[coord] y=[coord]** **width=[length] height=[length]** **rx=[length] ry=[length]**	x and y default to 0. rx,ry define the radii that round the corners of the rectangle.
script	**type=[mediatype]**	
set	begin=[timelist] end=[timelist] dur=[[clock] | indefinite] min=[clock] max=[clock] restart=[**always** | never | whenNotActive] repeatCount=[[clock] | indefinite] repeatDur=[[clock] | indefinite] **to=[legal]**	
stop	**offset=[[fr] | [percent]]** **stop-color=[color] stop-opacity=[fr]**	
style	**media=[comma separated list of media descriptors] title=[text] type=[mediatype]**	Usual to have the style sheet at the top of the document and surrounded by a **defs** element.
svg	**contentScriptType="text/ecamscript"** **contentStyleType="text/css"** **x=[coord] y=[coord]** **height=[length] width=[length]** preserveAspectRatio=[align] [meetOrSlice] xmlns=[resource] zoomAndPan=[**magnify** | disable | zoom] overflow=[visible | hidden | scroll | auto | inherit]	The default media type are given as examples for contentScriptType and contentStyleType.

Element	Attributes	Comment
switch	**requireFeatures=[org.w3c.svg.static | org.w3c.svg.dynamic | org.w3c.dom.svg | org.w3c.svg.lang | org.w3c.svg.animation etc]** **systemLanguage=[comma separated list of languages such as** en**]**	
symbol	All the presentation attributes **preserveAspectRatio=[align] [meetOrSlice]** **viewBox=[coord] [coord] [length] [length]**	Symbol is a container element for a set of graphics elements including **use** elements.
text	dx=[lengthlist] dy=[lengthlist] x=[coordlist] y=[coordlist] lengthAdjust=[spacing | spacingAndGlyphs] rotate=[degree] textLength=[length] transform=[transformlist]	Draws text with origin of text string or origin of individual characters defined by (x,y) offset by (dx,dy). Additional rotation can be specified for the text string or individual characters. The expected length of the text can be defined. If actual length is different, lengthAdjust decides whether it gets padded by just varying the spacing.
textPath	**lengthAdjust=[spacing | spacingAndGlyphs]** **method=[align | stretch] spacing=[auto | exact]** **startOffset=[length] textLength=[length]**	
title	Normally none	Title for document or element. Wise to only have one per element or document as browser may only look for first.
tref	dx=[lengthlist] dy=[lengthlist] x=[coordlist] y=[coordlist] lengthAdjust=[spacing | spacingAndGlyphs] rotate=[deglist] textLength=[length] xlink:href=[url]	Similar to tspan but text to be drawn is pointed at by the url rather than enclosed by the element as in tspan.
tspan	dx=[lengthlist] dy=[lengthlist] x=[coordlist] y=[coordlist] lengthAdjust=[spacing | spacingAndGlyphs] rotate=[deglist] textLength=[length]	Draws a substring within a text element with origin of substring or origin of individual characters characters defined by (x,y) offset by (dx,dy). Additional rotation can be specified and the expected length of the substring. If actual length is different, lengthAdjust decides whether it gets padded by just varying the spacing.
use	All the presentation attributes height=[length] width=[length] x=[coord] y=[coord] xlink:href=[url]	The **use** element can point either to a symbol, SVG document or a group. The use is effectively replaced by a group.

B.3 SVG Global Attributes

The table below gives a list of the attributes in SVG that can be used by most drawing elements etc.

Attribute	Possible Values	Comment
id	=[name]	The name must be unique in the document.
class	=[name]	The name is used to style sub classes of a set of drawing elements in their own way.
style		Style attribute as in CSS. List of styling declarations separated by semicolons.

B.4 SVG Style Properties and Attributes

The table below gives a list of the style properties in SVG that can also be used as style attributes.

Attribute	Possible Values	Comment
alignment-baseline	=[**auto** | baseline | before-edge | text-before-edge | middle | after-edge | text-after-edge | ideographic | alphabetic | hanging | mathematical | inherit]	
baseline-shift	=[**baseline** | sub | super | [percent] | [length] | inherit]	
clip	=[[shape] | auto | inherit]	
clip-path	=[[url] | none | inherit]	References the clipPath element that defines the clipping.
clip-rule	=[**nonzero** | evenodd | inherit]	Same as for fill-rule.
color	=[[color] | inherit]	CSS colour, better to use fill and stroke properties unless you need a common style across the SVG document and the page in which it is embedded.
color-interpolation		
color-rendering	=[**auto** | optimizeSpeed | optimizeQuality | inherit]	
direction	=[**ltr** | rtl | inherit]	Defines the base writing direction of the text.
dominant-baseline	=[**auto** | use-script | no-change | reset-size | ideographic | alphabetic | hanging | mathematical | inherit]	
fill	=[**none** | [color] | [url]]	The url points to a patten or gradient definition.
fill-opacity	=[fr]	Initial value is 1.
fill-rule	=[**nonzero** | evenodd | inherit]	
font-family	=[list of generic or font names as in CSS]	See CSS.
font-size	=[[length] | larger | smaller | [percent] | inherit]	See CSS.

Attribute	Possible Values	Comment
font-size-adjust	=[[num] | **none** | inherit]	Adjusts the font size to retain legibility. The number defines the required aspect ratio (for example, 0.58 for Verdana). If another font is used instead (for example, Times New Roman with an aspect ratio of 0.46, the font is scaled up in size by 0.58/0.46.
font-stretch	=[**normal** |wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit]	
font-style	=[**normal** | italic | oblique | inherit]	
font-variant	=[**normal** | small-caps | inherit]	
font-weight	=[**normal** | bold | bolder | lighter | 100 | 200 ! 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit]	
glyph-orientaion-vertical	=[**auto** | [degree] | inherit]	Top-down Latin text will be orientated 90 degrees unless this is set to 0.
glyph-orientation-horizontal	=[[degree] | inherit]	Default value is 0. Allowed values are 0, 90, 180, and 270.
kerning	=[**auto** | [length] | inherit]	kerning length is added to the inter-character spacing.
letter-spacing	=[**normal** | [length] | inherit]	
mask	=[[url] | **none** | inherit]	Defines the **mask** element to be used for masking.
onclick, onload etc	Script function call	
opacity	=[[fr] | inherit]	Initial value is 1.
shape-rendering	=[**auto** | optimizeSpeed | crispEdges | geometricPrecision | inherit]	A hint as to how to render the SVG document.
stroke	=[**none** | [color] | [url]]	The url points to a patten or gradient definition.
stroke-dasharrary	=[[length],[length],[length],[length], ...]	A list of lengths that should be even giving the dash and space lengths in order. If an odd number is specified, the list is repeated to make it even.
stroke-dashoffset	=[[length] | none]	Initial value is 0.
stroke-linecap	= [**butt** | round | square | inherit]	Defined for the end of paths and lines.

Attribute	Possible Values	Comment
stroke-linejoin	= [**miter** | round | bevel | inherit]	Specifies the shape to be used at the corners of paths and polylines.
stroke-miterlimit	=[[num] | inherit]	Initial value is 4. Limits the ratio of the miter length to the width of the lines joined by a miter.
stroke-opacity	=[fr]	Initial value is 1.
stroke-width	=[[length] | inherit]	Initial value is 1.
text-anchor	=[**start** | middle | end | inherit]	Starting position of the text string.
text-decoration	=[**none** | underline | overline | line-through | blink | inherit]	See CSS.
text-rendering	=[**auto** | optimizeSpeed | optimizeLegibility | geometricPrecision | inherit]	Allows renderer to make decisions on whether to anti-alias or use font hinting.
transform		
unicode-bidi	=[**normal** | embed | bidi-override | inherit]	See CSS for meaning.
visibility	=[visible | hidden | collapse | **inherit**]	
word-spacing	=[**normal** | [length] | inherit]	
writing-mode	=[**lr-tb** | rl-tb | tb-rl | lr | rl | tb | inherit]	

B.5 Filtering Elements

SVG has a range of image filtering operations that can be performed on the vector graphics image generated before it is displayed. These have not been described in this document.

Element	Attributes	Comment
definition-src		
feBlend	in2 mode=[**normal** | multiply | screen | darken | lighten]	
feColorMatrix	type=[**matrix** | saturate | hueRotate | luminanceToAlpha] values	
feComponentTransfer		
feComposite	in2 k1 k2 k3 k4 operator=[**over** | in | out | atop | xor | arithmetic]	
feConvolveMatrix	bias divisor edgeMode kernelMatrix kernelUnitLength order preserveAlpha targetX targetY	
feDiffuseLighting	diffuseConstant surfaceScale	
feDisplacementMap	in2 scale xChannelSelector=[R | G | B | **A**] yChannelSelector=[R | G | B | **A**]	
feDistantLight	azimuth elevation	

Element	Attributes	Comment
feFlood		
feFuncA		
feFuncB		
feFuncG		
feFuncR		
feGaussianBlur	stdDeviation	
feImage		
feMerge		
feMergeNode	in out	
feMorphology	operator=[**erode** | dilate] radius	
feOffset	dx dy	
fePointLight	x y z	
feSpecularLighting	specularConstant specularExponent surfaceScale	
feSpotLight	limitingConeAngle pointsAtX pointsAtY pointsAtZ x y z	
feTile		
feTurbulence	baseFrequency numOctaves seed stitchTiles=[stitch | **noStitch**] type=[fractalNoise | **turbulence**]	
filter	animate feColorMatrix feComposite feGaussianBlur feMorphology feTile filterRes filterUnits=[objectBoundingBox | userSpaceOnUse] height width primitiveUnits=[objectBoundingBox | userSpaceOnUse] x y	

B.6 Font Elements

SVG has a range of font and glyph definitional facilities that have not been described in this document.

Element	Attributes	Comment
altGlyph	dx dy format glyphRef rotate=[degree]	
altGlyphDef		
altGlyphItem		
font	horiz-adv-x horiz-origin-x horiz-origin-y vert-adv-y vert-origin-x vert-origin-y	
font-face	accent-height ascent bbox cap-height descent font-stretch font-style font-variant font-weight hanging ideographic mathematical overline-position overline-thickness panose-1 slope stemh stemv strikethrough-position strikethrough-thickness underline-position underline-thickness unicode-range units-per-em v-alphabetic widths x-height	
font-face-format		
font-face-name		
font-face-src		
font-face-uri		
glyph	arabic-form d glyph-name horiz-adv-x lang orientation unicode vert-adv-y vert-origin-x vert-origin-y	
glyphRef	dx dy format glyphRef	
hkern	g1 g2 k u1 u2	
missing-glyph	d horiz-adv-x vert-adv-y vert-origin-x vert-origin-y	
vkern	g1 g2 k u1 u2	

B.7 Other Elements

SVG has some other more specific elements that have not been described in this document.

Element	Attributes	Comment
color-profile	local name rendering-intent=[**auto** | perceptual | relative-colorimetric | saturation | absolute-colorimetric]	
cursor	x y	
definition-src		
foreignObject	x y	
marker	**markerHeight=[length] markerWidth=[length]** **markerUnits=[strokeWidth | userSpaceOnUse]** **orient=[auto | [degree]]** **preserveAspectRatio=[align] [meetOrSlice]** **refX=[coord] refY=[coord]** **viewBox=[coord] [coord] [length] [length]**	Defines a marker where (refX,refY) is the reference point of the marker. If attribute **orient** is set to auto, the marker is oriented in the current direction of the path (for example an arrow head).
metadata		
mpath	xmlns:xlink etc xlink:href externalResourcesRequired=[false | true]	Sub-element used by animateMotion to define a path instead of its **path** attribute.
pattern	**patternContentUnits=[objectBoundingBox | userSpaceOnUse]** **patternTransform=[transformlist]** **patternUnits=[objectBoundingBox | userSpaceOnUse]** **preserveAspectRatio=[align] [meetOrSlice]** **viewBox=[coord] [coord] [length] [length]** **x=[coord] y=[coord] width=[length] height=[length]**	
switch	**requireFeatures=[org.w3c.svg.static | org.w3c.svg.dynamic | org.w3c.dom.svg | org.w3c.svg.lang | org.w3c.svg.animation etc]** **systemLanguage=[comma separated list of languages such as** en**]**	
view	preserveAspectRatio=[align] [meetOrSlice] viewBox=[coord] [coord] [length] [length] viewTarget zoomAndPan=[disable | **magnify** | zoom]	

C. References

The main reference is the W3C site from which most other references are accessible.

URL	Comment
http://www.w3.org/Graphics/SVG/	Overview of SVG Activity
http://www.w3.org/Graphics/SVG/Group/	SVG Working Group Home Page
http://www.w3.org/TR/SVG/	Latest Version of the SVG Document
http://www.adobe.com/svg/	Adobe SVG Plug-in plus tutorial information and demonstrations
http://sis.cmis.csiro.au/svg/	CSIRO SVG Tool Kit. Can display SVGs and convert SVGs to JPEG.
http://www.alphaworks.ibm.com/tech/svgview	IBM SVG Viewer
http://xml.apache.org/batik/	Apache's Batik which derives from Jackaroo
http://www.jasc.com/webdraw.asp	Jasc WebDraw SVG Editor
http://www.mayura.com/	Mayura Draw Editor
http://www.levien.com/svg/	Gill: Gnome Illustration Application
http://www.digapp.com/newpages/svg2pdf.html	Converts SVGs to PDF
http://www.padc.mmpc.is.tsukuba.ac.jp/member/morik/fdssvg/	Converts bitmap images to SVG
http://broadway.cs.nott.ac.uk/projects/SVG/svgpl/	SVG-PL, a Perl library for creating legal SVG documents
http://www.w3.org/Graphics/SVG/Test/	SVG Conformance Test Suite
http://www.savagesoftware.com/products/svgtoolkit.html	Savage Software SVG Toolkit
http://www.square1.nl/index.htm	Graphics Connection Conversion Tools
http://www.celinea.com/	CR2V, Raster to Vector Converter
http://www.graphicservlets.com/wmf2svg.htm	WMF to SVG Converter