SVG Tutorial

David Duce [*], Ivan Herman [+], Bob Hopgood [*]

[*] Oxford Brookes University, [+] World Wide Web Consortium

Contents

1. Introduction

1.1 Images on the Web

The early browsers for the Web were predominantly aimed at retrieval of textual information. Tim Berners-Lee's original browser for the NeXT computer did allow images to be viewed but they popped up in a separate window and were not an integral part of the Web page. In January 1993, the Mosaic browser was released by NCSA. The browser was simple to download and, by the Autumn of 1993, was available for X workstations, PCs and the Mac. From 50 Web servers at the start of 1993, Web traffic had risen to 1% of internet traffic by October and 2.5% by the end of the year. About a million downloads of the Mosaic browser took place that year. In February of 1993, Mark Andreessen proposed the <IMG> element as an extension to Mosaic's HTML to provide a way of adding images to Web pages. In 1994, Dave Raggett developed an X-browser that allowed text to flow around images and tables and from then on images were an accepted part of the Web page. Web pages became glossier and the enormous growth of the Web started [1] [2]. Organisations could customise their home pages with the company logo. Maps, albeit images, could be added to show how to reach the organisation. Its products could be displayed on the Web. Eventually, the Web would become a major commercial outlet.

1.2 Supported Image Formats

The only image format supported by all the early browsers was the GIF format developed by CompuServe. The original GIF format only supported 256 colours. By 1995, the possibility of JPEG also being supported was growing and the lossy compression available with JPEG meant that real world images could be half or a third of the size of the same image stored in GIF format without loss of information for the designated use. Also, JPEG was a full 24-bit format allowing the possibility of 16 million colours [3].

The ability to add images of various types (maps, drawings, photographs etc) to Web pages enhanced the capabilities and made them more exciting. The downside was that the inclusion of images slowed the download time of the Web page by an order of magnitude. Browsers provided the option of turning off the images thus negating their use for core information. Browsers opened multiple channels to improve the download speed but, at the same time, congested the Internet for others even more.

In December 1994, CompuServe and Unisys announced that developers would need to pay a license fee to use the GIF format as the technique used to compress the image data, called LZW (after Lempel-Ziv-Welch), was patented by Unisys. In consequence (although in the end license fees were not charged to end users), a new image format, PNG (Portable Network Graphics) [4] was developed that does not have the patent problems associated with GIF. It provides an efficient lossless format for greyscale, true colour and palette-based images [5].

The latest versions of the main browsers provide reasonable support for GIF, PNG and JPEG images. The GIF format is being overtaken by PNG (especially where good colour representation of transparency is required) but quite slowly and JPEG 2000 will give that format a new lease of life with its improved compression techniques. The very latest mobile phones are likely to provide hardware/software support for the JPEG 2000 format.

1.3 Images are not Computer Graphics

But images are not computer graphics. The Oxford Dictionary of Computing has the following definition of computer graphics: **the creation, manipulation of, analysis of, and interaction with pictorial representations of objects and data using computers**. A digital image on the other hand is usually a 2-dimensional regular grid of pixels. The ability to interact with it is limited. Being just an array of pixels, most of the information that existed in the original object is lost. All that remains is what the eye can see. Figure 1.1 shows the difference between zooming in on the plane if the original is an image compared with what is seen if the drawing is defined as lines and areas.

The aim of this paper is to look at 2-dimensional computer graphics on the Web and to give some insight into why the Web has come so far without computer graphics being an integral part (given the importance of computer graphics in many applications).

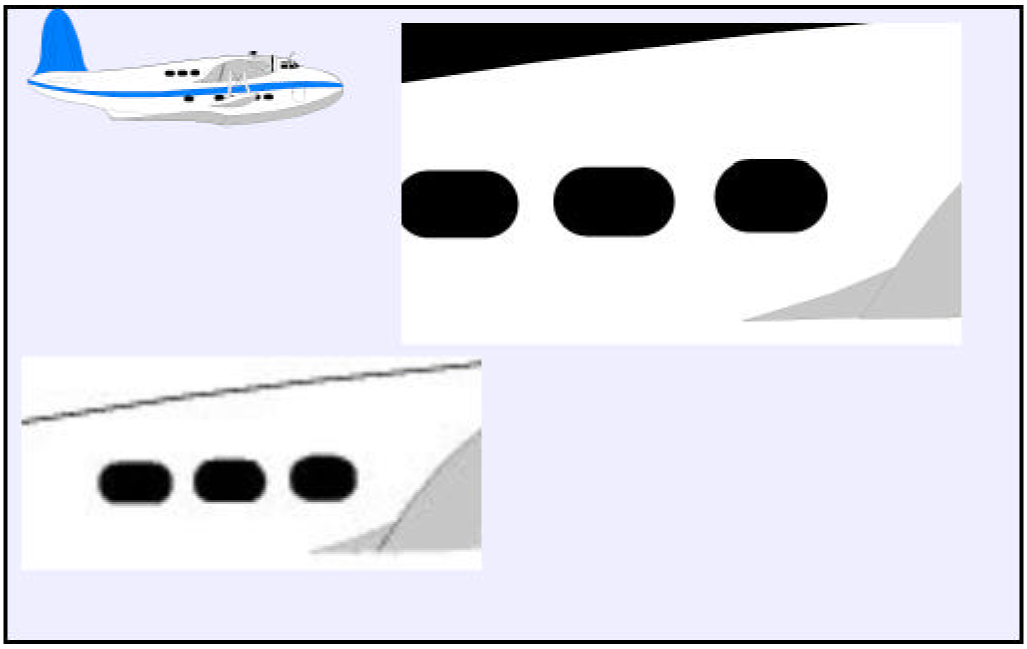


Figure 1.1: Images versus Vector Graphics

The image formats all share many disadvantages that are serious obstacles to the development and adoption of new technologies on the Web. Some of the major problems are listed below.

Bandwidth

Images are large. Improvements in network bandwidth have helped to hide this. Also image compression techniques have improved. Even so, images are a major bottleneck to accessing Web sites. This creates significant problems when designers want to follow their own style in creating new Web pages.

Flexibility

Images inherently have a fixed resolution. In consequence, an application destined to run on a range of PCs, PDAs and mobile phones is unable to adapt to the constraints of the device. Colour, resolution, aspect ratio and bandwidth often differ significantly between devices.

Hyperlinking

Hyperlinking is a fundamental requirement on the Web. However, to link to different places, dependent on where the user clicks on an image, is not simple. Early on, image maps were added to HTML. This allowed the coordinates of where the user clicked to be returned to the server where a program was run to determine which page to link to. Server side image maps are not efficient adding another round trip from client to server. The map is separate from the HTML page and is dependent on the server for translation. Different servers used different map file formats so that pages often could only be read by certain browsers. Client side image maps were added in HTML 3.0 and these allowed rectangular, elliptical or polygonal areas to be defined. Clicking on an area causes the link defined for that area to be taken. Creating image maps is cumbersome and is not related to the real objects being viewed but their image on the display.

Animation and Interaction

Many applications profit from the use of animation and interaction (cartography, CAD, remote teaching, etc). Image formats only provide crude animation limited to the sequential playback of a sequence of images combined into a single file. Interaction is limited to the use of image maps.

Separation of Style from Content

The same drawing in terms of meaning can be represented in many different ways dependent on the capabilities of the device. Dotted line on a mono display might be rendered as a different colour on a colour display. Images do not have the ability to make such changes.

Integration

In the early days of the Web, an HTML page was transmitted across the Internet using the HTTP protocol and there was a 1-1 relationship between documents and downloads. Today, the Web is much more complex. Separating style and content meant that a style sheet might be transmitted as well as the Web page. The move to XML [6] allows appropriate markup for different information in the Web page. No longer is it necessary to force the HTML elements defined for textual documents to be used for other purposes. Mathematical markup [7], multimedia [8], and chemical markup, for example, each use their own XML application. Any computer graphics on the Web should be integrated with this model of the Web. In consequence, the transmission of images as a final form rendering of something that has semantic content is likely to decrease. The image formats will be used for their primary purpose of transmitting real-world images where the photograph is the content.

This tutorial will concentrate on the way 2D vector computer graphics is being made a more integral part of the Web, in particular through open as opposed to vendor specific standards.

1.4 Multimedia is not Computer Graphics

Just as images are not computer graphics so multimedia presentations are not computer graphics. That is not to say that combining a variety of resources to create a meaningful presentation does not have merit. It is just that the emphasis is on integration and timing rather that the graphical content.

For example, SMIL is an open standard whose main aim is integrating a set of disparate resources scattered across the Web into a synchronised multimedia integration. Many problems arise such as layout, timing and bandwidth. Such systems are not considered further in this paper which concentrates on 2D graphics system in use on the Web.

Proprietary multimedia systems also exist that at times give the impression of being 2D graphics file formats. A good example is Macromedia Flash. Here we have two problems. It is neither multimedia or computer graphics in the strict sense as far as the Web is concerned. The multimedia integration occurs external to the Web. As far as the web is concerned, there is not a great deal of difference between a Flash presentation downloaded to a browser and the playback of an MPG video. Both show images that change over time. Neither make use of the Web as a distributed resource or the special features of high quality 2D graphics. A tutorial on Flash would start with the basic principle of a **timeline** followed by animation relative to that timeline and would eventually come round to describing the computer graphics and other objects to be integrated and animated.

At the other end of the spectra are proprietary systems such as Adobe Illustrator and its associated proprietary file format which have a much closer affinity to vector graphics. However, Illustrator is more the creation tool for the production of the computer graphics. Adobe has been a significant supported for the Scalable Vector Graphics (SVG) file format for the Web and Adobe Illustrator performs well as the creator of such files. For these reasons, this paper will concentrate on the open file format standards for the Web, WebCGM and SVG.

2. Early Vector Graphics on the Web

2.1 CGM

The ISO Computer Graphics Metafile (CGM) Standard [9] is a format for describing vector graphic pictures compactly. It has proven to be a very good format for a whole range of demanding 2-dimensional graphics presentation applications [10]. CGM first became an ISO standard in 1987 and has been enhanced over the years by enriching the drawing primitive set and providing more structural information. As it became richer, the concept of CGM Profiles for specific application sectors evolved.

In 1997, an analysis was done by W3C to see if it would be possible to define a CGM Web Profile that could satisfy the requirements for computer graphics on the Web. It passed most of the necessary criteria. CGM was an open specification that had been widely implemented. CGM separated abstract syntax from the concrete representation allowing multiple encodings to be defined. The vector drawing facilities were more than what was required. The HTML <OBJECT> element could be used to add CGM diagrams to a Web page. However, styling in CGM was provided by the bundle table approach also used in the ISO standards, PHIGS and GKS. This was a different approach to the one adopted in Cascading Style Sheets (CSS) on the Web to separate style and content. A major drawback was that linkage between drawings in the manner of the Web was not provided.

2.2 CGM on the Web

The CGM community saw major advantages in using CGM rather than images on the Web for schematic drawings:

- Vector graphics can be zoomed in and out while retaining the quality of the picture, unlike images.
- Vector graphics files are smaller and can be downloaded and viewed faster than images.
- Vector graphics can be interacted with in a meaningful way.
- Text in a CGM vector graphics drawing can be searched as easily as text in an HTML page.

These considerations in no way try to denigrate the use of image formats where they are appropriate.

In consequence, CGM suppliers provided CGM plug-ins to access CGM vector graphics on the Web using the existing encodings. A CGM MIME type was agreed in 1995. The only problem was that the CGMs produced by one vendor could not be read by viewers produced by another as different Profiles were implemented and the hyperlinking mechanisms introduced differed from one supplier to another.

A joint activity between the World Wide Web Consortium (W3C) and the CGM Open Consortium [11] (launched in May 1998) was initiated to define a common Web Profile for CGM that would be accepted both by ISO and W3C. This resulted in the WebCGM Profile, completed in January 1999 [12].

2.3 WebCGM Profile

WebCGM was based on the ATA CGM Profile for graphics interchange (GREXCHANGE). The Graphics Working Group (ATA 2100) of ATA, the Air Transport Association, had defined this CGM Profile for the aerospace industry. It was also working towards an intelligent graphics exchange profile (IGEXCHANGE) that associated semantic information to aid query, searching and navigation.

The structure of a WebCGM file is shown in Figure 2.1. Picture size and scaling and properties such as line width and background colour are defined in the Picture Descriptor. This is equivalent to styling provided for the <body> element in an HTML page.

Each picture contains CGM graphic elements. There are 4 groupings of graphical elements provided:

- **grobject**: a graphical object with a unique id and possibly linkURI and tooltip attributes. It is used to identify sources and destinations of hyperlinks. It is also possible to define the region of the group for picking and the initial view when the group is linked to. For example, more than the group may need to be visible to indicate the context.
- **layer**: this has a name and a list of objects. It allows a picture to be divided into a set of graphical layers that can be used to switch display to parts of an illustration.
- **para**: defines a paragraph as the grouping of several text drawing elements. The elements may be scattered across the drawing but for searching purposes are similar to an HTML paragraph.
- **sub-para**: a sub-paragraph used to identify fragments of text (for example as hotspots within a paragraph).

These provide the basis for searching and linking within and between CGM pictures. An object may be the target of a link. Browsers are expected to move the object into view and scale it to fit into the viewport. If the object has a **ViewContext** attribute the rectangle defining the view context must be within the viewport.

Links from WebCGM objects are defined by **linkURI** elements that are modelled on the XLink facilities [13]. Objects may have multiple links. Links can be bi-directional. Linkage can be from places outside the CGM and links from the CGM can be to any destination defined by a URL. Following a link can display the new picture in a separate window, load the picture into the current frame, load it over the parent of the current frame or replace the current picture.

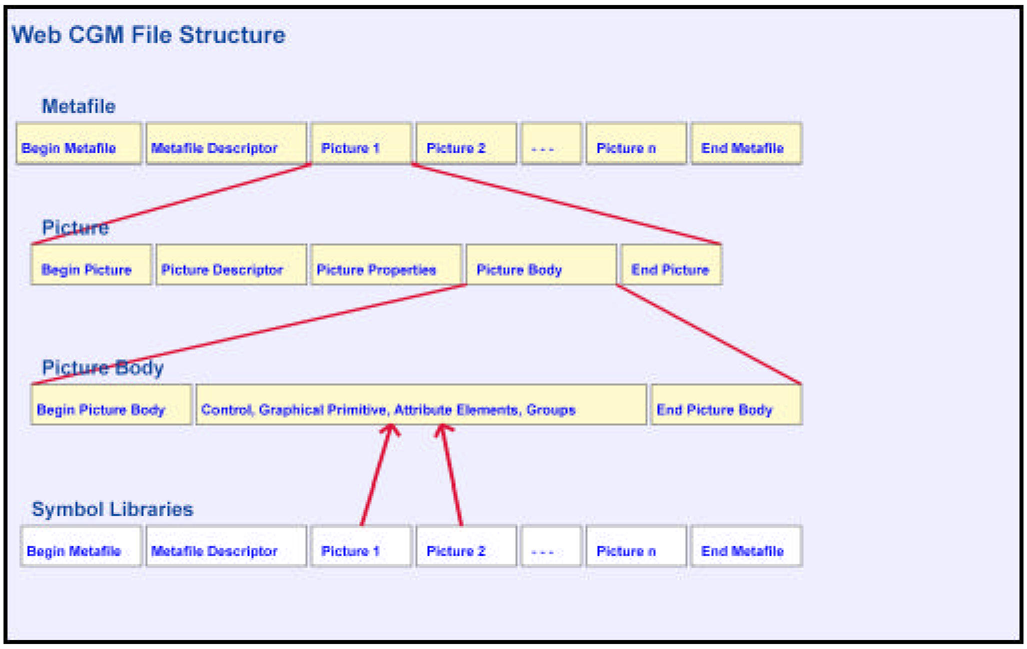


Figure 2.1: CGM Architecture

WebCGM is a reasonably full profile of CGM containing a rich set of graphics elements:

- Polylines, disjoint polylines, polygons, polygon sets.
- Rectangles, circles, ellipses, circular and elliptical arcs, pie slices.
- Text: both the Restricted Text primitive of CGM (which defines its extent box) and the Append Text element (continuation of a text string with a change of attributes).
- Closed Figure and Compound Line: allows complex paths to be defined as a sequence of other primitives.
- Polysymbol: placement of a sequence of symbols defined in the Symbol Library (another valid WebCGM metafile).
- Smooth curves: the smooth piece-wise cubic Bezier defined by CGM's Polybezier element.
- Cell Array and Tile Array allow PNG, and JPEG images to be integrated with the vector drawing.

Most of the line and fill attributes of CGM are included but only as INDIVIDUAL attributes. The bundled attribute functionality of CGM is omitted. Thus, WebCGM diagrams consider properties such as linestyle, color, fill types etc as content rather than styling.

The full set of CGM colour models is provided including sRGB and sRGB-alpha. International text is defined by selecting either Unicode UTF-8 or UTF-16.

2.4 WebCGM Viewers

Probably the most widely used Viewer is the Micrografx free ActiveCGM plug-in [14]. SDI has also released a CGM Plug-in [15] while Tech Illustrator has a TI/WebCGM Hotspot Plug-in module [16] to author hotspots for exporting to CGMs. There is good industrial support for WebCGM and it is widely used in the CAD and aerospace industries. A major interoperability demonstration took place at XML Open in Granada in May 1999.

A good source of further information on CGM is CGM Open [11], an organisation dedicated to open and interoperable standards for the exchange of graphical information. The W3C Web site is also a valuable source of news and reference information.

3. Introduction

3.1 Scalable Vector Graphics

Using an image format for evctor graphics has some major 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.

Using CGM was a solution on the Web prior to the arrival of XML. However it is aimed primarily at the CAD rather than graphics arts industry and is of the same generation as SGML. In consequence, there was interest within the World Wide Web Consortium (W3C) for a W3C 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 such a drawing allows greater and greater detail to be seen if the drawing is complex.

3.2 An XML Application

By 1997, it was becoming clear that the Extensible Markup Language (XML) [6] would make a profound difference to the way that the Web would develop. The first edition became a W3C Recommendation in February 1998 but by then its impact was already being felt. XML is a meta-language for defining markup languages. An XML application is a document format for storing structured information in an unambiguous and appropriate manner.

An XML application consists of 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 **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**. 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 (the root element) that encloses the complete drawing definition.
- Every start tag must have a correctly nested end tag
- The form of the start and end tags must be identical. If the start tag is upper case so must be the end tag.
- 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.

3.3 Submissions to W3C

The question now arose: could XML markup be used to express vector graphics? In 1998, there were four Submissions to W3C proposing an XML-based vector graphics markup language for the Web. In order, these were:

- **Web Schematics** **[17]**: similar to the troff pic language, it defined objects with anchor points that could be composed into pictures. This submission was made by Bob hopgood and david Duce of the Rutherford Appleton Laboratory and Vincent Quint of INRIA.
- **Precision Graphics Markup Language (PGML)** **[18]**: a lower level language that could be described as an XML-based version of PostScript.
- **Vector Markup Language (VML)** **[19]**: just as PGML has a relationship to Postscript, VML had a similar relationship to PowerPoint.
- **DrawML** **[20]**: a constraint-based higher level language that allowed the drawing to adjust to the content. Changing the text in a box would increase the size of the box and adjust the box surrounding that box etc.

In retrospect, it is surprising that the CGM community did not put forward a proposal for a CGM Profile defined using XML notation.

These submissions resulted in a Working Group being formed to define a single language for vector drawings on the Web called Scalable Vector Graphics (SVG) [21]. The Candidate Recommendation stage within the W3C process exists just before the full Recommendation stage and is to allow trial implementations to test the quality of the specification. The strong interest in SVG meant that there were implementations of the Candidate Recommendation early in 2001 even though the Candidate Recommendation was not issued until November 2000. SVG became a W3C Recommendation in September 2001. The W3C Recommendation is the subject of this Tutorial.

Of the four Submissions, PGML probably had the most impact on the functionality of SVG. Even so, SVG has evolved into a standard significantly different from all of the initial Submissions.

3.4 SVG: an Application of XML

SVG is an application of XML. This has the benefit that the overall syntactic structure of SVG is known and parsers exist to handle it. It also means that SVG can benefit from the other activities within W3C concerned with the XML Family of standards. In many cases, this is a strong plus but occasionally the constraints imposed by the other standards will mean that the functionality provided within SVG may be less elegant or have different characteristics from the form it would have taken if it had not been part of the XML Family. However the advantages far outweigh the disadvantages. Some examples of the influences on SVG are:

- **Cascading Style Sheets (CSS)** [22]: CSS is used to separate style from content initially in an HTML document. A CSS style sheet consists of a set of commands that specify the styling to be associated with a specific element. As CSS is not restricted to HTML elements, it can also be used to style an XML application.
- **Namespaces in XML** [23]: with many XML applications emerging, it is more likely that several will be used together in which case it is necessary to identify which elements belong to which application. XML achieves this by defining a prefix that identifies the namespace. For example, `<svg:text>` defines the start of the SVG text element. The prefix may be anything the user wants it to be as long as the appropriate namespace declaration identifies the application.
- **XML Linking Language (XLink)** [13]: as all XML applications are likely to require hyperlinking, a separate Recommendation, XLink, defines a flexible hyperlinking mechanism. Rather than define its own, SVG is able to use the XLink hyperlinking functionality via the XLink namespace.
- **XSL Transformations (XSLT)** [24]: XSLT defines a transformation functionality to be applied to XML documents. CSS effectively performs a single pass through an HTML document transforming the elements by defining their styling. XSLT provides similar functionality in terms of styling but also allows complex transformations of the XML documents. For SVG, higher-level functionality can be realised by defining an XSLT transformation down into SVG. In consequence, SVG need not define such functionality within the core version of SVG.
- **Synchronised Multimedia Integration Language (SMIL)** [8]: the SVG Working Group included animation functionality within its design objectives. SMIL was also considering similar functionality for multi-media presentations. The two Working Groups have, therefore, produced a single suite of animation functionality that can be used by both SMIL and SVG.
- **Document Object Model (DOM)** [25]: The DOM provides a standard method of interacting with an XML application. In consequence, SVG can use this functionality as the basis for interaction between a user and an SVG drawing.

3.5 Getting Started with SVG

Figure 3.1 shows the result that an SVG-enabled browser or viewer would make of the SVG document defined below.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN" "http://www.w3.org/TR/2001/REC-SVG-
20010904/DTD/svg10.dtd">
<svg width="320" height="220">
<rect width="320" height="220" fill="white" stroke="black" />
<g transform="translate(10 10)">
<g stroke="none" fill="lime">
<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 140Z"/>
</g>
</g>
</svg>
```

The initial XML declaration and the Document Type Declaration for SVG 1.0 can be omitted and future examples will do this.

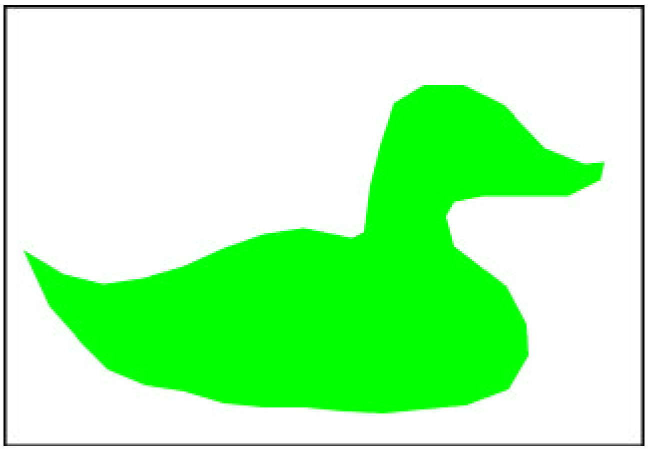


Figure 3.1: Simple SVG Drawing

This simple example reveals some of the basic characteristics of SVG:

SVG is an XML application

The root element is svg. All the elements are correctly nested. The attributes are enclosed in quotes and the **path** and **rect** elements do not have any content and so use the shorthand format.

SVG has a hierarchical structure

The g element in the example groups a set of elements. In this case there is just a single **path** element but normally there would be a sequence of drawing elements making up an object. Attributes can be defined on the g element that apply to the whole group. The hierarchical structure in SVG is similar to the scene graph approach used in systems like OpenInventor, PostScript and most graphics editors.

The simplest way to use SVG is to open an SVG file with an SVG-enabled web browser (either via an SVG plug-in or providing local support). An SVG diagram can be incorporated into a web page defined in HTML. The SVG document is defined and stored in a file with '.svg' as the file extension. To add it to the web page then requires, for example:

```
<p>This can be shown in the following diagram:</p>
<object width="320" height="220" data="myfirstsvg.svg" 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 the user 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. The user can 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 [39] and follow the instructions.

There are a number of stand-alone viewers for SVG that can also be used [40] [41] [42]. You just open the SVG file and it will be displayed in the viewer's window. There are also support tools for 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 [29].

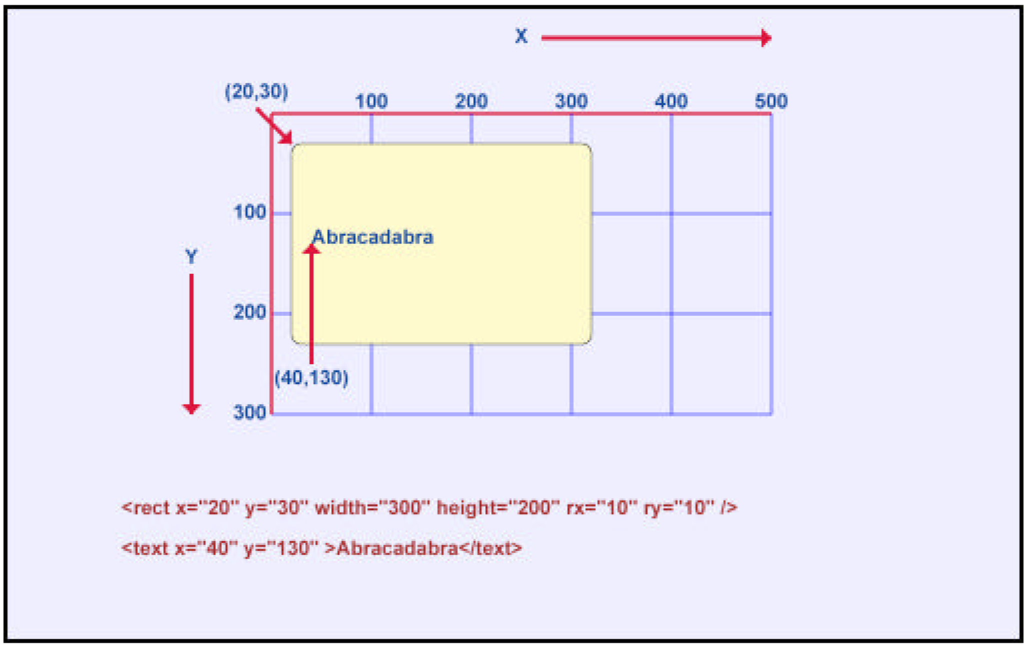
4. Coordinates and Rendering

4.1 Rectangles and Text

It is difficult to talk about either coordinates or rendering in a vacuum so we first need to specify two 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>
```



```
<rect x="20" y="30" width="300" height="200" rx="10" ry="10" />
<text x="40" y="130" >Abracadabra</text>
```

Figure 4.1: 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.

4.2 Coordinates

All graphics elements are rendered conceptually on to an SVG infinite canvas. A viewport can be established that defines a finite rectangular region within the canvas. Rendering uses the painter's model; elements later in the document are rendered on top of the preceding ones.

The viewport is specified by attributes of the **svg** element. Explicit **width** and **height** attributes can be set as in the example in Section 3.5. An alternative is to use the **viewBox** attribute which specifies the lower and upper bounds of the viewport in bot the X and Y directions.

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:

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

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.

The two approaches (width/height and viewport) are subtly different. In the first example using width and height, no units have been specified so pixels are the assumed coordinate system. The viewport required is 320 pixels wide and 220 pixels high. The local user coordinate system for the duck is also set to be 320 by 220 with one pixel equal to one local user coordinate. In the second case, the local user coordinate is set to 500 wide and 300 high and this is to be mapped to **fit** in the viewport. A small viewport would have the mapping from user coordinate to pixels different from a large viewport. If the aspect ratio of the viewport is different from that of the viewBox then various options are provided as to how the user would like the mapping to take place.

In SVG, if no units are specified the assumption is that the coordinates are defined in the local coordinate system. However, in defining the viewport size and in specifying the drawing, the complete set of units defined in CSS are available to the user (em, ex, px, pt, pc, cm, mm, in, %).

If the drawing is to be displayed as part of a Web page, a complex negotiation takes place between the SVG plug-in and the browser taking into account any constraints imposed by the user on inserting the drawing in the Web page or by the styling applied to the page as a whole. As a result of this negotiation, part of the image could be clipped, scaled or distorted, depending on how the constraints are resolved. The user can control the effect somewhat through a **preserveAspectRatio** attribute and by specifying whether all the drawing must be visible or whether some parts can be obscured.

We shall assume in our examples 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.

4.3 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.

4.4 Rendering Attributes and Styling Properties

The separation of style and content has been an issue in text processing and computer graphics for many years. In the Unix typesetting system, troff, for example, the raw text of a document could be "marked up" to indicate headings, paragraphs, enumerated lists, tables etc. The precise way in which these documents elements were to be presented was described through a macro language. Typically a set of macros (for example, the "ms" macro set) would be constructed to impart a particular appearance or "house style" to a collection of documents. The LaTeX document production system took a similar approach, essentially extending the TeX typesetting system with a particular markup command language. The task of constructing a document using LaTex was reduced (as Lamport puts it) to a "logical design" task. The LaTeX system provided typographic design, through particular style files, and the document's author provided the logical design. A whole class of documents (for example the papers in a journal) can thus be given a uniform appearance; an appearance furthermore, that is controlled by a design expert. Word processing systems such as MSWord and Framemaker provide stylesheets that can be used in a similar way.

Conceptual separation is, however, rarely so clean, and both text and word processing systems also provide functionality that enable the overall style rules to be modified (for example, functionality to change to a bold or italic font at a particular point in a document). One author might use bold text directly to emphasise a word while another might use italic for the same purpose, even though an **emphasis** style is provided. This might be laziness on the part of the document author, though sometimes bold and italic are used directly because bold and italic are intrinsic aspects of the presentation of the text, for example, a trademark might be set in a particular font and weight of text.

A similar separation between style and content in Web documents has been achieved by CSS [22]. The page author defines the content and structure (the logical design) of the Web document using HTML elements such as h1, h2, ul, etc. A separate style sheet controls the visual appearance (the visual design) of these elements when rendering in a Web browser or printed. Such style sheets can be embedded directly into Web pages or can be linked to the pages through an appropriate URL. This basic approach provides a mechanism to control the consistency of the visual rendering of a collection of Web documents. Advantages of this approach include:

- **Easy maintenance**: changing the colour of all **h1** elements can be done by changing just the style sheet, instead of scanning through the whole document.
- **House style**: can be defined by a collection of separate style sheet files.
- **Clarity**: pages using style sheets are usually structurally cleaner and hence easier to maintain.
- **Adaptation to the end-user**: style sheets may include special statements for audio browsers; browsers may allow the end user to use personal style sheets to adapt to personal disabilities or operating environment.
- **Design control**: style sheets may be prepared by professional designers, thus improving the overall visual quality and representation of the Web pages.

The use of style sheets is not limited to HTML; style sheets may also be used with XML documents in a similar way.

There are some parallels in the development of style control in computer graphics. In early graphics systems it was commonplace to control the visual appearance of graphical output primitives by attributes, for example to control properties such as linestyle, line width, colour, text font, etc. Attributes were typically set modally, for example by a set_colour function, the value remaining in force until a new value was set. If a particular attribute value was not supported by a particular device, it was permissible to simulate the effect of that value using other values for that attribute, other attributes and other primitives. A particular dashed linestyle could, for example, be simulated by a sequence of individual lines. The essence of this approach was that the application provided a precise specification of the required appearance and the system did its best to achieve the specified effect.

During the development of the Graphical Kernel System (GKS) it became clear that visual appearance could be either styling or an intrinsic part of the information to be presented. In architecture, different patterns denote different types of building material in a precise way; indiscriminate substitution may result in a house of sand rather than of stone! At other times, patterns are used purely to achieve differentiation between different types of object, the precise pattern used is unimportant, what matters is that pattern A should be visually distinguishable from pattern B. GKS [26] distinguished between global attributes which have the same value on all devices and attributes defined indirectly by a pointer into a table located on the device. The attribute values in the table could be different for different devices. Some attributes were always specified globally while others could be defined globally or indirectly depending on the application usage.

SVG is defined as an XML language and makes use of the styling functionality provided by CSS for XML documents. However, as hinted at above, styling for graphics is potentially more complex than for text (or at least more complex than the styling model for Web documents). Is colour in graphics, for example, style or content? If colour is used on a map to differentiate different countries, it is probably style. What is important is that the colour of one country should be distinguishable from that of another. Styling can be very valuable in this situation: the best choice of colour might depend on the context in which the map is used, specifying colour through the style mechanism makes it straightforward to change colours from one context to another.

However colour is not always style. The colour chosen in a logo, or in an artistic image, or in the precise representation of real world objects is inherently a part of the content of the picture. In GKS:94 terms [27], colour here is an attribute completely defined in the NDC picture, to be rendered exactly (so far as is possible) in every view of the picture. Changing colour through a style sheet mechanism in such a picture in a Web document would be fundamentally wrong.

These cases: colour as style and colour as an intrinsic property of a primitive, are recognised in SVG and two different mechanisms for setting visual attributes are provided. One method is to set rendering attributes directly. For example:

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

This defines two rectangles, the first is yellow with a black border; the second is hollow with a red border.

The second method using styling is illustrated by:

```
<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"/>
<rect class="different" x="20" y="330" width="300" height="200"/>
</svg>
```

This achieves the same visual result as the first approach. The "style" element encloses a style sheet expressed in the CSS syntax. (The CDATA annotation is used in order to escape the style language from the XML syntax checker.) Two styles are defined, the first for rectangles in general (filled in yellow with a black border) and a second for rectangles belonging to the class "different" defining rectangles with a red border and hollow interior.

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"/>
<rect class="different" x="20" y="330" width="300" height="200"/>
</svg>
```

Style can also be associated directly with an element through the style attribute. The example above could also be written:

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

SVG allows pictures to have arbitrary hierarchical structure. CSS provides powerful mechanisms for controlling appearance, both on the basis of the values of attributes (usually the class attribute, but other attributes could be used also) and the actual structure of the SVG element tree. It is also possible to write:

```
rect [class ~="different"] {stroke:red; fill:none}
```

which would select rectangles whose class attribute contains the value different in a set of space separated class attribute values. The "." notation introduced earlier in fact corresponds to the "~=" construct. A more complex example is shown below.

```
<svg width="400" height="250" >
<style type="text/css"> <![CDATA[
rect {stroke:black; fill:white}
rect.different {stroke:red; stroke-width:4; fill:none}
rect.different.again {stroke:none; fill:white}
rect.different.again.encore {stroke:blue; stroke-width:8; fill:none}
]]>
</style>
<rect x="20" y="20" width="100" height="100"/>
<rect class="different" x="20" y="140" width="100" height="100"/>
<rect class="different again" x="140" y="20" width="100" height="100"/>
<rect class="different again encore" x="140" y="140" width="100" height="100" />
</svg>
```

Figure 4.2 shows the result.

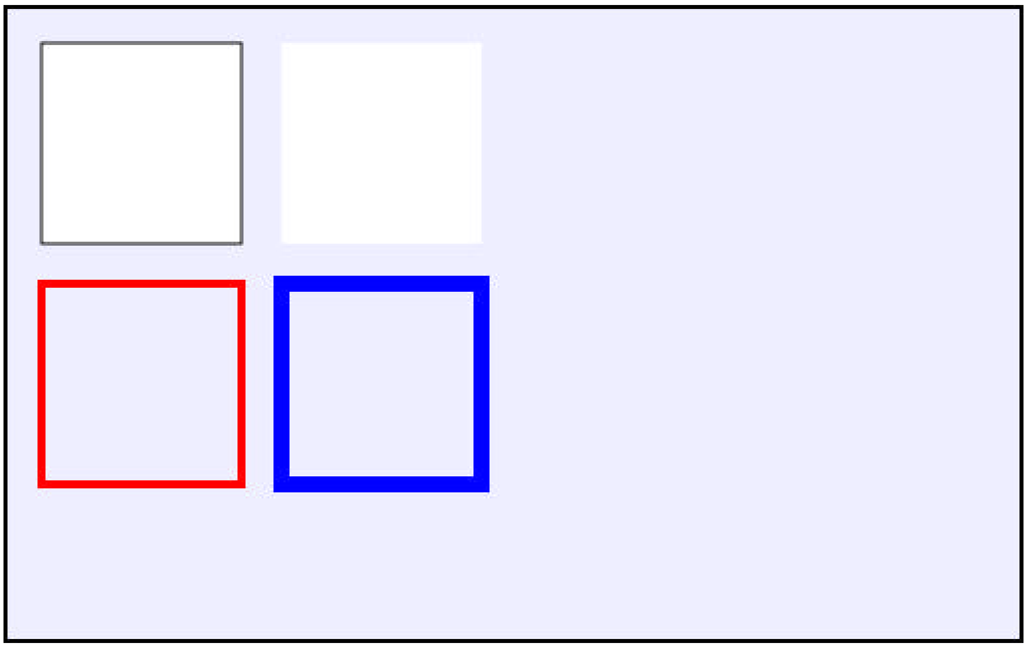


Figure 4.2: Class selection

The CSS functionality for matching the structure of the document tree is quite powerful, though slanted more towards the structures found in text documents than the more general structures found in graphics. A full discussion of this functionality is beyond the scope of this paper, but the example below illustrates the general idea.

```
<svg width="600" height="400" >
<style type="text/css"> <![CDATA[
rect {stroke:black; fill:yellow}
g > rect:first-child {stroke:red; fill:none}
]]>
</style>
<g>
<rect x="20" y="20" width="100" height="100"/>
<rect x="140" y="20" width="100" height="100"/>
</g>
</svg>
```

The selector '>' matches any **rect** element that is the first child of a **g** element.

One of the important concepts in CSS is the notion of **cascade**. Three different types of style sheet can be associated with a document: author, user and user agent. The author of a document can supply style information, so can the user and so can the user agent (usually a browser). In general, style sheets from these three sources may overlap in the styling they specify for a particular element (indeed there may be overlap from within a single style sheet - there is an example of this in Figure 4.4) and so the notion of cascade is introduced to define the effect. In essence, weights are assigned to each style rule and when several rules apply, the one with the greatest weight takes precedence. The details are quite involved and go beyond the scope of this paper. The interested reader is referred to the CSS2 specification [22]. One of the consequences though of this general mechanism is that presentation attributes attached to SVG elements have lower priority than other CSS style rules specified in author style sheets or style attributes. SVG and CSS do not have the equivalent of the GKS Aspect Source Flags [26], so there is no general way to ensure that the analogues of individual attribute specification (presentation attributes) will actually apply in all contexts in which SVG is used. From a graphics perspective this might be considered unfortunate, but it is the price paid for embedding graphics in a context where different priorities normally pertain.

SVG is now looking at the requirements of mobile devices. One of the problems that has arisen is that for some devices it would be useful if the attributes could be tailored to the particular device. Zooming in on an SVG diagram using a mobile phone soon results in a single line covering the whole display. One option being considered by the Working Group is the possibility of attributes being defined indirectly with the values pointed at being set differently on different devices. So it is possible that the final attribute model for SVG will be quite similar to the one used in GKS.

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.

4.5 Following Examples

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

```
<svg viewbox= "0 0 512 320" >
<title>Slide Title</title>
<rect x="2" y="2" width="508" height="318"/>
<!-- ************** Coloured Screen Area 512 by 320 ******************** -->
<!-- ************** 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 (512,320) space as follows:

Figure 4.3: Slide background, 512 by 320

The rectangle is set two pixes in from the edge to make sure all the border is visible.

5: SVG Drawing Elements

5.1 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.

5.2 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">
```

SVG is designed for a wide range of applications on the Web. The drawings may be complex and download times are important. In consequence, the conciseness of path expressions is of fundamental importance. It is for this reason that the path expressions themselves are not defined using a more verbose XML syntax. Every attempt is made to keep the number of characters in path expressions to the minimum. For this reason, paths are not restricted to polylines. Quadratic and cubic Bezier splines and elliptical arcs are also provided.

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 5.2 shows some more complex examples.

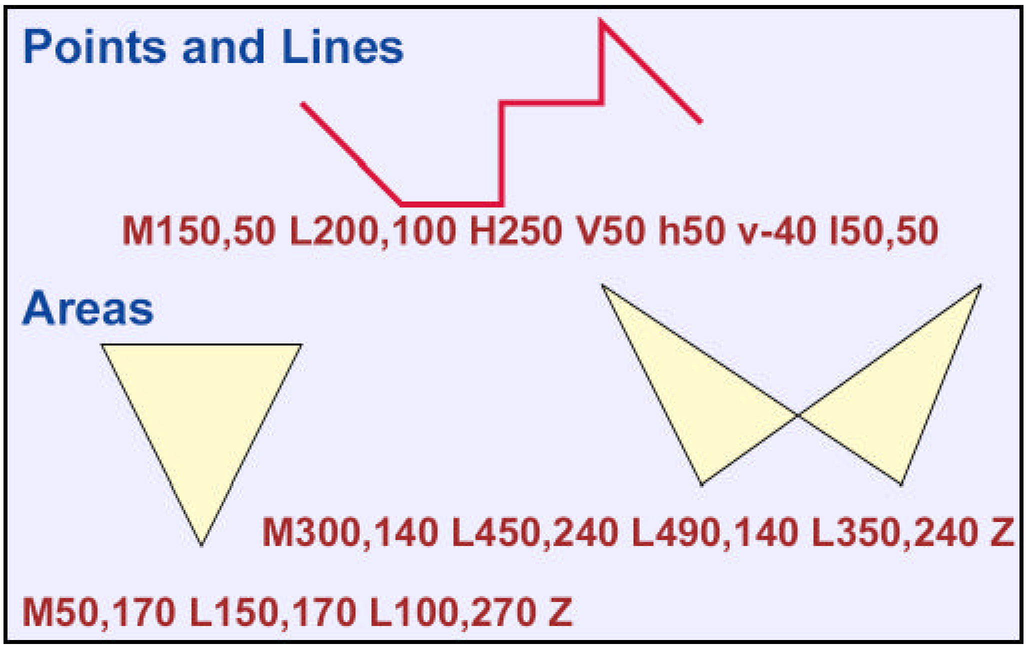

Points and Lines
M150,50 L200,100 H250 V50 h50 v-40 l50,50
Areas
M300,140 L450,240 L490,140 L350,240 Z
M50,170 L150,170 L100,270 Z

Figure 5.1: 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 5.2). The positioning of the control points change the shape of the curve under the user's control as can be seen in Figure 5.3. The coordinates used position the curves as they appear on the diagram.

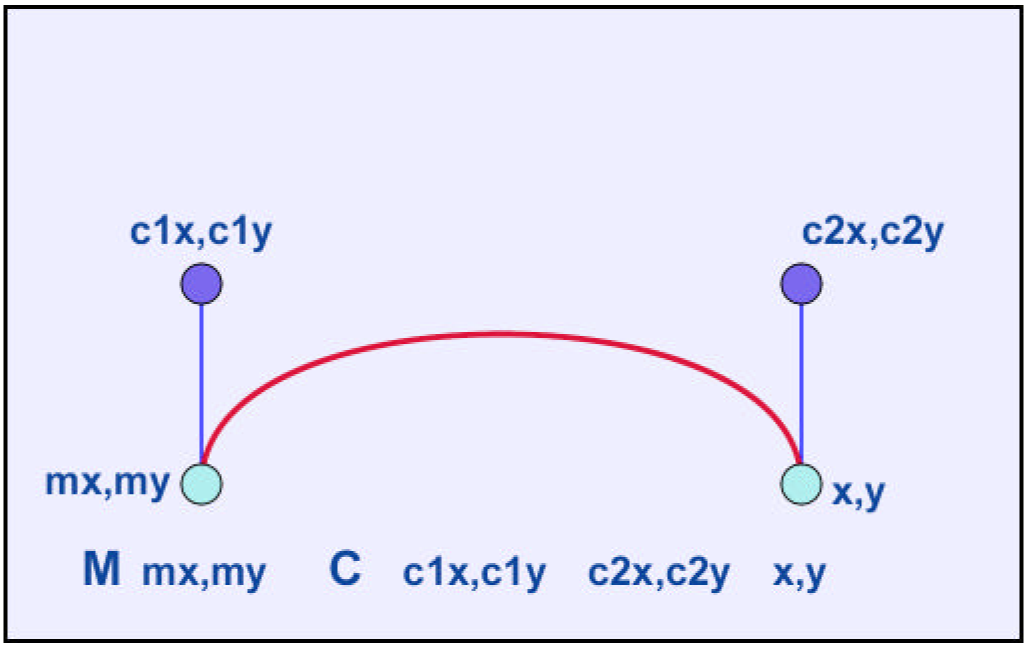


Figure 5.2: Path cubic bezier command

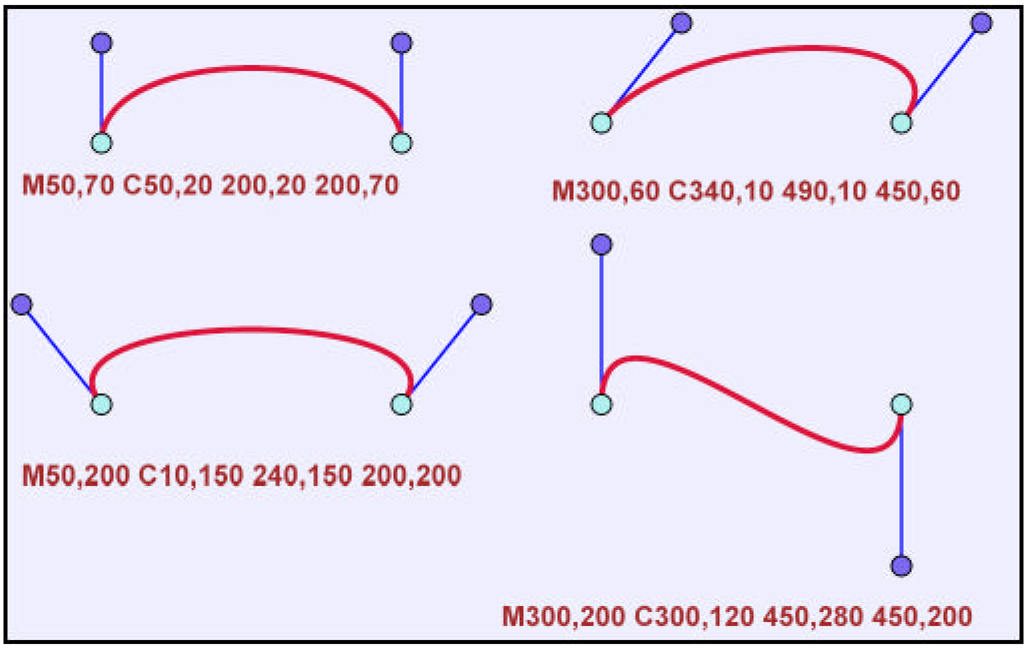

M50,70 C50,20 200,20 200,70
M300,60 C340,10 490,10 450,60
M50,200 C10,150 240,150 200,200
M300,200 C300,120 450,280 450,200

Figure 5.3: Path cubic bezier examples

A real world example is the creation of a duck as shown in Figure 5.4. 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"/>
```

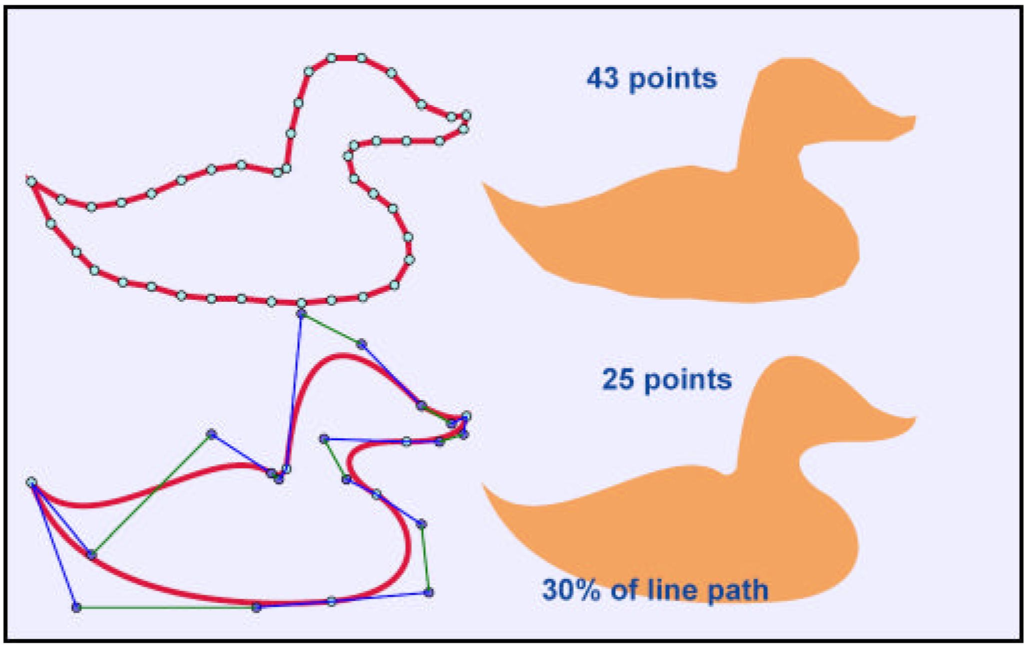


Figure 5.4: Path defined by lines and cubic beziers

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 end points are marked by circles and lines from the first point to the two control points and then the end point) and the duck without the 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"/>
```

SVG includes a number of semantic and syntactic measures to reduce the size of path expressions even further:

Using relative coordinates

Using relative instead of absolute coordinates can reduce the number of characters per coordinate significantly. Each command has a lower case equivalent which defines the coordinate values as relative to the current position.

Smooth curves

Often adjacent curves need to be joined smoothly, that is by sharing a tangential direction at the joint. This is achieved by defining the first control point of the second curve as the reflection of the second control point of the first curve. By defining separate commands for smooth joining curves (T and S for the quadratic and cubic splines) a number of coordinate pairs can be omitted.

Syntactic simplifications

White space can be omitted when the result is unambiguous. For example, no space is required between the command letter and the coordinate value; negative coordinates do not need a separation from the previous one; if the next command is the same as the previous one, the command letter can be omitted.

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"/>
```

SVG files can also be compressed using the gzip compression algorithms. The viewer decompresses the SVG picture on the fly. The SVG element definitions can be significantly compressed by this approach.

5.3 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 5.5 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

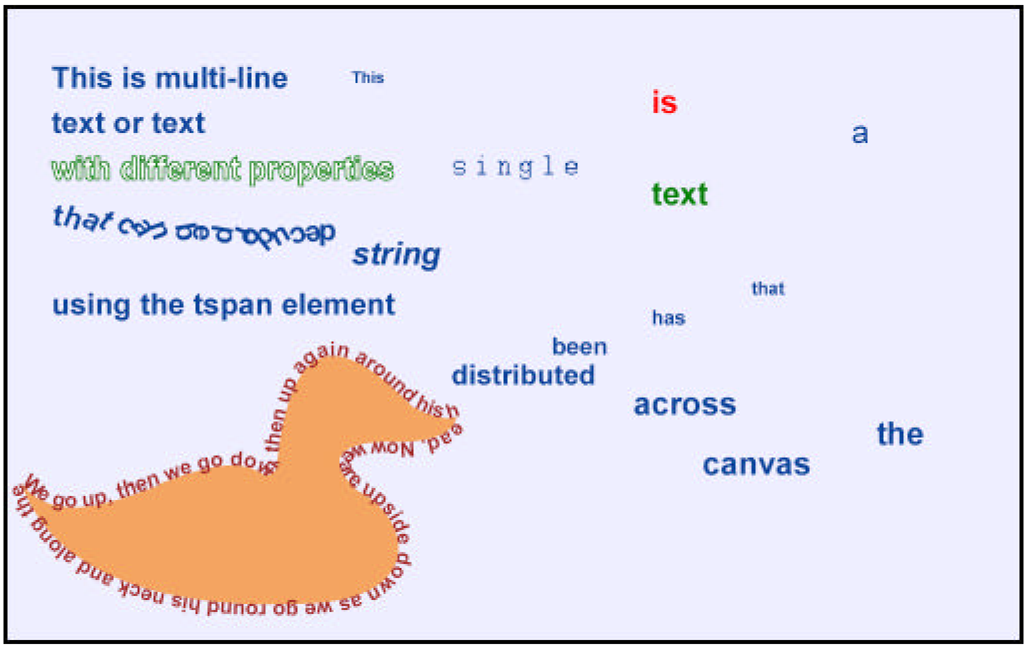

This is multi-line
text or text
with different properties
that can decorate
string
using the tspan element
This
single
is
a
text
that
has
been
distributed
across
the
canvas
We go up, then we go down, then up again around his head. Now we are upside down as we go round his neck and along the

Figure 5.5: Different text elements

```
<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="20" dy="30" rotate="10 20 30 40 0 50 60 70 0 80 90 0 100 110 120 140 150 160 170
180">
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>

<text>
<tspan x="30" dy="30" font-size="16">This </tspan>
<tspan x="330" dy="30" fill="red">is </tspan>
<tspan x="530" dy="30" font-weight="normal">a </tspan>
<tspan x="130" dy="30" font-family="Courier" font-size="28">single </tspan>
<tspan x="330" dy="30" fill="green">text </tspan>
<tspan x="30" dy="60" font-style="italic">string </tspan>
<tspan x="430" dy="30" font-size="18">that </tspan>
<tspan x="330" dy="30" font-size="20">has </tspan>
<tspan x="230" dy="30" font-size="24">been </tspan>
<tspan x="130" dy="30" font-size="28">distributed <</tspan>
<tspan dx="30" dy="30">across </tspan>
<tspan dx="130" dy="30">the </tspan>
<tspan dx="-230" dy="30">canvas</tspan>
</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 by using the **rotate** attribute. Again, a list of numbers can be provided to define the orientation of each character in the text sequence.

5.4 Basic Shapes

Path expressions are extremely versatile but can be unnecessarily powerful when defining simple shapes. As a consequence, SVG includes a number of basic shapes in its specification, such as rectangles (with optional rounded corners) circles, ellipses, single lines, simple polylines and polygons. These shapes are all equivalent to a particular path, and can be considered as simple shorthands.

As well as text, SVG also includes another drawing elements which cannot easily be derived from a path: image. Images can be included in an SVG drawing by using an external jpg, gif, or png image, in much the same way as it is done in HTML. Images can be positioned anywhere on the canvas and can also be transformed like any other geometric shape.

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 5.7) 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" />
```

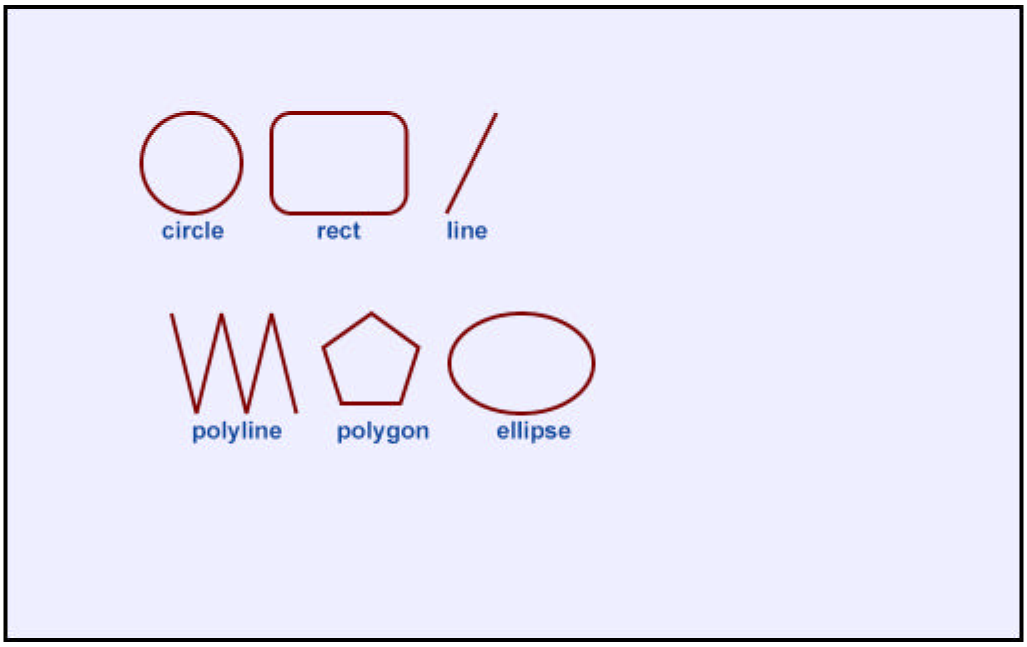


Figure 5.6: Basic elements

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

6. Grouping

6.1 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.

6.2 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 6.1 gives a montage of various transformations where the text defining the transformation is also transformed.

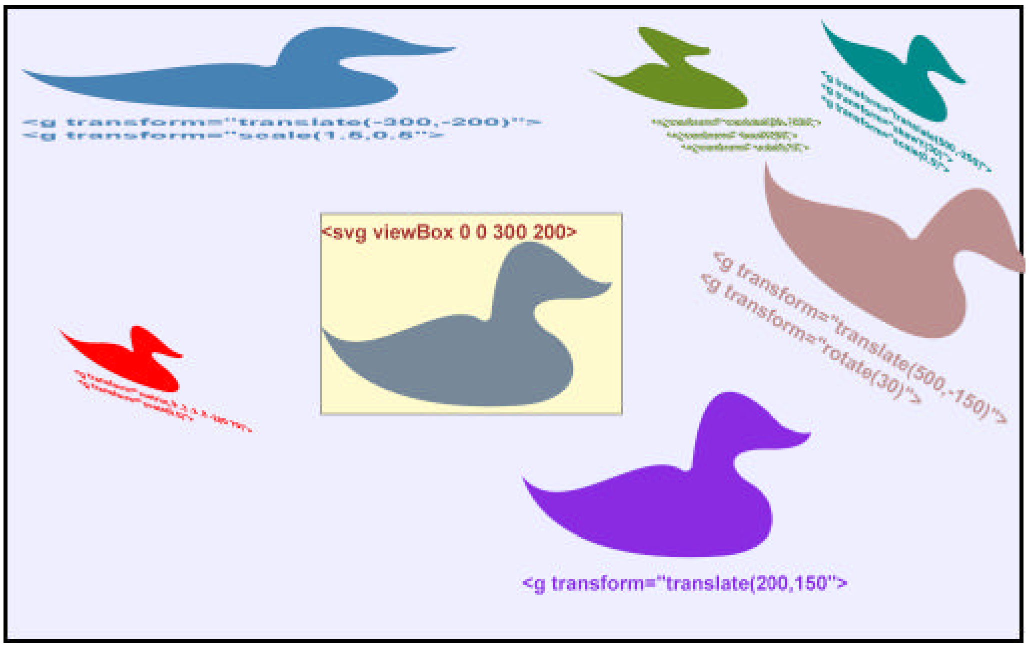


Figure 6.1: Transformations

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.

6.3 Clipping

A group of elements can be clipped against a clip path which is defined by a **clipPath** 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 6.2 shows the results of the three clipping paths defined above.

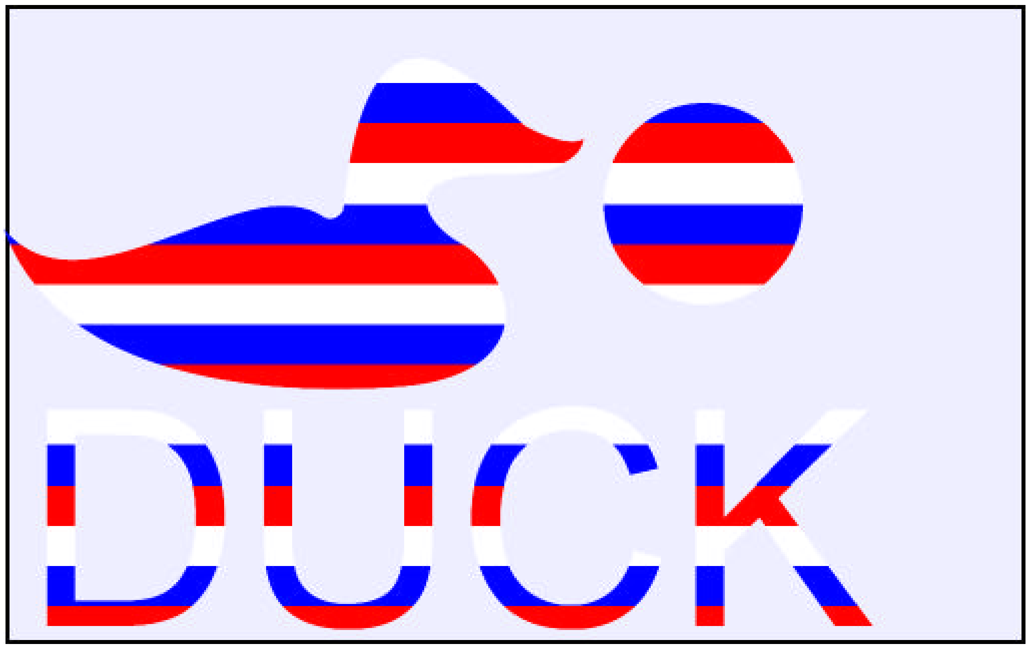


Figure 6.2: Clipping

7. Filling

7.1 Fill Properties

The geometry of an SVG element is largely controlled by the specific attributes associated with the element. The geometry can be transformed either on an element basis or as a group. It is also necessary to define the visual aspects of the drawing elements (colour, line styles, polygon fills, text size, etc).

SVG defines most of the aspects that are common in computer graphics:

- colour of the interior and the border of shapes both as RGB values and as 149 different colour names.
- opacity of the border and interior of shapes from opaque to transparent.
- clipping (any path or closed drawing element can serve as a clipping path).
- line width, style, join and end characteristics.
- fill rules (even-odd or non-zero).
- painting borders and areas using gradients or patterns.

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.

7.2 Colour

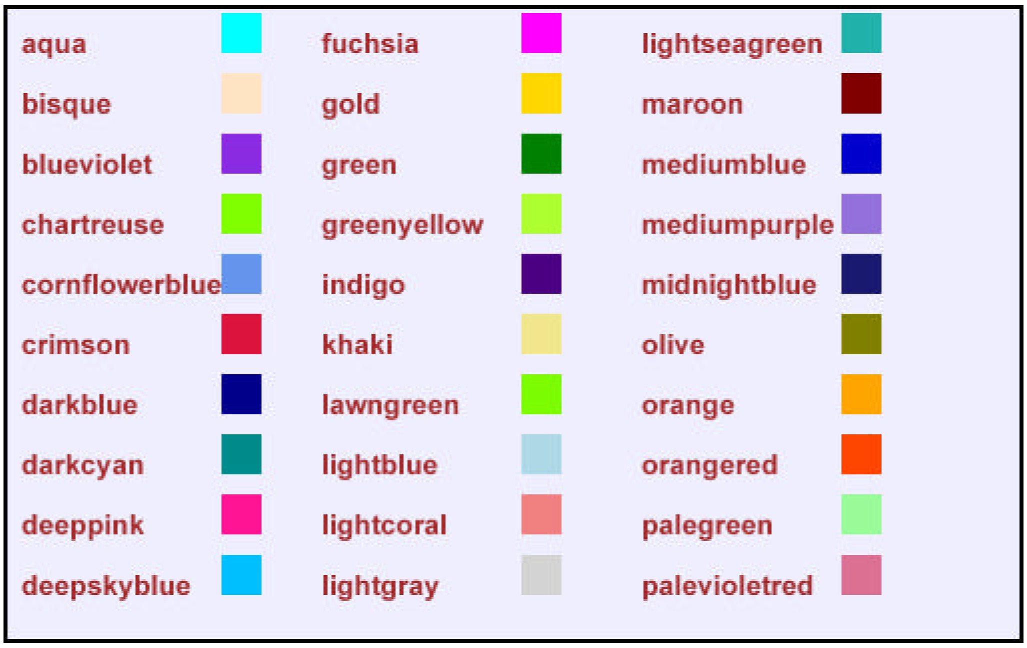
Colour values are used for various operations in SVG including filling and stroking. Colour can be defined in the same set of ways that it can be defined in CSS:

- 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 7.1 shows a sample of the colours available.



aqua	fuchsia	lightseagreen
bisque	gold	maroon
blueviolet	green	mediumblue
chartreuse	greenyellow	mediumpurple
cornflowerblue	indigo	midnightblue
crimson	khaki	olive
darkblue	lawngreen	orange
darkcyan	lightblue	orangered
deeppink	lightcoral	palegreen
deepskyblue	lightgray	palevioletred

Figure 7.1: Some SVG Colours

7.3 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 7.2 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.

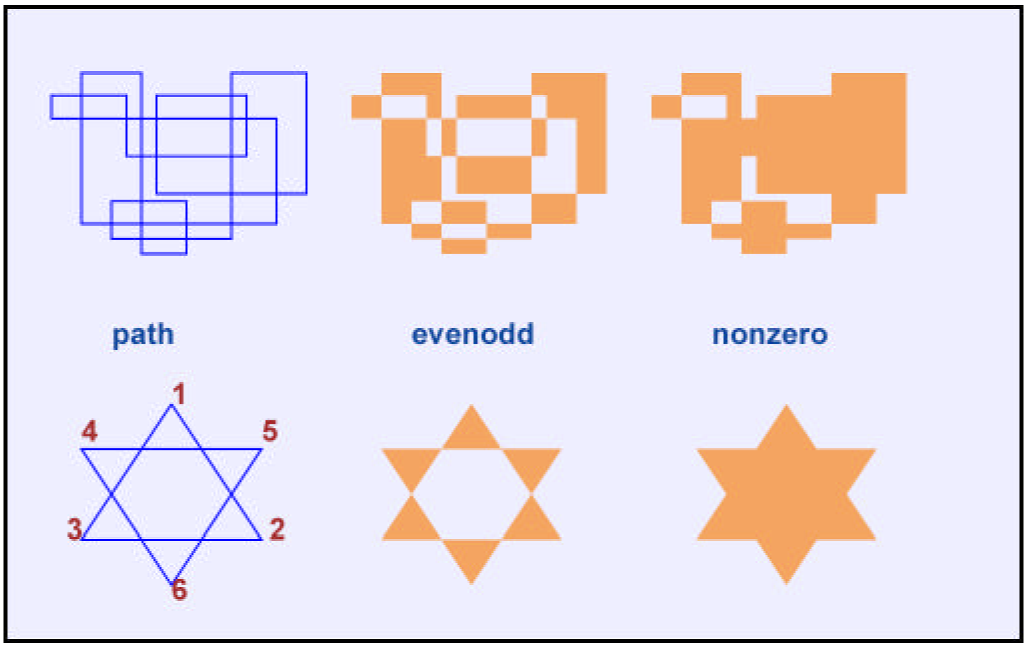


Figure 7.2: Fill Rules

7.4 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 7.3 shows various objects of different levels of transparency overlapping.

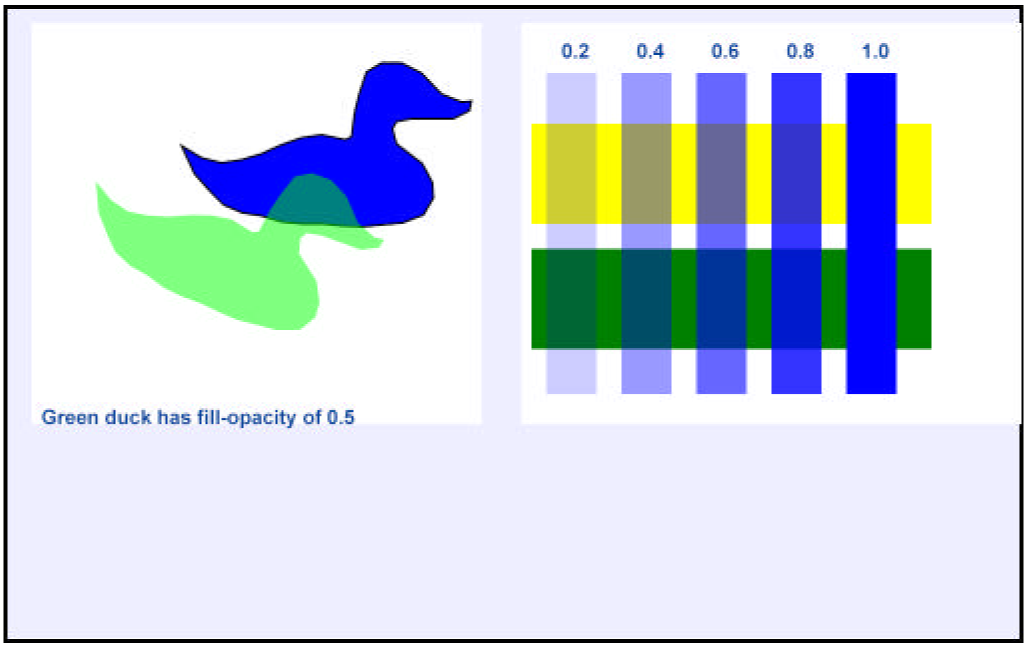


Figure 7.3: Opacity

7.5 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 7.4. 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.

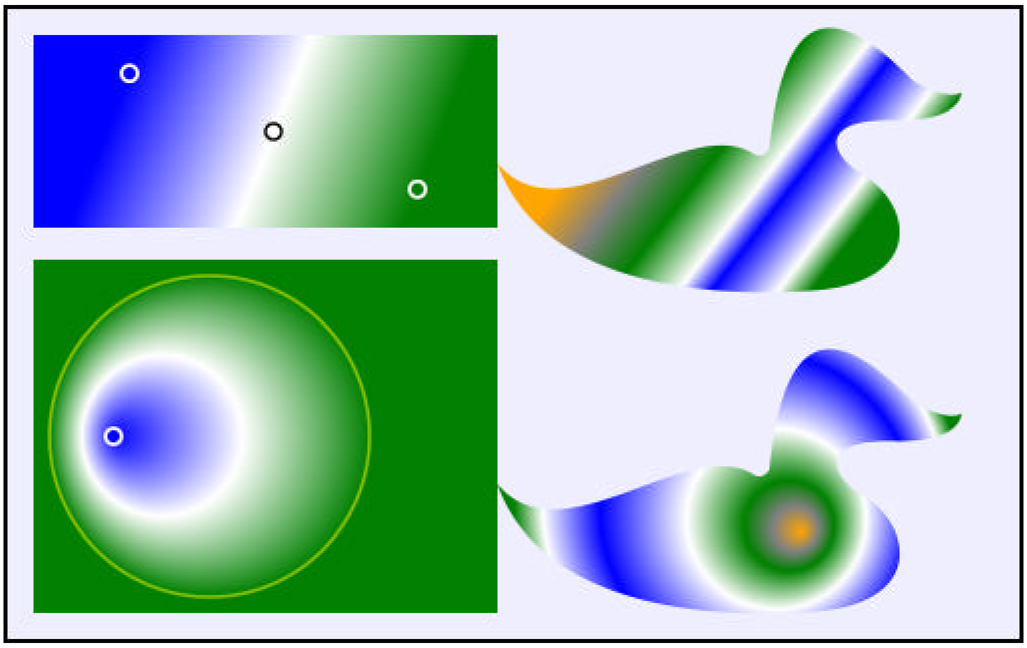


Figure 7.4: Colour Gradients

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

8. Stroking

8.1 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 8.1.

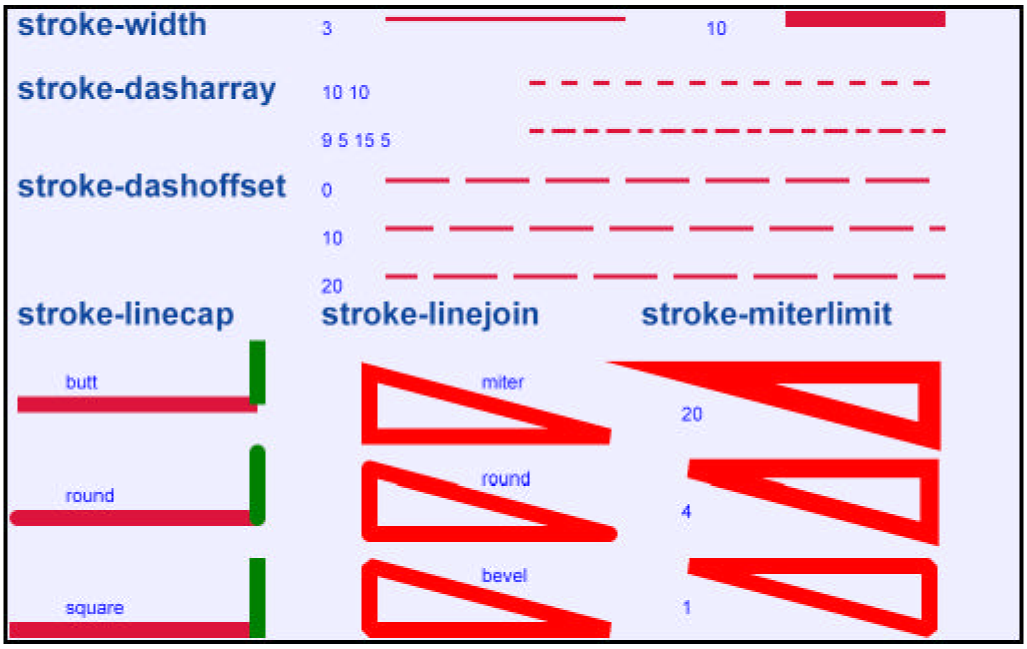


Figure 8.1: Stroke Properties

8.2 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 8.1 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.

8.3 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 8.1, 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 8.1). 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 8.1, the value of 4 bevels off the worst of the three joins while the value of 1 bevels all of the three joins.

9. Text

9.1 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.

9.2 Font Properties

Figure 9.1 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.

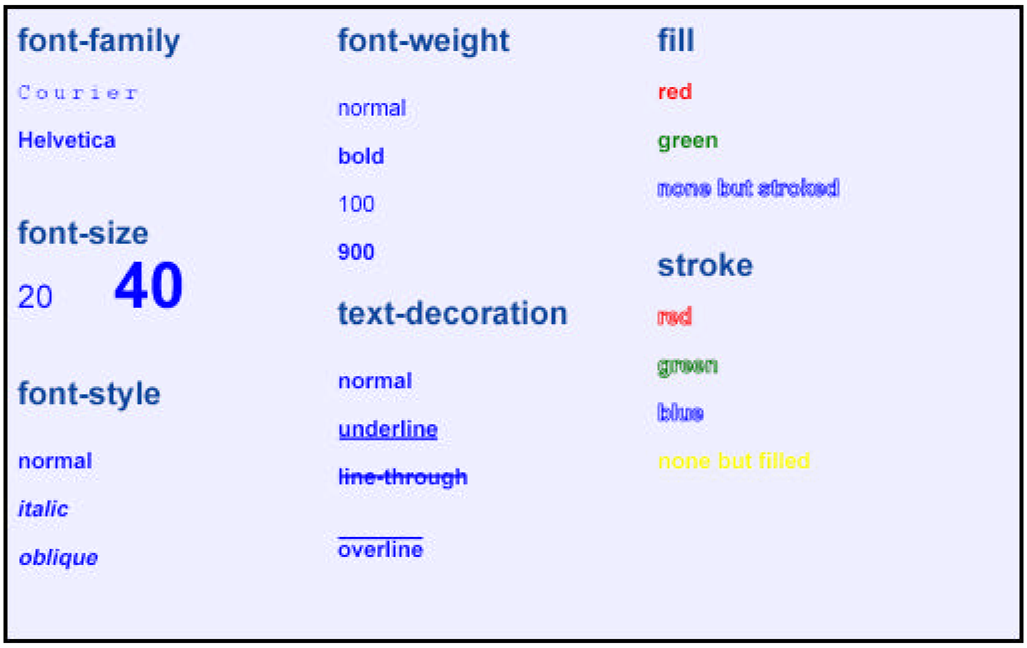


Figure 9.1: Font Properties

9.3 Text Properties

One of the most useful properties associated with the whole text string is the **text-anchor** property (Figure 9.2) 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 9.2 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**.

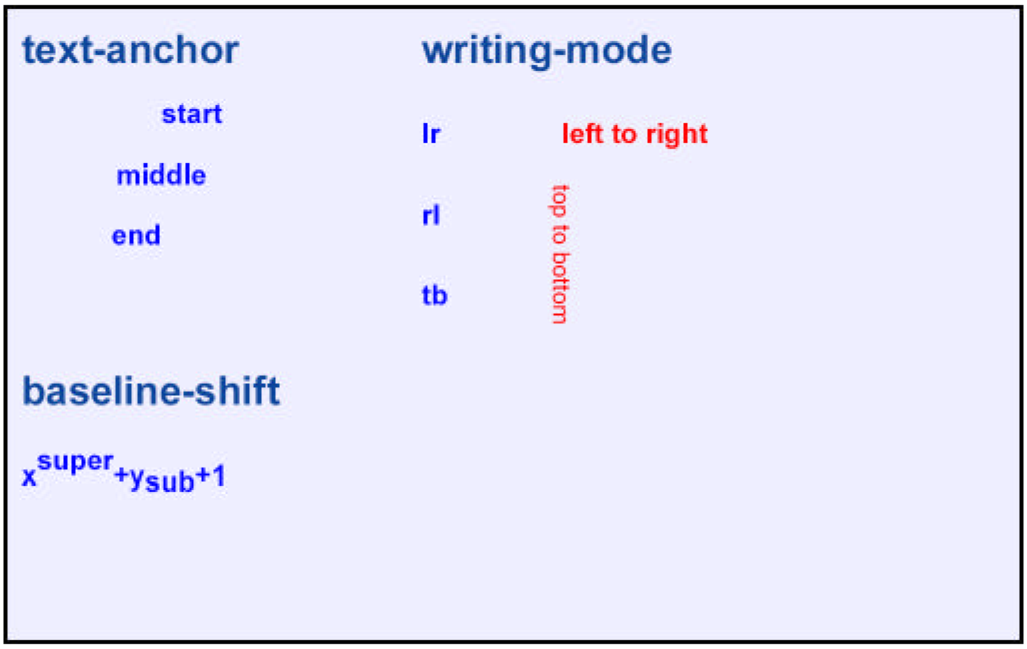


Figure 9.2: Text Properties

10. Animation

10.1 Simple Animation

The facilities of SVG described until now are comparable to a "classical" 2D graphics environment, such as GKS or PostScript. However, SVG also includes some so-called animation objects, which give a very different flavour to the system.

Animation in terms of SVG means the possibility to dynamically change of most of the attributes (both in XML in CSS) of SVG elements. For example, one might dynamically change the position, the geometry, various style elements (colour, linestyle, etc).

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>
```

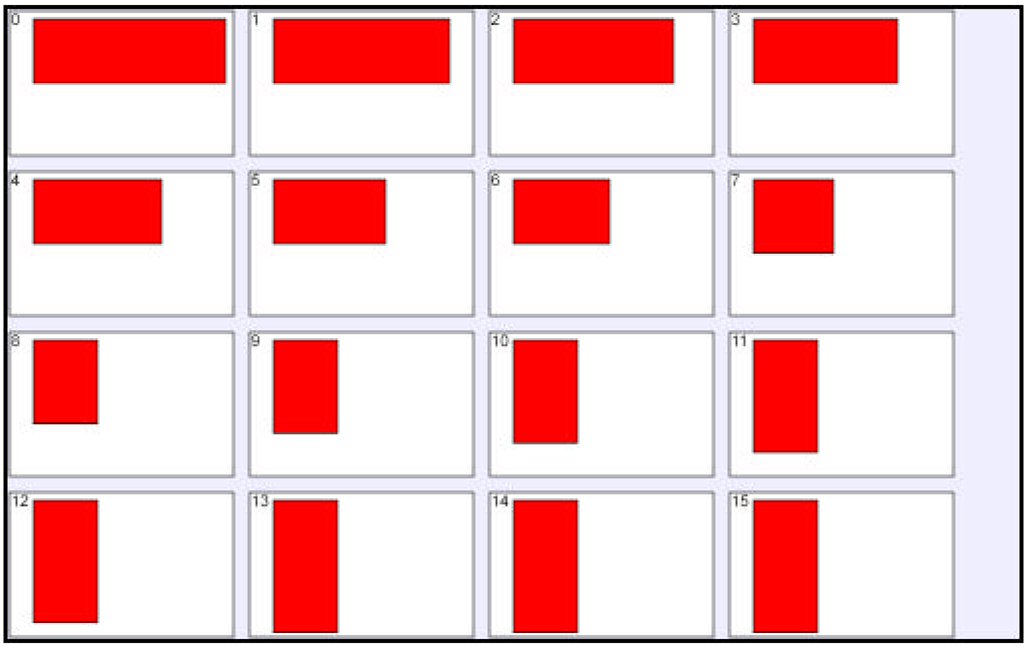


Figure 10.1: Simple Animation

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 10.1 where the rectangle is displayed at times 0 to 15 seconds.

It is worth noting that the animation concepts of SVG have not been defined by the SVG Working Group in isolation; instead, the functionality defined by another W3C document, called SMIL2.0, has been incorporated.

SMIL2.0 is concerned about multimedia synchronisation and presentation; it concentrates on specifying how various media presentation (audio, still or moving images, video, etc) can be presented. (In terms of SMIL2.0 an SVG image is but another type of media.) SMIL2.0 is a coordinating language in the sense that the various media objects are considered as black boxes for SMIL2.0, and the specification concentrates on the (two dimensional) layout, on the timing and synchronisation among media objects, on transition control, etc. SMIL2.0 also defines animation facilities (for example, moving objects around on the screen, changing their attributes); in terms of SMIL2.0, animation objects are yet another type of media objects, subject to the same timing and synchronisation constraints as other types of media objects. SVG took over the exact specification of the animation objects including their related timing; it is a nice example of interoperability among W3C recommendations.

When defining animation objects the author has to decide the following aspects:

1. **What** is animated?
2. **How** should the animation take place?
3. **When** should the animation take place and for how long?

Each of these issues is now considered separately.

As noted above, almost all attributes and CSS properties can be animated. There are five different types of animation objects in SVG:

animate
  to animate general attributes, as in the example above
set
  a shorthand notation for cases when attributes are 'discrete'(for example, visibility)
animateTransform
  an animation object tailored to the transform attribute
animateMotion
  to specify movement of an object along a specified path
animateColor
  an animation object tailored to colour changes

The semantics of the different animation objects, in terms of timing, animation control, etc, are identical; the differences reside solely in the way the target attributes are described.

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>
```

Note the **attributeType** attribute, which differentiates whether the target is XML (this is the default) or CSS.

To animate a property, the **attributeType** is given the value **CSS** and the CSS name is defined by the **attributeName** attribute. The results of this animation are shown in Figure 10.2.

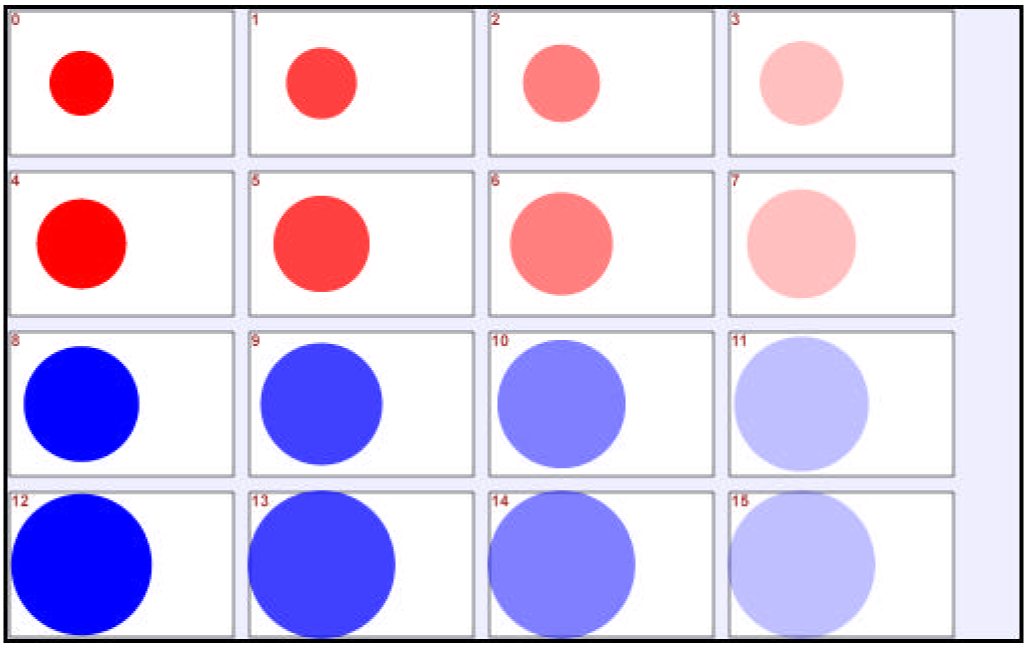


Figure 10.2: Animating Style properties

For values that do not have continuous ranges, these can be changed by the **set** element. The **set** element can be used as follows:

```
<circle cx="50" cy="50" r="20" style="fill:red; opacity:1">
<set attributeName="visibility" from="visible" to="invisible"/>
</circle>
```

There is no **dur** attribute: the effect of **set** is instantaneous. Note also that the values for **from** and **to** are not necessarily numerical, as shown in the example.

An example for the **animateTransform** element is:

```
<circle cx="50" cy="50" r="20">
<animateTransform attributeName="transform" type="translate" from="0,0" to="40,20"
dur="3s"/>
</circle>
```

which will move the centre of the circle from its initial position to the point (90,70) in 3 seconds.

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 10.3.

```
<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>
```

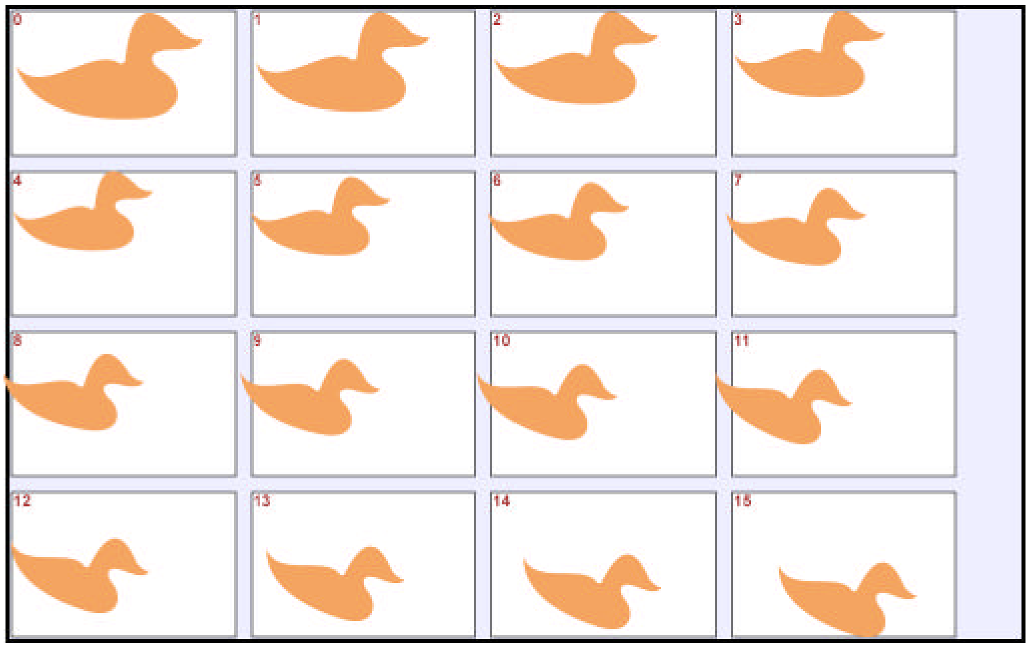


Figure 10.3: Animating Transforms

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.

10.2 How the Animation Takes Place

By default animation is linear. This means that a linear function is calculated between the **from** and the **to** values within the specific time duration. It is, however, possible to have finer control over the animation function through:

- specifying intermediate key time values
- replacing the linear animation functions by (cubic) splines

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.

The first possibility is shown in the following example:

```
<rect x="20" y="10" width="120" height="200">
<animate attributeName="width" begin="0s" fill="freeze"
values="120; 180; 190; 200" keyTimes="0; 2; 4; 8"/>
</rect>
```

This also specifies that the rectangle width should change in 8 seconds from 120 to 200, but in this case intermediate values that must be reached at 2 and 4 seconds (a very fast change at the beginning, but slowing down at the end) are also specified.

To achieve a somewhat similar (but much smoother) effect one can also define a spline function for the time change:

```
<rect x="20" y="10" width="120" height="200">
<animate attributeName="width" begin="0s" fill="freeze"
values="120; 200" keySplines="0 0.75 0.25 1"/>
</rect>
```

The spline used (defined in the [0,1] interval) is shown at the bottom left of Figure 6.3. The change of values is relatively fast at the beginning of the 8 second animation and slows down towards the end.

Finally, the two control methods can be combined: a separate key spline could be defined for each key time interval.

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 10.4, 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.

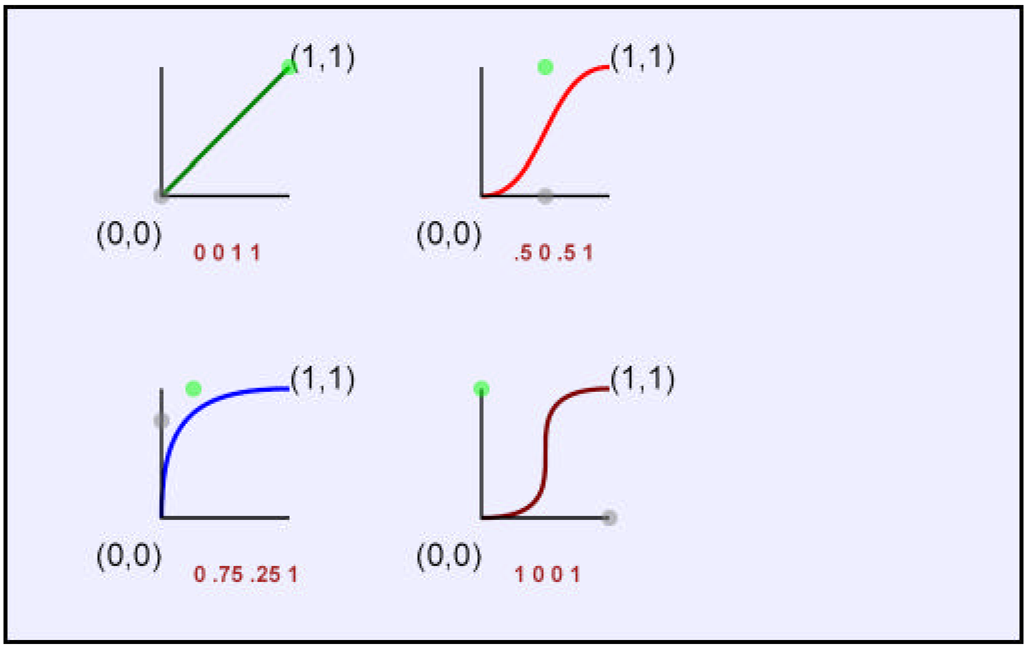


Figure 10.4: Spline Control

The animation of the circle defined above is shown in Figure 10.5 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.

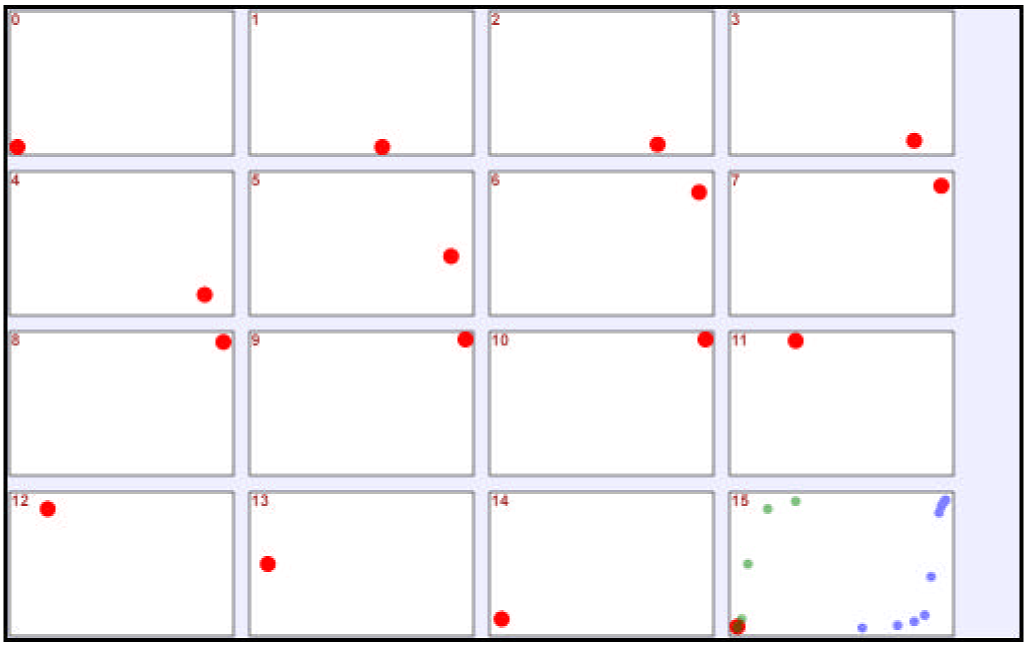


Figure 10.5: Animation Control

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 10.6.

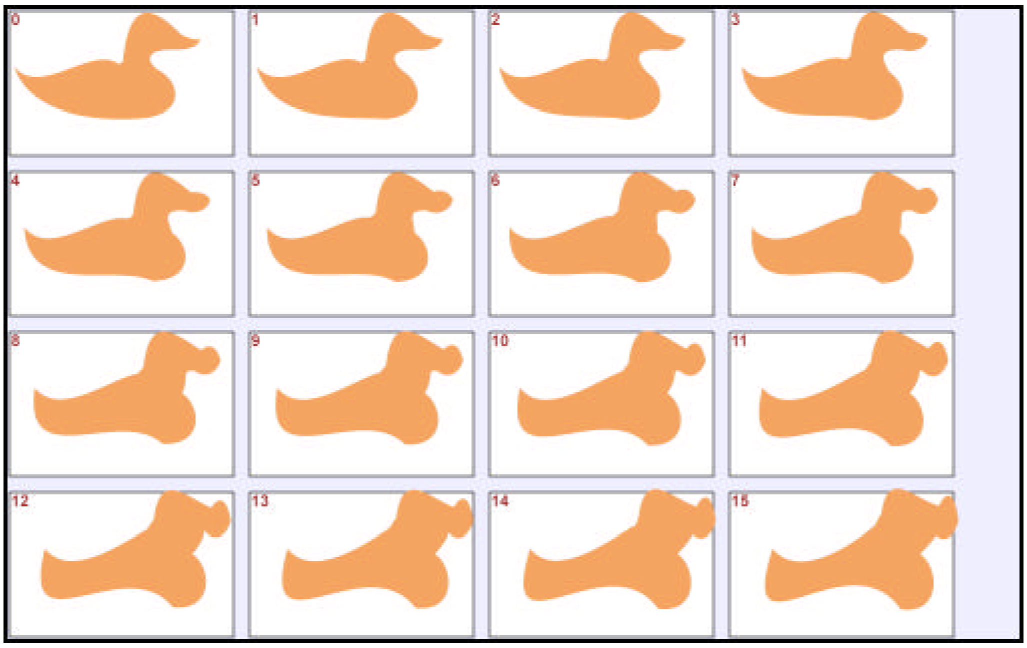


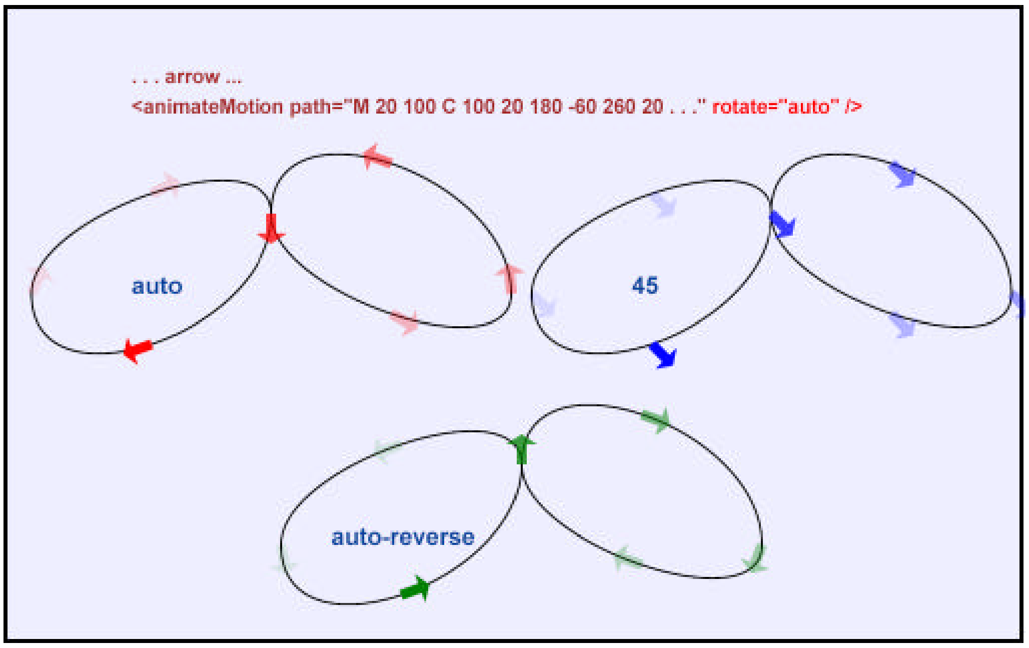
Figure 10.6: Animate path data

10.3 Animation along a Path

The animateMotion object can be used to move an object along a general path, rather than specifying a series of transformation objects. 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 10.7 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.



```
. . . arrow ...
<animateMotion path="M 20 100 C 100 20 180 -60 260 20 . . ." rotate="auto" />
```

Figure 10.7: Animate along a path

10.4 When the Animation Takes Place

In its simplest form, animation objects define a time range in which they operate. The attributes **begin**, **dur**, **end**, **repeatCount** and **repeatDuration** are the most important ones in this respect, and they all have a clear intuitive meaning. All these attributes can refer to time values in different formats (in practice, seconds and miliseconds are probably the most useful, but minutes or even hours can be used, too). Semantic conflicts can occur (for example, what happens if both duration and end times are defined and the values do not match?) and the SMIL2.0 document gives a very detailed account of these. However, in practice, authors use either one or the other. It is important to note that all the time count values are **relative to the load time of an object**, ie, `begin="3s"` means 3 seconds after the object has been loaded by the viewer.

Describing the animation solely on the basis of time is tedious and it also lacks the possibility for user interaction. However, the full specification of the **begin** (and **end**) attributes allows for **event based interaction**, too. These can be used for different purposes. Animation objects can be "chained", as in the following example:

```
<animate id="a1" begin="0s" ..../>
<animate id="a2" begin"a1.end" .... />
<animate id="a3" begin="a2.begin+8s" ... />
```

The second animation object would begin when the first one ends, while the third animation begins 8 seconds after the beginning of the second. Animation objects can be bound to user interactions:

```
<circle id="a1" ... />
...
<animate begin="a1.click" ..../>
```

Here the animation begins when the user clicks on the circle. While "click" is probably the most useful event type in this respect, one can also use "focusin", "focusout" (for example, when a text is selected).

The combination of the timing and the interaction facilities to control animation is quite powerful; it is possible to build up complex animated, and possibly interactive objects without using scripting (which is the "traditional" way of programming interaction on, for example, an HTML page). However, the animation objects also have their limitations: it is not possible to include complex calculations as part of attributes, nor can animation objects change over time. For this reason scripting is also included in the SVG specification, offering an alternative way of producing dynamic SVG images.

11. Linking and Templates

11.1 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.

11.2 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 11.1 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.

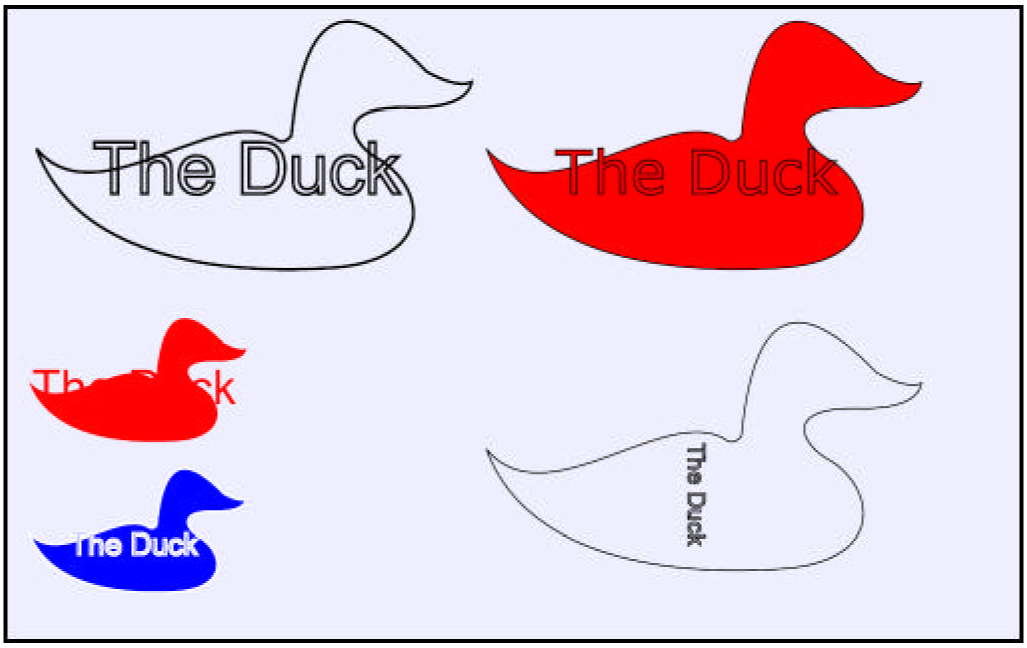


Figure 11.1: Symbols and their Use

11.3 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.

Note that a valid image type is SVG so that an SVG diagram can be embedded in another. The logo at the head of this document is an embedded SVG drawing.

11.4 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 11.2 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.

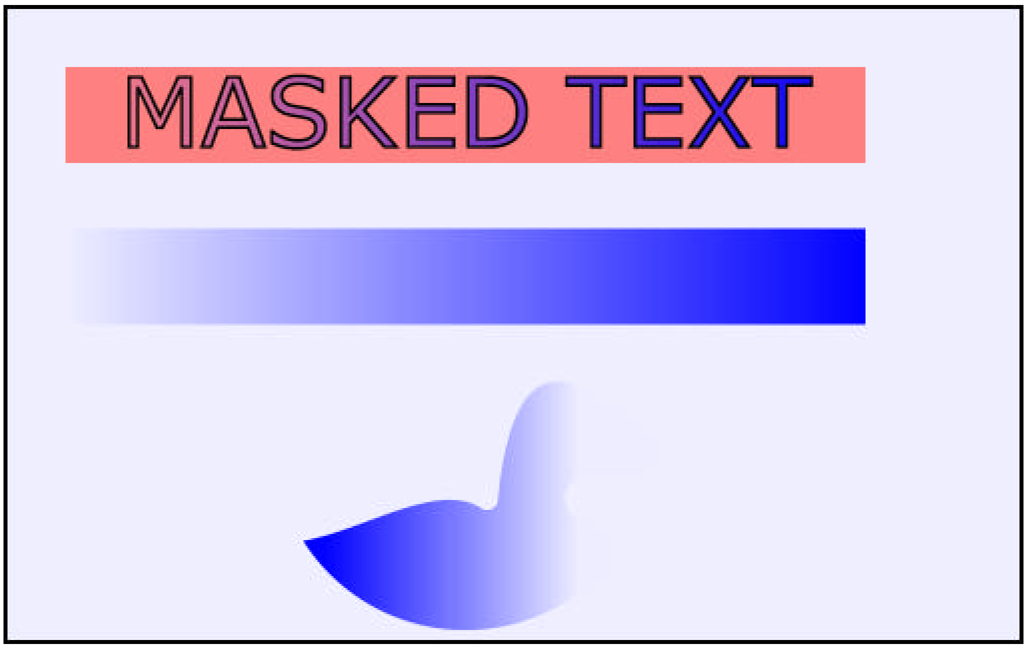


Figure 11.2: Masking

12. Interaction

12.1 Scripting and the DOM

When a Web document (be it HTML, XHTML, or some other XML language) is loaded into a browser, the document is parsed and internal structures to represent the document are created. The Document Object Model (DOM) is an API that can be used to manipulate these internal structures, as if the document were represented as a hierarchically structured collection of objects. The DOM does not mandate that the browser implement a document representation as a hierarchy of objects, though it might. Using a scripting language, the DOM permits the presentation of a Web document to be modified dynamically within a browser. This interface also permits content to be created dynamically within the browser. It should be noted that the copy of the Web document held on the server is not modified, it is purely the representation within the browser that is modified.

Modification to a document via the DOM is usually triggered by a browser event, the mouse being moved over a particular region of the document, a mouse click over a particular region, etc. The example below illustrates the main ideas.

```
<svg width="500" height = "500">
<script language="JavaScript"
type="text/javascript">
<![CDATA[
function onmouseover(evt) {
var elem = evt.target;
var style = elem.getStyle();
style.setProperty("fill-opacity", 0.5);
}
function onmouseout(evt) {
var elem = evt.target;
var style = elem.getStyle();
style.setProperty("fill-opacity", 1.0);
}
]]></script>
<rect x="20" y="20" width="250" height="250" style="fill:red; stroke:none"
onmouseover="onmouseover(evt)" onmouseout="onmouseout(evt)" />
<rect x="210" y="210" width="250" height="250" style="fill:green; stroke:none"
onmouseover="onmouseover(evt)" onmouseout="onmouseout(evt)" />
</svg>
```

The attributes **onmouseover** and **onmouseout** on the two rect specify the names of script functions to be invoked when the corresponding event occurs. The parameter passed to the function enables the corresponding element to be located in the document object tree.

The function onmouseover changes the **fill-opacity** property of the element to the value 0.5. This function is invoked when the mouse moves onto the region covered by the rectangle. The function onmouseout resets the **fill-opacity** property of the element to the value 1.0. This function is invoked when the mouse moves out of the region.

The functionality provided by CGM and SVG is very similar. In consequence, there is some work underway attempting to provide a common Document Object Model [25] for interacting with documents in both WebCGM and SVG formats. There are some minor differences that would not be accommodated by this common DOM. Disjoint polylines, pie slices, predefined dash styles are available with the WebCGM Profile but not available in SVG. SVG provides transformable symbols instantiated by the **use** element, bundled attribute styling using the **g** element, pattern and gradient fills. These facilities are available in CGM but not the WebCGM Profile.

12.2 Interaction Events

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

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

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.

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).

12.3 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 12.1. 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.

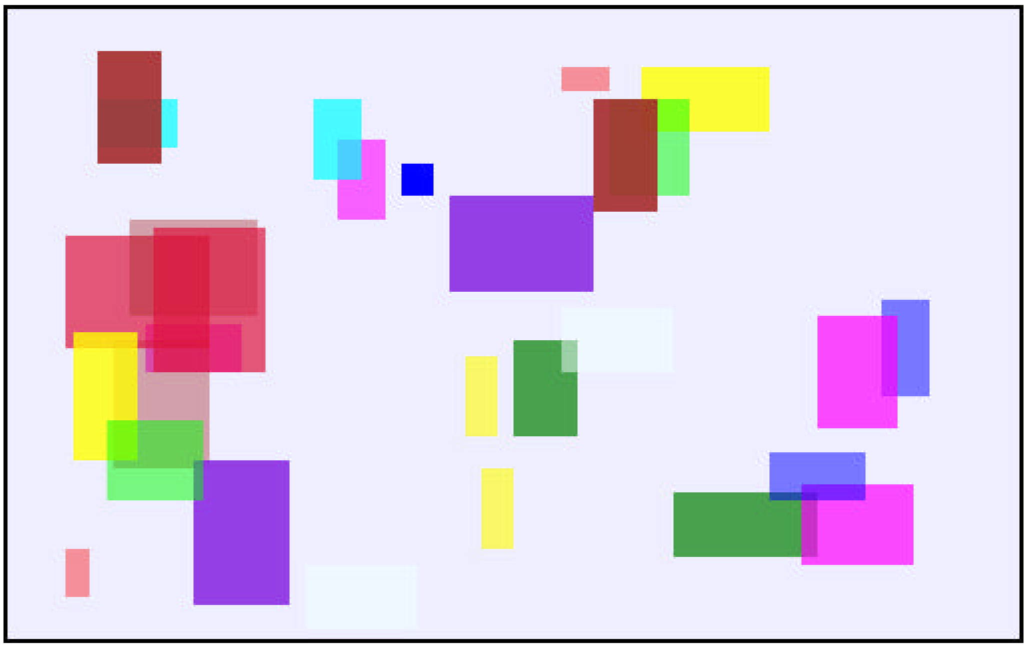


Figure 12.1: Cloning Rectangles

13 Filter Effects

13.1 Motivation

One important use for graphics on the Web is to include various types of artwork on the Web pages. These can be company logos, sophisticated advertisement images, computer arts, etc. In general, such images are produced by complex image processing tools resulting in images rich in shades and colours. SVG includes a number of tools, collectively called **filter primitives**, which can be used to achieve similar effects. The filters are obviously executed on the client side and that usually means that even very complex visual images can be described through relatively small files, rather than transmitting large pixel images. Also, the combination of filter effects with the more traditional graphics described in the earlier sections opens up rich possibilities for artwork creation.

As noted earlier, an SVG agent produces an image, conceptually, into a canvas. If no filter processing occurs, this image is then transmitted to the screen directly, using the painters' model. Filtering in SVG can occur between these two steps: the user can define filters which will operate on the output of the image generation on the canvas and it is the output of this filter which will, eventually, appear on the screen itself.

13.2 Filter Data Flow

The filtering operation follows a dataflow model used in other image processing environments such as Khoros [28] or the pbm toolkit widely used in Unix. A filter is a combination of filter primitives; each primitive can have one or more inputs and an output. The inputs to a primitive are either the source image (ie, produced by previous SVG operations) or the output of another primitive; the output of the last primitive is displayed on the screen. The RGB values of the source image or the alpha (opacity) values can be separated for the input to a primitive, for example a filter primitive might operate on the opacity values only.

Figure 13.1 shows the result of applying some filter operations to the duck.

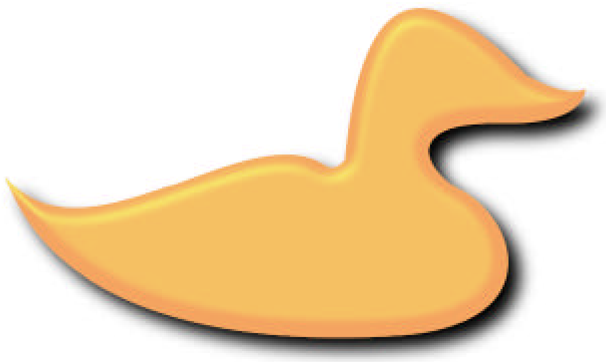


Figure 13.1: Filtered Duck

Figure 13.2 shows the dataflow network that creates the image.

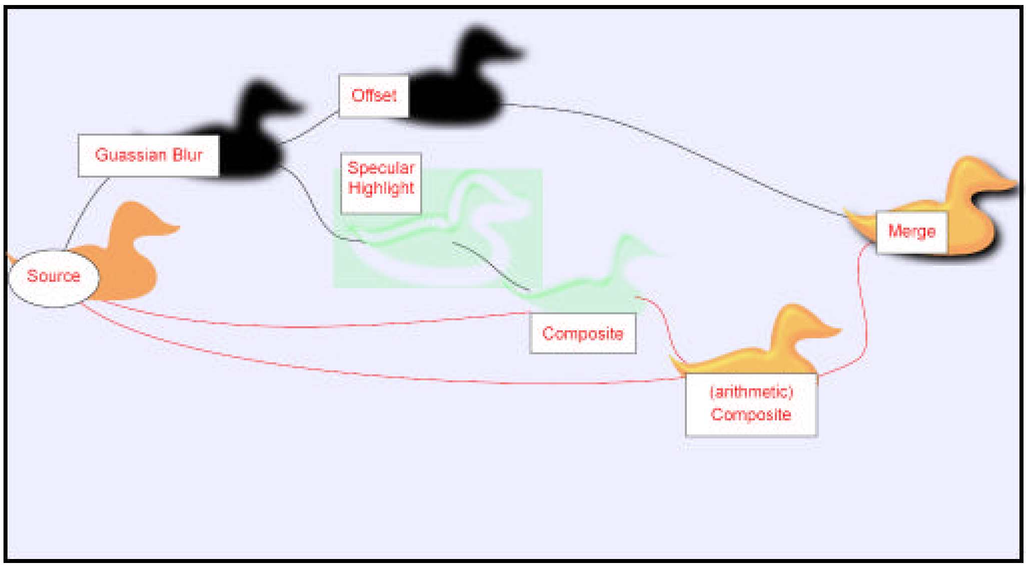


Figure 13.2: Data Flow of Filter Operations

13.3 Filter Primitives

An informal description of each filter primitive is:

1. A Gaussian Blur filter receives the opacity value of the source image and blurs it
2. An offset filter creates a slight offset in both the x and y direction of the blurred opacity value (this will be the dropped shadow of the final image)
3. The specular highlight will create a simulated highlight image on the blurred opacity image
4. The highlight image will be combined with the original opacity image to "cut off" the highlight so that the resulting image is no bigger than the original graphics
5. The original image is combined with the highlight to produce the original graphics with highlight (but without the shadow)
6. The shadow is combined with the previous step to produce the final image

The approximate SVG equivalent (trimming some of the details) is:

```
<defs>
<g transform="translate(10,-190)" id="theDuck" stroke="none" fill="sandybrown">
<path d="M 0 312 ... z"/>
</g>
<filter id="MyFilter">
<feGaussianBlur in="SourceAlpha" stdDeviation="4" result="blur"/>
<feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
<feSpecularLighting in="blur" surfaceScale="5" specularConstant="1"
specularExponent="10" lightColor="green" result="specOut">
<fePointLight x="-5000" y="-10000" z="20000"/>
</feSpecularLighting>
<feComposite in="specOut" in2="SourceAlpha" operator="in" result="specOut" />
<feComposite in="SourceGraphic" in2="specOut" operator="arithmetic"
k1="0" k2="1" k3="1" k4="0" result="litPaint"/>
<feMerge>
<feMergeNode in="offsetBlur"/>
<feMergeNode in="litPaint"/>
</feMerge>
</filter>
</defs>
<use xlink:href="#theDuck" filter="#MyFilter"/>
```

Given the complex visual effects of the result, the size of the SVG source is not particularly large compared to the original image of the duck.

The specification of each filter is quite complicated and relies on an intimate knowledge of various image processing techniques. However, it can be expected that authoring tools that can export SVG will hide the details from the casual user. Appendix A lists the filter primitives available in SVG with a short description of their functionalities.

14. Current State and the Future

14.1 Implementations

Current information concerning the state of browser plug-ins, stand-alone viewers, editors and converters is kept at the W3C SVG Working Group Home Page [29].

In October 2001, Adobe released the 3.0 version of their plug-in that works with most browsers (Opera, IE and Netscape at least) on Mac and PC platforms. Mozilla has a project for native support of SVG but the timescales keep slipping. Stand-alone viewers are available from IBM, Apache (Batik) and CSIRO . Toolkits are available from Apache and CSIRO. Editors are available from JASC (WebDraw), Mayura (Mayura Draw) and support for SVG is included in Adobe Illustrator and Corel Draw. This is not a definitive list but just gives an indication of the level of support. Many existing graphics design packages have added an SVG export or import function.

To help implementers debug their applications, W3C maintains an SVG Test Suite [30] of over 100 basic functionality tests for SVG. Each test comes with a PNG image that shows what the test should produce in terms of output.

14.2 Metadata

Searching for content is a major function on the Web. SVG attempts to make the textual content of a picture easy to find by allowing international text to appear as a sequence of text characters irrespective of writing direction. A search engine that is aware of the **tspan** element can search for multi-line text and text that may be scattered around the diagram.

SVG also provides a **metadata** element whose contents should be elements from other XML namespaces. In particular, the metadata could be expressed in RDF [31]. An ontology such as the Dublin Core [32] could be used to express information concerning the creator, the date it was produced etc. For CAD applications, the metadata might point into the application database to define the parts and materials to which the diagram refers.

SVG provides the mechanisms to allow rich metadata to be added to SVG diagrams. It will be interesting to see how this is used in practice.

14.3 Extensions to SVG

The W3C SVG Working Group does not believe that its job is done and early in 2001 they started collecting requirements for a new SVG 2.0.

SVG is seen as a basic platform which existing packages, such as Adobe Illustrator, can use as a Web delivery system. There has also been interest in extending SVG by defining richer languages that map down into SVG.

One example is CSVG [33] [34]. This is a constraint extension to SVG that defines a picture as a set of graphical objects. The properties of these objects including their position can be constrained by properties of other objects. For example, a rectangle can be specified to be above another rectangle, and a line can be constrained to connect the bottom of the top one to the top of the bottom one. To avoid over-constraining the system, each constraint can have its strength specified. Constraints are satisfied starting from the most important until a unique solution is found.

As XML applications become more common, there will be a need to define their inter-relationships more precisely. For example, MathML [7] is an XML application that renders mathematics correctly. The user may require mathematics to be included as text in an SVG diagram. SMIL [8] defines the layout and timing of a set of media objects. SVG is a specification that uses the SMIL animation facilities to create a media object. SVG allows media objects to be embedded within an SVG diagram. At some stage the relationships between these various specifications need to be more precisely defined.

SVG is a major advance in replacing images by vector graphics on the Web. However, it is just the first step and there is still a great deal of work to be done before the graphics environment on the Web reaches the maturity of existing computer graphics systems.

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 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	

Here is a brief description of the various filter primitives in SVG.

Blend
Pixel-wide combination of two images using various blending modes (normal, darken, lighten, multiply, screen).

Color Matrix
A general transformation of RGB and alpha values (using a 5x4 matrix). Some predefined matrices for, for example, hue rotation or saturation are also available.

Component Transfer
A component level remapping of the R, G, B, and alpha values. It allows operations like brightness or contrast adjustment, colour thresholding, etc. The transfer functions can be linear, table driven or gamma (for gamma correction).

Composition
Combination of two input images pixel-wise in image space using one of the [35] compositing operations or a simple arithmetic combination.

Convolution
A convolution matrix can be specified to perform convolution on the source image (used in edge detection, sharpening, etc)

Diffuse and Specular Lighting
The filters use the alpha channel as a bump map, ie, an imaginary surface is created in 3D with the alpha values. Lighting (using the usual Phong model) is then used to simulate lighting effects. Light sources can be distant, point, or spot.

Displacement maps
The pixel values of one image are displaced in function of the pixel values of another image.

Flood
Creation of a rectangle filled with a specific colour and opacity; to be used with other filter primitives.

Gaussian Blur
The effect is clear; the standard deviation of the blur can be controlled by special attributes.

Image
Refers to an external image which can then be used by other primitives as an input

Merge
Merge a number of input images.

Morphology
Performs "fattening" or "thinning" of an artwork.

Offset
Displacement of an image in x and y directions.

Tile
Fill a rectangle repeating an input image in a tiled pattern

Turbulence
It creates an image using the Perlin turbulence function [36], it allows the synthesis of various artificial textures.

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

[1] 'How the Web was Born'. James Gillies , Robert Cailliau. Oxford University Press, 2000.

[2] 'Weaving the Web'. Tim Berners-lee with Mark Fischetti. HarperCollins, 1999.

[3] 'Spinning the Web'. Andrew Ford. Thomson, 1995.

[4] 'http://www.w3.org/TR/REC-png: PNG (Portable Network Graphics) Specification'. World Wide Web Consortium, 1996.

[5] 'PNG, The Definitive Guide'. Greg Roelofs. O'Reilly, 1999.

[6] 'http://www.w3.org/TR/REC-xml : Extensible Markup Language (XML) 1.0 (Second Edition) '. World Wide Web Consortium, 2000.

[7] 'http://www.w3.org/TR/MathML2: Mathematical Markup Language (MathML) Version 2.0'. World Wide Web Consortium, 2001.

[8] 'http://www.w3.org/TR/smil20: Synchronized Multimedia Integration Language (SMIL 2.0) Specification'. World Wide Web Consortium, 2001.

[9] 'ISO/IEC 8632:1999: Information technology -- Computer graphics -- Metafile for the storage and transfer of picture description information

[10] 'The CGM Handbook'. Lofton Henderson, Anne Mumford. Academic Press, 1992.

[11] 'http://www.cgmopen.org'. CGM Open.

[12] 'http://www.w3.org/TR/REC-WebCGM: WebCGM Profile' World Wide Web Consortium, 1999.

[13] 'http://www.w3.org/TR/xlink/: XML Linking Language (XLink) Version 1.0'. World Wide Web Consortium, 2000.

[14] 'http://www.micrografx.com/resources/activeCGM.asp'. Micrografx ActiveCGM plug-in.

[15] 'http://www.sysdev.com/plugins/default.htm'. SDI CGM plug-in.

[16] 'http://www.tech-illustrator.com/'. Tech Illustrator WebCGM Hotspot Plug-in.

[17] 'http://www.w3.org/Submission/1998/05/: Web Schematics'. Submission to W3C, March 1998.

[18] 'http://www.w3.org/Submission/1998/06/: Precision Graphics Markup Language (PGML)'. Submission to W3C, April 1998.

[19] 'http://www.w3.org/Submission/1998/08/: Vector Markup Language (VML) '. Submission to W3C, May 1998.

[20] 'http://www.w3.org/Submission/1998/20/: DrawML'. Submission to W3C, December 1998

[21] 'http://www.w3.org/TR/SVG/ : Scalable Vector Graphics (SVG) 1.0 Specification'. World Wide Web Consortium, 2000.

[22] 'http://www.w3.org/TR/REC-CSS2: Cascading Style Sheets, level 2 (CSS2) Specification'. World Wide Web Consortium, 1998.

[23] 'http://www.w3.org/TR/REC-xml-names : Namespaces in XML'. World Wide Web Consortium, 1999.

[24] 'http://www.w3.org/TR/xslt: XSL Transformations (XSLT) Version 1.0' World Wide Web Consortium, 1999.

[25] 'http://www.w3.org/TR/DOM-Level-2-Core : Document Object Model (DOM) Level 2 Core Specification'. World Wide Web Consortium, 2000.

[26] 'ISO Standards for Computer Graphics - The First Generation'. D.B. Arnold, D.A. Duce. Butterworths, 1990

[27] 'GKS-94: An Overview'. K.W. Brodlie, L.B. Damnjanovic, D.A. Duce, F.R.A. Hopgood. IEEE Computer Graphics and Applications, pp. 64-71, 1995.

[28] 'http://www.khoral.com/'. Khoros.

[29] 'http://www.w3.org/Graphics/SVG/'. W3C SVG Overview.

[30] 'http://www.w3.org/Graphics/SVG/Test/'. SVG Test Suite.

[31] 'http://www.w3.org/TR/REC-rdf-syntax/: Resource Description Framework (RDF) Model and Syntax Specification', 1999.

[32] 'http://dublincore.org/'. Dublin Core.

[33] 'http://www.cs.washington.edu/homes/gjb/CSVG/'. CSVG Home Page

[34] 'A Constraint Extension to Scalable Vector Graphics'. G. J. Badros, J.J Tirtowidjojo, K.Marriott, B Meyer, W. Portnoy, A. Borning. Tenth International World Wide Web Conference, Hong Kong, may 2001, pp 489-498.

[35] 'Compositing Digital Images'. T. Porter and T. Duff. Computer Graphics, 1984. 18(3): p. 253-259.

[36] 'Texturing and Modeling, A Procedural Approach'. David Ebert, et al (Chapter by Ken Perlin). AP Professional, Cambridge, 1994.

[37] 'http://www.w3.org/Graphics/SVG/'. Overview of SVG Activity.

[38] 'http://www.w3.org/Graphics/SVG/Group/'. SVG Working Group Home Page.

[39] 'http://www.adobe.com/svg/'. Adobe SVG Plug-in plus tutorial information and demonstrations.

[40] 'http://sis.cmis.csiro.au/svg/'. CSIRO SVG Tool Kit. Can display SVGs and convert SVGs to JPEG.

[41] 'http://www.alphaworks.ibm.com/tech/svgview'. IBM SVG Viewer.

[42] 'http://xml.apache.org/batik/'. Apache's Batik which derives from Jackaroo.

[43] 'http://www.jasc.com/webdraw.asp'. Jasc WebDraw SVG Editor.

[44] 'http://www.mayura.com/'. Mayura Draw Editor.

[45] 'http://www.levien.com/svg/'. Gill: Gnome Illustration Application.

[46] 'http://www.digapp.com/newpages/svg2pdf.html'. Converts SVGs to PDF.

[47] 'http://www.padc.mmpc.is.tsukuba.ac.jp/member/morik/fdssvg/'. Converts bitmap images to SVG.

[48] 'http://broadway.cs.nott.ac.uk/projects/SVG/svgpl/'. .

[49] ''. SVG-PL, a Perl library for creating legal SVG documents.

[50] 'http://www.w3.org/Graphics/SVG/Test/'. SVG Conformance Test Suite.

[51] 'http://www.savagesoftware.com/products/svgtoolkit.html'. Savage Software SVG Toolkit.

[52] 'http://www.square1.nl/index.htm'. Graphics Connection Conversion Tools.

[53] 'http://www.celinea.com/'. CR2V, Raster to Vector Converter.

[54] 'http://www.graphicservlets.com/wmf2svg.htm'. WMF to SVG Converter.

[55] ''. .

[56] ''. .

[57] ''. .

[58] ''. .

[59] ''. .

[60] ''. .

[61] ''. .