Scalable Vector Graphics (SVG) Specification

W3C Working Draft *06 July 1999*

This version: http://www.w3.org/1999/07/06/WD-SVG-19990706/
Latest version: http://www.w3.org/TR/SVG
Previous version: http://www.w3.org/1999/06/25/WD-SVG-19990625/

Editor: Jon Ferraiolo <jferraio@adobe.com>

Authors: John Bowler, Microsoft Corporation <johnbo@microsoft.com>
Milt Capsimalis, Autodesk Inc. <milt@autodesk.com>
Richard Cohn, Adobe Systems Incorporated <cohn@adobe.com>
Andrew Donoho, IBM <awd@us.ibm.com>
David Duce, RAL (CCLRC) <dad@inf.rl.ac.uk>
Jerry Evans, Sun Microsystems <jerry.evans@Eng.sun.com>
Jon Ferraiolo, Adobe Systems Incorporated <jferraio@adobe.com>
Scott Furman, Netscape Communications Corporation <fur@netscape.com>
Peter Graffagnino, Apple <pgraff@apple.com>
Lofton Henderson, Inso Corporation <lofton@cgm.com>
Alan Hester, Xerox Corporation <Alan.Hester@usa.xerox.com>
Bob Hopgood, RAL (CCLRC) <frah@inf.rl.ac.uk>
Kelvin Lawrence, IBM <klawrenc@us.ibm.com>
Chris Lilley, W3C <chris@w3.org>
Philip Mansfield, Inso Corporation <philipm@paradigmdev.com>
Kevin McCluskey, Netscape Communications Corporation <kmcclusk@netscape.com>
Tuan Nguyen, Microsoft Corporation <tuann@microsoft.com>
Troy Sandal, Visio Corporation <TroyS@visio.com>
Peter Santangeli, Macromedia <psantangeli@macromedia.com>
Haroon Sheikh, Corel Corporation <haroons@corel.ca>
Gavriel State, Corel Corporation <gavriels@COREL.CA>
Robert Stevahn, Hewlett-Packard Company <rstevahn@boi.hp.com>
Shenxue Zhou, Quark <szhou@quark.com>

Abstract

This specification defines the features and syntax for Scalable Vector Graphics (SVG), a language for describing two-dimensional vector and mixed vector/raster graphics in XML.

Status of this document

This document is an intermediate public review draft version of the SVG specification.

The SVG working group has been using a staged approach. Initially, the working group developed an detailed set of SVG Requirements, which are listed in Appendix A: . These requirements were posted for public review initially in November 1998. For the most part, the specification has been developed to provide the feature set listed in the requirements document. Appendix A contains detailed editorial comments about which requirements have been addressed in this draft (along with hyperlinks to the relevant sections of the specification) and notes about which requirements have not been addressed yet and why.

The SVG working group has achieved significant progress toward translating the SVG requirements into an SVG specification. Much of the SVG language has been specified. A few major sections are still under development and many minor changes are still expected. There is still a need for considerable coordination work with other W3C working groups. Overall, it is likely that changes to the SVG specification will occur before a Proposed Recommendation is delivered by the working group.

Despite the preliminary nature of this draft specification, tools vendors and Web content creators are encouraged to experiment and develop preliminary versions of tools and Web sites according this draft specification, with the understanding that these tools and Web sites are experiemental/developmental in nature only and will need to be adapted to the final SVG Recommendation. The SVG working group is encouraged to see that several implementations of SVG are in progress, and encourages these implementations to track towards the present draft.

The main goal with this draft specification is to solicit public review and feedback. Public discussion of SVG features takes place on www-svg@w3.org, which is an automatically archived email list. Information on how to subscribe to public W3C email lists can be found at http://www.w3.org/Mail/Request. Review comments should be sent to www-svg@w3.org,

The home page for the W3C graphics activity is http://www.w3.org/Graphics/Activity.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Available formats

The SVG specification is available in the following formats. (In future versions, the specification's vector drawings will be available in both SVG and raster image formats. For now, only raster image formats are available.)

HTML 4.0:

http://www.w3.org/1999/07/06/WD-SVG-19990706/index.html

and a PDF file:

http://www.w3.org/1999/07/06/WD-SVG-19990706/WD-SVG-19990706.pdf.

Available languages

The English version of this specification is the only normative version. However, for translations in other languages see http://www.w3.org/Graphics/SVG/svg-updates/translations.html.

Quick Table of Contents

- 1 Introduction to SVG
- 2 SVG Concepts
- 3 Conformance Requirements and Recommendations
- 4 SVG Document Structure
- 5 SVG Rendering Model
- 6 Clipping, Masking and Compositing
- 7 CSS Properties, XML Attributes, Cascading, and Inheritance
- 8 Coordinate Systems, Transformations and Units
- 9 Filling, Stroking and Paint Servers
- 10 Color
- 11 Paths
- 12 Other Vector Graphic Shapes
- 13 Text
- 14 Images
- 15 Filter Effects
- 16 Interactivity
- 17 Animation
- 18 Backwards Compatibility, Descriptions and Titles
- 19 Embedding Foreign Object Types
- 20 Including Private Data
- 21 Extensibility
- 22 Metadata
- Appendix A. SVG Requirements
- Appendix B. Change History

- Appendix C. Document Type Definition
- Appendix D. SVG's Document Object Model (DOM)
- Appendix E. Sample SVG files
- Appendix F. Accessibility Support
- Appendix G. Minimizing SVG File Sizes
- Appendix H. Implementation and performance notes for fonts
- Appendix I. References

The following sections have not been written yet, but are expected to be be present in later versions of this specification:

- Appendix J. SVG Support for XML Fragments
- Appendix K. Internationalization Support
- Appendix L. Property and attribute index
- Appendix M. Index

Full Table of Contents

- 1 Introduction to SVG
  - 1.1 About SVG
  - 1.2 SVG MIME Type
  - 1.3 Compatibility with Other Standards Efforts
  - 1.4 Terminology
- 2 SVG Concepts
- 3 Conformance Requirements and Recommendations
  - 3.1 Introduction
  - 3.2 Conforming SVG Documents
  - 3.3 Conforming SVG Stand-Alone Files
  - 3.4 Conforming SVG Included Documents
  - 3.5 Conforming SVG Generators
  - 3.6 Conforming SVG Interpreters
  - 3.7 Conforming SVG Viewers
  - 3.8 General Implemenation Notes
    - 3.8.1 Forward and Undefined References
    - 3.8.2 Referenced objects are "pinned" to their own coordinate systems
- 4 SVG Document Structure
  - 4.1 Introduction

    - 4.2 Grouping and Naming Collections of Drawing Elements: the <g> Element
    - 4.3 Defining referenced and undrawn elements: the <defs> element
        - 4.3.1 The <style> sub-element to <defs>
        - 4.3.2 The <script> sub-element to <defs>
    - 4.4 The <symbol> element
    - 4.5 The <use> element
    - 4.6 The <image> element
- 5 SVG Rendering Model
    - 5.1 Introduction
        - 5.1.1 The painters model
        - 5.1.2 Rendering Order
        - 5.1.3 Grouping
        - 5.1.4 Specifying paint
        - 5.1.5 Specifying the painted region
        - 5.1.6 Use of external bitmap images
        - 5.1.7 Restricting painted regions
        - 5.1.8 Filtering painted regions
        - 5.1.9 Parent Compositing
    - 5.2 Rendering Properties
- 6 Clipping, Masking and Compositing
    - 6.1 Introduction
    - 6.2 Simple Alpha Blending/Compositing
    - 6.3 Clipping paths
    - 6.4 Masking
    - 6.5 Object And Group Opacity: the 'opacity' Property
- 7 CSS Properties, XML Attributes, Cascading, and Inheritance
    - 7.1 Introduction
    - 7.2 Cascading and Inheritance of XML Attributes and CSS Properties
    - 7.3 The Scope/Range of CSS Styles
- 8 Coordinate Systems, Transformations and Units
    - 8.1 Introduction
    - 8.2 Establishing the initial viewport
    - 8.3 Establishing A New User Space: Transformations
    - 8.4 Establishing an Initial User Coordinate System: the fit-box-to-viewport attribute

    - 8.5 Modifying the User Coordinate System: the transform attribute
    - 8.6 Establishing a New Viewport: the <svg> element within an SVG document
    - 8.7 Properties to Establish a New Viewport
    - 8.8 Units
    - 8.9 Redefining the meaning of CSS unit specifiers
    - 8.10 Further Examples
    - 8.11 Implementation Notes
- 9 Filling, Stroking and Paint Servers
    - 9.1 Introduction
    - 9.2 Fill Properties
    - 9.3 Stroke Properties
    - 9.4 Gradients
        - 9.4.1 Linear Gradients
        - 9.4.2 Radial Gradients
        - 9.4.3 Gradient Stops
    - 9.5 Patterns
    - 9.6 Inheritance of Painting Properties
- 10 Color
    - 10.1 Introduction
    - 10.2 The 'icc-profile' Property
- 11 Paths
    - 11.1 Introduction
    - 11.2 Path Data
        - 11.2.1 General information about path data
        - 11.2.2 The "moveto" commands
        - 11.2.3 The "closepath" command
        - 11.2.4 The "lineto" commands
        - 11.2.5 The curve commands
        - 11.2.6 The grammar for path data
    - 11.3 Markers
        - 11.3.1 The <marker> element
        - 11.3.2 Marker properties
        - 11.3.3 Details on How Markers are Rendered
    - 11.4 Implementation Notes

- 12 Other Vector Graphic Shapes
    - 12.1 Introduction
- 13 Text
    - 13.1 Introduction
    - 13.2 The <text> element
    - 13.3 White space handling
    - 13.4 Text selection
    - 13.5 Text and font properties
    - 13.6 Ligatures and alternate glyphs
    - 13.7 Text on a path
- 14 Images
- 15 Filter Effects
    - 15.1 Introduction
    - 15.2 Background
    - 15.3 Basic Model
    - 15.4 Defining and Invoking a Filter Effect
    - 15.5 Filter Effects Region
    - 15.6 Common Attributes
    - 15.7 Filter Processing Nodes
- 16 Interactivity
    - 16.1 Links: the <a> element
    - 16.2 Event Handling
    - 16.3 Zoom and pan control
- 17 Animation
    - 17.1 Introduction
    - 17.2 Animation Example Using the SVG DOM
- 18 Backwards Compatibility, Descriptions and Titles
    - 18.1 Introduction
    - 18.2 Backwards Compatibility
    - 18.3 The <description> and <title> elements
- 19 Embedding Foreign Object Types
- 20 Including Private Data
    - 20.1 Introduction
- 21 Extensibility

- 22 Metadata
  - 22.1 Introduction
  - 22.2 The SVG Metadata Schema
  - 22.3 An Example
- Appendix A. SVG Requirements
- Appendix B. Change History
- Appendix C. Document Type Definition
- Appendix D. SVG's Document Object Model (DOM)
- Appendix E. Sample SVG files
- Appendix F. Accessibility Support
- Appendix G. Minimizing SVG File Sizes
- Appendix H. Implementation and performance notes for fonts
- Appendix I. References
  - I.1 Normative references
  - I.2 Informative references

The following sections have not been written yet, but are expected to be be present in later versions of this specification:

- Appendix J. SVG Support for XML Fragments
- Appendix K. Internationalization Support
- Appendix L. Property and attribute index
- Appendix M. Index

1 Introduction to SVG

1.1 About SVG

This specification defines the features and syntax for Scalable Vector Graphics (SVG).

SVG is a language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects. The feature set includes nested transformations, clipping paths, alpha masks, filter effects, template objects and extensibility.

SVG drawings can be dynamic and interactive. The Document Object Model (DOM) for SVG allows for straightforward and efficient vector graphics animation via scripting. A rich set of event handlers such as onmouseover and onclick can be assigned to any SVG graphical object. Because of its compatibility and leveraging of other Web standards, features like scripting can be done on HTML and SVG elements simultaneously within the same Web page.

1.2 SVG MIME Type

The MIME type for SVG will be "`image/svg`". The W3C will register this MIME type around the time which SVG is approved as a W3C Recommendation.

1.3 Compatibility with Other Standards Efforts

SVG leverages and integrates with other W3C specifications and standards efforts. By leveraging and conforming to other standards, SVG becomes more powerful and makes it easier for users to learn how to incorporate SVG into their Web sites.

Here are some of the ways which SVG fits in and conforms to other standards:

- SVG is an application of the XML 1.0 Recommendation
- SVG conforms to the XML Namespace Recommendation
- SVG is tracking and will conform with XLink and XPointer (once these specifications become Recommendations)
- SVG conforms to the Cascading Style Sheets (CSS) level 2 Recommendation
- SVG conforms to the Document Object Model (DOM) level 1 Recommendation and is tracking the DOM level 2

- SVG utilises the switch and test concepts from the SMIL 1.0 Recommendation.
- SVG attempts to fit in with the HTML version 4 Recommendation, and is meant to work as a component grammar with future versions of HTML which are expressed in XML as a set of component XML grammars

Because SVG conforms to DOM, it will be scriptable just like HTML version 4 (sometimes called DHTML). This will allow a single scripting approach to be used simultaneously for both XML documents and SVG graphics. Thus, interactive and dynamic effects will be possible on multiple XML namespaces using the same set of scripts.

1.4 Terminology

Not yet written.

2 SVG Concepts

Not yet written.

3 Conformance Requirements and Recommendations

3.1 Introduction

Different sets of SVG conformance requirements exist for:

- Conforming SVG Documents
- Conforming SVG Stand-Alone Files
- Conforming SVG Included Documents
- Conforming SVG Generators
- Conforming SVG Interpreters
- Conforming SVG Viewers

3.2 Conforming SVG Documents

An SVG document is a *Conforming SVG Document* if it adheres to the specification described in this document (Scalable Vector Graphics (SVG) Specification) and also:

- is a well-formed XML document
- if all non-SVG namespace elements are removed from the given document, is a valid XML document
- conforms to the following W3C Recommendations:
  - the XML 1.0 specification (Extensible Markup Language (XML) 1.0)
  - (if any namespaces other than SVG are used in the document) Namespaces in XML
  - any use of CSS styles and properties needs to conform to Cascading Style Sheets, level 2 CSS2 Specification
  - any references to external style sheets should conform to Associating stylesheets with XML documents

3.3 Conforming SVG Stand-Alone Files

A file is a *Conforming SVG Stand-Alone File* if:

- it is a Conforming SVG Document
- its outermost XML element is an `svg` element

3.4 Conforming SVG Included Documents

SVG document can be included within parent XML documents using the XML namespace facilities described in Namespaces in XML.

An SVG document that is included within a parent XML document is a *Conforming Included SVG Document* if the SVG document, when taken out of the parent XML document, conforms to the SVG Document Type Definitions (DTD).

In particular, note that individual elements from the SVG namespace *cannot* be used by themselves. Thus, the SVG part of the following document is *not* conforming:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SomeParentXMLGrammar PUBLIC "-//SomeParent"
  "http://SomeParentXMLGrammar.dtd">
<ParentXML>
  <!-- Elements from ParentXML go here -->

  <!-- The following is not conforming -->
  <z:rect xmlns:z="http://www.w3.org/Graphics/SVG/svg-19990706.dtd"
            x="0" y="0" width="10" height="10" />

  <!-- More elements from ParentXML go here -->
</ParentXML>
```

Instead, for the SVG part to become a Conforming Included SVG Document, the file could be modified as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SomeParentXMLGrammar PUBLIC "-//SomeParent"
  "http://SomeParentXMLGrammar.dtd">
<ParentXML>
  <!-- Elements from ParentXML go here -->

  <!-- The following is conforming -->
  <z:svg xmlns:z="http://www.w3.org/Graphics/SVG/svg-19990706.dtd"
         width="100px" height="100px" >
    <z:rect x="0" y="0" width="10" height="10" />
  </z:svg>

  <!-- More elements from ParentXML go here -->
</ParentXML>
```

3.5 Conforming SVG Generators

A *Conforming SVG Generator* is a program which:

- always creates at least one of Conforming SVG Documents, Conforming SVG Stand-Alone Files or Conforming SVG Included Documents
- does not create non-conforming SVG documents of any of the above types
- conforms to the SVG accessibility guidelines

SVG generators are encouraged to follow W3C developments in the area of internationalization. Of particular interest is the *W3C Character Model* and the concept of *Webwide Early Uniform*

Normalization, which promises to enhance the interchangability of Unicode character data across users and applications. Future versions of the SVG Specification are likely to require support of the *W3C Character Model* in Conforming SVG Generators.

3.6 Conforming SVG Interpreters

An SVG interpreter is a program which can parse and process SVG documents. A *Conforming SVG Interpreters* must be able to:

- Successfully parse and process any Conforming SVG Document, (It is not required, however, that every possible SVG feature be supported beyond parsing. Thus, for example, a Conforming SVG Interpreter might bypass the processing of selected SVG elements.)
- If the program allows scripts to run against the SVG document's Document Object Model, then a Conforming SVG Interpreter must support the entire DOM model for SVG defined in this specification
- The XML parser must be able to handle arbitrarily long data streams.

3.7 Conforming SVG Viewers

An SVG viewer is a program which can parse and process an SVG document and render the contents of the document onto some sort of output medium such as a display or printer. Usually, an *SVG Viewer* is also an *SVG Interpreter*.

An SVG Viewer is a *Conforming SVG Viewer* if:

- in the typical case where the SVG Viewer is also an SVG Interpreter, then the program must also be a Conforming SVG Interpreter,
- all SVG features described in this specification (including all graphic elements, attributes and properties) must be supported and rendered.
- if display devices are supported, facilities must exist for zooming and panning of standalone SVG documents or SVG documents embedded within parent XML documents
- if printing devices are supported, SVG documents must be printable at printer resolutions with the same graphics features available as required for display (e.g., color must print correctly on color printers)
- the viewer should receive enough information from the parent environment to determine the device resolution. (In situations where this information is impossible to determine, the parent environment should pass a reasonable avalue for device resolution which tends to approximate most common target devices.)
- in web browser environments, the ability to search and select text strings within SVG documents
- if display devices are supported, ability to select and copy text from an SVG document to the system clipboard
- complete support for an ECMAScript binding of the SVG Document Object Model
- support for JPEG and PNG image formats
- support alpha channel blending of the SVG document image onto the target canvas
- supports the following W3C Recommendations with regard to SVG documents:

- ❍ complete support for the XML 1.0 specification (Extensible Markup Language (XML) 1.0)
- ❍ complete support for inclusion of non-SVG namespaces within an SVG document Namespaces in XML (Note that data from non-SVG namespaces can be ignored.)
- ❍ complete support for all features from CSS2 (Cascading Style Sheets, level 2 CSS2 Specification) that are described in this specification as applying to SVG
- ❍ complete support for external style sheets as described in Associating stylesheets with XML documents

Although anti-aliasing support isn't a strict requirement for a Conforming SVG Viewer, it is highly recommended. Lack of anti-aliasing support will generally result in poor results on display devices.

A higher class concept is that of a *Conforming High-Quality SVG Viewer* which must support the following additional features:

- Generally, professional-quality results with good processing and rendering performance and smooth, flicker-free animations
- Support for anti-aliasing of strokes and text
- Progressive rendering and animation effects (i.e., the start of the document will start appearing and animations will start running in parallel with downloading the rest of the document)
- Restricted screen updates (i.e., only required areas of the display are updated in response to redraw events)
- Background downloading of images and fonts retrieved from a web server, with updating of the display once the downloads are complete
- Color management via ICC profile support (i.e., the ability to support colors defined using ICC profiles)
- Resampling of image data using algorithms at least as good as bicubic resampling methods

3.8 General Implementation Notes

The following are implementation notes that correspond to features which span multiple sections of the SVG specifications. (Note that various other sections of this document contain additional section-specific implementation notes.)

3.8.1 Forward and undefined references

SVG makes extensive use of URI references to other objects. For example, to fill a rectangle with a linear gradient, you define a **<lineargradient>** element and give it an ID (e.g., **<lineargradient id="MyGradient"...>**, and then you can specify the rectangle as follows: **<rect style="fill:url(#MyGradient)"...>**.

In SVG, among the facilities that allow URI references are:

- the 'clippath' property
- the 'mask' property
- the 'fill' property

- the 'stroke' property
- the 'marker','marker-start','marker-mid' and 'marker-end properties
- the <use> element

Forward references are disallowed. All references should be to elements which are either defined in a separate document or defined earlier in same document. References to elements in the same document can only be to elements which are direct children of a <defs> element. (See Defining referenced and undrawn elements: the **<defs>** element.).

Invalid references should be treated as if no value were provided for the referencing attribute or property. For example, if there is no element with ID "BogusReference" in the current document, then **fill="url(#BogusReference)"** would represent an invalid reference. In cases like this, the element should be processed as if no 'fill' property were provided. Where a list of property values are possible and one of the property values is an undefined reference, then process the list as if the reference were removed from the list.

3.8.2 Referenced objects are "pinned" to their own coordinate systems

When a graphical object is referenced by another graphical object (such as when referenced graphical object is used as a clipping path), the referenced object does not change location, size or orientation. Thus, referenced graphical objects are "pinned" to the user coordinate system that is in place within its own hierarchy of ancestors and is not affected by the user coordinate system of the referencing object.

4 SVG Document Structure

4.1 Introduction

Each SVG document is contained within an **<svg>** outermost element.

An SVG "document" can range from a single SVG graphics element such as a rectangle to a complex, deeply nested collection of grouping and graphics elements. Also, an SVG document can be embedded inline as a **fragment** within a parent document (an expectedly common situation with an XML Web pages) or it can stand by itself as a **self-contained** graphics file.

The following example shows a simple SVG document embedded as a fragment within a parent XML document. Note the use of XML namespaces to indicate that the **<svg>** and **<rect>** elements belong to the SVG namespace:

```
<?xml version="1.0" standalone="yes"?>
<parent xmlns="http://someplace.org"
       xmlns:svg="http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
   <!-- parent stuff here -->
   <svg:svg width="5cm" height="8cm">
      <svg:ellipse rx="200" ry="130" />
   </svg:svg>
   <!-- ... -->
</parent>
```

Download this example

This example shows a slightly more complex (i.e., it contains multiple rectangles) stand-alone, self-contained SVG document:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Four separate rectangles
  </desc>
    <rect width="20" height="60"/>
    <rect width="30" height="70"/>
    <rect width="40" height="80"/>
    <rect width="50" height="90"/>
</svg>
```

Download this example

<svg> elements can appear in the middle of SVG documents. This is the mechanism by which SVG documents can be embedded within other SVG documents.

Another use for **<svg>** elements within the middle of SVG documents is to establish a new viewport and alter the meaning of CSS unit specifiers. See Establishing a New Viewport: the <svg> element within an SVG document and Redefining the meaning of CSS unit specifiers. .

4.2 Grouping and Naming Collections of Drawing Elements: the <g> Element

The **<g>** element is the element for grouping and naming collections of drawing elements. If several drawing elements share similar attributes, they can be collected together using a <g> element. For example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Two groups, each of two rectangles
  </desc>
  <g style="fillcolor:red">
    <rect x="100" y="100" width="100" height="100" />
    <rect x="300" y="100" width="100" height="100" />
  </g>
  <g style="fillcolor:blue">
    <rect x="100" y="300" width="100" height="100" />
    <rect x="300" y="300" width="100" height="100" />
  </g>
</svg>
```

Download this example

A group of drawing elements, as well as individual objects, can be given a name. Named groups are needed for several purposes such as animation and re-usable objects. The following example organizes the drawing elements into two groups and assigns a name to each group:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Two named groups
  </desc>
  <g id="OBJECT1">
    <rect x="100" y="100" width="100" height="100" />
  </g>
  <g id="OBJECT2">
    <circle cx="150" cy="300" r="25" />
  </g>
</svg>
```

Download this example

A <g> element can contain other <g> elements nested within it, to an arbitrary depth. Thus, the following is valid SVG:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Groups can nest
  </desc>
  <g>
     <g>
       <g>
       </g>
     </g>
   </g>
</svg>
```

Download this example

Any drawing element that is not contained within a <g> is treated (at least conceptually) as if it were in its own group.

4.3 Defining referenced and undrawn elements: the <defs> element

Every graphics element (???) in SVG can have a child **<defs>** element. The two main purpose of the <defs> element are:

- To identify those objects which will be referenced by other objects later in the document. It is a requirement that all referenced objects be defined within a <defs>. This requirement allows SVG user agents to potentially perform optimizations because only those elements defined in <defs> need to be retained as the remainder of the document is processed. (Additionally, all referenced elements from the same document must be located before the referencing element; thus, forward referencing is disallowed.).
- To provide a convenient mechanism for defining objects that are not drawn directly. For example, most clipping paths are only used for clipping purposes and not meant to be painted.

A <defs> element defines the child elements that are contained within it. The child elements are not drawn at definition.

The following example shows how a undrawn rectangle and a gradient can be defined within a <defs> element so that they can be referenced later in the document:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN" "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <rect id="TemplateObject01" width="100" height="37.34"/>
    <lineargradient id="Gradient01">
      <stop offset="30%" style="color:#39F"/>
    </lineargradient>
  </defs>
  <desc>Defining things for later use
  </desc>
  <!-- SVG elements in here would reference/use
       the elements defined in the <defs> -->
</svg>
```

Download this example

4.3.1 The <style> sub-element to <defs>

A **<style>** element can appear as a subelement to any <defs> element. A <style> element is equivalent to the <style> element in HTML and thus can contain any valid CSS definitions. Any CSS definitions within any <style> element have a "global" scope across the entire current SVG document. It is useful to structurally associate style with each definition, so that the style is available when the definition is used (possibly from another SVG graphic).

The following is an example of defining and using a text style:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <style><![CDATA[
      .TitleText { font-size: 16; font-family: Helvetica } ]]>
    </style>
  </defs>
  <text class="TitleText">Here is my title</text>
</svg>
```

Download this example

4.3.2 The <script> sub-element to <defs>

A **<script>** element can appear as a subelement to any <defs> element. A <script> element is equivalent to the <script> element in HTML and thus is the place for scripts (e.g., ECMAScript). Any functions defined within any <script> element have a "global" scope across the entire current SVG document.

The following is an example of defining an ECMAScript function and defining an event handler that invokes that function:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <script><![CDATA[
      /* Beep on mouseclick */
      MouseClickHandler() { beep(); }
     ]]>
    </script>
  </defs>
  <circle onclick="MouseClickHandler()" r="85"/>
</svg>
```

Download this example

4.4 The <symbol> element

The **<symbol>** element is used to define graphical objects which are meant for any of the following uses:

- A template object which will be used (i.e., instantiated) multiple times within a given document
- A member of a standard drawing symbol library that may be referenced by many different SVG documents
- The definition of a graphic to use as a custom glyph within a <text> element (e.g., generalize "text-on-a-path" to "SVG-on-a-path")
- Definition of a sprite for an animation

Closely related to the **<symbol>** element are the <marker> and <pattern> elements.

A <symbol> element can contain the same graphic elements that are allowed in <svg> and <g> elements. A <symbol> element has the following additional attributes to meet the needs of the above situations:

- **fit-box-to-viewport** and **preserve-aspect-ratio**, which are described in Establishing an Initial User Coordinate System: the fit-box-to-viewport attribute.
- **ref-x**, **ref-y**, which indicates a reference point for the symbol which is used in some cases (e.g., aligning an arrowhead onto a path).

4.5 The <use> element

Any <svg>, <symbol>, <g> or graphics element defined within a <defs> and assigned an ID is potentially a template object that can be re-used (i.e., "instanced") anywhere in the SVG document.

The **<use>** element references another graphics object and indicates that the contents of that graphics object should be included/drawn at that given point in the document. The <use> element conforms to XLink [??? Include reference to XLink]. (Note that the XLink specification is currently under development and is subject to change. The SVG working group will track and rationalize with XLink as it evolves.)

The **<use>** element can reference either:

- an element within the same SVG document whose immediate ancestor is a <defs> element

- an element within a different SVG document whose immediate ancestor is a <defs> element

Unlike <image>, the **<use>** element cannot reference entire files.

In the example below, the first **<g>** element has inline content. After this comes a **<use>** element whose *href* value indicates which predefined graphics object should be included/rendered at that point in the document. Finally, the second **<g>** element has both inline and referenced content. In this case, the referenced content will draw first, followed by the inline content.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <symbol id="TemplateObject01">
      <!-- symbol definition here -->
    </symbol>
  </defs>

  <desc>Examples of inline and referenced content
  </desc>

  <!-- <g> with inline content -->
  <g>
    <!-- Inline content goes here -->
  </g>

  <!-- referenced content -->
  <use href="#TemplateObject01" />

  <!-- <g> with both referenced and inline content -->
  <g>
    <use href="#TemplateObject01" />
    <!-- Inline content goes here -->
  </g>
</svg>
```

Download this example

The <use> element has optional attributes *x*=, *y*=, *width*= and *height*= which are used to map a referenced <symbol> element onto a rectangular region within the current coordinate system. The 'transform' property can also be applied to do a subsequent transformation of the symbol's coordinates after those coordinates are mapped onto the rectangular region.

Any graphics attributes specified on a <use> element override the attributes specified on the template/referenced element. For example, if the *stroke-width* on the template object is 10 but the <use> specifies a *stroke-width* of 20, then the object will draw with a *stroke-width* of 20.

The <use> element **does not** do the equivalent of a macro expansion. The SVG Document Object Model (DOM) only contains a <use> element and its attributes.

4.6 The <image> element

The **<image>** element indicates that the contents of a complete file should be rendered into a given rectangle within the current user coordinate system. The **<image>** element can refer to image files such as PNG or JPEG or to files with MIME type of "image/svg". Conforming SVG viewers need to support at least PNG, JPEG and SVG format files.

For more on image objects, refer to Images.

Unlike <use>, the **<image>** element cannot reference elements within an SVG file.

5 SVG Rendering Model

5.1 Introduction

Implementations of SVG are expected to behave as though they implement a rendering (or imaging) model corresponding to the one described in this chapter. A real implementation is not required to implement the model in this way, but the result on any device supported by the implementation should match that described by this model.

The chapter on conformance requirements describes the extent to which an actual implementation may deviate from this description. In practice an actual implementation will deviate slightly because of limitations of the output device (e.g. only a limited range of colors may be supported) and because of practical limitations in implementing a precise mathematical model (e.g. for realistic performance curves are approximated by straight lines, the approximation need only be sufficiently precise to match the conformance requirements.)

5.1.1 The painters model

SVG uses a "painters model" of rendering. "Paint" is applied in successive operations to the output device such that each operation paints over some area of the output device. When the area overlaps a previously painted area the new paint partially or completely obscures the old. This model is more sophisticated than older models in that the paint may be only partially opaque. When the paint is not completely opaque the result on the output device is defined by the (mathematical) rules for compositing described under Simple Alpha Blending.

5.1.2 Rendering Order

Elements in an SVG document have an implicit drawing order, with the first elements in the SVG document getting "painted" first. Subsequent elements are painted on top of previously painted elements.

5.1.3 Grouping

Grouping elements such as the <g> have the effect of producing a temporary separate canvas onto which child elements are painted. Upon the completion of the group, the effect is as if the group's canvas is painted onto the ancestors canvas using the standard rendering rules for individual graphic objects.

5.1.4 Specifying paint

Paint is specified as a color and an opacity. Color is normally specified using the RGB values traditionally used in computer systems. This is defined in Color. Opacity is simply a measure of the amount of the underlying color that the new paint obscures - a percentage or fractional value.

Paint may also be specified as a combined color and opacity value by specifying an external bitmap image. This is discussed in more detail below.

5.1.5 Specifying the painted region

Paint, when specified as color and opacity, is associated with a painting operation that fills a particular region or draws a line along a particular path. SVG specifies in Paths a syntax for defining such a path as well as rules which determine what parts of an output device are within the path and what parts are outside. These latter rules allow a painted region to be determined from a path.

[The rendering model also needs a precise definition of how line width and joins are interpreted]

5.1.6 Use of external bitmap images

An image specifies both paint and the region that it fills by giving an array of values that specify the paint color and opacity (often termed alpha) at a series of points normally on a rectangular grid.

SVG requires support for specified bitmap formats under conformance requirements.

When an image is rendered the original samples are "resampled" using standard algorithms to produce samples at the positions required on the output device. Resampling requirements are discussed under conformance requirements.

5.1.7 Restricting painted regions

SVG allows any painting operation to be limited to a sub-region of the output device by clipping and masking. This is described in Clipping, Masking and Compositing.

Clipping uses a path to define a region of the output device to which paint may be applied. Any painting operation executed within the scope of the clipping must be rendered such that only those parts of the device that fall within the clipping region are affected by the painting operation. "Within" is defined by the same rules used to determine the interior of a path for painting.

Masking uses the alpha channel or color information in a referenced SVG element to restrict the painting operation. In this case the opacity information within the alpha channel is used to define the region to which paint may be applied - any region of the output device that, after resampling the alpha channel appropriately, has a zero opacity must not be affected by the paint operation. All other regions composite the paint from the paint operation onto the the output device using the algorithms described in Clipping, Masking and Compositing.

Masking may also be specified by applying a "global" opacity to a set of rendering operations. In this case the mask defines an infinite alpha channel with a single opacity. (See 'opacity' property.) Additionally, opacity can be set for fill and stroke operations. (See 'fill-opacity' property and 'stroke-opacity' property.

In all cases the SVG implementation must behave as though all qualified painting is done first to an intermediate (imaginary) canvas then filtered through the clip area or mask. Thus if an area of the output device is painted with a group opacity of 50% using opaque red paint followed by opaque green paint the result is as though it had been painted with just 50% opaque green paint. This is because the opaque green paint completely obscures the red paint on the intermediate canvas before the intermediate as a whole is rendered onto the output device.

5.1.8 Filtering painted regions

SVG also allows any painting operation to be filtered. (See Filter Effects)

In this case the result must be as though the paint operations had been applied to an intermediate canvas, of a size determined by the rules given in Filter Effects then filtered by the processes defined in Filter Effects.

5.1.9 Parent Compositing

SVG documents can be semi-opaque. In many environments (e.g., web browsers), the SVG document has a final compositing step where the document as a whole is blended translucently into the background canvas.

5.2 Rendering Properties

The creator of an SVG document might want to provide a hint to the implementation about what tradeoffs to make as it renders vector graphics objects such as <path> elements and other vector graphic shapes such as circles and rectangles. The **'shape-rendering'** property provides these hints.

'shape-rendering'

Value:	default | optimize-speed | crisp-edges | geometric-precision | inherit
Initial:	false
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

'shape-rendering' provides a hint to the SVG user agent about how to optimize its shape rendering.

default

Indicates that the user agent should make appropriate tradeoffs to balance speed, crisp edges and geometric precision, but with geometric precision given more importance than speed and crisp edges.

optimize-speed

Indicates that the user agent should emphasize rendering speed over geometric precision and crisp edges. This option will sometimes cause the user agent to turn off shape anti-aliasing.

crisp-edges

Indicates that the user agent should attempt to emphasize the contrast between clean edges of artwork over rendering speed and geometric precision. To achieve crisp edges, the user agent might turn off anti-aliasing for all lines or possibly just for straight lines which are close to vertical or horizontal. Also, the user agent might adjust line positions and line widths to align edges with device pixels.

geometric-precision

Indicates that the user agent should emphasize geometric precision over speed and crisp edges.

The creator of an SVG document might want to provide a hint to the implementation about what tradeoffs to make as it renders text. The **'text-rendering'** property provides these hints.

'text-rendering'

Value:	default | optimize-speed | optimize-legibility | geometric-precision | inherit
Initial:	default
Applies to:	<text> elements
Inherited:	yes
Percentages:	N/A
Media:	visual

'text-rendering' provides a hint to the SVG user agent about how to optimize its text rendering.

default

Indicates that the user agent should make appropriate tradeoffs to balance speed, legibility and geometric precision, but with legibility given more importance than speed and geometric precision.

optimize-speed

Indicates that the user agent should emphasize rendering speed over legibility and geometric precision. This option will sometimes cause the user agent to turn off text anti-aliasing.

optimize-legibility

Indicates that the user agent should emphasize legibility over rendering speed and geometric precision. The user agent will often choose whether to apply anti-aliasing techniques, built-in font hinting or both to produce the most legible text.

geometric-precision

Indicates that the user agent should emphasize geometric precision over legibility and rendering speed. This option will usually cause the user agent to suspend the use of hinting so that glyph outlines are drawn with comparable geometric precision to the rendering of path data.

The creator of an SVG document might want to provide a hint to the implementation about how to make speed vs. quality tradeoffs as it performs image processing. The **'image-rendering'** property provides a hint to the SVG user agent about how to optimize its image rendering.

'image-rendering'

Value:	default | optimize-speed | optimize-quality | inherit
Initial:	default
Applies to:	<text> elements
Inherited:	yes
Percentages:	N/A
Media:	visual

default

Indicates that the user agent should make appropriate tradeoffs to balance speed and quality, but quality should be given more importance than speed.

optimize-speed

Indicates that the user agent should emphasize rendering speed over quality. This option will sometimes cause the user agent to use a bilinear image resampling algorithm.

optimize-quality

Indicates that the user agent should emphasize quality over rendering speed. This option will sometimes cause the user agent to use a bicubic image resampling algorithm.

The **'visibility'** indicates whether a given object should be rendered at all.

'visibility'

Value:	visible | hidden | inherit
Initial:	visible
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

'visibility' indicates whether a given object should be drawn.

visible

The current object should be drawn.

hidden

The current object should not be drawn.

6 Clipping, Masking and Compositing

6.1 Introduction

SVG supports the following clipping/masking features:

- **clipping paths**, which uses a vector graphics shape or text string to serve as the outline of a (in the absense of antialiasing) 1-bit mask, where everything on the "inside" of the outline is allowed to show through but everything on the outside is masked out
- **masks**, which uses an arbitrary grayscale or color bitmap (explicit or generated from SVG artwork) as a semi-transparent mask for compositing foreground objects into the current background.

SVG supports only simple alpha blending compositing (see Simple Alpha Blending/Compositing).

(Insert drawings showing a clipping path, a grayscale imagemask, simple alpha blending and more complex blending.)

6.2 Simple Alpha Blending/Compositing

[Insert discussion about color spaces and compositing techniques.]

It is likely that our default compositing colorspace will be linearized-sRGB, where it is linearized with respect to light energy. Any colors specified in sRGB would be composited after linearizing with the formula (alpha*src^2.2 + (1-alpha)*dst^2.2)^(1/2.2).

6.3 Clipping paths

The clipping path restricts the region to which paint can be applied. Conceptually, any parts of the drawing that lie outside of the region bounded by the currently active clipping path are not drawn. A clipping path can be thought of as a 1-bit mask.

A clipping path is defined with a **<clippath>** element. A clipping path is used/referenced using the **'clippath'** property.

A <clippath> element can contain <path> elements, <text> elements, other vector graphic shapes (such as <circle>) or a <use> element. If a <use> element is a child of a <clippath> element, it must directly reference path, text or vector graphic shape elements. Indirect references are an error and are processed as if the <use> element were not present. The silhouettes of the child elements are logically OR'd together to create a single silhouette which is then used to restrict the region onto which paint can be applied.

If the **'clippath'** property references a non-existent object or if the referenced object is not a **<clippath>** element, then the 'clippath' property will be ignored.

For a given drawing element, the actual clipping path used will be the intersection of its specified *clippath* with the *clippath*s of all its ancestors.

(There will be a mechanism for ensuring that an initial clipping path is set to the bounds of the entire viewport into which the SVG is being drawn. The working group is also investigating ways to allow an SVG drawing to draw outside of the initial viewport [i.e., initial clipping path goes beyond the bounds of the initial viewport].)

'clippath'

Value:	<uri> | none
Initial:	The bounds of the viewport (see ??? link to viewport).
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

<uri>

An XPointer to another graphical object within the same SVG document which will be used as the clipping path.

'cliprule'

Value:	evenodd | nonzero | inherit
Initial:	evenodd
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

evenodd

(??? Need detailed description plus drawings)

nonzero

(??? Need detailed description plus drawings)

6.4 Masking

In SVG, you can specify that any other graphics object or <g> element can be used as an alpha mask for compositing the current object into the background. One important distinction between a clipping path and a mask is that clipping paths are hard masks (i.e., the silhouette consists of either fully opaque pixels or fully transparent pixels, with the possible exception of antialiasing along the edge of the silhouette) whereas masks consist of a one-channel image where pixel values can range from fully transparent to semi-transparent to fully opaque.

A mask is defined with a **<mask>** element. A mask is used/referenced using the **'mask'** property.

A **<mask>** can contain any graphical elements or grouping elements such as a <g>.

If the **'mask'** property references a non-existent object or if the referenced object is not a **<mask>** element, then the 'mask' property will be ignored.

The effect is as if the child elements of the **<mask>** are rendered into an offscreen image. Any graphical object which uses/references the given **<mask>** element will be painted onto the background through the mask, thus completely or partially masking out parts of the graphical object.

The following processing rules apply:

- If the child elements of the <mask> define a one-channel image, then use that channel as the mask.
- If the child elements of the <mask> define a three-channel RGB image, then use the luminance from the image as the mask, where the luminance is calculated using the following formula: `1.0-(.2126*R^2.2+.7152*G^2.2+.0722*B^2.2)`.
- If the child elements of the <mask> define a four-channel RGBA image, then use the alpha channel as the mask.

Note that SVG <path>'s, shapes (e.g., <circle>) and <text> are all treated as four-channel RGBA images for the purposes of masking operations.

In the following example, an image is used to mask a rectangle:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Example of using a mask
  </desc>
  <g>
    <defs>
      <mask id="MyMask">
        <image href="transp.png" />
      </mask>
    </defs>
    <rect style="mask: url(#MyMask)" width="12.5" height="30" />
  </g>
</svg>
```

Download this example

A <mask> element can define a region on the canvas for the mask using the following attributes:

- **mask-units={ userspace | object-bbox }**. Defines the coordinate system for attributes **x, y, width, height**. If **mask-units="userspace"** (the default), **x, y, width, height** represent values in the current user coordinate system in place at the time when the <mask> element is defined. If **mask-units="object-bbox"**, then **x, y, width, height** represent values in an abstract coordinate system where (0,0) is the (minx,miny) in user space of the tight bounding box of the object referencing the mask, and (1,1) is the (maxx,maxy) corner of the bounding box. (Note: the bounding box represents the maximum extent of the shape of the object in X and Y with respect to the user coordinate system of the object exclusive of stroke-width.)
- **x, y, width, height**, which indicate the rectangle for the largest possible offscreen buffer, where the values are either relative to the current user coordinate system (if mask-units="userspace") or relative to the current object (if mask-units="target-object"). Note that the clipping path used to render any graphics within the mask will consists of the intersection of the current clipping path associated with the given object and the rectangle defined by **x, y, width, height**. The default values for **x, y, width, height** are 0%, 0%, 100% and 100%, respectively.

The following is a description of the **'mask'** property.

'mask'

Value:	<uri> | none
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual

<uri>

An XPointer to another graphical object which will be used as the mask.

6.5 Object And Group Opacity: the 'opacity' Property

There are several opacity properties within SVG:

- Fill opacity
- Stroke opacity
- Gradient stop opacity
- Object/group opacity

Except for object/group opacity (described just below), all other opacity properties are involved in intermediate rendering operations. Object/group opacity can be thought of conceptually as a postprocessing operation. Conceptually, after the object/group is rendered into an RGBA *SourcePixmap* in viewport space, the object/group opacity setting specifies how to blend the *SourcePixmap* into the current background.

'opacity'

Value:	<alphavalue> | <pct>
Initial:	100%
Applies to:	all elements
Inherited:	no
Percentages:	Yes, relative to the current viewport
Media:	visual

<alphavalue>

The uniform opacity setting to be applied across an entire object expressed as an <number> between 0 and 255. If the object is a <g>, then the effect is as if the contents of the <g> were blended against the current background using an 8-bit *MaskingPixmap* where the value of each pixel of *MaskingPixmap* is <alphavalue>.

<pct>

The uniform opacity setting to be applied across an entire object expressed as a percentage (100% means fully opaque). If the object is a <g>, then the effect is as if the contents of the <g> were blended against the current background using an 8-bit *MaskingPixmap* where the value of each pixel of *MaskingPixmap* is <pct> * 255.

7 CSS Properties, XML Attributes, Cascading, and Inheritance

7.1 Introduction: CSS Properties vs. XML Attributes

Any language which is an application of XML and supports Cascading Style Sheets (CSS) will necessarily have some of its element parameters as XML attributes and others as CSS properties. SVG is no exception.

In designing SVG, the following criteria were used to decide which parts of SVG were expressed as XML attributes and which as CSS properties. In general, CSS properties were used for the following:

- Parameters which are clearly visual in nature and thus lend themselves to styling. Examples include all attributes that define how an object is "painted" such as fill and stroke colors, linewidths and dash styles.
- Parameters having to do with text styling such as **font-family** and **font-size**. (In fact, SVG supports all of the text and font properties from CSS2.)
- Parameters which arguably fit in with CSS and which might have future use in multiple other XML grammars. By defining these attributes as CSS properties, we are making progress toward commonality across multiple languages. (Transformations are an example of this. ??? add link here)
- Parameters which arguably fit in with CSS and which could allow for smaller file sizes if they were defined as CSS properties rather than XML attributes.

All remaining parameters are XML attributes. The result is that the majority of parameters in SVG are expressed as CSS properties.

7.2 Cascading and Inheritance of XML Attributes and CSS Properties

SVG conforms fully to the cascading style rules of CSS (i.e., the rules by which the SVG user agent decides which property setting applies to a given element). See the CSS2 specification for a discussion of these rules.

In this document, the definition of each XML attribute and CSS property indicates whether that attribute/property can inherit the value of its parent.

7.3 The Scope/Range of CSS Styles

The following define the scope/range of CSS styles:

Stand-alone SVG graphic

> There is one parse tree, and all elements are in the SVG namespace. CSS styles defined anywhere within the SVG graphic (in style elements or style attributes, or in external style sheets linked with the stylesheet PI) apply across the entire SVG graphic.

Stand-alone SVG graphic embedded in an HTML document with the <img> or <object> elements

> There are two completely separate parse trees; one for the HTML document, and one for the SVG graphic. CSS styles defined anywhere within the HTML document (in style elements or style attributes, or in external style sheets linked with the stylesheet PI) apply across the entire HTML document. Since inheritance is down a parse tree, these styles do not affect the SVG graphic. CSS styles defined anywhere within the SVG document (in style elements or style attributes, or in external style sheets linked with the stylesheet PI) apply across the entire SVG document. These styles do not affect the containing HTML document. To get the same styling across both HTML document and SVG graphic, link them both to the same stylesheet.

Stand-alone SVG graphic textually included in an XML document

> There is a single parse tree, using multiple namespaces; one or more subtrees are in the SVG namespace. CSS styles defined anywhere within the XML document (in style elements or style attributes, or in external style sheets linked with the stylesheet PI) apply across the entire document including those parts of it in the SVG namespace. To get different styling for the SVG part, use the style attribute or <style> element on the <svg> element. Alternatively, put an ID on the <svg> element and use contextual CSS selectors.

8 Coordinate Systems, Transformations and Units

8.1 Introduction

For all media, the term *canvas* describes "the space where the SVG document is rendered." The canvas is infinite for each dimension of the space, but rendering occurs relative to a finite rectangular region of the canvas. This finite rectangular region is called the **viewport**. For visual media, the viewport is the viewing area where the user sees the SVG document.

The size of the viewport (i.e., its width and height) is determined by a negotiation process (see Establishing the size of the initial viewport) between the SVG document and its parent (real or implicit). Once that negotiation process is completed, the SVG user agent is provided the following information:

- an integer value that represents the width of the viewport in "pixels"
- an integer value that represents the height of the viewport in "pixels"
- (highly desirable but not required) a real number value that indicates how many millimeters a "pixel" represents

Using the above information, the SVG user agent establishes an initial **current coordinate system** for the SVG document such that the origin of the current coordinate system matches the origin of the viewport, and one unit in the current coordinate system equals one "pixel" in the viewport. This initial current coordinate system defines the initial **viewport space** and the initial **user space** (also called **user coordinate system**).

Lengths in SVG can be specified as:

- (if no unit designator is provided) values in user space -- for example, "15"
- (if a CSS unit specifier is provided) a length in CSS units -- for example, "15mm"

The supported CSS length unit specifiers are: em, ex, px, pt, pc, cm, mm, in, and percentages.

A new user space (i.e., a new current coordinate system) can be established at any place in the SVG document by specifying **transformations** in the form of **transformation matrices** or simple transformation operations such as rotation, skewing, scaling and translation. Establishing new user spaces via transformation operations are fundamental operations to 2D graphics and represent the typical way of controlling the size, position, rotation and skew of graphic objects.

New viewports also can be established, but are for more specialized uses. By establishing a new viewport, you can redefine the meaning of some of the various CSS unit specifiers (px, pt, pc, cm, mm, in, and percentages) and provide a new reference rectangle for "fitting" a graphic into a particular rectangular area. ("Fit" means that a given graphic is transformed in such a way that its bounding box in user space aligns exactly with the edges of a given viewport.)

8.2 Establishing the initial viewport

The attributes of the initial viewport are established by either the CSS positioning parameters that are defined by the outermost **<svg>** element in combination with the **width=** and **height=** XML attributes that are required on the **<svg>** element.

The size (i.e., width and height) of the initial viewport into which an SVG document should be rendered is determined as follows. If the outermost **<svg>** element contains CSS positioning properties which establish the width for the viewport, then the width of the viewport should be set to that size. If the CSS positioning properties on the outermost **<svg>** element do not provide sufficient information to determine the width of the viewport, then the XML attributes **width=** determines the width of the viewport. Similarly, if the outermost **<svg>** element contains CSS positioning properties which establish the height for the viewport, then the height of the viewport should be set to that size. If the CSS positioning properties on the **<svg>** element do not provide sufficient information to determine the height of the viewport, then the XML attributes **height=** determines the height of the viewport.

In the following example, an SVG graphic is embedded within a parent XML document which is formatted using CSS layout rules. The **width="100px"** and **height="200px"** attributes are used to set the size of the viewport:

```
<?xml version="1.0" standalone="yes"?>
<parent xmlns="http://some.url">

   <!-- SVG graphic -->
   <svg xmlns='http://www.w3.org/Graphics/SVG/svg-19990706.dtd'
      width="100px" height="200px">
      <path d="M100,100 Q200,400,300,100"/>
      <!-- rest of SVG graphic would go here -->
   </svg>

</parent>
```

Download this example

The initial clipping path for an SVG document is determined by the actual values of the **'clip'** and **'overflow'** properties that apply to the outermost **<svg>** element. (These concepts and properties are defined in the Cascading Style Sheets, level 2 CSS2 Specification.)

8.3 Establishing A New User Space: Transformations

To change the current user space coordinate system, you define a **transformation** which defines how to transform coordinates from the new user coordinate system into the previous user coordinate system. Mathematically, the transformation is represented by a **transformation matrix** which maps coordinates in the new user coordinate system into the previous user coordinate system. To illustrate:

(Insert an image which shows this concept.)

Transformation matrices define the mathematical mapping from one coordinate space into another. Of particular interest is the **current transformation matrix** (CTM) which defines the mapping from user space into viewport space.

(Insert an image showing the CTM mapping user space into device space.)

Transformation matrices are specified as 3x3 matrices of the following form:

(Insert an image showing [a b 0 c d 0 e f 1], but as a rectangular matrix.)

Because SVG's transformation matrices only have six entries that can be changed, these matrices will be called **2x3 transformation matrices**, which for convenience are often written as an array of six numbers: *[a b c d e f]*.

All coordinates in user space are expressed as (x,y) values. To calculate the transformation from the current user space coordinate system into viewport space, you multiply the vector (x,y,1) by the current transformation matrix (CTM) to yield (x',y',1):

(Insert an image showing [x',y',1]=[x,y,1][a b 0 c d 0 e f 1])

Whenever a new transformation is provided, a new CTM is calculated by the following formula. Note that the new transformation is pre-multiplied to the CTM:

(Insert an image which shows newCTM[a b c d e f]=[transformmatrix]*oldCTM[a b c d e f].)

It is important to understand the following key points regarding transformations in SVG:

- Transformations alter coordinate systems, not objects. All objects defined outside the scope of a transformation are unchanged by the transformation. All objects defined within the scope of a transformation will be drawn in the transformed coordinate system.
- Transformations specify how to map the transformed (new) coordinate system to the untransformed (old) coordinate system. All coordinates used within the scope the transformation are specified in the transformed coordinate system.
- Transformations are always premultiplied to the CTM.
- Matrix operations are not commutative - the order in which transformations are specified is significant. For example, a translation followed by a rotation will yield different results than a rotation followed by a translation:

  (Insert an image illustrates the above concept.)

Mathematically, all transformations can be expressed as matrices. To illustrate:

- Translations are represented as [1 0 0 1 tx ty], where *tx* and *ty* are the distances to translate the origin of the coordinate system in *x* and *y*, respectively.
- Scaling is obtained by [sx 0 0 sy 0 0]. This scales the coordinates so that one unit in the *x* and *y* directions of the new coordinate system is the same as *sx* and *sy* units in the previous coordinate system, respectively.
- Rotations are carried out by [cos(angle) sin(angle) -sin(angle) cos(angle) 0 0], which has the effect of rotating the coordinate system axes by *angle* degrees counterclockwise.
- (Similar examples can be given for skew, reflect and other simple transformations. At this time, the SVG working group is still investigating these other simple transformations.)

(Insert an image illustrates the above concept.)

8.4 Establishing an Initial User Coordinate System: the fit-box-to-viewport attribute

Various SVG elements have the effect of establishing a new viewport:

- Any **<svg>** element establishes a new viewport
- Any **<use>** element establishes a temporary new viewport for drawing instances of predefined graphics objects
- Markers establish a temporary new viewport for drawing arrowheads and polymarkers
- When the text on a path facility tries to draw a referenced <symbol> or <svg> element, it establishes a new temporary new viewport for the referenced graphic.
- When patterns are used to fill or stroke an object, a temporary new viewport is established for each drawn instance of the pattern.

It is very common to have the requirement that a given SVG document, <symbol>, <marker> or <pattern> should be scaled automatically to fit within a target rectangle. The **fit-box-to-viewport="<min-x> <min-y> <width> <height>"** attribute on the **<svg>** and **<symbol>** elements indicates that an initial coordinate system should be established such that the given rectangle in user space (specified by the four numbers <min-x> <min-y> <width> <height>) maps exactly to the current bounds of the viewport. For example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in" fit-box-to-viewport"0 0 40 30">
  <desc>This SVG drawing uses the fit-box-to-viewport
   attribute to automatically create an initial user coordinate
   system which causes the graphic to scale to fit into the
   viewport no matter what size the viewport is.
  </desc>
  <!-- This rectangle goes from (0,0) to (40,30) in user space.
       Because of the fit-box-to-viewport attribute above,
       the rectangle will end up filling the entire area
       reserved for the SVG document. -->
  <rect x="0" y="0" width="40" height="30" style="fill: blue" />
</svg>
```

Download this example

In some cases, it is necessary that the aspect ratio of the graphic be retained when utilizing **fit-box-to-viewport**. A supplemental attribute **preserve-aspect-ratio="<align> [<meet-or-slice>]"** indicates whether or not to preserve the aspect ratio of the original graphic. The **<align>** parameter indicates whether to preserve aspect ratio and what alignment method should be used if aspect ratio is preserved. The **<align>** parameter must be one of the following strings:

- **none** (the default) - Do not attempt to preserve aspect ratio. Scale the graphic non-uniformly if necessary such that the graphic's bounding box exactly matches the viewport rectangle.
- **xmin-ymin** - Attempt to preserve aspect ratio. Align the smallest X value of the graphic's bounding box with the smallest X value of the viewport. Align the smallest Y value of the graphic's bounding box with the smallest Y value of the viewport.
- **xmid-ymin** - Attempt to preserve aspect ratio. Align the midpoint X value of the graphic's bounding box with the midpoint X value of the viewport. Align the smallest Y value of the graphic's bounding box with the smallest Y value of the viewport.

- **xmax-ymin** - Attempt to preserve aspect ratio. Align the maximum X value of the graphic's bounding box with the maximum X value of the viewport. Align the smallest Y value of the graphic's bounding box with the smallest Y value of the viewport.
- **xmin-ymid** - Attempt to preserve aspect ratio. Align the smallest X value of the graphic's bounding box with the smallest X value of the viewport. Align the midpoint Y value of the graphic's bounding box with the midpoint Y value of the viewport.
- **xmid-ymid** - Attempt to preserve aspect ratio. Align the midpoint X value of the graphic's bounding box with the midpoint X value of the viewport. Align the midpoint Y value of the graphic's bounding box with the midpoint Y value of the viewport.
- **xmax-ymid** - Attempt to preserve aspect ratio. Align the maximum X value of the graphic's bounding box with the maximum X value of the viewport. Align the midpoint Y value of the graphic's bounding box with the midpoint Y value of the viewport.
- **xmin-ymax** - Attempt to preserve aspect ratio. Align the smallest X value of the graphic's bounding box with the smallest X value of the viewport. Align the maximum Y value of the graphic's bounding box with the maximum Y value of the viewport.
- **xmid-ymax** - Attempt to preserve aspect ratio. Align the midpoint X value of the graphic's bounding box with the midpoint X value of the viewport. Align the maximum Y value of the graphic's bounding box with the maximum Y value of the viewport.
- **xmax-ymax** - Attempt to preserve aspect ratio. Align the maximum X value of the graphic's bounding box with the maximum X value of the viewport. Align the maximum Y value of the graphic's bounding box with the maximum Y value of the viewport.

The **<meet-or-slice>** parameter is optional and must be one of the following strings:

- **meet** (the default) - Scale the graphic such that:
  - aspect ratio is preserved
  - the entire graphic (as defined by its bounding box) is visible within the viewport
  - the graphic is scaled up as much as possible, while still meeting the other criteria

  In this case, if the aspect ratio of the graphic does not match the viewport, some of the viewport will extend beyond the bounds of the graphic (i.e., the area into which the graphic will draw will be smaller than the viewport).
- **slice** - Scale the graphic such that:
  - aspect ratio is preserved
  - the entire viewport is covered by the graphic (as defined by its bounding box)
  - the graphic is scaled down as much as possible, while still meeting the other criteria

  In this case, if the aspect ratio of the graphic does not match the viewport, some of the graphic will extend beyond the bounds of the viewport (i.e., the area into which the graphic will draw is larger than the viewport).

8.5 Modifying the User Coordinate System: the transform attribute

All modifications to the user coordinate system are specified with the **transform** attribute: The **transform** attribute defines a new coordinate system transformation and thus implicitly a new user space and a new CTM. A transform attribute takes a list of transformations, which are applied in the order provided. The available transformations include:

- **matrix(<a> <b> <c> <d> <e> <f>)**, which specifies that the given transformation matrix should be premultiplied to the old CTM to yield a new CTM.

- **translate(<tx> [<ty>])**, which indicates that the origin of the current user coordinate system should be translated by *tx* and *ty* If <*ty*> is not provided, it is assumed to be zero.
  [A translate is equivalent to **matrix(1 0 0 1 tx ty)**].

- **scale(<sx> [<sy>])**, which indicates that the user coordinate system should be scaled by *sx* and *sy*. If <*sy*> is not provided, it is assumed to be equal to <*sy*>.
  [A scale is equivalent to **matrix(sx 0 0 sy 0 0)**].

- **rotate(<rotate-angle>)**, which indicates that the current user coordinate system should be rotated relative to its origin by <rotate-angle>, which is expressed in degrees.
  [A rotation is equivalent to **matrix(cos(angle) sin(angle) -sin(angle) cos(angle) 0 0)**].

- **skew-x(<skew-angle>)**, which indicates that the current user coordinate system should be transformed such that, for positive values of <skew-angle>, increasingly positive Y values will be tilted by increasing amounts in the direction of the positive X-axis. (??? Need picture). <skew-angle> is expressed in degrees.
  [A skew-x is equivalent to **matrix(1 0 tan(angle) 1 0 0)**].

- **skew-y(<skew-angle>)**, which indicates that the current user coordinate system should be transformed such that, for positive values of <skew-angle>, increasingly positive X values will be tilted by increasing amounts in the direction of the positive Y-axis. (??? Need picture). <skew-angle> is expressed in degrees.
  [A skew-y is equivalent to **matrix(1 tan(angle) 0 1 0 0)**].

All values are real numbers.

If a list of transforms is provided, then the net effect is as if each transform had been applied separately in the order provided. For example, **transform="translate(-10,-20) scale(2) rotate(45) translate(5,10)"** indicates that:

- the origin of the user coordinate system should be translated -10 units in X and -20 units in Y (equivalent to transformation matrix [1 0 0 1 -10 -20]),
- then the user coordinate system should be scaled uniformly by a factor of 2 (equivalent to transformation matrix [2 0 0 2 0 0]),
- then the user coordinate system should be rotated by 45 degrees (equivalent to transformation matrix [cos(45) sin(45) -sin(45) cos(45)]),
- then origin of the user coordinate system should be translated by 5 units in X and 10 units in Y (equivalent to transformation matrix [1 0 0 1 5 10]).

The result is equivalent to pre-multiplying all of the matrices together: [1 0 0 1 5 10] * [cos(45) sin(45) -sin(45) cos(45)] * [2 0 0 2 0 0] * [1 0 0 1 -10 -20], which would be roughly equivalent to the following transform definition: **matrix(1.414 1.414 -1.414 1.414 -17.07 1.213)**.

The **transform** attribute is applied to an element before processing any other coordinate or length values supplied for that element. Thus, in the element **`<rect x="10" y="10" width="20"`**

height="20" transform="scale(2)" />, the x, y, width and height values are processed after the current coordinate system has been scaled uniformly by a factor of 2 by the **transform** attribute. Thus, x, y, width and height (and any other attributes or properties) are treated as values in the new user coordinate system, not the previous user coordinate system.

8.6 Establishing a New Viewport: the <svg> element within an SVG document

At any point in an SVG drawing, you can establish a new viewport into which all contained graphics should be drawn by including an **<svg>** element inside an SVG document. By establishing a new viewport, you also implicitly establish a new initial user space, new meanings for many of the CSS unit specifiers and, potentially, a new clipping path.

To establish a new viewport, you use the positioning properties from CSS such as **left:**, **top:**, **right:**, **bottom:**, **width:**, **height:**, margin properties and padding properties. Here is an example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>This SVG drawing embeds another one,
    thus establishing a new viewport
  </desc>
  <!-- The following statement establishing a new viewport
       and renders SVG drawing B into that viewport -->
  <svg style="left: 25%; top: 25%" width="50%" height="50%">
     <!-- drawing B goes here -->
  </svg>
</svg>
```

Download this example

You can also specify values for the **'clip'** and **'overflow'** properties for <svg> elements within an SVG document. If specified on an <svg> element, these properties will change the current clipping path. (Note that these properties will be ignored if used on any other type of element.)

8.7 Properties to Establish a New Viewport

(Extract sections from chapter 8 of the CSS spec. Make modifications as necessary.).

8.8 Units

All coordinates and lengths in SVG can be specified in one of the following ways:

- **User units**. If no unit specifier is provided, a given coordinate or length is assumed to be in user units (i.e., a value in user space). For example:

  ```
  <text style="font-size: 50">Text size is 50 user units</text>
  ```

- **CSS units**. If a unit designator is provided on a coordinate or length value, then the given value is assumed to be in CSS units. Available unit designators are the absolute and relative unit designators from CSS (em, ex, px, pt, cm, mm, in and percentages). As in CSS, the *em* and *ex* unit designators are relative to the current font's *font-size* and *x-height*, respectively. Initially, the various absolute unit specifiers from CSS (i.e., px, pt, cm, mm, in) represent lengths within the initial user coordinate system and do not change their meaning as transformations alter the current coordinate system. Thus, "12pt" can be made to represent exactly 12 points on the actual visual medium even if the user coordinate system has been scaled. For example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Demonstration of coordinate transforms
  </desc>
  <!-- The following two text elements will both draw with a
        font height of 12 pixels -->
   <text style="font-size: 12">This prints 12 pixels high.</text>
   <text style="font-size: 12px">This prints 12 pixels high.</text>

   <!-- Now scale the coordinate system by 2. -->
   <g transform="scale(2)">

      <!-- The following text will actually draw 24 pixels high
           because each unit in the new coordinate system equals
           2 units in the previous coordinate system. -->
      <text style="font-size: 12">This prints 24 pixels high.</text>

      <!-- The following text will actually still draw 12 pixels high
           because the CSS unit specifier has been provided. -->
      <text style="font-size: 12px">This prints 12 pixels high.</text>

   </g>
</svg>
```

Download this example

If possible, the SVG user agent should be passed the actual size of a *px* unit in inches or millimeters by its parent user agent. (See Conformance Requirements and Recommendations.) If such information is not available from the parent user agent, then the SVG user agent should assume a *px* is defined to be exactly .28mm.

8.9 Redefining the meaning of CSS unit specifiers

The process of establishing a new viewport via an **<svg>** element inside of an SVG document changes the meaning of the following CSS unit specifiers: px, pt, cm, mm, in, and % (percentages). A "pixel" (the px unit) becomes equivalent to a single unit in the user coordinate system for the given **<svg>** element. The meaning of the other absolute unit specifiers (pt, cm, mm, in) are determined as an appropriate multiple of a *px* unit using the actual size of *px* unit (as passed from the parent user agent to the SVG user agent). Any percentage values that are relative to the current viewport will also represent new values.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="300px" height="3oopx">
  <desc>Transformation with establishment of a new viewport
  </desc>
  <!-- The following two text elements will both draw with a
        font height of 12 pixels -->
   <text style="font-size: 12">This prints 12 pixels high.</text>
   <text style="font-size: 12px">This prints 12 pixels high.</text>
   <!-- Now scale the coordinate system by 2. -->
   <g style="transform: scale(2)">

      <!-- The following text will actually draw 24 pixels high
           because each unit in the new coordinate system equals
           2 units in the previous coordinate system. -->
      <text style="font-size: 12">This prints 24 pixels high.</text>

      <!-- The following text will actually still draw 12 pixels high
           because the CSS unit specifier has been provided. -->
      <text style="font-size: 12px">This prints 12 pixels high.</text>
   </g>

   <!-- This time, scale the coordinate system by 3. -->
   <g style="transform: scale(3)">

      <!-- Establish a new viewport and thus change the meaning of
           some CSS unit specifiers. -->
      <svg style="left:0; top:0; right:100; bottom:100"
           width="100%" height="100%">

         <!-- The following two text elements will both draw with a
              font height of 36 screen pixels. The first text element
              defines its height in user coordinates, which have been
              scaled by 3. The second text element defines its height
              in CSS px units, which have been redefined to be three times
              as big as screen pixels due the <svg> element establishing
              a new viewport. -->
         <text style="font-size: 12">This prints 36 pixels high.</text>
         <text style="font-size: 12px">This prints 36 pixels high.</text>

      </svg>
   </g>
</svg>
```

Download this example

8.10 Further Examples

(Include an example which shows multiple viewports, multiple user spaces and multiple use of different units.)

8.11 Implementation Notes

Any values expressed in CSS units or percentages of the current viewport should be implemented such that these values map to corresponding values in user space as follows:

- Coordinate values:
    - If both X and Y are expressed in viewport-relative coordinates (e.g., **<rect x="3in" y="2in" ... />**), then convert from CSS units into viewport space so that you have a coordinate pair (X,Y) in viewport space. Then apply the inverse of the current transformation matrix (CTM) onto (X,Y) to provide an (X',Y') coordinate value in user space.
    - If X is expressed in viewport-relative units, but Y is expressed in user space units (e.g., in **<rect x="2in" y="15" ... />**, X is expressed in viewport-relative units but Y is expressed in user space units), then determine the equation of the line in user space which corresponds to the vertical line through the point (X,0) in viewport space. (If user space is

rotated or skewed relative to the viewport, then the line in user space will no longer be vertical.) Then the result point (X',Y') in user space that you want is the intersection of the above equation with the horizontal line through the point (0,Y) in user space.
    As a example, suppose your viewport is 10 cm wide and 10 cm high and the CTM is [10 0 0 10 0 0]. If you encounter **<rect x="5cm" y="1" ... />**, then determine the equation of the line in user space that goes through (5cm, 0) in viewport space (the equation is `x = .5`). The result point (X',Y') in user space is the intersection of the line `x = .5` with the line `y = 1`, which results in the point `(X',Y')=(.5,1)` in user space.
    - If Y is expressed in viewport-relative units, but X is expressed in user space units (e.g., in **<rect x="43" y="2cm" ... />**, X is expressed in user space units but Y is expressed in viewport-relative units), then determine the equation of the line in user space which corresponds to the horizontal line through the point (0,Y) in viewport space. (If user space is rotated or skewed relative to the viewport, then the line in user space will no longer be vertical.) Then the result point (X',Y') in user space that you want is the intersection of the above equation with the horizontal line through the point (0,Y) in user space.
    As a example, suppose your viewport is 10 cm wide and 10 cm high and the CTM is [10 0 0 10 0 0]. If you encounter **<rect x="1" y="5cm" ... />**, then determine the equation of the line in user space that goes through (0, 5cm) in viewport space (the equation is `y = .5`). The result point (X',Y') in user space is the intersection of the line `y = .5` with the line `x = 1`, which results in the point `(X',Y')=(1,.5)` in user space.
- For any width value represented in a viewport-relative coordinate system (i.e., CSS units or percentages), transform the points (0,0) and (width,0) from viewport space to current user space using the inverse of the current transformation matrix, yielding two points in userspace Q1 and Q2. Do a distance calculation between Q1 and Q2 (sqrt((Q2x-Q1x)**2,(Q2y-Q1y)**2)) and use that as the width value for the given operation.
- For any height value represented in a viewport-relative coordinate system (i.e., CSS units or percentages), do the same thing as above, but use points (0,0) and (0,height) instead.
- For any length value which isn't tied to an axis, we use an approach which gives appropriate weighting to the contribution of the two dimensions of the viewport. Determine an angle ang=atan(viewport-height/viewport-width), then determine a point P=(length*cos(ang), length*sin(ang)) in viewport space. Transform the two points (0,0) and P from viewport space into current userspace using the inverse of the current transformation matrix, yielding two points in userspace Q1 and Q2. Do a distance calculation between Q1 and Q2 (sqrt((Q2x-Q1x)**2,(Q2y-Q1y)**2)) and use that as the length value for the given operation.

9 Filling, Stroking and Paint Servers

9.1 Introduction

Vector graphics shapes and text objects can be **filled** (which means painting the interior of the object) and **stroked** (which means painting along the outline of the object). Filling and stroking both can be thought of in more general terms as **painting** operations.

With SVG, you can paint (i.e., fill or stroke) with:

- a single color
- a gradient (linear or radial)
- a pattern (vector or image, possibly tiled)
- custom paints available via the extensibility mechanism

SVG uses the general notion of a **paint server**. Gradients and patterns are just specific types of paint servers. For example, first you define a gradient by including a <gradient> element within a <defs>, assign an ID to that <gradient> object, and then reference that ID in a **'fill'** or **'stroke'** property:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Linear gradient example
  </desc>
  <g>
    <defs>
      <lineargradient id="MyGradient">
        <stop offset="0%" style="color:#F60"/>
        <stop offset="70%" style="color:#FF6"/>
      </lineargradient>
    </defs>
    <rect style="fill: url(#MyGradient)" width="20" height="15.8"/>
  </g>
</svg>
```

Download this example

9.2 Fill Properties

'fill'

Value:	none | current-color | <color> [icc-color(<colorvalue>*)] | inherit | <uri> [none | current-color | <color> [icc-color(<colorvalue>*)] | inherit]
Initial:	current-color
Applies to:	all elements
Inherited:	see Inheritance of Painting Properties below
Percentages:	N/A
Media:	visual

none

Indicates that the object should not be filled.

current-color

Indicates that the object should filled with the color specified by the **'color'** property. This mechanism is provided to facilitate sharing of color attributes between parent grammars such as other (non-SVG) XML. This mechanism allows you to define a style in your HTML which sets the 'color' property and then pass that style to the SVG user agent so that your SVG text will draw in the same color.

<color> [icc-color(<colorvalue>*)]

<color> is the explicit color (in the sRGB color space) to be used to fill the current object. SVG supports all of CSS2's <color> specifications. If an optional **[icc-color(<colorvalue>*)]** is provided, then the list of **<colorvalue>**'s is a set ICC-profile-specific color values. On platforms which support ICC-based color management, the **icc-color** gets precedence over the <color> (which is in the sRGB color space). For more on ICC-based colors, refer to Color.

<uri> [none | current-color | <color> | inherit]

The <uri> is how you identify a fancy paint style such as a gradient, a pattern or a custom paint from extensibility. The <uri> should provide the ID of the paint server (e.g., a gradient [??? see link] or a pattern [??? see link]) to be used to paint the current object. If the XPointer is not valid (e.g., it points to an object that doesn't exist or the object is not a valid paint server), then the paint method following the <uri> (i.e., **none | current-color | <color> | inherit**) is used if provided; otherwise, no gradient will occur.

Note that graphical objects that are not closed (e.g., a <path> without a closepath at the end or a <polyline>) still can be filled. The fill operation automatically closes all open subpaths by connecting the last point of the subpath with the first point of the subpath before painting the fill.

'fillrule'

Value:	evenodd | nonzero | inherit
Initial:	evenodd
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

evenodd

(??? Need detailed description plus drawings)

nonzero

(??? Need detailed description plus drawings)

'fill-opacity'

Value:	<opacity-value>
Initial:	evenodd
Applies to:	all elements
Inherited:	yes
Percentages:	Allowed
Media:	visual

'fill-opacity' specifies the opacity of the painting operation used to fill the current object. It is important to note that any given object can have three different opacity properties: **'fill-opacity'**, **'stroke-opacity'** and **'opacity'**. The **'fill'** painting operation is done and blended into the current background (or temporary offscreen buffer, if **'opacity'** is not 1.0) using the value of **'fill-opacity'**. Next, The **'stroke'** painting operation is done and blended into the current background (or temporary offscreen buffer, if **'opacity'** is not 1.0) using the value of **'stroke-opacity'**. Finally, if **'opacity'** is not 1.0, the offscreen holding the object as a whole is blended into the current background.

(The above paragraph needs to be moved someplace else, such as SVG Rendering Model.)

<opacity-value>

The opacity of the painting operation used to fill the current object. If a <number> is provided, then it must be in the range of 0.0 (fully transparent) to 1.0 (fully opaque). If a percentage is provided, then it must be in the range of 0% to 100%. Any values outside of the acceptable range are rounded to the nearest acceptable value.

'fill-params'

Value:	<string> | inherit
Initial:	Empty string
Applies to:	all elements
Inherited:	yes
Percentages:	Paint server-specific.
Media:	visual

'fill-params' specifies an arbitrary <string> which is passed to the current fill paint server. The meaning of <string> is paint server-specific. None of the built-in paint servers use **'fill-params'**. It is meant as a way to pass parameters to a custom paint servers defined via paint server extensibility.

<string>

A <string> containing parameters which should be passed to the current fill paint server.

9.3 Stroke Properties

'stroke'

Value:	none | current-color | <color> [icc-color(<number>*)] | inherit | <uri> [none | current-color | <color> | inherit]
Initial:	none
Applies to:	all elements
Inherited:	see Inheritance of Painting Properties below

Percentages:	N/A
Media:	visual

none

(Same meaning as with **'fill'**.)

current-color

(Same meaning as with **'fill'**.)

<color>

(Same meaning as with **'fill'**.)

<uri> [none | current-color | <color> [icc-color(<colorvalue>*)] | inherit]

(Same meaning as with **'fill'**.)

'stroke-width'

Value:	<width> | inherit
Initial:	1
Applies to:	all elements
Inherited:	yes
Percentages:	Yes
Media:	visual

<width>

The width of the stroke on the current object, either expressed as a <length> in user units, a <length> in Transformed CSS units (??? add link) or as a percentage. If a percentage is used, the <width> is expressed as a percentage of the current viewport (??? add link).

'stroke-linecap'

Value:	butt | round | square | inherit
Initial:	butt
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

'stroke-linecap' specifies the shape to be used at the end of open subpaths when they are stroked.

butt

See drawing below.

round

See drawing below.

square

See drawing below.

(??? insert drawing here)

'stroke-linejoin'

Value:	miter | round | bevel | inherit
Initial:	miter
Applies to:	all elements
Inherited:	yes

Percentages:	N/A
Media:	visual

'stroke-linejoin' specifies the shape to be used at the corners of paths (or other vector shapes) that are stroked. when they are stroked.

miter

See drawing below.

round

See drawing below.

bevel

See drawing below.

(??? insert drawing here)

'stroke-miterlimit'

Value:	<miterlimit> | inherit
Initial:	8
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

When two line segments meet at a sharp angle and **miter** joins have been specified for **'stroke-linejoin'**, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The **'stroke-miterlimit'** imposes a limit on the ratio of the miter length to the **'stroke-linewidth'**.

<miterlimit>

The limit on the ratio of the miter length to the **'stroke-linewidth'**. The value of **<miterlimit>** must be a number greater than or equal to 1.

(??? insert drawing here)

'stroke-dasharray'

Value:	none | <dasharray> | inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	Yes. See below.
Media:	visual

'stroke-dasharray' controls the pattern of dashes and gaps used to stroke paths. The value of **<dasharray>** is a list of space- or comma-separated <number>'s that specify the lengths of alternating dashes and gaps.

none

Indicates that no dashing should be used. If stroked, the line should be drawn solid.

<dasharray>

A list of space- or comma-separated <length>'s which can be in user units or in any of the CSS units, including percentages. A percentage represents a distance as a percentage of the current viewport (??? Add link here).

An empty string for **'stroke-dasharray'** is equivalent to the value **none**.

(??? insert drawing here)

'stroke-dashoffset'

Value:	<dashoffset> | inherit
Initial:	0
Applies to:	all elements
Inherited:	yes
Percentages:	Yes. See below.
Media:	visual

'stroke-dashoffset' specifies the distance into the dash pattern to start the dash.

<dashoffset>

A <length> which can be in user units or in any of the CSS units, including percentages. A percentage represents a distance as a percentage of the current viewport (??? Add link here).

(??? insert drawing here)

'stroke-opacity'

Value:	<opacity-value> | inherit
Initial:	evenodd
Applies to:	all elements
Inherited:	yes
Percentages:	Allowed
Media:	visual

'stroke-opacity' specifies the opacity of the painting operation used to stroke the current object. (??? Add link about how different opacity parameters interact.)

<opacity-value>

The opacity of the painting operation used to stroke the current object. If a <number> is provided, then it must be in the range of 0.0 (fully transparent) to 1.0 (fully opaque). If a percentage is provided, then it must be in the range of 0% to 100%. Any values outside of the acceptable range are rounded to the nearest acceptable value.

'stroke-params'

Value:	<string> | inherit
Initial:	Empty string
Applies to:	all elements
Inherited:	yes
Percentages:	Paint server-specific.
Media:	visual

'stroke-params' specifies an arbitrary <string> which is passed to the current stroke paint server. The meaning of <string> is paint server-specific. None of the built-in paint servers use **'stroke-params'**. It is meant as a way to pass parameters to a custom paint servers defined via paint server extensibility.

<string>

A <string> containing parameters which should be passed to the current stroke paint server.

9.4 Gradients

Gradients consist of continuously smooth color transitions along a vector from one color to another, possibly followed by additional transitions along the same vector to other colors. SVG provides for two types of gradients, **linear gradients** and **radial gradients**.(??? Include drawing)

Gradients are specified within a <defs> element and are then referenced using **'fill'** or **'stroke'** or properties on a given graphics object (e.g., a <rect> element) to indicate that the given element should be filled or stroked with the referenced gradient.

9.4.1 Linear Gradients

Linear gradients are defined by a **<lineargradient>** element. A <lineargradient> element can have the following attributes:

- **gradient-units={ userspace | object-bbox }**. Defines the coordinate system for attributes **x1, y1, x2, y2**. If **gradient-units="userspace"** (the default), **x1, y1, x2, y2** represent values in the current user coordinate system in place at the time when the <lineargradient> element is defined. If **gradient-units="object-bbox"**, then **x1, y1, x2, y2** represent values in an abstract coordinate system where (0,0) is the (minx,miny) in user space of the bounding box of the object getting filled with the gradient, and (1,1) is the (maxx,maxy) corner of the bounding box. (Note: the bounding box represents the maximum extent of the shape of the object in X and Y with respect to the user coordinate system of the object exclusive of stroke-width.)
- **x1, y1, x2, y2** define a *gradient vector* for the linear gradient. This *gradient vector* provides starting and ending points onto which the **<stops>** are mapped (i.e., a gradient stop at 0% is mapped to the start of the gradient vector and a gradient stop at 100% is mapped to the end of the gradient vector). Defaults values are x1="0%", y1="0%", x2="100%", y2="0%". (??? Add drawing.)
- **gradient-transform** contains the definitions of an optional additional transformation from the gradient coordinate system onto the target coordinate system (i.e., userspace or object-bbox). This allows for things such as skewing the gradient. **gradient-transform** can take on the same values as the transform attribute.
- **spread-method** indicates what happens if the the gradient starts or ends inside the bounds of the *target rectangle*. Possible values are: *stick* (??? Need a better word. We mentioned one at the face-to-face), which says to use the terminal colors of the gradient to fill the remainder of the target region, *reflect*, which says to reflect the gradient pattern start-to-end, end-to-start, start-to-end, etc. continuously until the *target rectangle* is filled, and *repeat*, which says to repeat the gradient pattern start-to-end, start-to-end, start-to-end, etc. continuously until the target region is filled.

Percentages are allowed for **x1, y1, x2, y2**. For gradient-units="userspace", percentages represent values relative to the current viewport. For gradient-units="object-bbox", percentages represent values relative to the bounding box for the object.

(??? Need to include some drawings here showing these attributes)

9.4.2 Radial Gradients

Radial gradients are defined by a **<radialgradient>** element. A <radialgradient> element can have the following attributes:

- **gradient-units={ userspace | object-bbox }**. Defines the coordinate system for attributes **cx, cy, r, fx, fy**. If **gradient-units="userspace"** (the default), **cx, cy, r, fx, fy** represent values in the current user coordinate system in place at the time when the <radialgradient> element is defined. If **gradient-units="object-bbox"**, then **cx, cy, r, fx, fy** represent values in an abstract coordinate system where (0,0) is the (minx,miny) in user space of the bounding box of the object getting filled with the gradient, and (1,1) is the (maxx,maxy) corner of the bounding box. (Note: the bounding box represents the maximum extent of the shape of the object in X and Y with respect to the user coordinate system of the object exclusive of stroke-width.)
- **cx, cy, r** define the largest/outermost circle for the radial gradient. The gradient will be drawn such that the 100% gradient stop is mapped to the perimeter of this largest/outermost circle. (The default value for each attribute is 50%.) (??? Add drawing.)
- **fx, fy** define the focal point for the radial gradient. The gradient will be drawn such that the 0% gradient stop is mapped to (fx, fy). (The default value for each attribute is 50%.) (??? Add drawing.)
- **gradient-transform** has the same meaning as for linear gradients.

Percentages are allowed for **cx, cy, r, fx, fy**. For gradient-units="userspace", percentages represent values relative to the current viewport. For gradient-units="object-bbox", percentages represent values relative to the bounding box for the object.

(??? Need to include some drawings here showing these attributes)

9.4.3 Gradient Stops

The ramp of colors to use on a gradient is defined by the **<stop>** elements that are child elements to either the <lineargradient> element or the <radialgradient> element. Here is an example of the definition of a linear gradient that consists of a smooth transition from white-to-red-to-black:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Radial gradient example with three gradient stops
  </desc>
  <g>
    <defs>
      <radialgradient id="MyGradient">
         <stop offset="0%" style="color:white"/>
         <stop offset="50%" style="color:red"/>
         <stop offset="100%" style="color:black"/>
      </radialgradient>
    </defs>
    <circle style="fill: url(#MyGradient)" r="42"/>
  </g>
</svg>
```

Download this example

The **offset** attribute is either a <number> (usually ranging from 0 to 1) or a percentage (correspondingly usually ranging from 0% to 100%) which indicates where the gradient stop should be placed. For linear gradients, the offset attribute represents a location along the *gradient vector*. For radial gradients, it represents a percentage distance from (fx,fy) to the edge of the outermost/largest circle.

The **color** property indicates what color to use at that gradient stop. All valid CSS2 color property specifications are available.

An **opacity** property can be used to define the opacity of a given gradient stop.

Some notes on gradients:

- Gradient offset values less than 0 (or less than 0%) are rounded up to 0%. Gradient offset values greater than 1 (or greater than 100%) are rounded down to 100%.
- There needs to be at least two stops defined to have a gradient effect. If no stops are defined, then painting should occur as if 'none' were specified as the paint style. If one stop is defined, then paint with the solid color fill using the color defined for that gradient stop.
- Each gradient offset value should be equal to or greater than the previous gradient stop's offset value. If a given gradient stop's offset value is not equal to or greater than all previous offset values, then the offset value is adjusted to be equal to the largest of all previous offset values.
- If two gradient stops have the same offset value, then the latter gradient stop controls the color value at the overlap point.

9.5 Patterns

A pattern is used to fill or stroke an object using a pre-defined graphic object which can be replicated ("tiled") at fixed intervals in *x* and *y* to cover the areas to be painted.

Patterns are defined using a **<pattern>** element and then referenced by properties **fill:** and **stroke:**. The <pattern> element the same attributes as <symbol>, plus the following pattern-specific attributes:

- **pattern-units={ userspace | object-bbox }**. Defines the coordinate system for attributes **x, y, width, height**. If **pattern-units="userspace"** (the default), **x, y, width, height** represent values in the current user coordinate system when the <pattern> element is defined. If **pattern-units="object-bbox"**, then **x, y, width, height** represent values in an abstract coordinate system where (0,0) is the (minx,miny) in user space of the bounding box of the object getting filled with the pattern, and (1,1) is the (maxx,maxy) corner of the bounding box. (Note: the bounding box represents the maximum extent of the shape of the object in X and Y with respect to the user coordinate system of the object exclusive of stroke-width.)
- **x, y, width, height** indicate how the pattern tiles should be placed and spaced and represent coordinates and values in the coordinate space specified by **pattern-units**.
- **pattern-transform** contains the definitions of an optional additional transformation from the pattern coordinate system onto the target coordinate system (i.e., userspace or object-bbox). This allows for things such as skewing the pattern tiles. **pattern-transform** can take on the same values as the transform attribute.

An example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in" >
  <defs>
```

```
    <pattern id="TrianglePattern"
             pattern-units="userspace"
             x="0" y="0" width="25" height="25"
             pattern-transform="skew-x(45)"
             fit-bbox="0 0 10 10" >
      <path d="M 0 0 L 10 0 L 5 10 z" />
  </defs>
  <!-- Fill this ellipse with the above pattern -->
  <ellipse style="fill: url(#TrianglePattern)" rx="40" ry="27" />
</svg>
```

Download this example

9.6 Inheritance of Painting Properties

The values of any of the painting properties described in this chapter can be inherited from a given object's parent. Painting, however, is always done on each leaf-node individually, never at the <g> level. Thus, for the following SVG, two distinct gradients are painted (one for each rectangle):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Gradients apply to leaf nodes
  </desc>
  <g>
    <defs>
      <lineargradient id="MyGradient">
        <stop offset="0%" style="color:#F60"/>
        <stop offset="70%" style="color:#FF6"/>
      </lineargradient>
    </defs>
    <g style="fill: url(#MyGradient)">
      <rect width="20" height="15.8"/>
      <rect width="35" height="8"/>
   </g>
  </g>
</svg>
```

Download this example

10 Color

10.1 Introduction

All SVG colors are specified in the sRGB color space (see [SRGB]). At a minimum, SVG user agents should conform to the color behavior requirements specified in the Colors chapter of the CSS2 specification (see [CSS2]).

Additionally, SVG documents can specify an alternate color specification using an ICC profiles (see [ICC32]). If ICC-based colors are provided and the SVG user agent support ICC color, then the ICC-based color takes precedence over the sRGB color specification.

For more on specifying color properties, refer to the descriptions of the 'fill' property and the 'stroke' property.

10.2 The 'icc-profile' Property

The **'icc-profile'** property identifies the ICC profile which should be used to process all <icc-color> definitions within the current object.

'icc-profile'

Value:	sRGB | <profile-name> [<uri>] | inherit
Initial:	sRGB
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual

sRGB

Indicates that the currently active ICC profile is sRGB.

<profile-name> [<uri>]

Provides a **<profile-name>** for the profile which is present mainly for caching and performance reasons (e.g., if the profile is already installed on the user's system or is already in use, then don't download the profile from its URL). The **<uri>** indicates the location of the profile itself. Although optional, the **<uri>** is recommended in order to guarantee that the given profile will be available to the end user.

11 Paths

11.1 Introduction

Path objects are used to represent an outline which can be filled, stroked (see Filling, Stroking and Paint Servers) or used as a clipping path (see Clipping, Masking and Compositing), or for any combination of the three.

A path is described using the concept of a current point. In an analogy with drawing on paper, the current point can be thought of as the location of the pen. The position of the pen can be changed, and the outline of a shape (open or closed) can be traced by dragging the pen in either straight lines or curves.

Paths represent an outline of an object which is defined in terms of *moveto* (set a new current point), *lineto* (draw a straight line), *curveto* (draw a curve using a cubic bezier), *arc* (elliptical or circular arc) and *closepath* (close the current shape by drawing a line to the last *moveto*) elements. Compound paths (i.e., a path with subpaths, each consisting of a single *moveto* followed by one or more line or curve operations) are possible to allow effects such as "donut holes" in objects.

11.2 Path Data

11.2.1 General information about path data

A path is defined by including a **<path>** element which contains a **d="(path data)"** attribute, where the **d** attribute contains the *moveto*, *line*, *curve* (both cubic and quadratic beziers), *arc* and *closepath* instructions. The following example specifies a path in the shape of a triangle. (The **M** indicates a *moveto*, the **L**'s indicate *lineto*'s, and the **z** indicates a *closepath*:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
     xmlns = 'http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
  <path d="M 100 100 L 140 100 L 120 140 z"/>
</svg>
```

Download this example

A **<path>** element can also contain child **<data>** elements which also contain **d="(path data)"** attributes. This ability to have child **<data>** elements allows for very long path data strings to be broken up into more manageable smaller strings. Here is a <path> element equivalent to the <path> element above, but whose path data is divided into four pieces:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
```

```
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <path d="M 100 100">
    <data d="L 140 100"/>
    <data d="L 120 140"/>
    <data d="z"/>
  </path>
</svg>
```

Download this example

All **d=** attributes are concatenated together to yield a single path specification, with the **d=** attribute (if present) on the **<path>** element preceding the **d=** attributes on the **<data>** elements. Each **d=** attribute is restricted to 1023 characters of data. Path data commands cannot be broken across different **d=** attributes. For example, you cannot split a "moveto" command such that the X-coordinate in on one **d=** attribute and the Y-coordinate is on another.

The syntax of path data is very abbreviated in order to allow for minimal file size and efficient downloads, since many SVG files will be dominated by their path data. Some of the ways that SVG attempts to minimize the size of path data are as follows:

- All instructions are expressed as one character (e.g., a *moveto* is expressed as an **M**)
- Superfluous white space and separators such as commas can be eliminated (e.g., "M 100 100 L 200 200" contains unnecessary spaces and could be expressed more compactly as "M100 100L200 200")
- The command letter can be eliminated on subsequent commands if the same command is used multiple times in a row (e.g., you can drop the second "L" in "M 100 200 L 200 100 L -100 -200" and use "M 100 200 L 200 100 -100 -200" instead)
- Relative versions of all commands are available (upper case means absolute coordinates, lower case means relative coordinates)
- Alternate forms of *lineto* are available to optimize the special cases of horizontal and vertical lines (absolute and relative)
- Alternate forms of *curve* are available to optimize the special cases where some of the control points on the current segment can be determined automatically from the control points on the previous segment

The path data syntax is a prefix notation (i.e., commands followed by parameters). The only allowable decimal point is a period (".") and no other delimiter characters are allowed. (For example, the following is an invalid numeric value in a path data stream: "13,000.56". Instead, you should say: "13000.56".)

The following sections list the commands.

11.2.2 The "moveto" commands

The "moveto" commands (**M** or **m**) establish a new current point. The effect is as if the "pen" were lifted and moved to a new location. A path data segment must begin with either one of the "moveto" commands or one of the "arc" commands. Subsequent "moveto" commands (i.e., when the "moveto" is not the first command) represent the start of a new *subpath*:

Command	Name	Parameters	Description

M (absolute) **m** (relative)	moveto	(x y)*	Start a new sub-path at the given (x,y) coordinate. **M** (uppercase) indicates that absolute coordinates will follow; **m** (lowercase) indicates that relative coordinates will follow. If a relative moveto (**m**) appears as the first element of the path, then it is treated as a pair of absolute coordinates. If a moveto is followed by multiple pairs of coordinates, the subsequent pairs are treated as implicit lineto commands.

11.2.3 The "closepath" command

The "closepath" (**z**) causes an automatic straight line to be drawn from the current point to the initial point of the current subpath. "Closepath" differs in behavior from what happens when "manually" closing a subpath via a "lineto" command in how 'stroke-linejoin' and 'stroke-linecap' are implemented. With "closepath", the end of the final segment of the subpath is "joined" with the start of the initial segment of the subpath using the current value of 'stroke-linejoin' . If you instead "manually" close the subpath via a "lineto" command, the start of the first segment and the end of the last segment are not joined but instead are each capped using the current value of 'stroke-linecap':

Command	Name	Parameters	Description
z	closepath	(none)	Close the current subpath by drawing a straight line from the current point to current subpath's most recent starting point (usually, the most recent moveto point).

11.2.4 The "lineto" commands

The various "lineto" commands draw straight lines from the current point to a new point:

Command	Name	Parameters	Description
L (absolute) **l** (relative)	lineto	(x y)*	Draw a line from the current point to the given (x,y) coordinate which becomes the new current point. **L** (uppercase) indicates that absolute coordinates will follow; **l** (lowercase) indicates that relative coordinates will follow. A number of coordinates pairs may be specified to draw a polyline. At the end of the command, the new current point is set to the final set of coordinates provided.

H (absolute) **h** (relative)	horizontal lineto	x*	Draws a horizontal line from the current point (cpx, cpy) to (x, cpy). **H** (uppercase) indicates that absolute coordinates will follow; **h** (lowercase) indicates that relative coordinates will follow. Multiple x values can be provided (although usually this doesn't make sense). At the end of the command, the new current point becomes (x, cpy) for the final value of x.
V (absolute) **v** (relative)	vertical lineto	y*	Draws a vertical line from the current point (cpx, cpy) to (cpx, y). **V** (uppercase) indicates that absolute coordinates will follow; **v** (lowercase) indicates that relative coordinates will follow. Multiple y values can be provided (although usually this doesn't make sense). At the end of the command, the new current point becomes (cpx, y) for the final value of y.

11.2.5 The curve commands

These three groups of commands that draw curves:

- Cubic bezier commands (**C**, **c**, **S** and **s**). A cubic bezier segment is defined by a start point, an end point, and two control points.
- Quadratic bezier commands (**Q**, **q**, **T** and **T**). A quadratic bezier segment is defined by a start point, an end point, and one control point.
- Elliptical arc commands (**A**, **a**, **B** and **b**). An elliptical arc segment draws a segment of an ellipse defined by the formulas:

  ```
  x = cx + rx * cos(theta)
  y = cy + ry * sin(theta)
  ```

  where the elliptical arc is drawn as a sweep for every possible *theta* between a given start angle and end engle.

The cubic bezier commands are as follows:

Command	Name	Parameters	Description

C (absolute) **c** (relative)	curveto	(x1 y1 x2 y2 x y)*	Draws a cubic bezier curve from the current point to (x,y) using (x1,y1) as the control point at the beginning of the curve and (x2,y2) as the control point at the end of the curve. **C** (uppercase) indicates that absolute coordinates will follow; **c** (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybezier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybezier.
S (absolute) **s** (relative)	shorthand/smooth curveto	(x2 y2 x y)*	Draws a cubic bezier curve from the current point to (x,y). The first control point is assumed to be the reflection of the second control point on the previous command relative to the current point. (If there is no previous command or if the previous command was not an C, c, S or s, assume the first control point is coincident with the current point.) (x2,y2) is the second control point (i.e., the control point at the end of the curve). **S** (uppercase) indicates that absolute coordinates will follow; **s** (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybezier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybezier.

The quadratic bezier commands are as follows:

Command	Name	Parameters	Description

Q (absolute) **q** (relative)	quadratic bezier curveto	(x1 y1 x y)*	Draws a quadratic bezier curve from the current point to (x,y) using (x1,y1) as the control point. **Q** (uppercase) indicates that absolute coordinates will follow; **q** (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybezier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybezier.
T (absolute) **t** (relative)	Shorthand/smooth quadratic bezier curveto	(x y)*	Draws a quadratic bezier curve from the current point to (x,y). The control point is assumed to be the reflection of the control point on the previous command relative to the current point. (If there is no previous command or if the previous command was not an Q, q, T or t, assume the control point is coincident with the current point.) **T** (uppercase) indicates that absolute coordinates will follow; **t** (lowercase) indicates that relative coordinates will follow. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybezier.

The elliptical arc commands are as follows:

Command	Name	Parameters	Description

A (absolute) **a** (relative)	elliptical arc	(rx ry x-axis-rotation large-arc-flag sweep-flag x y)*	Draws an elliptical arc from the current point to (**x**, **y**). The size and orientation of the ellipse is defined two radii (**rx**, **ry**) and an **x-axis-rotation**, which indicates how the ellipse as a whole is rotated relative to the current coordinate system. The center (**cx**, **cy**) of the ellipse is calculated automatically to satisfy the constraints imposed by the other parameters. **large-arc-flag** and **sweep-flag** contribute to the automatic calculations and help determine how the arc is drawn.

The elliptical arc command draws a section of an ellipse which meets the following constraints:

- the arc starts at the current point
- the arc ends at point (**x**, **y**)
- the ellipse has the two radii (**rx**, **ry**)
- the X-axis of the ellipse is rotated by **x-axis-rotation** relative to the X-axis of the current coordinate system.

For most situations, there are actually four different arcs (two different ellipses, each with two different arc sweeps) that satisfy these constraints: (Pictures will be forthcoming in a future version of the spec) **large-arc-flag** and **sweep-flag** indicate which one of the four arcs should be drawn, as follows:

- Of the four candidate arc sweeps, two will represent an arc sweep of greater than or equal to 180 degrees (the "large-arc"), and two will represent an arc sweep of less than or equal to 180 degrees (the "small-arc"). If **large-arc-flag** is '1', then one of the two larger arc sweeps will be chosen; otherwise, if **large-arc-flag** is '0', one of the smaller arc sweeps will be chosen,
- If **sweep-flag** is '1', then the arc will be drawn in a "positive-angle" direction (i.e., the ellipse formula x=**cx**+**rx***cos(theta) and y=**cy**+**ry***sin(theta) is evaluated such that theta starts at an angle corresponding to the current point and increases positively until the arc reaches (x,y)). A value of 0 causes the arc to be drawn in a "negative-angle" direction (i.e., theta starts at an angle value corresponding to the current point and decreases until the arc reaches (x,y)).

(We need examples to illustrate all of this! Here is one for the moment. Suppose you have a circle with center (5,5) and radius 2 and you wish to draw an arc from 0 degrees to 90 degrees. Then one way to achieve this would be `M 7,5 A 2,2 0 0 1 5,7`. In this example, you move to the "0 degree" location on the circle, which is (7,5), since the center is at (5,5) and the circle has radius 2. Since we have circle, the two radii are the same, and in this example both are equal to 2. Since our sweep is 90 degrees, which is less than 180, we set large-arc-flag to 0. We want to draw the sweep in a positive angle direction, so we set sweep-flag to 1. Since we want to draw the arc to the point which is at the 90 degree location of the circle, we set (x,y) to (5,7).)

11.2.6 The grammar for path data

(??? This requires clean-up and a more formal write-up on the terminology.) The following is the BNF for SVG paths. The following notation is used:

- *: 0 or more
- +: 1 or more
- ?: 0 or 1
- (): grouping
- |: separates alternatives
- double quotes surround literals

```
svg-path:
    wsp* subpaths?

subpaths:
    subpath
    | subpath subpaths

subpath:
    moveto subpath-elements?

subpath-elements:
    subpath-element-wsp
    | subpath-element-wsp subpath-elements

subpath-element-wsp:
    subpath-element wsp*

subpath-element:
    closepath
    | lineto
    | horizontal-lineto
    | vertical-lineto
    | curveto
    | smooth-curveto
    | quadratic-bezier-curveto
    | smooth-quadratic-bezier-curveto
    | elliptical-arc

moveto:
    ( "M" | "m" ) wsp* moveto-argument-sequence

moveto-argument-sequence:
    coordinate-pair
    | coordinate-pair lineto-argument-sequence

closepath:
    ("Z" | "z") wsp*

lineto:
    ( "L" | "l" ) wsp* lineto-argument-sequence

lineto-argument-sequence:
    coordinate-pair
    | coordinate-pair lineto-argument-sequence

horizontal-lineto:
    ( "H" | "h" ) wsp* horizontal-lineto-argument-sequence

horizontal-lineto-argument-sequence:
    horizontal-lineto-argument
    | horizontal-lineto-argument horizontal-lineto-argument-sequence

horizontal-lineto-argument:
    coordinate

vertical-lineto:
    ( "V" | "v" ) wsp* vertical-lineto-argument-sequence

vertical-lineto-argument-sequence:
    vertical-lineto-argument
    | vertical-lineto-argument vertical-lineto-argument-sequence

vertical-lineto-argument:
    coordinate

curveto:
    ( "C" | "c" ) wsp* curveto-argument-sequence

curveto-argument-sequence:
    curveto-argument
    | curveto-argument curveto-argument-sequence

curveto-argument:
    coordinate-pair coordinate-pair coordinate-pair

smooth-curveto:
    ( "S" | "s" ) wsp* smooth-curveto-argument-sequence

smooth-curveto-argument-sequence:
    smooth-curveto-argument
    | smooth-curveto-argument smooth-curveto-argument-sequence

smooth-curveto-argument:
    coordinate-pair coordinate-pair

quadratic-bezier-curveto:
    ( "Q" | "q" ) wsp* quadratic-bezier-curveto-argument-sequence

quadratic-bezier-curveto-argument-sequence:
    quadratic-bezier-curveto-argument
    | quadratic-bezier-curveto-argument
        quadratic-bezier-curveto-argument-sequence

quadratic-bezier-curveto-argument:
    coordinate-pair coordinate-pair

smooth-quadratic-bezier-curveto:
    ( "T" | "t" ) wsp* smooth-quadratic-bezier-curveto-argument-sequence

smooth-quadratic-bezier-curveto-argument-sequence:
    coordinate-pair
    | coordinate-pair smooth-quadratic-bezier-curveto-argument-sequence

elliptical-arc:
    ( "A" | "a" ) wsp* elliptical-arc-argument-sequence

elliptical-arc-argument-sequence:
    elliptical-arc-argument
    | elliptical-arc-argument elliptical-arc-argument-sequence

elliptical-arc-argument:
    nonnegative-number-comma-wsp nonnegative-number-comma-wsp number-comma-wsp
        flag-comma-wsp flag-comma-wsp coordinate-pair

coordinate-pair:
    coordinate coordinate

coordinate:
    number-comma-wsp

nonnegative-number-comma-wsp:
    nonnegative-number wsp* comma? wsp*

number-comma-wsp:
    number wsp* comma? wsp*

nonnegative-number:
    integer-constant
    | floating-point-constant

number:
    sign? integer-constant
    | sign? floating-point-constant

flag-comma-wsp:
    flag wsp* comma? wsp*

flag:
    "0" | "1"

comma:
    ","

integer-constant:
    digit-sequence

floating-point-constant:
    fractional-constant exponent?
    | digit-sequence exponent

fractional-constant:
    digit-sequence? "." digit-sequence
    | digit-sequence "."

exponent:
    ( "e" | "E" ) sign? digit-sequence

sign:
    "+" | "-"

digit-sequence:
    digit
    | digit digit-sequence

digit:
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

wsp:
    (#x20 | #x9 | #xD | #xA)+
```

11.3 Markers

To use a marker symbol for arrowheads or polymarkers, you need to define a **<marker>** element which defines the marker symbol and then refer to that <marker> element using the various marker properties (i.e., 'marker-start', 'marker-end', 'marker-mid' or 'marker') on the given <path> element or vector graphic shape. Here is an example which draws a triangular marker symbol that is drawn as an arrowhead at the end of a path:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
```

```
<svg width="4in" height="4in"
     fit-box-to-viewport="0 0 4000 4000" >
  <defs>
    <marker id="Triangle"
      fit-bbox="0 0 10 10" ref-x="0" ref-y="5"
      marker-width="1.25" marker-height="1.75"
      orient="auto">
      <path d="M 0 0 L 10 5 L 0 10 z" />
    </marker>
  </defs>
  <desc>Placing an arrowhead at the end of a path.
  </desc>
  <path d="M 1000 1000 L 2000 1000 L 3000 2000"
        style="fill:none; stroke:black; stroke-width:100;
        marker-end:url(#Triangle)" />
</svg>
```

Download this example

11.3.1 The <marker> element

The **<marker>** element defines the graphics that is to be used for drawing arrowheads or polymarkers on a given <path> element or vector graphic shape.

The **<marker>** element has all of the same attributes as the <symbol> element. Additionally, it has the following marker-specific attributes:

- **marker-units**, which can have values *stroke-width* (the default) or *userspace*. **marker-units** indicates how to interpret the values of **marker-width** and **marker-height** (described as follows). If **marker-units="userspace"**, then **marker-width** and **marker-height** represent values in the user coordinate system in place for the graphic object referencing the marker. If **marker-units="stroke-width"**, then **marker-width** and **marker-height** represent scale factors relative to the stroke width in place for graphic object referencing the marker.
- **marker-width** and **marker-height** represent the width and height, respectively, of the temporary viewport that is to be created when drawing the marker. Default values for both attributes is "3".
- **orient**, which can have values *auto* or *<angle>*. (The default value is an angle of zero degrees.) The **orient** attribute indicates how the marker should be rotated. A value of *auto* indicates that the marker should be oriented such that its positive X-axis is pointing in a direction that is the average of the ending direction of path segment going into the vertex and the starting direction of the path segment going out of the vertex. (Refer to Implementation Notes for a more thorough discussion directionality of path segments.) A value of *<angle>* represents a particular orient in the user space of the graphic object referencing the marker. For example, if a value of "0" is given, then the marker will be drawn such that its X-axis will align with the X-axis of the user space of the graphic object referencing the marker.

Markers are drawn such that their reference point (i.e., attributes **ref-x** and **ref-y**) is positioned at the given vertex.

11.3.2 Marker properties

'marker-start' defines the arrowhead or polymarker that should be drawn at the first vertex of the given **<path>** element or vector graphic shape. **'marker-end'** defines the arrowhead or polymarker that should be drawn at the final vertex. **'marker-mid'** defines the arrowhead or polymarker that should be drawn at every other vertex (i.e., every vertex except the first and last).

'marker-start', 'marker-end', marker-mid'

Value:	none | inherit | <uri>
Initial:	none
Applies to:	all elements
Inherited:	see Inheritance of Painting Properties below
Percentages:	N/A
Media:	visual

none

Indicates that no marker symbol should be drawn at the given vertex (vertices).

<uri>

The <uri> is an XPointer (???) reference to the ID of a <marker> element which should be used as the arrowhead symbol or polymarker at the given vertex (vertices). If the XPointer is not valid (e.g., it points to an object that is undefined or the object is not a <marker> element), then the marker(s) should not be drawn.

The **'marker'** property specifies the marker symbol that should be used for all points on the sets the value for all vertices on the given **<path>** element or vector graphic shape. It is a short-hand for the three individual marker properties:

'marker'

Value:	see individual properties
Initial:	see individual properties
Applies to:	all elements
Inherited:	see Inheritance of Painting Properties below
Percentages:	N/A
Media:	visual

11.3.3 Details on How Markers are Rendered

The following provides details on how markers are rendered:

- Markers are drawn after the given object is filled and stroked.
- Each marker is drawn on the path by first creating a temporary viewport such that the origin of the viewport coordinate system is at the given vertex and the axes are aligned according to the **orient** attribute on the **<marker>** element.
- The width and height of the viewport is established by evaluating the values of <marker-units>, <marker-width> and <marker-height> and calculating temporary values **computed-width** and **computed-height** in the user coordinate system of the object referencing the markers. **computed-width** and **computed-height** are used to determine the dimensions of the temporary viewport.
- The marker is drawn into the viewport.

For illustrative purposes, we'll repeat the marker example shown earlier:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
```

```
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="4in"
     fit-box-to-viewport="0 0 4000 4000" >
  <defs>
    <marker id="Triangle"
      fit-bbox="0 0 10 10" ref-x="0" ref-y="5"
      marker-width="1.25" marker-height="1.75"
      orient="auto">
      <path d="M 0 0 L 10 5 L 0 10 z" />
    </marker>
  </defs>
  <desc>Placing an arrowhead at the end of a path.
  </desc>
  <path d="M 1000 1000 L 2000 1000 L 3000 2000"
        style="fill:none; stroke:black; stroke-width:100;
        marker-end:url(#Triangle)" />
</svg>
```

Download this example

The rendering effect of the above file will be visually identical to the following:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="4in"
     fit-box-to-viewport="0 0 4000 4000" >
  <defs>
    <!-- Note: to illustrate the effect of "marker",
      replace "marker" with "symbol" and remove the various
      marker-specific attributes -->
    <symbol id="Triangle"
      fit-bbox="0 0 10 10" ref-x="0" ref-y="5">
      <path d="M 0 0 L 10 5 L 0 10 z" />
    </symbol>
  </defs>
  <desc>File which produces the same effect
      as the marker example file, but without
      using markers.
  </desc>
  <!-- The path draws as before, but without the marker properties -->
  <path d="M 1000 1000 L 2000 1000 L 3000 2000"
        style="fill:none; stroke:black; stroke-width:100" />

  <!-- The following logic simulates drawing a marker
       at final vertex of the path. -->

  <!-- First off, move the origin of the user coordinate system
       so that the origin is now aligned with the end point of the path. -->
  <g transform="translate(3000 2000)" >

    <!-- Now, rotate the coordinate system 45 degrees because
         the marker specified orient="auto" and the final segment
         of the path is going in the direction of 45 degrees. -->
    <g transform="rotate(45)" >

      <!-- Establish a new viewport with an <svg> element.
           The width/height of the viewport are 1.25 and 1.75 times
           the current stroke-width, respectively. Since the
           current stroke-width is 100, the viewport's width/height
           is 125 by 175. Apply the fit-box-to-viewport attribute
           from the <marker> element onto this <svg> element.
           Transform the marker symbol to align (ref-x,ref-y) with
           the origin of the viewport. -->
      <svg width="125" height="175"
           fit-box-to-viewport="0 0 10 10"
           transform="translate(0,-5)" >
        <!-- Expand out the contents of the <marker> element. -->
        <path d="M 0 0 L 10 5 L 0 10 z" />
      </svg>
    </g>
  </g>
</svg>
```

Download this example

11.4 Implementation Notes

A conforming SVG user agent must implement path rendering as follows:

- Error handling:
  - The general rule for error handling in path data is that the SVG user agent should render a <path> element up to (but not including) the path command containing the first error in the path data specification. This will provide a visual clue to the user/developer about where the error might be in the path data specification.
  - Wherever feasible, all SVG user agents should report all errors to the user. This rule will greatly discourage generation of invalid SVG path data.
- Markers, directionality and zero-length path segments:
  - If markers are specified, then a marker should be drawn on every applicable vertex, even if the given vertex is the end point of a zero-length path segment and even if "moveto" commands follow each other.
  - Certain line-capping and line-joining situations and markers require that a path segment have directionality at its start and end points. Zero-length path segments have no directionality. In these cases, the following algorithm should be used to establish directionality: to determine the directionality of the start point of a zero-length path segment, go backwards in the path data specification within the current subpath until you find a segment which has directionality at its end point (e.g., a path segment with non-zero length) and use its ending direction; otherwise, temporarily consider the start point to lack directionality. Similarly, to determine the directionality of the end point of a zero-length path segment, go forwards in the path data specification within the current subpath until you find a segment which has directionality at its start point (e.g., a path segment with non-zero length) and use its starting direction; otherwise, temporarily consider the end point to lack directionality. If the start point has directionality but the end point doesn't, then the end point should use the start point's directionality. If the end point has directionality but the start point doesn't, then the start point should use the end point's directionality. Otherwise, set the directionality for the path segment's start and end points to align with the positive X-axis in user space.
  - If **'stroke-linecap'** is set to **butt** and the given path segment has zero length, do not draw the linecap for that segment; however, do draw the linecap for zero-length path segments when **'stroke-linecap'** is set to either **round** or **square**. (This allows round and square dots to be drawn on the canvas.)
- 1023 character limitation on **d=** attributes:
  - A conforming implementation must treat any SVG document which has a **d=** attribute with more than 1023 characters as in error. The proper behavior is to draw all valid and complete path data commands which are completely specified within the 1023 character restriction and discard (i.e., do not draw) all other path data commands that follow,

including path data commands on subsequent <data> elements within the same <path> element.

- ❍ Changes to the path data stream via the DOM which cause the 1023 character restriction to be is not an error. A conforming implementation must shuffle path data commands with the various <path> and <data> elements to ensure that the 1023 character restriction is not violated by shifting entire path data commands from any **d=** that exceeds 1023 characters to the following <data> element until the 1023 character restriction is honored. If the shifting causes a subsequent **d=** attribute to be greater than 1023, then shift whole path data commands off the end of that **d=** attribute, and so on. If necessary, create additional <data> elements to accommodate the overflow.
- ❍ The S/s commands indicate that the first control point of the given cubic bezier segment should be calculated by reflecting the previous path segments second control point relative to the current point. The exact math that should be used is as follows. If the current point is (curx, cury) and the second control point of the previous path segment is (oldx2, oldy2), then the reflected point (i.e., (newx1, newy1), the first control point of the current path segment) is:

```
(newx1, newy1) = (curx - (oldx2 - curx), cury - (oldy2 - cury))
               = (2*curx - oldx2, 2*cury - oldy2)
```

- ❍ A non-positive radius value should be treated as an error.
- ❍ Unrecognized contents within a path data stream (i.e., contents that are not part of the path data grammar) should be treated as an error.

12 Other Vector Graphic Shapes

12.1 Introduction

SVG contains the following set of predefined graphic objects:

- <rect/> (a rectangle with optional rounding attributes rx and ry which represents the radii of an ellipse [axis-aligned with the rectangle] to use to round off the corners of the rectangle);
- <circle/>
- <ellipse/>
- <polyline/>
- <polygon/>
- <line/>

Mathematically, these shape elements are equivalent to the path objects that would construct the same shape. They may be stroked, filled and used as clip paths, and all the properties described above for paths apply equally to them.

For example, the following will draw a blue circle with a red outline:

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>This is a blue circle with a red outline
  </desc>
  <g>
    <circle style="fill: blue; stroke: red"
      cx="200" cy="200" r="100"/>
  </g>
</svg>
```

Download this example

This ellipse uses default values for the center.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>This is an ellipse, axis aligned and centered on the origin
  </desc>
  <g>
    <ellipse rx="85" ry="45"/>
  </g>
</svg>
```

Download this example

Here is a polyline; for comparison, the equivalent path element is also given.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>A sample polyline, and equivalent path
  </desc>
  <polyline points="20,20 50,100 200,80 70,300"/>
  <path d="M20,20 L50,100 L200,80 L70,300"/>
</svg>
```

Download this example

A polygon is exactly the same as a polyline, except that the figure is automatically closed.

The **points** attribute on <polyline> and <polygon> is restricted to 1023 characters. Thus, these elements should only be used for graphics with a relatively small number of vertices. If you have geometry that has any possibility of exceeding the 1023 character limit, use the <path> element instead.

13 Text

13.1 Introduction

SVG allows text to be inserted into the drawing. All of the same styling attributes available for paths and vector graphic shapes are also available for text. (See Filling, Stroking and Paint Servers.)

13.2 The <text> element

The **<text>** element adds text to a drawing.

In the example below, the string "Hello, out there" is drawn in blue:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <text x=".5in" y="2in"
        style="fill:blue">Hello, out there</text>
</svg>
```

Download this example

<text> elements which are not positioned along a path (see Text On A Path below) can supply optional attributes *x*= and *y*= which represents the location in user coordinates (or Transformed CSS units or a percentage of the current viewport) for the initial *current text position*. Default values for *x*= and *y*= are zero in the user coordinate system. For left-aligned character strings, the placement point of the first glyph will be positioned exactly at the *current text position* and the glyph will be rendered based on the current set of text and font properties. After the glyph is rendered, the *current text position* is advanced based on the metrics of the glyph that were used along with the current set of text and font properties. For Roman text, the *current text position* is advanced along the x-axis by the width of that glyph. The cycle repeats until all characters in the <text> element have been rendered.

Within a <text> element, text and font properties and the *current text position* can be modified by including a **<tspan>** element. The <tspan> element can contain attributes **style** (which allows new visual rendering attributes to be specified) and the following attributes, which perform adjustments on the *current text position*:

- **x=** and **y=** - If provided, these attributes indicate a new (absolute) *current text position* within the user coordinate system.
- **dx=** and **dy=** - If provided, these attributes indicate a new (relative) *current text position* within the user coordinate system.

The **x=** and **dx=** values are cumulative; thus, if both are provided, the new *current text position* will have an X-coordinate of `x+dx`. Similarly, if both **x=** and **dx=** are provided, the new *current text position* will have a Y-coordinate of `y+dy`.

(Internationalization issues need to be addressed.)

A <tspan> element can also be used to specify that the character data from a different element should be used as character data for the given <tspan> element. In the example below, the first <text> element (with id="TextToUse") will not draw because it is part of a <defs> element. The second <text> element draws the string "ABC". The third text element draws the string "XYZ" because it includes a <tspan> element which is a reference to element "TextToUse", and that element's character data is "XYZ":

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <text id="TextToUse">XYZ</text>
  </defs>
  <text>ABC</text>
  <text>
    <tspan href="#TextToUse"/>
  </text>
</svg>
```

Download this example

If a <tspan> element has both an *href* attribute and its own character data, the character data from the *href* attribute draws before its own character data.

13.3 White space handling

SVG supports the standard XML attribute **xml:space** for determining how it should handle white space characters within a given **<text>** element's character data. **xml:space** is an inheritable attribute which can have one of two values:

- **default** (the initial/default value for xml:space) - When `xml:space="default"`, the SVG user agent will do the following. First, it will convert all carriage returns, linefeeds and tab characters into space characters. Then, it will drop all strip off all leading and trailing space characters. Finally, it will consolidate all contiguous space characters into a single space character. (This algorithm is very close to standard practice in HTML4 web browsers.)
- **preserve** - When `xml:space="preserve"`, the SVG user agent will do the following. It will convert all carriage returns, linefeeds and tab characters into space characters. Then, it will draw all space characters, including leading, trailing and multiple contiguous space characters. Thus, when drawn with `xml:space="preserve"`, the string `"a   b"` (three spaces between "a" and "b") will produce a larger separation between "a" and "b' than `"a b"` (one space between "a" and "b").

13.4 Text selection

SVG user agents running on systems with have clipboards for copy/paste operations and which are equipped with input devices that allow for text selection should support the selection of text from an SVG document and the ability to copy selected text strings to the system clipboard.

Within an SVG user agent which supports text selection and pointer devices such as a mouse, the following behavior should exist. When the pointing device is clicked over an SVG character and then dragged, then whenever the mouse goes over another character defined within the same <text> elements, all characters whose position in the document is between the initial character and the current character should be highlighted, no matter where they might be located on the canvas.

When feasible, generators of SVG should attempt to order their text strings to facilitate properly ordered text selection within SVG viewing applications such as Web browsers.

13.5 Text and font properties

(Descriptions (or references to the CSS2 spec) for all of the text and font properties from CSS2 go here. The following are SVG-specific text properties which go beyond the text and font properties already defined in CSS2:)

13.6 Ligatures and alternate glyphs

There are situations such as ligatures, special-purpose fonts (e.g., a font for music symbols) or alternate glyphs for Asian text strings where a different glyph should be used to render some text than the glyph which normally corresponds to the given character data. Also, the W3C Character Model (??? add link) requires early normalization of character data to facilitate meaningful and unambiguous exchange of character data and correct comparisons between character strings. The W3C Character Model will bring about many common situations where the normalized character data will be different than the glyphs which the user want to use to render that character data.

To allow for control over the glyphs used to render particular character data, the **'altglyph'** property is available.

'altglyph'

Value:	unicode(<value>) | glyphname(<string>) | glyphid(<value>) | ROS(<value>) cid(<value>) | inherit
Initial:	none
Applies to:	<text> elements
Inherited:	yes
Percentages:	N/A
Media:	visual

unicode(<value>))

where <value> indicates a string of Unicode characters that should replace the text within the <text> element

glyphname(<string>))

where <string> provides a string of which is the name of the glyph that should be used to replace the text within the <text> element

glyphid(<value>))

where <value> a string of which is numeric ID/index of the glyph that should be used to replace the text within the <text> or <t> element

ROS and ***cid***

are required for Web fonts in OpenType/CFF format and operate similar to *glyphid*

13.7 Text on a path

In addition to text drawn in a straight line, SVG also includes the ability to place text along the shape of a <path> element. The basic algorithm is such that each glyph is positioned at a specific point on the path, with the glyph getting rotated such that the baseline of the glyph is either parallel or tangent to the tangent of the curve at the given point. Here are some examples of text on a path:

(??? include some drawings here)

The following is a simple example which illustrates many of the basic concepts:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>Simple text on a path
  </desc>
  <path id="MyPath" style="visibility: hidden"
        d="M 100 100 C 125 125 175 125 200 100" />
  <text>
    <textpath href="#MyPath">Text on path</textpath>
  </text>
</svg>
```

Download this example

The **start-offset** attribute defines the distance from the start of the path where the first character should be positioned.

Various CSS properties control aspects of text drawn along a path. The standard set of text and font properties from CSS2 apply, including **'text-align'** and **'vertical-align'**. Additional, the following additional properties control how the text is formatted and rendered:

- **text-direction:** Possible values are: *natural*, which indicates that text advance from character to character should be dictated by the natural text direction from the font, *l-to-r* (text should start toward the beginning ofthe path and proceed toward the end of the path and the baseline of the glyphs should be parallel to the path), *r-to-l* (text should start at the end of the path and the baseline of the glyphs should be parallel to the path), *t-to-b* (text should start at the beginning of the path and proceed toward the end of the path and the baseline of the glyphs should be perpendicular to the path) and *b-to-t* (text should start at the end of the path and the baseline of the glyphs should be parallel to the path),
- **orient-to-path:** Possible values are *true* and *false*, with a default of true. If true, the glyph/symbol is rotated to have its coordinate system line up with the tangent of the curve.
- **text-transform:** which defines a transformation that should be applied (conceptually) after the given glyph/symbol is properly sized, placed and oriented by all of the other CSS text properties. The options for this property are the same as the **transform:** property (??? add link).

Text on a path opens the possibility of significant implementation differences due to different methods of calculating distance along a path. In order to ensure that distance calculations are sufficiently precise, the following two attributes are available on <path> elements. (??? Obviously, this section needs to be moved to the <path> element section.)

- **flatness=** A distance measurement in either local coordinates or in CSS units which serves as a hint for the allowable error tolerance allowable in approximating a curve with a series line segments. Commercial implementations should honor this attribute.
- **nominal-length=** The distance measurement (A) for the given <path> element computed at authoring time. The SVG user agent should compute its own distance measurement (B). The SVG user agent should then scale all distance-along-a-curve computations by A divided by B.

(??? Insert drawings here)

13.7.1 Vector graphics along a path

SVG's text on a path features set has been generalized to allow for arbitrary SVG along a path, by adding the **use** element as a valid child of **text**.

14 Images

Images are specified using the **<image>** element, which includes all relevant attributes from the HTML <img> element and is used in similar ways. Textual descriptions are held in a **<desc>** sub element, rather than in the alt attribute as with HTML.

The SVG **<image>** element has the additional attributes *x* and *y* (both have default values of zero) to indicate the location of the left/top corner of the image in user space.

The SVG **<image>** element is defined as an XLink and thus has the attribute *href*. (Note that the XLink specification is currently under development and is subject to change. The SVG working group will track and rationalize with XLink as it evolves.)

The default values for *width* and *height* are 1 user unit. A valid example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>This graphic links to an external image
  </desc>
  <image x="200" y="200" style="width: 100px; height: 100px"
  href="myimage.png">
    <title>My image</title>
  </image>
</svg>
```

Download this example

A well-formed example:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
 xmlns='http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
  <desc>This links to an external image
  </desc>
  <image x="200" y="200" style="width: 100px; height: 100px"
   xml:link = 'simple' show = 'embed' actuate = 'auto'
  href="myimage.png">
    <title>My image</title>
  </image>
</svg>
```

Download this example

For more information, see the <image> element.

15 Filter Effects

15.1 Introduction

A model for adding declarative raster-based rendering effects to a 2D graphics environment is presented. As a result, the expressiveness of the traditional 2D rendering model is greatly enhanced, while still preserving the device independence, scalability, and high level geometric description of the underlying graphics.

15.2 Background

On the Web, many graphics are presented as bitmap images in gif, jpg, or png format. Among the many disadvantages of this approach is the general difficulty of keeping the raster data in sync with the rest of the Web site. Many times, a web site designer must resort to a bitmap editor to simply change the title of a button. As the Web gets more dynamic, we desire a way to describe the "piece parts" of a site in a more flexible format. This chapter describes SVG's declarative filter effects model, which when combined with the 2D power of SVG can describe much of the common artwork on the web in such a way that client-side generation and alteration can be performed easily.

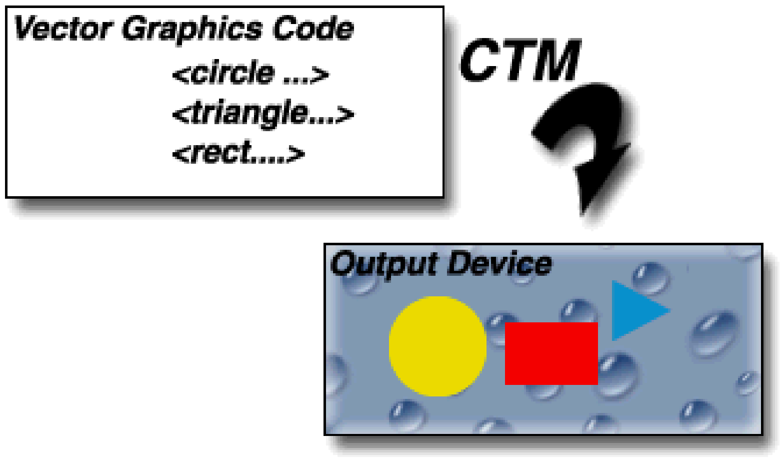
15.3 Basic Model

The filter effects model consists of a set of filtering operations (called "processing nodes" in the descriptions below) on one or more graphic primitives. Each processing node takes a set of graphics primitives as input, performs some processing, and generates revised graphics primitives as output. Because nearly all of the filtering operations are some form of image processing, in almost all cases the output from most processing nodes consists of a single RGBA image.

For example, a simple filter could replace one graphic by two -- by adding a black copy of original offset to create a drop shadow. In effect, there are now two layers of graphics, both with the same original set of graphics primitives. In this example, the bottommost shadow layer could be blurred and become a raster layer, while the topmost layer could remain as higher-order graphics primitives (e.g., text or vector objects). Ultimately, the two layers are composited together and rendered into the background.

Filter effects introduce an additional step into the traditional 2D graphics pipeline. Consider the traditional 2D graphics pipeline:

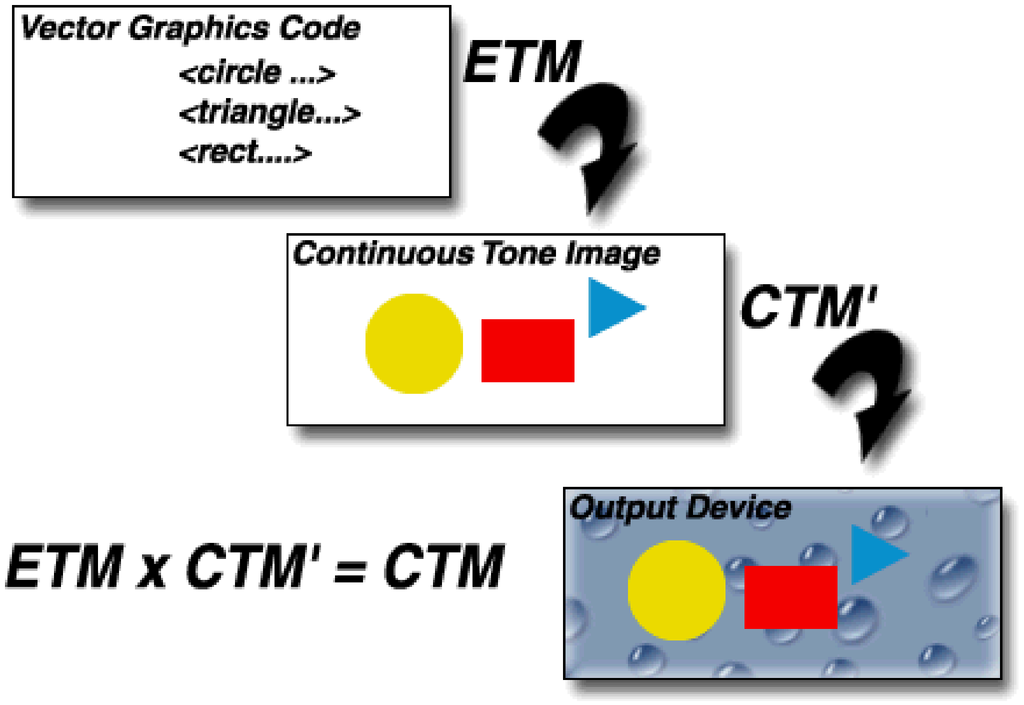
Traditional 2D graphics pipeline



Vector graphics primitives are specified abstractly and rendered onto the output device through a geometric transformation called the *current transformation matrix*, or *CTM*. The CTM allows the vector graphics code to be specified in a device independent coordinate system. At rendering time, the CTM accounts for any differences in resolution or orientation of the input vector description space and the device coordinate system. According to the "painter's model", areas on the device which are outside of the vector graphic shapes remain unchanged from their previous contents (in this case the droplet pattern).

Consider now, altering this pipeline slightly to allow rendering the graphics to an intermediate continuous tone image which is then rendered onto the output device in a second pass:

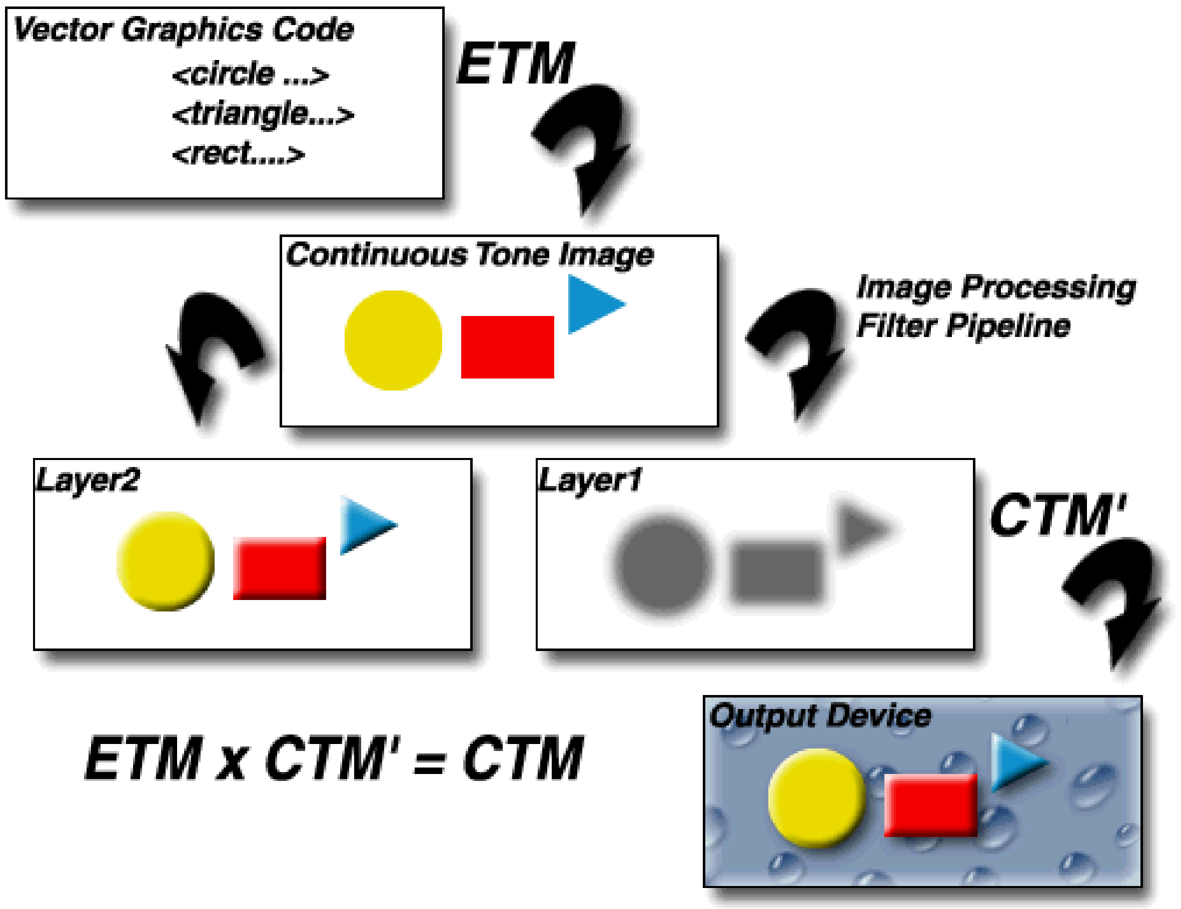
Rendering via Continuous Tone Intermediate Image



We introduce a new transformation matrix called the *Effect Transformation Matrix*, or *ETM*. Vector primitives are rendered via the ETM onto an intermediate continuous tone image. This image is then rendered onto the output device using the standard 2D imaging path via a modified transform, *CTM'*, such that the net effect of *ETM* followed by *CTM'* is equivalent to the original CTM. It is important to note that the intermediate continuous tone image contains coverage information so that non rendered parts of the original graphic are transparent in the intermediate image and remain unaffected on the output device, as required by the painter's model. A physical analog to this process is to imagine rendering the vector primitives onto a sheet of clear acetate and then transforming and overlaying the acetate sheet onto the output device. The resulting imaging model remains as device-independent as the original one, except we are now using the 2D imaging model itself to generate images to render.

So far, we really haven't added any new expressiveness to the imaging model. What we *have* done is reformulated the traditional 2D rendering model to allow an intermediate continuous tone rasterization phase. However, now we can extend this further by allowing the application of image processing operations on the intermediate image, still without sacrificing device independence. In our model, the intermediate image can be operated on by a number of image processing operations which can effect both the color and coverage channels. The resulting image(s) are then rendered onto the device in the same way as above.

Rendering via Continuous Tone Intermediate Step with Image Processing



In the picture above, the intermediate set of graphics primitives was processed in two ways. First a simple bump map lighting calculation was applied to add a 3D look, then another copy of the original layer was offset, blurred and colored black to form a shadow. The resulting transparent layers were then rendered via the painter's model onto the output device.

15.4 Defining and Invoking a Filter Effect

Filter effects are defined by a **<filter>** element with an associated ID. Filter effects are applied to elements which have a **filter:** property which reference a **<filter>** element. Here is an example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <defs>
    <filter id="CoolTextEffect">
      <!-- Definition of filter goes here -->
    </filter>
  </defs>
  <text style="filter:url(#CoolTextEffect)">Text with a cool effect</text>
</svg>
```

Download this example

When applied to grouping elements such as <g>, the **filter:** property applies to the contents of the group as a whole. The effect should be as if the group's children did not render to the screen directly but instead just added their resulting graphics primitives to the group's *graphics display list (GDL)*, which is then passed to the filter for processing. After the group filter is processed, then the result of the filter should be rendered to the target device (or passed onto a parent grouping element for further processing in cases such as when the parent has its own group filter).

The **<filter>** element consists of a sequence of *processing nodes* which take a set of graphics primitives as input, apply filter effects operations on the graphics primitives, and produce a modified set of graphics primitives as output. The processing nodes are executed in sequential order. The resulting set of graphics primitives from the final processing node (*feMerge* in the example below) represents the result of the filter. Here is an example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
  <filter id="Shadow">
    <feGaussianBlur     in="SourceAlpha"
                        radius="3"
                        nodeid="blurredAlpha" />
    <feOffset           in="blurredAlpha"
                        dx="2" dy="1"
                        nodeid="offsetBlurredAlpha" />
    <feDiffuseLighting  in="blurredAlpha"
                        diffuse-constant=".5"
                        nodeid="bumpMapDiffuse" >
      <feDistantLight/>
    </feDiffuseLighting>
    <feComposite        in="bumpMapDiffuse" in2="SourcePaint"
                        operator="arithmetic" k1="1"
                        nodeid="litPaint" />
    <feSpecularLighting in="blurredAlpha"
                        specular-constant=".5"
                        specular-exponent="10"
                        light-color="feDistantLight"
                        nodeid="bumpMapSpecular" >
      <feDistantLight/>
    </feSpecularLighting>
    <feComposite        in="litPaint" in2="bumpMapSpecular"
                        operator="arithmetic" k2="1" k3="1"
                        nodeid="litPaint" />
    <feComposite        in="litPaint"
                        in2="SourceAlpha"
                        mode="AinB"
                        nodeid="litPaint" />
    <feMerge>
      <feMergeNode in="litPaint" />
      <feMergeNode in="offsetBlurredAlpha" />
    </feMerge>
  </filter>

  <text style="font-size:36; fill:red; filter:url(#Shadow)"
        x="10" y="250">Shadowed Text</text>
</svg>
```

Download this example

For most processing nodes, the **in** (and sometimes **in2**) attribute identifies the graphics which serve as input and the **nodeid** attribute gives a name for the resulting output. The **in** and **in2** attributes can point to either:

- A built-in keyword. In the example above, the <feGaussianBlur> processing node specifies keyword *SourceGraphic*, which indicates that the original set of graphics primitives available at the start of the filter should be used as input to the processing node.
- A reference to an earlier **nodeid**. In the example above, the <feOffset> processing node refers to the most recent processing node whose nodeid is *blurredAlpha*. (In this case, that would be the <feGaussianBlur> processing node.)

The default value for **in** is the output generated from the previous processing node. In those cases where the output from a given processing node is used as input only by the next processing node, it is not necessary to specify the **nodeid** on the previous processing node or the **in** attribute on the next processing node. In the example above, there are a few cases (show highlighted) where **nodeid** and **in** did not have to be provided.

Filters *do not* use XML IDs for **nodeid**s; instead, **nodeid**s can be any arbitrary attribute string value. **nodeid**s only are meaningful within a given <filter> element and thus have only local scope. If a nodeid appears multiple times within a given <filter> element, then a reference to that nodeid will use the closest preceding processing node with the given nodeid. Forward references to nodeids are invalid.

15.5 Filter Effects Region

A <filter> element can define a region on the canvas on which a given filter effect applies and can provide a resolution for any intermediate continuous tone images used in to process any raster-based processing nodes. The <filter> element has the following attributes:

- **filter-units={ userspace | object-bbox }**. Defines the coordinate system for attributes **x, y, width, height**. If **filter-units="userspace"** (the default), **x, y, width, height** represent values in the current user coordinate system in place at the time when the <filter> element is defined. If **filter-units="object-bbox"**, then **x, y, width, height** represent values in an abstract coordinate system where (0,0) is the (minx,miny) in user space of the tight bounding box of the object referencing the filter, and (1,1) is the (maxx,maxy) corner of the bounding box. (Note: the bounding box represents the maximum extent of the shape of the object in X and Y with respect to the user coordinate system of the object exclusive of stroke-width.)
- **x, y, width, height**, which indicate the rectangle for the largest possible offscreen buffer, where the values are either relative to the current user coordinate system (if filter-units="userspace") or relative to the current object (if filter-units="target-object"). Note that the clipping path used to render any graphics within the filter will consists of the intersection of the current clipping path associated with the given object and the rectangle defined by **x, y, width, height**. The default values for **x, y, width, height** are 0%, 0%, 100% and 100%, respectively.
- **filter-res**, which has the form `x-pixels [y-pixels])` and indicates the width/height of the intermediate images in pixels. If not provided, then a reasonable default resolution appropriate for the target device will be used. (For displays, an appropriate display resolution, preferably the current display's pixel resolution, should be the default. For printing, an appropriate common printer resolution, such as 400dpi, should be the default.)

For performance reasons on display devices, it is recommended that the filter effect region is designed to match pixel-for-pixel with the background pixmap.

It is often necessary to provide padding space because the filter effect might impact bits slightly outside the tight-fitting bounding box on a given object. For these purposes, it is possible to provide negative percentage values for **x, y** and percentages values greater than 100% for **width, height**. For example, x="-10%" y="-10%" width="110%" height="110%".

15.6 Common Attributes

The following two attributes are available for all processing nodes (the exception is feMerge and feImage, which do not have an **in** attribute):

Common Attributes	*nodeid* Assigned name for this node. If supplied, then the GDL resulting after processing the given node is saved away and can be referenced as input to a subsequent processing node. *in* If supplied, then indicates that this processing node should use either the output of previous node as input or use some standard keyword to specify alternate input. Available keywords include: ● **SourceGraphic**. This built-in input represents the graphics elements that were the original input into the <filter> element. For raster effects processing nodes, the graphics elements will be rasterized into an initially clear RGBA raster in image space. Pixels left untouched by the original graphic will be left clear. The image is specified to be rendered in linear RGBA pixels. The alpha channel of this image captures any anti-aliasing specified by SVG. (Since the raster is linear, the alpha channel of this image will represent the exact percent coverage of each pixel.) ● **SourceAlpha**. Same as SourceGraphic except only the alpha channel is specified. The color channels of this image are implicitly black and are unaffected by any image processing operations. Again, pixels unpainted by the SourceGraphic will be 0. The SourceAlpha image will also reflects any Opacity settings in the SourceGraphic. If this option is used, then some implementations may need to rasterize the graphics elements in order to extract the alpha channel. ● **BackgroundImage** This built-in input represents an image snapshot of the rendering surface under the filter region at the time that the <filter> element was invoked. ● **BackgroundAlpha** Same as BackgroundImage except only the alpha channel is specified. ● **FillPaint** This image represents the color data specified by the current SVG rendering state, transformed to image space. The FillPaint image has conceptually infinite extent in image space. (Since it is usually either just a constant color, or a tile). Frequently this image is opaque everywhere, but it might not be if the "paint" itself has alpha, as in the case of an alpha gradient or transparent pattern. For the simple case where the source graphic represents a simple filled object, it is guaranteed that: `SourceGraphic = In(FillPaint,SourceAlpha)` where In(A,B) represents the resulting image of Porter-Duff compositing operation A in B (see below). ● **StrokePaint** Similar to FillPaint, except for the stroke color as specified in SVG. Again for the simple case where the source graphic represents stroked path, it is guaranteed that: `SourceGraphic = In(StrokePaint,SourceAlpha)` where In(A,B) represents the resulting image of Porter-Duff compositing operation A in B (see below).

15.7 Filter Processing Nodes

The following is a catalog of the individual processing nodes. Unless otherwise stated, all image filters operate on linear premultiplied RGBA samples. Filters which work more naturally on non premultiplied data (feColorMatrix and feComponentTransfer) will temporarily undo and redo premultiplication as specified. All raster effect filtering operations take 1 to N input RGBA images, additional attributes as parameters, and produce a single output RGBA image.

NodeType	**feBlend**
Processing Node-Specific Attributes	*mode*, One of the image blending modes (see table below). Default is: normal *in2*, The second image ("B" in the formulas) for the compositing operation.

Description	This filter composites two objects together using commonly used high-end imaging software blending modes. Performs the combination of the two input images pixel-wise in image space.

Implementation Notes

The compositing formula, expressed using premultiplied colors:

```
qr = qb*(1-qa) + qa*(1-qb) + qa*qb
cr = (1-qa)*cb + (1-qb)*ca + qa*qb*(Blend(ca/qa,cb/qb))
where:
  qr = Result opacity
  cr = Result color (RGB) - premultiplied
  qa = Opacity value at a given pixel for image A
  qb = Opacity value at a given pixel for image B
  ca = Color (RGB) at a given pixel for image A - premultiplied
  cb = Color (RGB) at a given pixel for image B - premultiplied
  Blend = Image compositing function, depending on the compositing mode
```

The following table provides the list of available image blending modes:

Image Blending Mode	Blend() function
normal	Ca
multiply	Ca*Cb
screen	1-(1-Ca)*(1-Cb)
darken	(to be provided later)
lighten	(to be provided later)

NodeType	**feColor**
Processing Node-Specific Attributes	*color*, RGBA color (floating point?)
Description	Creates an image with infinite extent filled with *color*

NodeType	**feColorMatrix**

Processing Node-Specific Attributes

type, string (one of: matrix, saturate, hue-rotate, luminance-to-alpha)

values

- For matrix: space-separated list of 20 element color transform (a00 a01 a02 a03 a04 a10 a11 ... a34). For example, the identity matrix could be expressed as:

  ```
  type="matrix"
  values="1 0 0 0 0  0 1 0 0 0  0 0 1 0 0  0 0 0 1 0"
  ```
- For saturate: one real number (0 to 1)
- For hue-rotate: one real number (degrees)
- Not applicable for luminance-to-alpha

Description

This filter performs

```
   | R' |     | a00 a01 a02 a03 a04 |   | R |

   | G' |     | a10 a11 a12 a13 a14 |   | G |

   | B' |  =  | a20 a21 a22 a23 a24 | * | B |

   | A' |     | a30 a31 a32 a33 a34 |   | A |

   | 1  |     |  0   0   0   0   1  |   | 1 |
```

for every pixel. The RGBA and R'G'B'A' values are automatically *non-premultiplied* temporarily for this operation.

The following shortcut definitions are provide for compactness. The following tables show the mapping from the shorthand form to the corresponding longhand (i.e., matrix with 20 values) form:

```
saturate value (0..100)

   s = value/100

   | R' |     |0.213+0.787s  0.715-0.715s  0.072-0.072s 0  0 |   | R |

   | G' |     |0.213-0.213s  0.715+0.285s  0.072-0.072s 0  0 |   | G |

   | B' |  =  |0.213-0.213s  0.715-0.715s  0.072+0.928s 0  0 | * | B |

   | A' |     |           0            0             0  1  0 |   | A |

   | 1  |     |           0            0             0  0  1 |   | 1 |

hue-rotate value (0..360)

   | R' |     | a00  a01  a02  0  0 |   | R |

   | G' |     | a10  a11  a12  0  0 |   | G |

   | B' |  =  | a20  a21  a22  0  0 | * | B |

   | A' |     | 0    0    0    1  0 |   | A |

   | 1  |     | 0    0    0    0  1 |   | 1 |

   where the terms a00, a01, etc. are calculated as follows:

      | a11 a12 a13 |    [+0.213 +0.715 +0.072]
      | a21 a22 a13 | =  [+0.213 +0.715 +0.072] +
      | a11 a12 a13 |    [+0.213 +0.715 +0.072]

                                  [+0.787 -0.715 -0.072]
          cos(hue-rotate value) * [-0.212 +0.285 -0.072] +
                                  [-0.213 -0.715 +0.928]

                                  [-0.213 -0.715+0.928]
          sin(hue-rotate value) * [+0.143 +0.140-0.283]
                                  [-0.787 +0.715+0.072]

   Thus, the upper left term of the hue matrix turns out to be:

         .213 + cos(hue-rotate value)*.787 - sin(hue-rotate value)*.213


luminance-to-alpha

   | R' |     |      0        0        0  0  0 |   | R |

   | G' |     |      0        0        0  0  0 |   | G |

   | B' |  =  |      0        0        0  0  0 | * | B |

   | A' |     |  0.299    0.587    0.114  0  0 |   | A |

   | 1  |     |      0        0        0  0  1 |   | 1 |
```

Implementation issues	These matrices often perform an identity mapping in the alpha channel. If that is the case, an implementation can avoid the costly undoing & redoing of the premultiplication for all pixels with A = 1.

NodeType	**feComponentTransfer**
Processing Node-Specific Attributes	None.
Processing Node-Specific Sub-Elements	Each <feComponentTransfer> element needs to have at most one each of the following sub-elements, each of which is an empty element: *<feFuncR>*, transfer function for red component *<feFuncG>*, transfer function for green component *<feFuncB>*, transfer function for blue component *<feFuncA>*, transfer function for alpha component Each of these sub-elements (i.e., *<feFuncR>*, *<feFuncG>*, *<feFuncB>*, *<feFuncA>*) can have the following attributes: Common parameters to all transfer modes: *type*, string (one of: identity, table, linear, gamma) Parameters specific to particular transfer modes: For *table*: *table-values*, list of real number values *v0,v1,...vn*. For *linear*: *slope*, real number value giving slope of linear equation. *intercept*, real number value giving Y-intercept of linear equation. For *gamma* (see descriptiong below for descriptions): *amplitude*, real number value. *exponent*, real number value. *offset*, real number value.

Description	This filter performs component-wise remapping of data as follows:
	`R' = feFuncR( R )`
	`G' = feFuncG( G )`
	`B' = feFuncB( B )`
	`A' = feFuncA( A )`
	for every pixel. The RGBA and R'G'B'A' values are automatically *non-premultiplied* temporarily for this operation.
	When *type="table"*, the transfer function consists of a linearly interpolated lookup table.
	k/N <= C < (k+1)/N => C' = vk + (C - k/N)*N * (vk+1 - vk)
	When *type="linear"*, the transfer function consists of a linear function describes by the following equation:
	C' = slope*C + offset
	When *type="gamma"*, the transfer function consists of the following equation:
	C' = amplitude*pow(C, exponent) + offset

Comments	This filter allows operations like brightness adjustment, contrast adjustment, color balance or thresholding. We might want to consider some predefined transfer functions such as identity, gamma, sRGB transfer, sine-wave, etc.

Implementation issues	Similar to the feColorMatrix filter, the undoing and redoing of the premultiplication can be avoided if feFuncA is the identity transform and A = 1.

NodeType	**feComposite**
Processing Node-Specific Attributes	*operator*, one of (*over, in, out, atop, xor, arithmetic*). Default is: over. *arithmetic-constants*, k1,k2,k3,k4 *in2*, The second image ("B" in the formulas) for the compositing operation.
Description	This filter performs the combination of the two input images pixel-wise in image space. *over, in, atop, out, xor* use the Porter-Duff compositing operations. For these operations, the extent of the resulting image can be affected. In other words, even if two images do not overlap in image space, the extent for over will essentially include the union of the extents of the two input images. *arithmetic* evaluates k1*i1*i2 + k2*i1 + k3*i2 + k4, using componentwise arithmetic with the result clamped between [0..1].
Comments	*arithmetic* are useful for combining the output from the feDiffuseLighting and feSpecularLighting filters with texture data. *arithmetic* is also useful for implementing *dissolve*.

NodeType	**feDiffuseLighting**
Processing Node-Specific Attributes	*result-scale* (Multiplicative scale for the result. This allows the result of the feDiffuseLighting nodeto represent values greater than 1) *surface-scale* height of surface when Ain = 1. *diffuse-constant* kd in Phong lighting model. Range 0.0 to 1.0 *light-color* RGB

Processing Node-Specific Sub-Elements	One of
	`<feDistantLight  azimuth= elevation= >`
	`<fePointLight    x= y= z= >`
	`<feSpotLight     x= y= z=`
	`points-at-x=`
	`points-at-y=`
	`points-at-z=`
	`specular-exponent=>`

	Light an image using the alpha channel as a bump map. The resulting image is an RGBA opaque image based on the light color with alpha = 1.0 everywhere. The lighting caculation follows the standard diffuse component of the Phong lighting model. The resulting image depends on the light color, light position and surface geometry of the input bump map. Color or texture is mean to be applied via a multiply (mul) composite operation.
	
	`Dr = (kd * N.L * Lr) / result-scale`
	`Dg = (kd * N.L * Lg) / result-scale`
	`Db = (kd * N.L * Lb) / result-scale`
	`Da = 1.0 / result-scale`
	
	where
	
	kd = diffuse lighting constant
	N = surface normal unit vector, a function of x and y
	L = unit vector pointing from surface to light, a function of x and y in the point and spot light cases
	Lr,Lg,Lb = RGB components of light, a function of x and y in the spot light case
	result-scale = overall scaling factor

Description	N is a function of x and y and depends on the surface gradient as follows:
	
	The surface described by the input alpha image Ain (x,y) is:
	`Z (x,y) = surface-scale * Ain (x,y)`
	
	Surface normal is calculated using the Sobel gradient 3x3 filter:
	`Nx (x,y)= - surface-scale * 1/4*(( I(x+1,y-1) + 2*I(x+1,y)`
	`+ I(x+1,y+1))`
	`                                  -  (I(x-1,y-1) + 2*I(x-1,y)`
	`+ I(x-1,y+1)))`
	`Ny (x,y)= - surface-scale * 1/4*(( I(x-1,y+1) + 2*I(x,y+1) + I(x+1,y+1))`
	
	`                                  -  (I(x-1,y-1) + 2*I(x,y-1)`
	`+ I(x+1,y-1)))`
	`Nz (x,y) = 1.0`
	
	`N = (Nx, Ny, Nz) / Norm((Nx,Ny,Nz))`
	
	L, the unit vector from the image sample to the light is calculated as follows:
	
	For Infinite light sources it is constant:
	`Lx = cos(azimuth)*cos(elevation)`
	`Ly = -sin(azimuth)*cos(elevation)`
	`Lz = sin(elevation)`
	
	For Point and spot lights it is a function of position:
	`Lx = Lightx - x`
	`Ly = Lighty - y`
	`Lz = Lightz - Z(x,y)`
	
	`L = (Lx, Ly, Lz) / Norm(Lx, Ly, Lz)`
	
	where Lightx, Lighty, and Lightz are the input light position.
	
	Lr,Lg,Lb, the light color vector is a function of position in the spot light case only:
	`Lr = Lightr*pow((-L.S),specular-exponent)`
	`Lg = Lightg*pow((-L.S),specular-exponent)`
	`Lb = Lightb*pow((-L.S),specular-exponent)`
	
	where S is the unit vector pointing from the light to the point (points-at-x, points-at-y, points-at-z) in the x-y plane:
	`Sx = points-at-x - Lightx`
	`Sy = points-at-y - Lighty`
	`Sz = points-at-z - Lightz`
	
	`S = (Sx, Sy, Sz) / Norm(Sx, Sy, Sz)`
	
	If L.S is positive no light is present. (Lr = Lg = Lb = 0)

Comments	This filter produces a light map, which can be combined with a texture image using the multiply term of the *arithmetic* <Composite> compositing method. Multiple light sources can be simulated by adding several of these light maps together before applying it to the texture image.

NodeType	**feDisplacementMap**
Processing Node-Specific Attributes	*scale* *x-channel-selector* one of *R*,*G*,*B* or *A*. *y-channel-selector* one of *R*,*G*,*B* or *A* *in2*, The second image ("B" in the formulas) for the compositing operation.

Description	Uses Input2 to spatially displace Input1, (similar to the Photoshop displacement filter). This is the transformation to be performed: `P'(x,y) <- P( x + scale * ((XC(x,y) - .5), y + scale * (YC(x,y) - .5))` where P(x,y) is the source image, Input1, and P'(x,y) is the destination. XC(x,y) and YC(x,y) are the component values of the designated by the *x-channel-selector* and *y-channel-selector*. For example, to use the R component of Image2 to control displacement in x and the G component of Image2 to control displacement in y, set *x-channel-selector* to "R" and *y-channel-selector* to "G".
Comments	The displacement map defines the inverse of the mapping performed.
Implementation issues	This filter can have arbitrary non-localized effect on the input which might require substantial buffering in the processing pipeline. However with this formulation, any intermediate buffering needs can be determined by *scale* which represents the maximum displacement in either x or y.

NodeType	**feGaussianBlur**
Processing Node-Specific Attributes	*std-deviation.*

Description	Perform gaussian blur on the input image.
	
	The Gaussian blur kernel is an appoximation of the normalized convolution:
	`H(x) = exp(-x2/ (2s2)) / sqrt(2* pi*s2)`
	
	where 's' is the standard deviation specified by std-deviation.
	
	This can be implemented as a separable convolution.
	
	For larger values of 's' (s >= 2.0), an approximation can be used: Three successive box-blurs build a piece-wise quadratic convolution kernel, which approximates the gaussian kernel to within roughly 3%.
	`let d = floor(s * 3*sqrt(2*pi)/4  +  0.5)`
	
	... if d is odd, use three box-blurs of size 'd', centered on the output pixel.
	
	... if d is even, two box-blurs of size 'd' (the first one centered one pixel to the left, the second one centered one pixel to the right of the output pixel one box blur of size 'd+1' centered on the output pixel.

Implementation Issues	Frequently this operation will take place on alpha-only images, such as that produced by the built-in input, SourceAlpha. The implementation may notice this and optimize the single channel case. If the input has infinite extent and is constant, this operation has no effect. If the input has infinite extent and is a tile, the filter is evaluated with periodic boundary conditions.

NodeType	**feImage**
Processing Node-Specific Attributes	*href,* reference to external image data. *transform,* supplemental transformation specification
Description	Refers to an external image which is loaded or rendered into an RGBA raster. If *imaging-matrix* is not specified, the image takes on its natural width and height and is positioned at 0,0 in image space. The imageref could refer to an external image, or just be a reference to another piece of SVG. This node produces an image similar to the builtin image source *SourceGraphic* except from an external source.

NodeType	**feMerge**
Processing Node-Specific Attributes	none
Processing Node-Specific Sub-Elements	Each <feMerge> element can have any number of <feMergeNode> subelements, each of which has an **in** attribute.
Description	Composites input image layers on top of each other using the *over* operator with *Input1* on the bottom and the last specified input, *InputN*, on top.
Comments	Many effects produce a number of intermediate layers in order to create the final output image. This filter allows us to collapse those into a single image. Although this could be done by using n-1 Composite-filters, it is more convenient to have this common operation available in this form, and offers the implementation some additional flexibility (see below).
Implementation issues	The canonical implementation of feMerge is to render the entire effect into one RGBA layer, and then render the resulting layer on the output device. In certain cases (in particular if the output device itself is a continuous tone device), and since merging is associative, it may be a sufficient approximation to evaluate the effect one layer at a time and render each layer individually onto the output device bottom to top.

NodeType	**feMorphology**
Processing Node-Specific Attributes	*operator*,one of *erode* or *dilate.* *radius*, extent of operation
Description	This filter is intended to have a similar effect as the min/max filter in Photoshop and the width layer attribute in ImageStyler. It is useful for "fattening" or "thinning" an alpha channel, The dilation (or erosion) kernel is a square of side 2**radius* + 1.
Implementation issues	Frequently this operation will take place on alpha-only images, such as that produced by the built-in input, *SourceAlpha.* In that case, the implementation might want to optimize the single channel case. If the input has infinite extent and is constant, this operation has no effect. If the input has infinite extent and is a tile, the filter is evaluated with periodic boundary conditions.

NodeType	**feOffset**
Processing Node-Specific Attributes	*dx*,*dy*
Description	Offsets an image relative to its current position in the image space by the specified vector.
Comments	This is important for effects like drop shadow etc.

NodeType	**feSpecularLighting**
Processing Node-Specific Attributes	*surface-scale* height of surface when Ain = 1. *specular-constant* ks in Phong lighting model. Range 0.0 to 1.0 *specular-exponent* exponent for specular term, larger is more "shiny". Range 1.0 to 128.0. *light-color* RGB

Processing Node-Specific Sub-Elements	One of `<feDistantLight  azimuth= elevation= >` `<fePointLight    x= y= z= >` `<feSpotLight     x= y= z= points-at-x= points-at-y= points-at-z= specular-exponent=>`
Description	Light an image using the alpha channel as a bump map. The resulting image is an RGBA image based on the light color. The lighting caculation follows the standard specular component of the Phong lighting model. The resulting image depends on the light color, light position and surface geometry of the input bump map. The result of the lighting calculation is added. We assume that the viewer is at infinity the z direction (i.e the unit vector in the eye direction is (0,0,1) everywhere. `Sr = ks * pow(N.H, specular-exponent) * Lr` `Sg = ks * pow(N.H, specular-exponent) * Lg` `Sb = ks * pow(N.H, specular-exponent) * Lb` `Sa = max(Sr, Sg, Sb)` `where` `ks = specular lighting constant` `N = surface normal unit vector, a function of x and y` `H = "halfway" unit vectorbetween eye unit vector and light unit vector` `Lr,Lg,Lb = RGB components of light` See feDiffuseLighting for definition of N and (Lr, Lg, Lb). The definition of H reflects our assumption of the constant eye vector E = (0,0,1): `H = (L + E) / Norm(L+E)` where L is the light unit vector. Unlike the feDiffuseLighting, the feSpecularLighting filter produces a non-opaque image. This is due to the fact that specular result (Sr,Sg,Sb,Sa) is meant to be added to the textured image. The alpha channel of the result is the max of the color components, so that where the specular light is zero, no additional coverage is added to the image and a fully white highlight will add opacity.
Comments	This filter produces an image which contains the specular reflection part of the lighting calculation. Such a map is intended to be combined with a texture using the *add* term of the *arithmetic* Composite method. Multiple light sources can be simulated by adding several of these light maps before applying it to the texture image.
Implementation issues	The feDiffuseLighting and feSpecularLighting filters will often be applied together. An implementation may detect this and calculate both maps in one pass, instead of two.

NodeType	**feTile**
Processing Node-Specific Attributes	none
Description	Creates an image with infinite extent by replicating source image in image space.

NodeType	**feTurbulence**
Processing Node-Specific Attributes	*base-frequency* *num-octaves* *type*, one of*fractal-noise* or *turbulence*.

Description	Adds noise to an image using the Perlin turbulence-function. It is possible to create bandwidth-limited noise by synthesizing only one octave. For a detailed description the of the Perlin turbulence-function, see "Texturing and Modeling", Ebert et al, AP Professional, 1994. If the input image is infinite in extent, as is the case with a constant color or a tile, the resulting image will have maximal size in image space.
Comments	This filter allows the synthesis of artificial textures like clouds or marble.
Implementation issues	It might be useful to provide an actual implementation for the turbulence function, so that consistent results are achievable.

16 Interactivity

16.1 Links: the <a> element

SVG provides an **<a>** element, analogous to like HTML's <a> element, to indicate hyperlinks; those parts of the drawing which when clicked on will cause the current browser frame to be replaced by the contents of the URL specified in the *href* attribute.

The <a> element uses Xlink. (Note that the XLink specification is currently under development and is subject to change. The SVG working group will track and rationalize with XLink as it evolves.)

The following is a valid example of a hyperlink attached to a path (which in this case draws a triangle):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
  <desc>This valid svg document draws a triangle which is a hyperlink
  </desc>
  <a href="http://www.w3.org">
    <path d="M 0 0 L 200 0 L 100 200 z"/>
  </a>
</svg>
```

Download this example

This is the well-formed equivalent example:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
 xmlns = 'http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
  <desc>This well formed svg document draws a triangle which is a hyperlink
  </desc>
  <a xml:link="simple" show="replace" actuate="user" href="http://www.w3.org">
    <path d="M 0 0 L 200 0 L 100 200 z"/>
  </a>
</svg>
```

Download this example

In both examples, if the path is clicked on, then the current browser frame will be replaced by the W3C home page.

16.2 Event Handling

Any <g>, <image>, <path>, <text> or vector graphic shape (such as a <rect>) can be assigned any of the following standard HTML event handlers:

Mouse Events

- **onmousedown**
- **onmouseup**
- **onclick**
- **ondblclick**
- **onmouseover**
- **onmousemove**
- **onmouseout**

Keyboard Events

- **onkeydown**
- **onkeypress**
- **onkeyup**

State Change Events

- **onload**
- **onunload** (only applicable to outermost <svg> elements which are to be mapped into a rectangular region/viewport)
- **onzoom** (only applicable to outermost <svg> elements which are to be mapped into a rectangular region/viewport)

Additionally, SVG's scripting engine needs to have the *altKey*, *ctrlKey* and *shiftKey* properties available.

16.3 Zoom and pan control

The outermost **<svg>** element in an SVG document can have the optional attribute **allow-zoom-and-pan**, which takes the possible values of *true*and *false*. If true, the user agent should allow the user to zoom in and pan around the given document. If false, the user agent should not allow the user to zoom and pan on the given document.

17 Animation

17.1 Introduction

The Web is a dynamic medium and that SVG needs to support the ability to change vector graphics over time. SVG documents can be animated in the following ways:

- (Syntax under development) SVG will include a syntax for describing simple time-based modifications on any attribute or property on any of its elements. Thus, there will be elements in the SVG language that will allow you to do the following, among others: motion path animation, fade-in/fade-out effects, and objects which grow, shrink or spin. One objective is to provide sufficient animation capability such that SVG can be used for banner ads as an alternative to animated GIF.
- Using the SVG DOM. The SVG DOM conforms to key aspects of Document Object Model (DOM) Level 1 Specification and Document Object Model (DOM) Level 2 Specification. Every attribute and style sheet setting is accessible to scripting, and SVG offers a set of additional DOM interfaces to support efficient animation via scripting. As a result, virtually any kind of animation can be achieved, including motion paths, color changes over time, and transparency effects. The timer facilities in scripting languages such as ECMAScript can be used to start up and control the animations. (See example below.)
- SVG documents can be components within SMIL (Synchronized Multimedia Integration Language) documents.
- In the future, it is expected that future versions of SMIL) will be modularized and that components of it could be used in conjunction with SVG and other XML grammars such as XHTML to achieve animation effects.

17.2 Animation Example Using the SVG DOM

The following example shows a simple animation:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG July 1999//EN"
  "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in"
     fit-box-to-viewport="0 0 400 300"
     onload="StartAnimation()" >

  <defs>
    <script><![CDATA[
      var timer_increment = 50.
      var max_time = 10000
      var text_element
      StartAnimation() {
        text_element = document.getElementById("TextElement");
        ShowAndGrowElement(0);
      }
      ShowAndGrowElement(timevalue) {
        timevalue = timevalue + timer_increment
        if (timevalue > max_time)
          timevalue = timevalue - floor(timevalue/max_time) * max_time

        // Scale the text string gradually until it is 20 times larger
        scalefactor = (timevalue * 20.) / max_time
        text_element.SetAttribute("transform", "scale(" + scalefactor + ")")

        // Make the string more opaque
        opacityfactor = timevalue / max_time
        text_element.getStyle().setProperty("opacity", "opacity:" + opacityfactor, "")

        // Call ShowAndGrowElement again <timer_increment> milliseconds later.
        setTimeout("ShowAndGrowElement(" + timer_increment + ")")
      }
    ]]></script>
  </defs>

  <g transform="translate(50,300)" style="fill:red; font-size:10">
    <text id="TextElement">SVG</text>
  </g>
</svg>
```

Download this example

The above SVG file contains a single graphics element, a text string that says "SVG". The animation loops continuously. The text string starts out small and transparent and grows to be large and opaque. Here is an explanation of how this example works:

- The <svg> element's `width` and `height` attributes indicate that the document should take up a rectangle of size 4inches by 3inches. The `fit-box-to-viewport` attribute indicates that the initial coordinate system should have (0,0) at its top left and (400,300) at its bottom right. (Thus, 1 inch equals 100 user units.) The `onload="StartAnimation()"` attribute indicates that when the document has been fully loaded and processed, then invoke ECMAScript function StartAnimation().
- The <script> element defines the ECMAScript which makes the animation happen. The `StartAnimation()` function is only called once to give a value to global variable `text_element` and to make the initial call to `ShowAndGrowElement()`. `ShowAndGrowElement()` is called every 50 milliseconds and resets the `transform` and `style` attributes on the text element to new values each time it is called. At the end of `ShowAndGrowElement`, the function tells the ECMAScript engine to call itself again after 50 more milliseconds.
- The <g> element shifts the coordinate system so that the origin is shifted toward the lower-left of the viewing area. It also defines the fill color and font-size to use when drawing the text string.
- The <text> element contains the text string and is the element whose attributes get changed during the animation.

18 Backwards Compatibility, Descriptions and Titles

18.1 Introduction

SVG offers features to allow for:

- Backwards Compatibility Reasonable fallback behavior in the event an SVG file is viewed in a browser which doesn't support SVG
- **<desc>** Descriptive information on a section of graphics elements which is packaged in a convenient way so that the visually impaired can understand the nature of the graphics
- **<title>** A short description on a section of graphics elements which can be used to provide tooltips in a viewing environment

18.2 Backwards Compatibility

A user agent (UA) might not have the ability to process and view SVG documents. The following list outlines two of the backwards compatibility scenarios associated with SVG documents:

- For XML grammars with the ability to embed SVG documents, it is assumed that some sort of alternate representation capability such as the <switch> element and some sort of feature-availability test facility (such as what is described in the SMIL 1.0 specification (??? add links)) will be available.

  This <switch> element and feature-availability test facility (or their equivalents) are the recommended way for XML authors to provide an alternate representation to an SVG document, such as an image or a text string. The following example shows how to embed an SVG drawing within a SMIL 1.0 document such that an alternate image will display in the event the UA doesn't support SVG. (In this example, the SVG document is included via a URL reference. With some parent XML grammars it will also be possible to include an SVG document inline within the same file as its parent grammar.)

```
<?xml version="1.0" standalone="yes"?>
<smil>
  <body>
    <!-- The SMIL <switch> element will process the
         first child element which tests true and skip
         past all others. -->
    <switch>
      <!-- The system-required attribute tests to see if
           the user agent supports SVG. If true, then
           render the file drawing.svg. -->
      <ref system-required="http://www.w3.org/Graphics/SVG/svg-19990706.dtd"
```

```
             type="image/svg" src="drawing.svg" />
        <!-- Else, render the alternate image. -->
        <img src="alternate_image.jpg" />
      </switch>
    </body>
  </smil>
```

Download this example

- For HTML 4.0, SVG drawings should be embedded using the <object> element. The alternate representation should be included as the content of the <object> element. In this case, the SVG document usually will be included via a URL reference. The following example shows how to use the <object> element to include an SVG drawing via a URL reference with an image serving as the alternate representation in the absence of an SVG user agent:

```
<html>
  <body>
    <object type="image/svg" data="drawing.svg">
      <!-- The contents of the <object> element (i.e., an alternate
         image) are drawn in the event the user agent cannot process
         the SVG drawing. -->
      <img src="alternate_image.jpg" alt="short description" />
    </object>
  </body>
</html>
```

18.3 The <desc> and <title> elements

Each <g> or graphics object in an SVG drawing can supply a <desc> and/or a <title> description string where the description is text-only. These description strings provide information about the graphics to visually impaired users. User agents which can process SVG in most cases will ignore the description strings (except that the <title> might be used to provide a tooltip).

The following is an example. In typical operation, the SVG user agent would ignore (i.e., not display) the <desc> element and the <title> elements and would render the remaining contents of the <g> element.

If this file were processed by an older browser (i.e., a browser that doesn't have SVG support), then the browser would treat the file as HTML. All SVG elements would not be recognized and therefore ignored. The browser would render all character data (including any character data within <desc> and <title> elements) within the current text region using current text styling parameters.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg SYSTEM "http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
<svg width="4in" height="3in">
<g>
  <title>
    Company sales by region
  </title>
  <desc>
    This is a bar chart which shows
    company sales by region.
  </desc>
  <!-- Bar chart defined as vector data -->
</g>
</svg>
```

Download this example

Description and title elements can contain marked-up text from other namespaces. Here is an example:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
 xmlns="http://www.w3.org/Graphics/SVG/svg-19990706.dtd">
   <desc xmlns:mydoc="http://foo.org/mydoc">
      <mydoc:title>This is an example SVG file</mydoc:title>
      <mydoc:para>The global description uses markup from the
        <mydoc:emph>mydoc</mydoc:emph> namespace.</mydoc:para>
   </desc>
   <g>
   <!-- the picture goes here -->
   </g>
</svg>
```

Download this example

19 Embedding Foreign Object Types

One goal for SVG is to provide a mechanism by which other XML language processors can render into an area within an SVG drawing, with those renderings subject to the various transformations and compositing parameters that are currently active within the SVG document. One particular example of this is to provide a frame for the HTML/CSS processor so that dynamically reflowing text (subject to SVG transformations and compositing) could be inserted into the middle of an SVG document. Another example is inserting a MathML expression into an SVG drawing.

The <foreignobject> element allows for inclusion of foreign namespaces which has graphical content drawn by a different user agent, where the graphical content that is drawn is subject to SVG transformations and compositing. The contents of <foreignobject> are assumed to be from a different namespace. Any SVG elements within a <foreignobject> will not be drawn, except in the situation where a properly defined SVG subdocument is recursively embedded within the different namespace (e.g., an SVG document contains an XHTML document which in turn contains yet another SVG document).

Additionally, there is a capability for alternative representations so that something meaningful will appear in SVG viewing environments which do not have the ability to process a given <foreignobject>. To accomplish this, SVG has a **<switch>** element and **system-required** attribute similar to the corresponding facilities within the SMIL 1.0 Recommendation. The rules for **<switch>** are that the first child element whose **system-required** evaluates to "true" will be processed and all others ignored.

Here is an example:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
 xmlns = 'http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
  <desc>This example uses the switch element to provide a
  fallback graphical representation of an equation, if
  MathML is not supported.
  </desc>
  <!-- The <switch> element will process the first child element
       whose testing attributes evaluate to true.-->
  <switch>

    <!-- Process the MathML if the system-required attribute
         evaluates to true (i.e., the user agent supports MathML
         embedded within SVG). -->
    <foreignobject
       system-required="http://www.w3.org/TR/REC-MathML-19980407"
       width="100" height="50">
      <!-- MathML content goes here -->
    </foreignobject>

    <!-- Else, process the following alternate SVG.
         Note that there are no testing attributes on the <g> element.
         If no testing attributes are provided, it is as if there
         were testing attributes and they evaluated to true.-->
    <g>
      <!-- Draw a red rectangle with a text string on top. -->
      <rect style="fill: red"/>
      <text>Formula goes here</text>
    </g>

  </switch>
</svg>
```

Download this example

It is not required that SVG user agent support the ability to invoke other arbitrary user agents to handle embedded foreign object types; however, all conforming SVG user agents would need to support the **<switch>** element and should be able to render valid SVG elements when they appear as one of the alternatives within a **<switch>** element.

Ultimately, it is expected that commercial Web browsers will support the ability for SVG to embed content from other XML grammars which use CSS layout or XSL to format their content, with the resulting CSS- or XSL-formatted content subject to SVG transformations and compositing. At this time, such a feature represents an objective, not a requirement.

(The exact mechanism for providing these capabilities hasn't been decided yet. Many details need to be worked out.)

20 Private Data

20.1 Introduction

SVG allows inclusion of elements from foreign namespaces anywhere with the SVG document tree. In general, the SVG user agent will ignore any unknown elements except to include them within the DOM. (The notable exception is described under Embedding Foreign Object Types.)

SVG's ability to include foreign namespaces can be used for the following purposes:

- Application-specific information so that authoring applications can include model-level data in the SVG document to serve their "roundtripping" purposes (i.e., the ability to write, then read a file without loss of higher-level information).
- Supplemental data for extensibility. For example, suppose you have an extrusion extension which takes any 2D graphics and extrudes it in three dimensions. When applying the extrusion extension, you probably will need to set some parameters. The parameters can be included in the SVG document by inserting elements from an extrusion extension namespace.

To illustrate, a business graphics authoring application might want to include some private data within an SVG document so that it could properly reassemble the chart (a pie chart in this case) upon reading it back in:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
     xmlns = 'http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
  <defs>
    <myapp:piechart xmlns:myapp="http://mycompany/mapapp"
                    title="Sales by Region">
      <myapp:pieslice label="Northern Region" value="1.23"/>
      <myapp:pieslice label="Eastern Region" value="2.53"/>
      <myapp:pieslice label="Southern Region" value="3.89"/>
      <myapp:pieslice label="Western Region" value="2.04"/>
      <!-- Other private data goes here -->
    </myapp:piechart>
  </defs>
  <desc>This chart includes private data in another namespace
  </desc>
  <!-- In here would be the actual graphics elements which
       draw the pie chart -->
</svg>
```

Download this example

21 Extensibility

It is highly desirable that SVG provides a general, robust extensibility mechanism which allows the feature set of SVG to be enhanced without having to wait for SVG specification to be updated and new SVG implementations to be deployed. The general model is that extensions could be implemented in scripting (e.g., ECMAScript), Java or CLSID/ActiveX, with each option offering different degrees of platform-independence, performance, and universal availability.

Here is a mini requirements definition for an extensibility mechanism:

1. Provisions for alternative renderings in standard SVG (i.e., SVG without extensions) in the event that a given extension is not available.
2. Allows for the possibility of extensions that are built into a given SVG user agent. (For example, a particular SVG implementation might choose to have custom built-in SVG functionality in order to serve the special needs of a particular set of users.)
3. Allows for the possibility of extensions that are added on dynamically at run-time via scripting (e.g., ECMAScript), Java/JAR or CLSID/ActiveX.
4. Leverages and conforms to others Web standards directions.
5. Provides the extensibility mechanisms for the following:
    - custom paint servers (i.e., fill and stroke types)
    - custom object servers (e.g., 3D lettering) that allow for whole new object types to be added to SVG beyond the basic types of paths, images and text.
    - Filter effects that allow transforming vector and/or raster data into other vector/raster data before painting.
    - Compositing effects that provide hooks for custom blending the object into its background.

The exact mechanism for providing this capability hasn't been decided yet.

22 Metadata

22.1 Introduction

Metadata is information about a document.

RDF is the appropriate language for metadata. The specifications for RDF can be found at:

- Resource Description Framework Model and Syntax Specification
- Resource Description Framework (RDF) Schema Specification

Metadata within an SVG document should be expressed in the appropriate RDF namespaces and should be placed within the **<metadata>** child element to the document's **<svg>** root element. (See Example below.)

Here are some suggestions for content creators regarding metadata:

- Content creators should refer to W3C Metadata Recommendations and activities when deciding which metadata schema to use in their documents.
- Content creators should refer to the Dublin Core, which is a set of generally applicable core metadata properties (e.g., Title, Creator/Author, Subject, Description, etc.).
- Additionally, SVG Metadata Schema (below) contains a set of additional metadata properties that are common across most uses of vector graphics.

Individual industries or individual content creators are free to define their own metadata schema, but everyone is encouraged to follow existing metadata standards and use standard metadata schema wherever possible to promote interchange and interoperability. If a particular standard metadata schema does not meet your needs, then it is usually better to define an additional metadata schema in RDF which is used in combination with the given standard metadata schema than to totally avoid the standard schema.

22.2 The SVG Metadata Schema

(This schema has not yet been defined. Here are some candidate attributes for the schema: MeetsAccessibilityGuidelines, UsesDynamicElements, ListOfExtensionsUsed, ListOfICCProfilesUsed, LiistOfFontsUsed, ListOfImagesUsed, ListOfForeignObjectsUsed, ListOfExternalReferences.)

22.3 An Example

Here is an example of how metadata can be included in an SVG document. The example uses the Dublin Core version 1.0 schema and the SVG metadata schema:

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in"
    xmlns = 'http://www.w3.org/Graphics/SVG/svg-19990706.dtd'>
    <desc>Floor layout for MyCompany office space</desc>
    <metadata>
      <rdf:RDF
               xmlns:rdf = "http://www.w3.org/...-rdf-syntax-ns"
               xmlns:rdfs = "http://www.w3.org/TR/...-schema"
               xmlns:dc = "http://purl.org/dc/elements/1.0/"
               xmlns:svgmetadata = "http://www.w3.org/..." >
        <rdf:Description about=""
                 dc:title="MyCompany Floor Layout"
                 dc:description="Floor layout for MyCompany office space"
                 dc:publisher="MyCompany Incorporated"
                 dc:date="1999-03-03"
                 dc:format="image/svg"
                 dc:language="en" >
          <dc:creator>
            <rdf:Bag>
                <rdf:li>Billy Potts</rdf:li>
                <rdf:li>Mary Graham</rdf:li>
            </rdf:Bag>
          </dc:creator>
          <svgmetadata:General MeetsAccessibilityGuidelines="true"/>
        </rdf:Description>
      </rdf:RDF>
    </metadata>
</svg>
```

Download this example

Appendix A: SVG Requirements

Introduction to SVG Requirements

This appendix contains the extensive list of Design Goals and Detailed Requirements which were drafted initally before the SVG specification was written and then revised as the specification developed.

Interspersed within the Design Goals and Detailed Requirements are references into those sections of the specification which correspond to the given Design Goal or Detailed Requirement.

SVG Requirements Table of Contents

SVG Design Goals

Open specification

Widely implemented and supported

Relationship to other Web standards efforts

Graphics features

Interactivity and Dynamic Behaviors

Interchange features

SVG Detailed Requirements

General requirements

1. Consistent visual results and behaviors across implementations
2. Elegant, uniform, complete and consistent
3. Packaging
4. Performance
5. Alternate representations

6. Backward compatibility
7. Well internationalized
8. Accessibility features

Graphical facilities

9. Vector graphics shapes
10. Text data
11. Image data
12. Color support
13. Transparency support
14. Grouping and layering
15. Template objects/symbols
16. Fill options
17. Stroke options
18. Transformations
19. Coordinate systems, relationship to CSS positioning
20. Antialiasing
21. Stenciling and masking
22. Client-side filter effects such as shadowing
23. Compositing
24. CSS support
25. Connectable reference points
26. Parameter substitution and formulas
27. Print control

Interaction

28. Zoom and pan
29. Links
30. Event handling
31. Object selection, clipboard
32. DOM access
33. Animation

Miscellaneous

34. Inclusion of private data (metadata)
35. Extensibility
36. Embedded fonts and images

SVG Design Goals

The following are the Design Goals for SVG. Besides providing a set of high-level objectives for SVG, these goals act as the criteria by which proposed features are judged. Thus, the features list shown below under SVG Detailed Requirements should reflect the higher-level goals listed here.

These SVG Design Goals are not listed in any particular order. It is recognized that some of the goals might conflict or be unachievable and that tradeoffs will need to be made.

Open specification

1. The specification should be controlled by the members in the W3C, not by a single vendor. Thus, the specification should not subject to sudden change by a single vendor.
2. The specification should be vendor neutral and thus should not contain features biased towards a particular vendor.

   EDITOR: None of the above goals has a direct impact on the specification.

Widely implemented and supported

3. SVG should be a standard feature in Web browsers
4. Implementations of SVG should be consistent so that the same visual results and behaviors exist in all conforming SVG user agents.
5. There should not be subset problems and incompatible generator/reader sets. Thus, there should be a single language specification, not a set of layered language specifications.
6. There should be widespread support in authoring applications and related tools
7. To promote widespread adoption and support, SVG should be specified to be as basic and simple as possible while still providing necessary features to satisfy the needs of graphics on the Web. While the chief goal is to aim at the middle ground, a basic and simple feature set will allow it to be used on devices with a wide range of resolutions and capabilities, from small mobile devices through office computer monitors to high resolution printers.
8. Straightforward generation via hand-authoring with a text editor or server-side scripts (e.g., CGI)
9. SVG should be as self-contained as possible. While SVG should leverage and be compatible with other W3C work, it should attempt to do so without introducing excessive dependencies on other specifications.
10. Ready availability to the casual implementor is desirable

    EDITOR: The current draft of SVG attempts to meet these goals.

11. Reference source code is desirable

    EDITOR: The above goal is independent of the specification.

Relationship to other Web standards efforts

12. Defined as an application of XML
13. Compatible with and/or leverages other relevant standards efforts, including XML namespaces, XML links (XLink), DOM, CSS, XSL and metadata. For example:
    - the elements and attributes of an SVG drawing should be scriptable via the DOM
    - text should be expressed as XML character data so that it can be found by search engines
    - attributes which make sense to be part of a style should be expressed in CSS

    The SVG working group will need to coordinate proactively with other working groups when it is more appropriate to meet the SVG requirements through modifications to other Recommendations.

    EDITOR: The current draft of SVG attempts to meet these goals.

Graphics features

14. Complete, general-purpose Web vector graphics format that satisfactorily meets the graphics delivery needs for all creators and consumers of Web vector graphics
15. Sufficiently powerful and precise to meet the needs of professional Web graphic designers such that they will utilize SVG instead of raster formats in those cases where vector graphics is a more natural or appropriate format

    EDITOR: The current draft of SVG attempts to meet these goals.
16. Sufficiently powerful to meet the needs of business presentation and diagramming applications such that these drawings will be published on the Web using SVG instead of raster formats

    EDITOR: The current draft of SVG attempts to meet the final-form needs of these applications, but the higher-level (e.g., intelligent layout) needs of these applications are not addressed in the current draft specification. It is unclear at this time to what level the first version of SVG will attempt to address these needs.
17. Feature set is rich enough such that a reasonable conversion of most existing standard static graphics files into SVG is possible
18. Sufficiently compatible with the graphics design and publishing industries' feature sets and file formats such that there is (as lossless as possible) a straightforward mapping from these applications and file formats into SVG. The goals are to facilitate conversion of existing artwork into SVG, promote the creation of lots of compelling new SVG artwork, make it as easy as possible for the graphics design and publishing industries to adapt existing authoring tools, and provide for new SVG authoring tools.
19. Compatible with the needs of non-technical persons who want a straightforward way to generate relatively simple graphics either by hand-editing or via server-based graphics generation (e.g., CGI scripts).
20. Compatible with high-quality, efficient printing at full printer resolution and with reliable color

21. Focused on basic foundation, presentation-level graphics facilities, with a critical eye toward higher-level (model-level) constructs which might be better suited to a higher-level standard which would sit on top of SVG
22. The working group is investigating whether other Web standards (e.g., CSS, XSL, MathML, Web Schematics) could use SVG as its low-level rendering specification. (This goal has significant dependencies on other W3C initiatives such as the W3C Formatting Model and might change accordingly.)

    EDITOR: The current draft of SVG attempts to meet these goals.
23. The specification for SVG should take into account the possibility of building a future 3D graphics specification which either sits on top of SVG or which is entirely independent but with a similar syntax; however, the SVG working group should not let itself be slowed down or constrained by 3D upgrade path issues.

    EDITOR: This goal hasn't been addressed yet.

Interactivity and Dynamic Behaviors

24. Allows for interactive and dynamic Web pages

    EDITOR: The current draft of SVG attempts to meet these goals.

Interchange features

25. Suitable as a platform-independent graphics exchange format
26. Mechanisms for inclusion of application-specific (or industry specific) private data which would facilitate use of SVG as a application-specific (or industry-specific) native file format for authoring applications
27. Capable of use as a print metafile: sufficiently expressive such that a higher-level print metafile XML grammar could use SVG for its page imaging operators

    EDITOR: The current draft of SVG attempts to meet these goals.

SVG Detailed Requirements

The following is the detailed list of required features and capabilities in SVG. Items which are already listed as a Design Goal are not repeated here. It is recognized that some of these requirements might conflict or may not be possible.

General requirements

1. Consistent visual results and behaviors across implementations
    1. The specification needs to be complete and unambiguous
    2. The goal is to have at least two independent implementations each of SVG viewers and

    SVG generators under development as the specification is defined to help find and remove ambiguities from the specification.
    3. An extensive conformance test suite needs to be delivered in conjunction with the final specification.

    EDITOR: The above goals are independent of the specification.

2. Elegant, uniform, complete and consistent
    1. Wherever possible, all graphical attributes and operations should be available to all object types. For example, if 2x3 transformation matrices are available for shape objects, then 2x3 transformation matrices should also be available for text and image objects.

    EDITOR: The current draft of SVG attempts to meet these goals.

3. Packaging
    1. Stand-alone packaging: Compatible with the need of stand-alone graphics authoring packages. It should be possible for a graphics authoring application to save a stand-alone SVG file. ("Stand-alone packaging" means that an SVG file is saved as a separate, self-contained file, probably with a .svg extension on platforms where the extension is important.)
    2. Fragment packaging: Compatible with the need of consolidated full-page Web authoring packages which would like to author/deliver full pages of XML with optional vector graphics elements interspersed as isolated components within the page. ("Fragment packaging" means that snippets of SVG would appear inside the content of a parent XML grammar. For example, a picture of an arrow expressed in SVG might appear inline within an XML file. The expected way this would be achieved would be by referencing the SVG namespace within the XML file.)

        EDITOR: In spec. See Two Indended Uses: Documents and Fragments.

4. Performance
    1. Reasonable display performance on commonly used end-user machines. (EDITOR: The current draft of SVG attempts to meet this goal.)
    2. Highly efficient representation to minimize download times
        1. The working group should investigate abbreviation schemes (particularly for path data) to minimize file sizes

            EDITOR: The current draft of SVG attempts to meet this goal. This draft uses the following strategies in order to achieve minimal download times on uncompressed SVG files:

            - A rich feature set on the client-side such as gradient fills, pattern fills and filter effects allow complicated graphics to be specified with a minimal number of characters.
            - Path data can be expressed in an abbreviated form (see Path data)
            - The elements and attributes which are most commonly used in files have short names.
            - Because most attributes are expressed via CSS properties, it will be possible in some cases to minimize file size and download times by combining common sets of attributes into named, reused CSS styles.

        2. The working group should also collaborate with other working groups, such as the XML working group. on binary compression alternatives.

           EDITOR: Flate compression is very suitable for compressing XML grammars such as SVG and it is already part of HTTP 1.1.
    3. Streamable (i.e., support progressive rendering)
    4. Simple primitives (e.g., lines) should exhibit fast performance

       EDITOR: The current draft of SVG attempts to meet these goals. See Other Vector Graphic Shapes
5. Alternate representations
    1. The working group should investigate the issue of alternative representations in the form of text strings or image data. Alternate representations as text or images might provide a good answer for backwards compatibility in some cases or provide for a faster display option for some drawings. Alternate text strings would be particularly valuable to visually impaired users.
6. Backward compatibility
    1. The working group should develop a reasonable backward compatibility strategy for when a user attempts to view an SVG drawing in a browser which doesn't yet support SVG.

   EDITOR: The current draft describes suggested methods for achieving backward compatibility with browsers that don't support SVG.
7. Well internationalized
    1. By virtue of being written in XML, SVG will already have baseline internationalisation capability (Unicode characters, language tagging). The working group will collaborate with the I18N working group to ensure that SVG is suitable for worldwide Web graphics.

   EDITOR: The current draft of SVG has not yet addressed internationalization issues.
8. Accessibility
    1. The working group should ensure that adaptive interfaces for people with disabilities are fully supported, and that mapping SVG content to these contexts is easy and graceful.

   EDITOR: The current draft of SVG attempts to meet this goal. See Accessibility Features.

Graphical facilities

9. Vector graphics shapes

   (Note: in the remainder of this document, the terms "vector graphic shape" will be abbreviated to "shape" or "shape objects")
    1. Path data
        1. Paths can be made up of any combination of the following:
            - Straight line segments
            - Cubic bezier curved segments

        - Quadratic bezier curved segments
        - Elliptical and circular arcs
        - No other curve types (Other curve types such as splines or nurbs are either technically very difficult, industry-specific and/or have not established themselves as industry standards as much as beziers.)
    2. Compound paths (i.e., paths with donut holes) should be supported. Both the non-zero winding and even-odd fill rules should be supported in both fill and clip operations to ensure compatibility with the design and publishing industries' existing file formats and authoring applications. (Also, it is much easier to implement these fill rules inside a renderer than it is in a file format converter.)

    EDITOR: In spec. See Paths.

2. A set of predefined shape types such as rectangles, rounded rectangles, circles and ellipses should be available so that simple objects can be defined without having to learn bezier mathematics. The exact list of predefined object types has not been decided yet.

    EDITOR: In spec. See Other Vector Graphic Shapes.

3. Any shape can be filled, stroked or used as a clipping path and/or mask. (See Fill options, Stroke options and Stenciling and masking, below.)

    EDITOR: In spec. See Filling, Stroking and Paint Servers and and Clipping, Masking and Compositing.

10. Text data
    1. Text strings should be expressed as XML character data which could be marked up with arbitrary XML name spaces. (Text strings as XML character data allow search engines to find the strings.)

        EDITOR: Text strings can either be expressed as XML character data or as XPointer references to other objects where the text string is expressed as XML character data. See The <text> element

    2. All text/font attributes from CSS should be supported. Complete support for CSS is what the Web community will expect. (See CSS support, below)

        EDITOR: All CSS text/font attributes that seem to make sense for final-form text are available. See Text and Font Properties

    3. It is clear that precise text sizing/positioning is a requirement for the graphics design community. Thus, SVG should allow for precise control of text sizing, positioning, rotation, and skewing on a character-by-character basis. In particular, the working group should look to see if this feature could be packaged as to provide for precise text on a path.

        EDITOR: Precise text positioning is available because it is possible to position individual characters and to locate the glyph origin at a particular location. See The <text> element. Also, SVG will include a built-in text-on-a-path feature. See Text on a path.

    4. It is clear that precise control of fonts and glyphs is a requirement for the graphics design community. It should be possible to achieve precise control of the exact font and the exact glyphs within a given font to use for a given character sequence so that graphic artists have a way to ensure that the end user sees what was intended

EDITOR: Precise control of font by name is possible using the font specification from CSS which SVG will use. (However, this still leaves open the possibility of the wrong version of the given font being used; however, there are ways to set up your font specification such that correct versioning can be achieved in all but highly unusual circumstances.) Precise control of glyphs is available in SVG. See Ligatures and Alternate Glyphs.

5. The same fill/stroke/clip/mask capabilities that can be used with vector data should also be available to text data. For example, it should be possible to fill a text string with a gradient or stroke it with a dashed line pattern. This is a widely used feature in the graphics design world and is required for compatibility with existing graphics content.

   EDITOR: In spec. See Filling, Stroking and Paint Servers.

6. The working group hasn't investigated yet whether it is advisable to provide a capability to automatically position a text string relative to another graphical element (e.g., automatically centering a text string within a rectangle). It is clear that such a feature would be a great convenience for hand-coders of certain types of drawings, such as diagrams. The working group should investigate whether there is an easy-to-use (for a hand-coder) yet robust and elegant way of achieving this.

   EDITOR: Not in current draft spec of SVG.

7. Having a text string determine the dimensions of a parent graphical object (e.g., a box is drawn to fit around a text string) is too high-level of a construct and is not planned for version 1 of SVG.

   EDITOR: Not in current draft spec of SVG.

8. Adding a text-on-a-path capability to SVG is still under consideration. The working group recognizes that there are many technical complexities which make it difficult to design the feature in such a way that it will be both sufficiently powerful and sufficiently compatible with the needs of various authoring scenarios.

   EDITOR: SVG will include a built-in text-on-a-path feature. See Text on a path.

11. Image data
    1. The same image formats available to HTML should be available to SVG

       EDITOR: In spec. See Images.

    2. The working group hasn't had time yet to address the issue of color management on image data. The goals will be to:
       1. achieve compatibility with how other specifications (e.g., HTML) perform color management on image data
       2. (just like the rest of SVG) take full advantage of color management systems when supported by the browser and/or the platform operating system (see Color support below)
       3. if different ICC profile are specified both as an attribute on an image and embedded within the image, the attribute should win

       EDITOR: Not addressed yet.

12. Color support
    1. All colors need to be specified in the sRGB color space for compatibility with other Web

standards.

EDITOR: In spec. See Colors.

2. It is clear that reliable color is important to both end users and Web content creators. sRGB does not answer all precise color needs, and most desktop platforms support the more complete ICC-based color management system. Thus, an alternative ICC-based color representation should be available for all color attributes. If the SVG user agent supports color management, then the ICC color takes precedence over the sRGB color.

   EDITOR: In spec. See Colors.

3. There has been little discussion so far on spot color support. Spot colors might be possible by specifying then using ICC profiles (perhaps via a Named Color profile), or by storing spot color information as private data (metadata) in the SVG file.

   EDITOR: Not addressed yet.

13. Transparency support

    Transparency support is becoming commonplace in authoring applications and is widely used today in Web animations.

    1. Many existing authoring applications achieve transparency by adding an opacity property wherever a color property is allowed. For compatibility with these applications, and because this represents a good technical solution to many transparency needs, wherever a color property is allowed, there will be a corresponding opacity property which indicates how transparent a given object/component/attribute should be when blended into its background

       EDITOR: In spec. See 'fill-opacity', 'stroke-opacity' and Object And Group Opacity: the 'opacity' Property.

    2. In group opacity, the collection of objects that makes up a group is (in effect) drawn into a temporary area in memory, and then the temporary area is blended as a single unit into the background graphics. Group opacity is necessary when you have an aggregate object such as a automobile which is drawn as a collection of overlapping, opaque components (e.g., the hubcap might be an opaque circle that is drawn on top of an opaque tire) and you want to blend that object as a unit into the background. Group opacity is a requirement for many animation applications and has utility with static graphics also. It is straightforward to implement, particularly once you have support for transparent image objects, which is needed for GIF and PNG. It is infeasible to achieve these effects without this feature. Thus, group opacity should be supported in SVG.

       EDITOR: In spec. See Object And Group Opacity: the 'opacity' Property.

    3. The working group decided that identification of a transparency/blue-screen/chromaKey color will not be supported for shapes and text because the opacity features are sufficient. However, the transparency color feature that exists in image file formats such as GIF and PNG will be supported.

       EDITOR: Not in spec because we decided not to do support chromaKey.

    4. See Compositing, below.

14. Grouping and layering

    1. There needs to be an ability to define groups (i.e., collections) of objects to allow for

convenient access for scripting applications. Groups should have the following attributes (among others), all of which are accessible and/or controllable via scripting:

1. Unique ID
2. z-index for explicit layering
3. visibility
4. transformations
5. opacity

EDITOR: In spec. See Grouping and Naming Collections of Drawing Elements: the <g> Element.

2. Implicit layering: unless an explicit z-index value is provided, objects will be drawn bottom-to-top in the order which they appear in the input stream. This strategy is compatible with most popular graphics file formats.

   EDITOR: In spec. See SVG Rendering Model.

3. Explicit layering: objects or collections of objects can be assigned an explicit z-index value, which takes precedence over the implicit bottom-to-top drawing order. (The bottom-to-top drawing order of objects with the same z-index relative to each other is the the order in which they appear in the input stream.) This explicit layering is a requirement for achieving certain types of animation effects.

   EDITOR: Not in spec. The working group decided that a z-index effect can be achieved either by having CSS manage multiple SVG drawings or by rearranging graphical elements via the DOM. A z-index option would complicate implementation and streaming for little gain.

15. Template objects/symbols
    1. There should be a capability for defining template objects that can be instantiated multiple times with different attributes.
    2. The template objects should be able to establish a full set of default attributes, and the instantiated objects should be able to override any of these default attriubutes.

    EDITOR: In spec. See Using template objects: the <use> element.

16. Fill options (i.e., painting the interior of a shape or text object)

    The following fill options should be available:

    1. Solid-color fill (see Color support above)

       EDITOR: In spec. See Filling, Stroking and Paint Servers.

    2. Gradients
       1. Multiple gradient stops will be supported (not just two)
       2. Both linear and radial gradients will be supported
       3. It is still an open issue within the working group whether additional gradient types should be supported. Examples: conical, elliptical, square, rectangular, Gouraud shaded triangle and rectangular meshes.
       4. Gradients should allow for both a linear ramp between gradient stops and a well-defined sigma function for the calculation of the intermediate colors between

the stops.

5. For all gradients, the specification should provide details for the exact behavior of the gradient ramp. If there is a disagreement about what the gradient ramps should be, priority should be given to: (1) making sure the gradient ramp rules are consistent with real-world lighting effects, (2) retaining compatibility with as much existing content as possible (with a bias towards existing content whose users really care about how their gradients look)
6. The suggestion has been made to allow for substitution of a different color table for use with a given gradient. (The color stop positions stay the same, but the colors used change.) This suggestion hasn't been discussed yet in the working group.

EDITOR: In spec. See Gradients.

3. Patterns (i.e., tiled object fill)
   1. Any objects (i.e., shapes, images, text) can be used to fill any other objects
   2. When objects are used to fill other objects, parameters can be set to achieve tiling effects.
   3. The tiling options should be at least as capable as those found in the file formats used by the design and publishing industries to ensure compatibility with existing artwork and authoring applications.

   EDITOR: In spec. See Patterns.
4. Other fill styles - The working group hasn't decided yet whether other fill styles such as fractal patterns are appropriate.

   EDITOR: Not addressed yet.
5. Custom fill styles - See Extensibility, below.

   EDITOR: In spec. See Extensibility.

17. Stroke options (i.e., drawing the outline of a shape or text object)
    1. The same attributes available for filling an object (see Fill options, above) should be available to stroke an object. For example, you should be allowed to stroke with a pattern.

       EDITOR: In spec. See Stroke Properties.
    2. The set of stroke options should be at least as capable as the stroke options in the file formats used by the design and publishing industries
       1. Arbitrary, continuous values for line width
       2. The working group hasn't reached consensus yet on whether SVG should support hairlines (i.e., instructing the device to draw the thinnest possible line)
       3. User-defined dash patterns and initial phase offset
       4. Caps, joins and miterlimits

       EDITOR: In spec. See Stroke Properties.
    3. Arrowheads and polymarkers
       1. There should be a small set of predefined arrowheads

    2. Arrowheads are optional and can be placed at the start and/or end of path objects
    3. The working group should investigate the possibility of providing for custom "arrowheads" at the start and/or end of a path object which are defined by referencing a different graphical object or group. This capability would solve the following needs:
        - Diagramming applications might use arrowheads extensively, but the predefined arrowheads might not be satisfactory
        - Graphic designers who want to attach arrowheads to drawings will want complete control of the visual characteristics of their arrowheads
        - This feature can be leveraged to provide a polymarker capability for applications such as scatter diagrams

    EDITOR: In spec. See Markers.

18. Transformations
    1. Arbitrarily nested 2x3 matrix transformations should be available. This facility is necessary to ensure compatibility with existing artwork and authoring tools.
        1. 2x3 matrices are sufficient to achieve any types of translation, scaling, rotation and skewing
        2. Transformation matrices can be nested to an arbitrary depth. A given transformation matrix is concatenated with the transformation matrices of all parent objects.

       EDITOR: In spec. See Coordinate systems and transformation matrices.
    2. In the spirit of making SVG reasonably easy to use for content creators who would have difficulty constructing 2x3 matrices, SVG should offer an alternative set of simpler transformation attributes (e.g., rotation=, scale=). The SVG specification, however, needs to defined unambiguously what should happen if both the simpler transformation attributes and a 2x3 transformation matrix is provided.

       EDITOR: In spec. See Coordinate systems and transformation matrices.
    3. These transformations should apply to all object types (shapes, images and text) in a uniform and consistent manner

       EDITOR: In spec. See Coordinate systems and transformation matrices.
    4. Transformed objects should have the option of exhibiting true scalability (where all attributes should scale uniformly, and linewidths, images and text should scale along with the sizes of the shapes). Additionally, transformed objects should also have the option of selective scalability such that certain attributes (e.g., stroke size and textsize) are invariant.

       EDITOR: True scalability is in the current draft spec. There are some capabilities for selective scalability by setting properties such as linewidths and font sizes using Transformed CSS units. See Coordinate systems and transformation matrices.
19. Coordinate systems, relationship to CSS positioning
    1. SVG should use CSS positioning for establishing a viewport for an outermost SVG element, such as an <svg>...</svg> complete drawing or an SVG fragment within an XML Web page.

    2. SVG will support local user coordinates
    3. Real number values (i.e., an integer followed by a possible decimal fractional component) should be possible for all appropriate attributes and coordinates. The language definition itself should not inhibit infinite precision.

    EDITOR: In spec. See Coordinate Systems, Transformations and Units.

20. Antialiasing
    1. It is clear that the graphics design community not only wants antialiasing to be available, but also wants to have the ability to turn antialiasing on/off on an object basis in order to achieve precise control of the rendering and possibly to control drawing speed. Thus, there will be an antialias control attribute on each graphics object in SVG.

       EDITOR: In spec. See 'stroke-antialiasing' and 'text-antialiasing'.
    2. Antialiasing adds a significant burden to the casual implementor and isn't a requirement for all potential applications of SVG, such as viewing CAD drawings. Thus, antialiasing control will be regarded as a hint to the SVG user agent, which can choose whether or not to honor it. Major general-purpose implementations of SVG user agents such as commercial Web browsers should honor and implement the antialiasing control hint.

       EDITOR: This had been in spec but got dropped accidentally between working drafts and will be corrected.
21. Stenciling and masking
    1. Clipping paths

       Clipping paths are a commonly used feature in existing artwork and an integral part of all authoring products used by graphic designers. Clipping paths are as fundamental to the design and publishing industries' existing file formats and authoring tools as are nested transformation matrices. Clipping is relatively easy to implement in viewers using a raster mask approach in device space if you have code for drawing a filled path into an offscreen buffer. Nested clipping paths are easy to achieve on top of an unnested clipping path implementation by just looping through each parent clipping path one after another. The performance overhead with nested clipping paths in animation scenarios is probably small relative to other necessary computations (e.g, filling and stroking) and can be overcome on the user end by simply avoiding/minimizing the use of the feature. Without nested clipping paths, converters from the design and publishing industries' existing file formats will have a significant burden in determining how to flatten the nested clipping paths they encounter. Nested clipping paths are necessary to provide lossless translation from the design and publishing industries' existing file formats.
       1. Any shape or text object can be used as a hard-edge clipping path
       2. Clipping paths can be nested arbitrarily to ensure compatibility with existing artwork and authoring tools.

       EDITOR: In spec. See Clipping Path Properties.
    2. Masks

       8-bit masking is a fundamental feature on all operating systems, and a key feature for both static and dynamic Web pages.
       1. Any graphics object (i.e., shape, text, or image) can be used as an 8-bit alpha mask to control the alpha-compositing of a different object (or group/collection of

objects) into other objects

EDITOR: In spec. See Masking Properties.

22. Client-side filter effects such as shadowing

    Client-side filters provide for the possibility of significant file size and download time savings in many applications. An example is text drawn with a glow effect and a drop shadow. If client-side glow and drop shadow filters were available, then only the text string and the names of the filters would need to be downloaded, instead of today, where the text needs to be converted to a raster by the author.

    1. SVG should include a set of built-in client-side filter effects for commonly used effects such as shadowing. The definitions of these effects should be unambiguous so that all implementations produce the same visual results.
    2. Additionally, the working group should investigate the feasibility of a general filter mechanism which would allow for custom filter effects to be downloaded and applied to a given graphical object (see Extensibility, below)

       EDITOR: In spec to some level. See Filters. Extensibility features are mentioned in spec but aren't detailed.

23. Compositing

    Compositing means the rules by which a foreground object's colors are blended into a background object's colors. There are many different approaches to compositing. Alpha compositing (where each pixel has an 8-bit alpha channel which determines how opaque that pixel is) is the most common, but even this method has multiple variations.

    1. The working group needs to define standard rules for how alpha compositing should work. Which color space (sRGB? Lab?)? What are the exact formulas?

       EDITOR: In spec. This document will call for alpha compositing in the sRGB color space using simple alpha and (1-alpha) algorithms. Details haven't been provided. See Simple Alpha Blending/Compositing.

    2. The working group needs to decide whether SVG should offer multiple compositing options as standard features. Should SVG support all of the Porter/Duff compositing options? Additionally, Photoshop offers many Adjustment Layer options. Should these be supported?

       EDITOR: Not addressed yet.

    3. See Extensibility, below.

24. CSS support

    1. All CSS text/font attributes from the most recently approved CSS recommendation should be honored by all conforming SVG user agents.

       EDITOR: All CSS text/font attributes that seem to make sense for final-form text are available. See Text and Font Properties

    2. The issue of CSS inheritance is still open. It is unclear that the standards world will have addressed how a parent XML/HTML grammar passes its current style table to a child grammar. Until this issue is addressed, the SVG working group cannot promise that version 1 of SVG will support the inheritance of CSS styles from a parent grammar. The working group should work towards achieving this highly desirable capability, however.

Representatives from the CSS and XSL have requested working with the SVG working on style sheet issues. They would like to discuss:

1. use either XSL or CSS stylesheets within SVG
2. use stylesheets to apply properties to both the text AND graphic objects/assemblies within an embedded SVG component (to provide/enforce "house style" or common properties across all illustrations in a document).

EDITOR: Not addressed yet.

25. Connectable reference points

    Connectable reference points are useful constructs for many applications. In particular, diagramming applications could use connectable reference points to define the start and end points for lines that connect two different objects. The difficulty with connectable reference points is that they might introduce a large degree of complexity into the specification and the various implementations. It might be better to leave connectable reference points to a higher-level XML grammar which sits on top of SVG.

    1. It is not a requirement for version 1 of SVG to provide for a mechanism for defining connectable reference points. This is a complex issue which might be better served by a higher-level XML grammar.
    2. However, the working group should not exclude this as an option for version 1 of SVG. If one of the working group members can come up with a good proposal, then the working group should consider the proposal seriously.
    3. See DOM access, below, for more on connectable reference points.

    EDITOR: Not in current draft spec of SVG.

26. Parameter substitution and formulas on coordinates

    Parameterized graphic objects, possibly built using a set of formulas to define how the object grows and stretches based on the values of its parameters and other aspects of the graphics, would provide for a powerful "intelligent graphics" capability. However, such as feature opens up many complex issues which might be better off in a higher-level XML grammar which sits on top of SVG.

    1. Parameter substitution and formulas on coordinates are not requirements for version 1 of SVG.
    2. A simple parameter substitution strategy, however, might be easy to define, simple to implement, and provide lots of value. In other words, a simple parameter substitution strategy might provide 90% of the value at 10% of the effort. If one of the working group members can come up with a good proposal, then the working group should consider the proposal seriously.
    3. It is harder to conceive of a simple formula-based coordinate capability. However, if someone of the working group members can come up with a good proposal, then the working group should consider the proposal seriously.

    EDITOR: Not in current draft spec of SVG.

27. Ability to control whether a given object is printed.

    1. When the working group considers this feature, it should see whether CSS's print control features are adequate.

EDITOR: Not addressed yet.

Interaction

28. Zoom and pan
    1. An SVG user agent should support zoom and pan on graphics, with true scalability. Thus, all objects and attributes (including such things as text and linewidths) should grow/shrink uniformly with the zoom level.

    EDITOR: In spec. See Zoom and pan control.
29. Links
    1. It should be possible to assign a link to any graphic object or group.
    2. SVG should support all of the kinds of links into and out of a drawing as would be appropriate. For example, it should provide for links to other views in the same file or links to external media (i.e., a URL). Also, it should be possible to link into a particular view within an SVG drawing.
    3. As much as is appropriate, SVG should be compatible with XLink.
    4. The working group discussed briefly the concept of linking into a moment of time within an animation application. This should be investigated by the working group as the specification is developed.

    EDITOR: Basically linking out of SVG is in the spec. Ideas are presented for linking into a particular view within an SVG drawing. SVG linking is compatible with XLink. There is nothing in the spec about linking into a moment of time (this is probably best served by an animation grammar on top of SVG). See Links: the <a> element
30. Event handling
    1. It should be possible to assign event handlers to an individual graphic object or group.
    2. The list of event handlers should at least be as extensive as what is available for HTML (e.g., mouseovers, mouseclicks).
    3. Additional event handlers might prove to be valuable. For example, an onzoom event handler might prove very useful to control what content appears based on zoom level. The working group should investigate onzoom and other possible event handlers.

    EDITOR: In spec (including an onzoom event handler). See Event Handling
31. Object selection, copying/pasting to clipboard
    1. It is highly desirable but not required that SVG viewing user agents have the ability to selectively copy/paste graphical elements, particularly from the browser to the desktop environment.
    2. The working group has not investigated yet whether it makes sense to specify an object selection mechanism in SVG. However, it is clear that the ability to select part of a drawing is a requirement for clipboard/exchange purposes.
    3. A particular detail is selecting text for the purposes of copying to the clipboard. The working group hasn't discussed yet whether it should be possible to select text strings (complete or partial) from within an SVG user agent.

    EDITOR: The spec includes a provision for selecting text for the purposes of copying to the clipboard. The spec doesn't say anything yet about other graphic object types, although this seems like a desirable feature. See Defining Text Flows: the <textflow> element

32. DOM access
    1. SVG should provide for complete DOM support for all attributes and elements.
    2. To provide robust hooks for animation applications, the DOM should expose graphic objects down to the individual point.
    3. Supplemental utility methods (e.g., query an object's bounds, its center, its perimeter as expressed as a path, access to "connectable" points for callouts or connection lines) would be very helpful to people writing scripts that drive SVG drawings or to higher level grammars (e.g., Web Schematics or MathML) which might want to perform layout based on the bounds of a given graphics object. The SVG working group should collaborate with the DOM working group to investigate whether it is possible within the constraints of DOM to provide such utility methods.

    EDITOR: We have concluded that SVG should have utility functions. These utility methods have been mentioned for accessing individual points within a Path and for controlling an animation Animation.

33. Animation

    Built-in animation primitives will not be part of SVG. Animation will only be possible via the DOM or a different specification, which might sit on top of SVG.

    EDITOR: Animation utility functions will be part of the SVG DOM. See Animation.

Miscellaneous

34. Inclusion of private data (metadata)

    To promote the use of SVG as an interchange format or as a component of higher-level graphics languages, there needs to be a provision for inclusion of application-specific or industry-specific private data within an SVG drawing.

    EDITOR: In spec. See Private Data.

35. Extensibility

    It is highly desirable that SVG be extensible to cope with changing requirements and for providing many valuable hooks to allow for creation of more efficient and compelling Web pages. A well-designed extensibility mechanism can allow for tomorrow's innovations to be available in today's browsers (i.e., no need to wait for a new version of the standard to be defined an implemented in browsers). A well-designed extensibility mechanism could be the best solution for many valuable features such as client-side filters on graphics data. Possible extensibility facilities are custom paint servers (for filling and stroking), custom object types, custom filters, custom compositing engines and custom color spaces. However, defining a useful and workable extensibility mechanism is very difficult and frought with many obstacles, such as deficiencies in cross-platform language standards. Thus, the working group should look into an extensibility mechanism as a highly desirable feature and review proposals from the members, but an extensibility mechanism is not an absolute requirement for version 1 of SVG.

    EDITOR: In spec, except color space extensibility was dropped as this should be achieved by providing an ICC profile. See Extensibility.

36. Embedded components to achieve self-contained graphics files
    1. Images - There has been some strong initial feedback has been that SVG should provide for embedded images. The working group should collaborate with other working groups (e.g., XML) to investigate the feasibility of allowing for embedded images within an SVG drawing.

       EDITOR: Not in spec. With the performance improvements in HTTP 1.1, images can be referenced just as in HTML without significant performance penalty.
    2. Fonts - There has been some strong initial feedback has been that SVG should provide for embedded fonts. The working group has yet to make decisions on this issue. The working group should collaborate with other working groups (e.g., XML) to investigate the feasibility of allowing for embedded fonts within an SVG drawing.

       EDITOR: Not in spec. CSS already provides for Web fonts. We are currently counting on CSS Web fonts and are waiting on public feedback whether some sort of built-in font strategy is also needed.

Appendix B: Change History

Changes with the 06-July-1999 SVG Draft Specification

- Changes to Conformance Requirements and Recommendations:
    - In Conforming SVG Viewers, dropped GIF from the list of required formats. Now, only JPEG and PNG are listed.
    - In Forward and undefined references, indicated that forward references are disallowed and included a link to the decription of the <defs> element.
- Changes to Document Structure:
    - Modified the description of the <defs> element to discuss how all referenced elements must be direct children of a <defs> element.
    - Modified the description of the <use> element to indicate that <use> can only refer to elements within an SVG file (not entire files).
    - Added a section on the <image> element. The <image> element is very comparable to <use> except that it can only refer to whole file (not elements within a file).
- Changes to Rendering Model:
    - Moved the recently modified/renamed properties shape-rendering, text-rendering and image-rendering into this chapter. (There used to be properties 'stroke-antialiasing' and 'text-antialiasing'.)
- Changes to Clipping, Masking and Compositing:
    - For Clipping paths, reformulated how clipping paths are specified. Now, there is a <clippath> element whose children can include <path> elements, <text> elements and other vector graphic shapes such as <circle>. The silhouettes of the child elements are logically OR'd together to create a single silhouette which is then used to restrict the region onto which paint can be applied. Also, fixed a bug in the spec by replacing the 'inherit' value on 'clippath' with a 'none' value and fixed the spec to say that 'clippath' does *not* inherit the 'clippath' property from its parent.
    - For Masking, reformulated how clipping paths are specified. Now, there is a <mask> element whose children can include any graphical object. The <mask> element can have attributes mask-units, x, y, width and height to indicate a sub-region of the canvas for the masking operation. These changes obsolete the following old properties: 'mask-method', 'mask-width', 'mask-height', 'mask-bbox'.

- Changes to Filling, Stroking and Paint Servers:
    - Renamed stroke-antialiasing to shape-rendering, with possible values of default, crisp-edges, optimize-speed and geometric-precision. The revised property is now just a hint to the implementation. Moved to Rendering chapter.
    - Revised the wording on gradient stops to indicate that out-of-order gradient stops should be resolved by adjusting offset values until the offset values become valid. (Previously, the spec said that gradient stops would be sorted.)
- Changes to Paths:
    - Removed the old elliptical arc commands A|a and B|b and inserted a new elliptical arc command called A|a, which has a different set of parameters than the previous two formulations. The new arc command matches the formulation of the other path data commands in that it starts with the current point and ends at an explicit (x,y) value.
- Changes to Other Vector Graphic Shapes:
    - In the sentence, "Mathematically, these shape elements are equivalent to the cubic bezier path objects that would construct the same shape", removed the words "cubic bezier".
- Changes to Text:
    - Replaced the old <textflow>, <textblock>, <text> and <textsrc> with the new <text> and <tspan>, which is a subelement to <text> and has optional attributes x=, y=, dx=, dy=, style= and href= (which allows it to take the place of <textsrc>). The only lost functionality from this simplification is the ability to select text across discontiguous blocks of text elements.
    - Made <textpath> a container element which can contain <tspan> elements or character data. This reformulation was necessary given the changes in the previous bullet.
    - Renamed text-antialiasing to text-rendering, with possible values of default, optimize-legibility, optimize-speed and geometric-precision. The revised property is now just a hint to the implementation. Moved to Rendering chapter.
- Changes to Images:
    - Added new property image-rendering, with possible values of default, optimize-speed and optimize-quality. The new property is just a hint to the implementation. The new property is documented in the Rendering chapter.
- Changes to Filter Effects:
    - Removed vector effects, including VEAdjustGraphics and VEPathTurbulence -- the working group decided that we hadn't found a critical mass of vector graphics effects functionality sufficient to warrant the additional complexity
    - Modified the names of all of the filter effects processing nodes to have the prefix "fe". The prefix is meant to prevent name clashes (e.g., <feImage> won't clash with <image>).
    - Removed the section on parameter substitution -- the WG didn't see why filter effects deserved macro expansion over other features.
- Changes to Animation chapter to indicate that SVG will include declarative animation. (Syntax still under development.)
- One line change in the SVG DOM chapter to change getStyle() to style property, per feedback from the DOM working group.
- Minor changes to the example in the Metadata chapter to fix incorrect references to Dublin Core

elements.

- Changes to DTD
    - Changes to DTD to reflect all of the changes described earlier in this section.
    - Flattened some double-indirect entity referencing into only single-indirect referencing. Fixed bug where pattern used x,y,width,height twice.
    - Changed rx,ry on <rect> to be #IMPLIED so that if one of them is missing the other one will be assigned the same value (for circular fillets).

Changes with the 25-June-1999 SVG Draft Specification

- General editorial activities:
    - Modified the titles and content of chapters 1 and 2. Chapter 1 is now a Introduction to SVG and chapter 2 is now SVG Concepts.
    - Included a first pass of information about conformance requirements, including a discussion of what makes a conforming document, generator, interpreter and viewer.
    - Included updated wording on the Rendering Model.
    - Reorganized the appendices. Added the beginnings of Appendix D. SVG's Document Object Model (DOM), Appendix E. Sample SVG files, Appendix F. Accessibility Support, Appendix G. Minimizing SVG File Sizes, Appendix H. Implementation and performance notes for fonts and Appendix I. References.
    - Included an example of DOM-based animation>.
    - Removed some of the wording that indicated tentativeness about certain features as the specification of various features is firming up.
- Coordinate Systems, Transformations and Units modifications:
    - Changed the 'transform' property into the transform attribute. The **transform** attribute can now accept a list of transformations such as **transform="translate(-10,-20) scale(2) rotate(45) translate(5,10)"**. Added skew-x and skew-y convenience transformations. Removed the fit() options from the old transform property and created new attributes **fit-box-to-viewport=** and **preserve-aspect-ratio**, described in new section Establishing an Initial User Coordinate System: the fit-box-to-viewport attribute.
    - Added an Implementation Notes section to the chapter on Coordinate Systems, Transformations and Units.
    - Added a note to the description of the transform attribute to indicate that the transform attribute is applied before other attributes or properties are processed.
- Paths modifications:
    - The J|j commands (elliptical quadrant) have been dropped from the list of path data commands because the working group felt the J|j commands would not receive wide usage.

- The path data commands for switching between absolute and relative coordinates in the middle of a command (the former A and r commands) have been dropped because of their high complexity relative to their limited space-saving value.
- The various arc commands in path data have been consolidated, renamed, and then expanded. The new commands are: A|a (an arc whose sweep is described by a start angle and end angle) and B|b (an arc whose sweep is described by two vectors whose intersections with the ellipse define the start point and end points of the arc).
- Reformulated the T/t path data commands to be consistent with the rest of the path data commands (i.e., vertices provided, control points automatically calculated as in S/s).
- Broke up the path data commands into separate tables to improve understandability.
- Modified the write-up on markers so that the <marker> element no longer is a subelement to <path>. <marker> is now defined to be just like <symbol>, but with marker-specific attributes **marker-units**, **marker-width**, **marker-height** and **orient**. To use a marker on a given <path> or vector graphic shape, we have new properties **'marker-start'**, **'marker-end'**, **'marker-mid'** and **'marker'**. See Markers.
- Indicated that each **d=** attribute in a <path> element is restricted to 1023 characters. See Path Data.
- Added an Implementation Notes section to the document that describes various details about expected processing and rendering behavior when drawing paths.
- Added The grammar for path data, a BNF for path data.

- Filling, Stroking and Paint Servers modifications:
  - Included a note under 'fill' property that indicates that open paths and polylines still can be filled.
  - Provided a more detailed write-up on patterns to make the <pattern> element consistent in various ways with <symbol>, <marker>, <lineargradient> and <radialgradient>.
  - Modified gradients in various ways. Replaced attribute target-type with gradient-units. Replaced <lineargradient> attributes vector-start-x, vector-start-y, vector-length, vector-angle with x1, y1, x2, y2. Replaced <radialgradient> attributes outermost-origin-x, outermost-origin-y, outermost-radius, innermost-x, innermost-y with cx, cy, r, fx, fy. Removed attributes target-left, target-top, target-right, target-bottom, which were deemed superfluous. Renamed attribute matrix to gradient-transform. Added gradient-transform back to linear gradients (they were in an earlier draft). Renamed <gradientstop> to <stop> to save space since the working group decided it didn't want to offer non-linear gradient ramps. Removed attribute color from <stop> and included new paragraphs indicating that color and opacity are set via the 'color' and 'opacity' properties.
  - Added a value of **none** to property 'stroke-dasharray'.
- Text modifications:
  - Broke the <textflow> element into two elements <textblock> and <textflow> to greatly simplify the feature, to remove the need to maintain consistent doubly linked lists, and to remove the possibility of cyclic references. Removed <tf> and renamed <t> to <tref>
  - Renamed the <src> subelement to <text> to <textsrc> for more consistency in nomenclature and to avoid use of such a generic element name for such a specific purpose.
- General/Miscellaneous:

    - Added a syntax and various processing details for Filter Effects
    - Altered the description of the <symbol> element to reflect the changes in transform-related attributes and properties.
    - In the chapter on Other Vector Graphic Shapes, changed the attributes on <ellipse> from major/minor to rx/ry for consistency with other parts of the spec, removed the angle attribute on ellipse, reformulated polygon to be exactly line polyline except that it automatically closes, changed "verts" to "points", and added rounding radii rx and ry to rectangle. Also, included a note about the 1023 character limit on the "points" attribute for <polyline> and <polygon>.
    - Removed property 'z-index'. The working group decided that a z-index effect can be achieved either by having CSS manage multiple SVG drawings or by rearranging graphical elements via the DOM. A z-index option would complicate implementation and streaming for little gain.
    - Add a chapter on Metadata, with an initial description of how metadata would work with SVG.
    - Removed the <private> element after concluding it is unnecessary given XML namespaces and the new W3C approach to validating namespaces. As a consequence, modified the write-up under Private Data.
    - Updated the descriptions under Embedding Foreign Object Types to reflect increased certainty about the direction SVG is headed in this area.
    - Added a General Implementation Notes section to the chapter on Conformance Requirements and Recommendations which discusses implementation issues that apply across the entire SVG language. In particular, added sections Forward and Undefined References (which explains implementation rules involving references that aren't valid at initial processing time) and Referenced objects are "pinned" to their own coordinate systems.
    - Changed all occurrences of "SVG processor" to "SVG user agent".
    - Fixed all incorrect references to <description> and replaced them with <desc>.
- Summary of changes to the DTD:
    - Gave the <a> element have the same content model as the <g> element.
    - Add transform attribute to most graphic objects.
    - Added attributes fit-box-to-viewport and preserve-aspect-ratio to <svg> and <symbol> elements
    - Added attributes x and y to the <svg> element.
    - For symbol_descriptor_attributes, renamed attributes x-min, y-min, x-max, y-max to x, y, width, height, respectively.
    - Modified the <marker> element to reflect the revised formulation for markers.
    - Added a <pattern> element which reflects the modified write-up on patterns. (The <pattern> element was missing from the previous DTD.).
    - Modified the definitions of <lineargradient> and <radialgradient> to reflect the modified write-up on gradients.
    - Renamed <gradientstop> to <stop>.
    - Removed attribute color from <stop>.

    - Changed the attributes on <ellipse> from major/minor to rx/ry for consistency with other parts of the spec, removed the angle attribute on ellipse, reformulated polygon to be exactly line polyline except that it automatically closes, changed "verts" to "points", and added rounding radii rx and ry to rectangle.
    - Removed the <private> element after concluding it is unnecessary given XML namespaces and the new W3C approach to validating namespaces.
    - Added xml:space to every element that might have character data content somewhere inside of it. This will allow content developers to control whether white space is preserved on <text> elements.
    - Text-related: renamed <src> to <textsrc> for more consistency in nomenclature and to avoid use of such a generic element name for such a specific purpose. Because of modifications in the area of defining textflows, added <textblock>, renamed <t> to <tref> and changed <textflow> so that if can only contain <tref> subelements.
    - Added a syntax for Filter Effects
    - Modified <foreignobject> such that it can only be the child of a <switch> element.
    - Added an href attribute to the <script> element. (Oversight that it wasn't there before.)
    - General clean-up in the area of anything using attributes x, y, width or height. Defined standard entities xy_attributes, bbox_attributes_optional and bbox_attributes_wh_required. In particular, the following elements now require width and height attributes: <image>, <rect>, <foreignobject>, <pattern>.

Changes with the 12-April-1999 SVG Draft Specification

- Included a DTD in Appendix C.
- There is now an <svg> element which is the root for all stand-alone SVG documents and for any SVG fragments that are embedded inline within a parent XML grammar. (See SVG Document Structure>.)
- Added initial descriptions of how text-on-a-path and SVG-along-a-path might work. (See Text on a Path.)
- Added <symbol> and <marker> elements to provide packaging for the following:
    - Necessary additional attributes on template objects
    - A clean way of defining standard drawing symbol libraries
    - The definition of a graphic to use as a custom glyph within a <text> element (e.g., generalize "text-on-a-path" to "SVG-on-a-path")
    - Necessary additional attributes for pattern definitions (for pattern fill)
    - Definition of a sprite for an animation
    - Marker symbols
    - Arrowheads

  Also added a new optional <data> subelement to the <path> element to provide the necessary hook to provide for custom arrowheads.

- Many changes to Coordinate Systems, Transformations and Units to make the section more complete and more readable. The specific changes to this chapter include:
  - Relatively minor changes in terminology to better match the terminology used in the CSS2 specification. For example, the definitions of the terms *canvas* and *viewport* were modified to be as close as possible to the corresponding definitions in the CSS2 specification.
  - The initial coordinate system is now based on the parent document's notion of pixels rather than points.
  - When embedded inline within a parent XML grammar, the outermost <svg> element in an SVG document acts like a block-level formatting object in the CSS layout model and thus supports CSS positioning properties such as **'left'** and **'width'** and the CSS properties **'clip'** and **'overflow'**.
  - Nested <svg> elements are the mechanism for recursively including nested SVG drawings, but also provide the one and only means of establishing a new viewport and thus changing the meaning of the various CSS unit specifiers such as px, pt, cm and % (percentages). Nested <svg> elements support the same CSS positioning properties as an outermost <svg> element,
- Removed <pieslice>, which was considered to be of lesser general utility than the other predefined vector graphic shapes, and added <line>, which allows a one-segment line to be drawn. See Other Vector Graphic Shapes.
- Replaced the <althtml> element with a description for how to use the <switch> (or equivalent) elements in XML grammars or the <object> element in HTML 4.0 as the recommended way to provide for alternate representations in the event the user agent cannot process an SVG drawing. (See Backwards Compatibility.)
- Removed the comment in the discussion under <description> and <title> which said that the given text string could be specified as an attribute. The text string now can only be supplied as character data. (See The <description> and <title> elements.
- Changed the wording about text strings to say that the current point is advanced by the metrics of the glyph(s) used rather than the character used. (See text positioning.)
- Added some details to the description of the <textflow> element to indicate that <text> elements can be directly embedded within <textflow> and that the current text position is remembered within a <textflow> from one <text> element to the next <text> element. (See Text Flows.)
- Added a new property **'text-antialiasing'** to provide a hint to the user agent about whether or not text should be antialiased. The lack of such a property was an inadvertant omission from previous versions of the spec and was called for in the SVG Requirements document.
- Removed the 'matrix' property from linear gradients because it was unnecessary (overspecification) and the 'spread-method' property from radial gradients because it was difficult to specify and implement, it didn't match current common usage and is of little apparent utility. (See Gradients.)
- Included a new section 2.1 with a brief discussion about the "image/svg" MIME type. Subsequent sections in chapter 2 have been renumbered accordingly. (See SVG MIME Type.)
- Added another bullet to the Accessibility section to indicate that SVG's zooming feature aids those with partial visual impairment. (See Accessibility.)
- Elaborated to a small level on how Embedded Foreign Object Types might work to reflect progess within the working group on the issue.

- Changed altglyph from a subelement to <text> to a CSS property in response to discussion on the W3C Character Model. See Alternate Glyphs.
- In the discussion about the <use> element, made clear that template objects could come from either the same document or an external document.
- Minor changes to description under Event Handling to indicate that any element can have an onload or onunload event handler to provide additional control via scripting as parts of the drawing download progressively.

Changes with the 05Feb1999 SVG Draft Specification

This was the first public working draft.

Appendix C: Document Type Definition

The DTD is also available for download.

```
<!ENTITY % namespace
  "xmlns CDATA #FIXED 'http://www.w3.org/Graphics/SVG/SVG-19990706'">

<!ENTITY % class
  "class NMTOKENS #IMPLIED">

<!ENTITY % id
  "id ID #IMPLIED">

<!ENTITY % lang
  "xml:lang NMTOKEN #IMPLIED">

<!ENTITY % style
  "style CDATA #IMPLIED">

<!ENTITY % class_id_lang
  "%class;
   %id;
   %style;">

<!ENTITY % class_id_lang_style
  "%class;
  %id;
  %lang;
  %style;">

<!ENTITY % structured_text
  "content CDATA #FIXED 'structured text'">

<!ENTITY % xmlspace
  "xml:space (default|preserve) #IMPLIED">

<!ENTITY % g_eventhandlers
  "onmousedown CDATA #IMPLIED
   onmouseup CDATA #IMPLIED
   onclick CDATA #IMPLIED
   ondblclick CDATA #IMPLIED
   onmouseover CDATA #IMPLIED
   onmousemove CDATA #IMPLIED
   onmouseout CDATA #IMPLIED
   onkeydown CDATA #IMPLIED
   onkeypress CDATA #IMPLIED
   onkeyup CDATA #IMPLIED
   onload CDATA #IMPLIED
   onselect CDATA #IMPLIED">

<!ENTITY % r_eventhandlers
  "onunload CDATA #IMPLIED
   onzoom CDATA #IMPLIED ">

<!ENTITY % system_required
  "system-required CDATA #IMPLIED">

<!ENTITY % replaced
  "xml:link CDATA #FIXED 'simple'
```

```
   show CDATA #FIXED 'embed'
   actuate CDATA #FIXED 'auto'
   href CDATA #REQUIRED ">

<!ENTITY % hyperlink
  "xml:link CDATA #FIXED 'simple'
   show CDATA #FIXED 'replace'
   actuate CDATA #FIXED 'user'
   href CDATA #REQUIRED ">

<!ENTITY % xy_attributes
  "x CDATA #IMPLIED
   y CDATA #IMPLIED">

<!ENTITY % dxdy_attributes
  "dx CDATA '0'
   dy CDATA '0'">

<!ENTITY % bbox_attributes_optional
  "%xy_attributes;
   width CDATA #IMPLIED
   height CDATA #IMPLIED">

<!ENTITY % bbox_attributes_wh_required
  "%xy_attributes;
   width CDATA #REQUIRED
   height CDATA #REQUIRED">

<!ENTITY % ref_xy_attributes
  "ref-x CDATA #IMPLIED
   ref-y CDATA #IMPLIED">

<!ENTITY % fit_attributes
  "fit-box-to-viewport CDATA #IMPLIED
   preserve-aspect-ratio CDATA 'xmid-ymid meet'">

<!ENTITY % transform_attributes
  "transform CDATA #IMPLIED">

<!ENTITY % shapes
  "rect|circle|ellipse|polyline|polygon|line">

<!ENTITY % g_elements
  "(defs?,title?,desc?,(use|image|text|path|%shapes;|g|switch|svg|a)*)">

<!ENTITY % g_elements_and_foreignobject
  "(defs?,title?,desc?,(use|image|text|path|%shapes;|g|switch|svg|a|foreignobject)*)">

<!ENTITY % filter_node_attributes
  "in CDATA #IMPLIED
  nodeid CDATA #IMPLIED">

<!ENTITY % component_transfer_function_attributes
  "type CDATA #REQUIRED
  table-values CDATA #IMPLIED
  slope CDATA #IMPLIED
  intercept CDATA #IMPLIED
  amplitude CDATA #IMPLIED
  exponent CDATA #IMPLIED
  offset CDATA #IMPLIED">

<!ELEMENT svg %g_elements; >
<!ATTLIST svg
  %namespace;
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %r_eventhandlers;
  %transform_attributes;
  %bbox_attributes_wh_required;
  %ref_xy_attributes;
  %fit_attributes;
  allow-zoom-and-pan (true | false) "true" >

<!ELEMENT g %g_elements; >
<!ATTLIST g
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %transform_attributes;>

<!ELEMENT switch %g_elements_and_foreignobject; >
<!ATTLIST switch
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %transform_attributes;>


<!ELEMENT path (data)* >
<!ATTLIST path
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  d CDATA #REQUIRED
  flatness CDATA #IMPLIED
  nominal-length CDATA #IMPLIED>

<!ELEMENT data EMPTY >
<!ATTLIST data
  d CDATA #REQUIRED >

<!ELEMENT rect EMPTY >
<!ATTLIST rect
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  %bbox_attributes_wh_required;
  rx CDATA "#IMPLIED"
  ry CDATA "#IMPLIED">

<!ELEMENT circle EMPTY >
<!ATTLIST circle
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  cx CDATA "0"
  cy CDATA "0"
  r CDATA #REQUIRED>

<!ELEMENT ellipse EMPTY >
<!ATTLIST ellipse
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  cx CDATA "0"
  cy CDATA "0"
  rx CDATA #REQUIRED
  ry CDATA #REQUIRED>
```

```
<!ELEMENT polyline EMPTY >
<!ATTLIST polyline
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  points CDATA #REQUIRED>

<!ELEMENT polygon EMPTY >
<!ATTLIST polygon
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  points CDATA #REQUIRED>

<!ELEMENT line EMPTY >
<!ATTLIST line
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  x1 CDATA "0"
  x2 CDATA "0"
  y1 CDATA "0"
  y2 CDATA "0">

<!ELEMENT text (#PCDATA|tspan|textpath)* >
<!ATTLIST text
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %transform_attributes;
  %xy_attributes;>

<!ELEMENT textpath (#PCDATA|tspan)* >
<!ATTLIST textpath
  start-offset CDATA "0"
  %replaced;>

<!ELEMENT tspan (#PCDATA)* >
<!ATTLIST tspan
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %xy_attributes;
  %dxdy_attributes;
  %replaced;>

<!ELEMENT use (desc?, title?) >
<!ATTLIST use
  %class_id_lang;
  %system_required;
  %g_eventhandlers;
  %transform_attributes;
  %bbox_attributes_optional;
  %replaced;>

<!ELEMENT image (desc?, title?) >
<!ATTLIST image
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %transform_attributes;
  %bbox_attributes_wh_required;
  %replaced;>

<!ELEMENT foreignobject (#PCDATA)* >
<!ATTLIST foreignobject
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %transform_attributes;
  %bbox_attributes_wh_required;
  %structured_text; >

<!ELEMENT a %g_elements; >
<!ATTLIST a
  %hyperlink;>

<!ELEMENT desc (#PCDATA)* >
<!ATTLIST desc
  %class_id_lang_style;
  %system_required;
  %g_eventhandlers;
  %xmlspace;
  %structured_text;>

<!ELEMENT title (#PCDATA)* >
<!ATTLIST title
  %class_id_lang_style;
  %xmlspace;
  %structured_text;>

<!ELEMENT defs (path|use|image|text|%shapes;|g|switch|svg|
  script|style|
  symbol|marker|
  lineargradient|radialgradient|pattern|
  clippath|mask|filter)* >
<!ATTLIST defs
  %class_id_lang_style;
  %xmlspace;>

<!ELEMENT script (#PCDATA)* >
<!ATTLIST script
  language CDATA "text/ecmascript"
  %replaced; >

<!ELEMENT style (#PCDATA)* >
<!ATTLIST style type CDATA #FIXED "text/css">

<!ELEMENT symbol %g_elements; >
<!ATTLIST symbol
  %class_id_lang_style;
  %xmlspace;
  %bbox_attributes_optional;
  %ref_xy_attributes;
  %fit_attributes;>

<!ELEMENT marker %g_elements; >
<!ATTLIST marker
  %class_id_lang_style;
  %xmlspace;
  %bbox_attributes_optional;
  %ref_xy_attributes;
  %fit_attributes;
  marker-units (stroke-width | userspace) "stroke-width"
  marker-width  CDATA "3"
  marker-height CDATA "3"
  orient CDATA "0">
```

```
<!ENTITY % gradient.attrs
  "gradient-units (userspace | object-bbox) 'userspace'
   gradient-transform CDATA #IMPLIED" >

<!ELEMENT lineargradient (stop)* >
<!ATTLIST lineargradient
  id ID #IMPLIED
  %gradient.attrs;
  x1 CDATA #IMPLIED
  y1 CDATA #IMPLIED
  x2 CDATA #IMPLIED
  y2 CDATA #IMPLIED
  spread-method (stick | reflect | repeat) "stick">

<!ELEMENT radialgradient (stop)* >
<!ATTLIST radialgradient
  id ID #IMPLIED
  %gradient.attrs;
  cx CDATA #IMPLIED
  cy CDATA #IMPLIED
  r CDATA #IMPLIED
  fx CDATA #IMPLIED
  fy CDATA #IMPLIED>

<!ELEMENT stop EMPTY >
<!ATTLIST stop
  %style;
  id ID #IMPLIED
  offset CDATA #REQUIRED>

<!ELEMENT pattern %g_elements; >
<!ATTLIST pattern
  %class_id_lang_style;
  %xmlspace;
  %bbox_attributes_wh_required;
  %ref_xy_attributes;
  %fit_attributes;
  pattern-units (userspace | object-bbox) "userspace"
  pattern-transform CDATA #IMPLIED>

<!ELEMENT clippath (path|text|%shapes;|use)* >
<!ATTLIST clippath
  %class_id_lang_style;
  %xmlspace;>

<!ELEMENT mask %g_elements; >
<!ATTLIST mask
  %class_id_lang_style;
  %xmlspace;
  mask-units (userspace | object-bbox) "userspace"
  %bbox_attributes_optional;>

<!ELEMENT filter (feBlend|feColor|
  feColorMatrix|feComponentTransfer|
  feComposite|feDiffuseLighting|feDisplacementMap|
  feGaussianBlur|feImage|feMerge|
  feMorphology|feOffset|feSpecularLighting|
  feTile|feTurbulence)* >
<!ATTLIST filter
  %class_id_lang;
  filter-units (userspace | object-bbox) "userspace"
  %bbox_attributes_optional;
  filter-res CDATA #IMPLIED>

<!ELEMENT feBlend EMPTY >
<!ATTLIST feBlend
  %filter_node_attributes;
  mode (normal | multiple | screen | darken | lighten) "normal"
  in2 CDATA #REQUIRED>

<!ELEMENT feColor EMPTY >
<!ATTLIST feColor
  %filter_node_attributes;
  color CDATA #IMPLIED>

<!ELEMENT feColorMatrix EMPTY >
<!ATTLIST feColorMatrix
  %filter_node_attributes;
  type CDATA #REQUIRED
  values CDATA #IMPLIED>

<!ELEMENT feComponentTransfer (feFuncR?,feFuncG?,feFuncB?,feFuncA?) >
<!ATTLIST feComponentTransfer
  %filter_node_attributes;>

<!ELEMENT feFuncR EMPTY >
<!ATTLIST feFuncR
  %component_transfer_function_attributes;>

<!ELEMENT feFuncG EMPTY >
<!ATTLIST feFuncG
  %component_transfer_function_attributes;>

<!ELEMENT feFuncB EMPTY >
<!ATTLIST feFuncB
  %component_transfer_function_attributes;>

<!ELEMENT feFuncA EMPTY >
<!ATTLIST feFuncA
  %component_transfer_function_attributes;>

<!ELEMENT feComposite EMPTY >
<!ATTLIST feComposite
  %filter_node_attributes;
  operator (over | in | out | atop | xor | arithmetic) "over"
  k1 CDATA #IMPLIED
  k2 CDATA #IMPLIED
  k3 CDATA #IMPLIED
  k4 CDATA #IMPLIED
  in2 CDATA #REQUIRED>

<!ELEMENT feDiffuseLighting (feDistantLight|fePointLight|feSpotLight) >
<!ATTLIST feDiffuseLighting
  %filter_node_attributes;
  result-scale CDATA #IMPLIED
  surface-scale CDATA #IMPLIED
  diffuse-constant CDATA #IMPLIED
  light-color CDATA #IMPLIED>

<!ELEMENT feDistantLight EMPTY >
<!ATTLIST feDistantLight
  azimuth CDATA #IMPLIED
  elevation CDATA #IMPLIED>

<!ELEMENT fePointLight EMPTY >
<!ATTLIST fePointLight
  x CDATA #IMPLIED
  y CDATA #IMPLIED
  z CDATA #IMPLIED>

<!ELEMENT feSpotLight EMPTY >
<!ATTLIST feSpotLight
  x CDATA #IMPLIED
  y CDATA #IMPLIED
  z CDATA #IMPLIED
  points-at-x CDATA #IMPLIED
  points-at-y CDATA #IMPLIED
  points-at-z CDATA #IMPLIED
  specular-exponent CDATA #IMPLIED>

<!ELEMENT feDisplacementMap EMPTY >
<!ATTLIST feEDisplacementMap
  %filter_node_attributes;
  scale CDATA #IMPLIED
  x-channel-selector (R | G | B | A) "A"
  y-channel-selector (R | G | B | A) "A"
  in2 CDATA #REQUIRED>

<!ELEMENT feGaussianBlur EMPTY >
<!ATTLIST feGaussianBlur
  %filter_node_attributes;
  std-deviation CDATA #IMPLIED>

<!ELEMENT feImage EMPTY >
<!ATTLIST feImage
  nodeid CDATA #IMPLIED
  %replaced;
  %transform_attributes;>

<!ELEMENT feMerge (feMergeNode)* >
<!ATTLIST feMerge
  %filter_node_attributes;>

<!ELEMENT feMergeNode EMPTY >
<!ATTLIST feMergeNode
  in CDATA #IMPLIED>

<!ELEMENT feMorphology EMPTY >
<!ATTLIST feEMorphology
  %filter_node_attributes;
  operator (erode | dilate) "erode"
  radius CDATA #IMPLIED>

<!ELEMENT feOffset EMPTY >
<!ATTLIST feOffset
  %filter_node_attributes;
  dx CDATA #IMPLIED
  dy CDATA #IMPLIED>

<!ELEMENT feSpecularLighting (feDistantLight|fePointLight|feSpotLight) >
<!ATTLIST feSpecularLighting
  %filter_node_attributes;
  surface-scale CDATA #IMPLIED
  specular-constant CDATA #IMPLIED
  specular-exponent CDATA #IMPLIED
  light-color CDATA #IMPLIED>

<!ELEMENT feTile EMPTY >
<!ATTLIST feTile
  %filter_node_attributes;>

<!ELEMENT feTurbulence EMPTY >
<!ATTLIST feTurbulence
  %filter_node_attributes;
  base-frequency CDATA #IMPLIED
  num-octaves CDATA #IMPLIED
  type (fractal-noise | turbulence) "turbulence">
```

Appendix D: SVG's Document Object Model (DOM)

D.1 SVG DOM Overview

The SVG DOM has the following general characteristics:

- It supports all appropriate and relevant facilities defined by the two documents Document Object Model (DOM) Level 1 Specification and Document Object Model (DOM) Level 2 Specification
- Wherever possible, the SVG DOM maintains consistency with the DOM for HTML 4.0, which was defined initially in Document Object Model (HTML) Level 1 of the DOM Level 1 Specification and which has been enhanced in various ways in the DOM Level 2 Specification
- In those cases where the above two approaches do not provide sufficient capabilities, SVG-specific DOM facilities are provided.

In particular, the following should be noted:

- The SVG DOM supports the CSS DOM facilities described in the DOM Level 2 Specification document
- The SVG DOM supports the event handling facilities described in the DOM Level 2 Specification document

D.2 Naming Conventions

The SVG DOM follows similar naming conventions to the HTML DOM Level 1.

All names are defined as one or more English words concatenated together to form a single string. Property or method names start with the initial keyword in lowercase, and each subsequent word starts with a capital letter. For example, a property that returns document meta information such as the date the file was created might be named "fileDateCreated". In the ECMAScript binding, properties are exposed as properties of a given object. In Java, properties are exposed with get and set methods.

The return value of an attribute that has a data type that is a value list is always capitalized, independent of the case of the value in the source document. For example, if the value of the align attribute on a P element is "left" then it is returned as "Left". For attributes with the CDATA data type, the case of the return value is that given in the source document.

D.3 Objects related to SVG documents

Interface *SVGDocument*

An `SVGDocument` is the root of the SVG hierarchy and holds the entire content. Beside providing access to the hierarchy, it also provides some convenience methods for accessing certain sets of information from the document.

IDL Definition

```
interface SVGDocument : Document {

  // Same meanings as in HTML DOM Level 1
           attribute  DOMString            title;
  readonly attribute  DOMString            referrer;
  readonly attribute  DOMString            domain;
  readonly attribute  DOMString            URL;
  void                      open();
  void                      close();
  void                      write(in DOMString text);
  void                      writeln(in DOMString text);
  Element                   getElementById(in DOMString elementId);

  // Methods to eliminate flicker in animations.
  unsigned long suspend_redraw(in unsigned long max_wait_milliseconds);
  void          unsuspend_redraw(in unsigned long suspend_handle_id)
                                 raises(DOMException);
  void          unsuspend_redraw_all();

  // The following utility methods for matrix arithmetic will be
  // available from the DOM. (Details not yet available.)
  // matrix += Vector2D           - translation
  // matrix -= Vector2D           - translation
  // matrix *= number             - scaling
  // matrix /= number             - scaling
  // matrix *= Vector2D           - non-orthogonal scaling
  // matrix /= Vector2D           - non-orthogonal scaling
  // matrix *= Matrix2D           - matrix concatenation
  // matrix == number             - test scaling matrix identity
  // matrix == Vector2D           - test for simple scaling
  // matrix.FSimple()             - transformation is an offset/scale
  // matrix.Rotate(angle)         - rotation
  // matrix.Rotate(Vector2D)      - rotation defined by vector (atan(y/x))
  // matrix.FlipX()               - flip +x <-> -x
  // matrix.FlipY()               - flip +y <-> -y
  // matrix.SkewX()               - skew in direction of X axis
  // matrix.SkewY()               - skew in direction of Y axis
  // matrix.Inverse()             - invert the matrix
};
```

Attributes

`title`

The title of a document as specified by the `title` element within the `defs` sub-element of the `svg` element that encompasses the document (i.e., `<svg><defs><title>Here is the title</title></defs><svg>`

`referrer`

Returns the URI of the page that linked to this page. The value is an empty string if the user navigated to the page directly (not through a link, but, for example, via a bookmark).

`domain`

The domain name of the server that served the document, or a null string if the server cannot be identified by a domain name.

`URL`

The complete URI of the document.

Methods

`open`

Note. This method and the ones following allow a user to add to or replace the structure model of a document using strings of unparsed SVG. Open a document stream for writing. If a document exists in the target, this method clears it.
This method has no parameters.
This method returns nothing.
This method raises no exceptions.

`close`

Closes a document stream opened by `open()` and forces rendering.
This method has no parameters.
This method returns nothing.
This method raises no exceptions.

`write`

Write a string of text to a document stream opened by `open()`. The text is parsed into the document's structure model.

Parameters

`text`	The string to be parsed into some structure in the document structure model.

This method returns nothing.
This method raises no exceptions.

`writeln`

Write a string of text followed by a newline character to a document stream opened by `open()`. The text is parsed into the document's structure model.

Parameters

`text`	The string to be parsed into some structure in the document structure model.

This method returns nothing.
This method raises no exceptions.

`getElementById`

Returns the Element whose `id` is given by elementId. If no such element exists, returns `null`. Behavior is not defined if more than one element has this `id`.

Parameters

`elementId`	The unique `id` value for an element.

Return Value

The matching element.

This method raises no exceptions.

`suspend_redraw`

Takes a time-out value which indicates that redraw should not occur until: (a) the corresponding `unsuspend_redraw(suspend_handle_id)` call has been made, (b) an `unsuspend_redraw_all()` call has been made, or (c) its timer has timed out. In environments that do not support interactivity (e.g., print media), then redraw should not be suspended. `suspend_handle_id = suspend_redraw(max_wait_milliseconds)` and `unsuspend_redraw(suspend_handle_id)` should be packaged as balanced pairs. When you want to suspend redraw actions as a collection of SVG DOM changes occur, then precede the changes to the SVG DOM with a method call similar to `suspend_handle_id = suspend_redraw(max_wait_milliseconds)` and follow the changes with a method call similar to `unsuspend_redraw(suspend_handle_id)`. Note that multiple `suspend_redraw` calls can be used at once and that each such method call is treated independently of the other `suspend_redraw` method calls.

Parameters

`max_wait_milliseconds`	The amount of time in milliseconds to hold off before redrawing the device. Values greater than 60 seconds will be truncated down to 60 seconds.

Return Value

A number which acts as a unique identifier for the given `suspend_redraw()` call. This value should be passed as the parameter to the corresponding `unsuspend_redraw()` method call.

This method raises no exceptions.

`unsuspend_redraw`

Cancels a specified `suspend_redraw()` by providing a unique `suspend_handle_id`.

Parameters

`suspend_handle_id`	A number which acts as a unique identifier for the desired `suspend_redraw()` call. The number supplied should be a value returned from a previous call to `suspend_redraw()`.

Return Value

None.

This method will raise a DOMException with value **NOT_FOUND_ERR** if an invalid value (i.e., no such `suspend_handle_id` is active) for `suspend_handle_id` is provided.

`unsuspend_redraw_all`

Cancels all currently active `suspend_redraw()` method calls. This method is most useful at the very end of a set of SVG DOM calls to ensure that all pending `suspend_redraw()` method calls have been cancelled.

Parameters

None.

Return Value

None.

This method raises no exceptions.

Interface *SVGElement*

All SVG element interfaces derive from this class.

IDL Definition

```
interface SVGElement : Element {
  readonly attribute  SVGDocument ownerSVGDocument;
  CSSStyleDeclaration  style;
};
```

Interface *SVGPathElement*

Corresponds to the <path> element.

IDL Definition

```
interface SVGPathElement : SVGElement {
  // Create an empty SVGPathSeg, specifying the type via a number.
  // All values initialized to zero.
  SVGPathSeg  createSVGPathSeg(in unsigned short pathsegType)
                                   raises(DOMException);

  // Create an empty SVGPathSeg, specifying the type via a single character.
  // All values initialized to zero.
  SVGPathSeg  createSVGPathSegFromLetter(in DOMString pathsegTypeAsLetter)
                                             raises(DOMException);

  // Create an SVGPathSeg, specifying the path segment as a string.
  // For example, "M 100 200". All irrelevant values are set to zero.
  SVGPathSeg  createSVGPathSegFromString(in DOMString pathsegString)
                                             raises(DOMException);

  // This set of functions allows retrieval and modification
  // to the path segments attached to this path object.
  // All 20 defined types of path segments are available
  // through these attributes and methods.

  readonly attribute unsigned long  number_of_pathsegs;
  SVGPathSeg      getSVGPathSeg(in unsigned long index);
  DOMString       getSVGPathSegAsString(in unsigned long index);
  SVGPathSeg      insertSVGPathSegBefore(in SVGPathSeg newSVGPathSeg,
                                         in unsigned long index)
                                         raises(DOMException);
  SVGPathSeg      replaceSVGPathSeg(in SVGPathSeg newSVGPathSeg,
                                    in unsigned long index)
                                    raises(DOMException);
  SVGPathSeg      removeSVGPathSeg(in unsigned long index)
```

```
                                     raises(DOMException);
  SVGPathSeg      appendSVGPathSeg(in SVGPathSeg newSVGPathSeg)
                                     raises(DOMException);

  // This alternate set of functions also allows retrieval and modification
  // to the path segments attached to this path object.
  // These attributes and methods provide a "normalized" view of
  // the path segments where the path is expressed in terms of
  // the following subset of SVGPathSeg types:
  // kSVG_PATHSEG_MOVETO_ABS (M), kSVG_PATHSEG_LINETO_ABS (L),
  // kSVG_PATHSEG_CURVETO_CUBIC_ABS (C) and kSVG_PATHSEG_CLOSEPATH (z).
  // Note that number_of_pathsegs and number_of_normalized_pathsegs
  // may not be the same. In particular, elements such as arcs may
  // be expanded into multiple kSVG_PATHSEG_CURVETO_CUBIC_ABS (C)
  // pieces when retrieved in the "normalized" view of the path object.

  readonly attribute unsigned long  number_of_normalized_pathsegs;
  SVGPathSeg      getNormalizedSVGPathSeg(in unsigned long index);
  DOMString       getNormalizedSVGPathSegAsString(in unsigned long index);
  SVGPathSeg      insertNormalizedSVGPathSegBefore(in SVGPathSeg newSVGPathSeg,
                                in unsigned long index)
                                raises(DOMException);
  SVGPathSeg      replaceNormalizedSVGPathSeg(in SVGPathSeg newSVGPathSeg,
                                in unsigned long index)
                                raises(DOMException);
  SVGPathSeg      removeNormalizedSVGPathSeg(in unsigned long index)
                                raises(DOMException);
  SVGPathSeg      appendNormalizedSVGPathSeg(in SVGPathSeg newSVGPathSeg)
                                raises(DOMException);
};
```

Attributes

Details will be provided later.

Methods

Details will be provided later.

Implementation Notes

The following is recommended behavior (not required to be conforming) for SVG user agents regarding **d="..."** and the DOM.

- The XML DOM already provides access to the raw strings as entered for the **d="..."** attributes on <path> and <data> elements. The recommended behavior is that if the path data has not been changed via the DOM, then any XML DOM calls to get the attribute values of the **d="..."** attributes should return (to the best degree possible) the strings as originally entered. (The simplest way to implement path data logic to make this possible is to retain all path data strings in their original form.)
- Whenever the path data has been changed via the SVG DOM, the SVG DOM should attempt to preserve as much of the original **d="..."** attribute strings as possible. All path segments that were on a particular <path> or <data> elements should stay with the original element. The exception is when modifications to the path data via the SVG DOM causes particular **d="..."** attribute strings to exceed the 1023 character limit. The rule should be as follows: whenever a change to the DOM causes a particular **d="..."** attribute to have more characters that 1023, take segments off the end of the given **d="..."** attribute and push onto the beginning of the **d="..."** attribute of the subsequent <data> element (creating a new such element if necessary). If pushing new segments onto the beginning of a subsequent <data> element causes its **d="..."** attribute to exceed 1023 characters, then take segments off of its **d="..."** attribute and push onto the beginning of the

d="..." attribute of its subsequent <data> element, until all **d="..."** attributes satisfy the 1023 character limit restriction.

Interface *SVGPathSeg*

Corresponds to a single segment within a path data specification.

IDL Definition

```
interface SVGPathSeg {
  // Path Segment Types
  const unsigned short kSVG_PATHSEG_UNKNOWN                      = 0;  // ?
  const unsigned short kSVG_PATHSEG_CLOSEPATH                    = 1;  // z
  const unsigned short kSVG_PATHSEG_MOVETO_ABS                   = 2;  // M
  const unsigned short kSVG_PATHSEG_MOVETO_REL                   = 3;  // m
  const unsigned short kSVG_PATHSEG_LINETO_ABS                   = 4;  // L
  const unsigned short kSVG_PATHSEG_LINETO_REL                   = 5;  // l
  const unsigned short kSVG_PATHSEG_CURVETO_CUBIC_ABS            = 6;  // C
  const unsigned short kSVG_PATHSEG_CURVETO_CUBIC_REL            = 7;  // c
  const unsigned short kSVG_PATHSEG_CURVETO_QUADRATIC_ABS        = 8;  // Q
  const unsigned short kSVG_PATHSEG_CURVETO_QUADRATIC_REL        = 9;  // q
  const unsigned short kSVG_PATHSEG_ARC_SWEEP_ABS                = 10; // A
  const unsigned short kSVG_PATHSEG_ARC_SWEEP_REL                = 11; // a
  const unsigned short kSVG_PATHSEG_ARC_VECTOR_ABS               = 12; // B
  const unsigned short kSVG_PATHSEG_ARC_VECTOR_REL               = 13; // b
  const unsigned short kSVG_PATHSEG_LINETO_HORIZONTAL_ABS        = 14; // H
  const unsigned short kSVG_PATHSEG_LINETO_HORIZONTAL_REL        = 15; // h
  const unsigned short kSVG_PATHSEG_LINETO_VERTICAL_ABS          = 16; // V
  const unsigned short kSVG_PATHSEG_LINETO_VERTICAL_REL          = 17; // v
  const unsigned short kSVG_PATHSEG_CURVETO_CUBIC_SMOOTH_ABS     = 18; // S
  const unsigned short kSVG_PATHSEG_CURVETO_CUBIC_SMOOTH_REL     = 19; // s
  const unsigned short kSVG_PATHSEG_CURVETO_QUADRATIC_SMOOTH_ABS = 20; // T
  const unsigned short kSVG_PATHSEG_CURVETO_QUADRATIC_SMOOTH_REL = 21; // t

  readonly attribute  unsigned short pathsegType;
  readonly attribute  DOMString      pathsegTypeAsLetter;

  // Attribute values for a path segment.
  // Each pathseg has slots for any possible path seg type.
  // (We don't want to define 20 different subclasses to PathSeg -
      our document is long enough already.)
  attribute  double       x;   // end point for pathseg (or center for A/a/B/b)
  attribute  double       y;   // end point for pathseg (or center for A/a/B/b)
  attribute  double       x0,y0;    // for control points (start point for B/b)
  attribute  double       x1,y1;    // for control points (end point for B/b)
  attribute  double       r1,r2;    // radii for A/a/B/b
  attribute  double       a1,a2,a3; // angles for A/a

  readonly attribute  Path parentPath;
  readonly attribute  SVGDocument ownerSVGDocument;
  readonly attribute  SVGPathSeg previousSibling;
  readonly attribute  SVGPathSeg nextSibling;
};
```

Appendix E: Sample SVG Files

Not yet written.

Appendix F: Accessibility Support

G.1 Accessibility and SVG

Drawings done in SVG will be much more accessible that drawings done as image formats for the following reasons:

- Text strings in SVG are represented as regular XML character data rather than bits in an image. (See Text.)
- At any place in the SVG hierarchy, a drawing can include a long set of descriptive text and/or a short description in the form of a title. Both of these features can be used to help the visually impaired interpret both the intent and specific content of a drawing. The drawing can be architected such that there is a single description for the drawing as a whole or there are multiple descriptions which are distributed within the drawing and describe each separate component within the drawing.
- Because of SVG's support of Cascading Style Sheets Level 2 (??? need to add link), there will be the ability to set up personal style sheets to adjust the color contrast of graphic elements.
- Because SVG documents are scalable, people with partial visual impairment will be able to zoom in on graphics for easier viewing.

(Additional information on accessibility is forthcoming.)

G.1 SVG Accessibility Guidelines

(??? Still under development)

Appendix G: Minimizing SVG File Sizes

Considerable effort has been made to make SVG file sizes as small as possible while still retaining the benefits of XML and achieving compatibility and leverage with other W3C specifications.

Here are some of the features in SVG that promote small file sizes:

- SVG's path data definition was defined to produce a compact data stream for vector graphics data: all commands are one character in length; relative coordinates are available; separator characters don't have to be supplied when tokens can be identified implicitly; smooth curve formulations are available (cubic beziers, quadratic beziers and elliptical arcs) to prevent the need to tesselate into polylines; and shortcut formulations exist for common forms of cubic bezier segments, quadratic bezier segments, and horizontal and vertical straight line segments so that the minimum number of coordinates need to be specified.
- Text can be specified using XML character data -- no need to convert to outlines.
- SVG contains a facility for defining symbols once and referencing them multiple times using different visual attributes and different sizing, positioning, clipping and client-side filter effects
- SVG supports CSS selectors and property inheritance, which allows commonly used sets of attributes to be defined once as named styles.
- Filter effects allow for compelling visual results and effects typically found only in image-authoring tools using small amounts of vector and/or raster data

Additionally, HTTP 1.1 allows for compressed data to be passed from server to client, which can result in significant file size reduction. Here are some sample compression results using gzip compression on SVG documents:

Uncompressed SVG	**With gzip compression**	**Compression ratio**
30,203	8,680	71%
12,563	8,048	83%
7,106	2,395	66%
6,216	2,310	63%
4,381	2,198	50%

A related issue is progressive rendering. Some SVG viewers will support:

- the ability to display the first parts of an SVG document as the remainder of the document is downloaded from the server; thus, the user will see part of the SVG drawing right away and interact with it, even if the SVG file size is large.
- delayed downloading of images and fonts. Just like some HTML browsers, some SVG viewers will download images and Web fonts last, substituting a temporary image and system fonts, respectively, until the given image and/or font is available.

Here are techniques for minimizing SVG file sizes and minimizing the time before the user is able to start interacting with the SVG document:

- Construct the SVG file such that any links which the user might want to click on are included at the beginning of the SVG file
- Use default values whenever possible rather than defining all attributes and properties explicitly.
- Take advantage of the path data data compaction facilities: use relative coordinates; use *h* and *v* for horizontal and vertical lines; use *s* or *t* for cubic and quadratic bezier segments whenever possible; eliminate extraneous white space and separators.
- Utilize symbols if the same graphic appears multiple times in the document
- Utilize CSS property inheritance and selectors to consolidate commonly used properties into named styles or to assign the properties to a parent <g> element.
- Utilize filter effects to help construct graphics via client-side graphics operations.

Appendix H: Implementation and Performance Notes for Fonts

Reliable delivery of fonts is considered a critical requirement for SVG. Designers should be able to create SVG graphics with whatever fonts they care to use and then the same fonts should appear in the end user's browser when viewing an SVG drawing, even if the given end user hasn't purchased the fonts in question. This parallels the print world, where the designer uses a given font when authoring a drawing for print, but when the end user views the same drawing within a magazine the text appears with the correct font.

SVG utilizes CSS2's WebFont facility as a key mechanism for reliable delivery of font data to end users. A common scenario is that SVG authoring applications will generate compressed, subsetted web fonts for all text elements included in a given SVG document. Typically, the web fonts will be saved in a nearby location to the SVG document itself.

Appendix I. References

Contents

- I.1 Normative references
- I.2 Informative references

I.1 Normative references

[COLORIMETRY]

"Colorimetry, Second Edition", CIE Publication 15.2-1986, ISBN 3-900-734-00-3.
Available at http://www.hike.te.chiba-u.ac.jp/ikeda/CIE/publ/abst/15-2-86.html.

[CSS2]

"Cascading Style Sheets, level 2", B. Bos, H. W. Lie, C. Lilley, I. Jacobs, 12 May 1998.
Available at http://www.w3.org/TR/REC-CSS2.

[HTML40]

"HTML 4.0 Specification", D. Raggett, A. Le Hors, I. Jacobs, 8 July 1997.
Available at http://www.w3.org/TR/REC-html40/. The Recommendation defines three document type definitions: Strict, Transitional, and Frameset, all reachable from the Recommendation.

[ICC32]

"ICC Profile Format Specification, version 3.2", 1995.
Available at ftp://sgigate.sgi.com/pub/icc/ICC32.pdf.

"Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", N. Freed and N. Borenstein, November 1996.
Available at ftp://ftp.internic.net/rfc/rfc2045.txt. Note that this RFC obsoletes RFC1521, RFC1522, and RFC1590.

[RFC2068]

"HTTP Version 1.1 ", R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, January 1997.
Available at ftp://ftp.internic.net/rfc/rfc2068.txt.

[SRGB]

"Proposal for a Standard Color Space for the Internet - sRGB", M. Anderson, R. Motta, S. Chandrasekar, M. Stokes.
Available at http://www.w3.org/Graphics/Color/sRGB.html.

[UNICODE]

"The Unicode Standard: Version 2.0", The Unicode Consortium, Addison-Wesley Developers Press, 1996. For bidirectionality, see also the corrigenda at http://www.unicode.org/unicode/uni2errata/bidi.htm. For more information, consult the Unicode

Consortium's home page at http://www.unicode.org/.

The latest version of Unicode. For more information, consult the Unicode Consortium's home page at http://www.unicode.org/.

[URI]

"Uniform Resource Identifiers (URI): Generic Syntax and Semantics", T. Berners-Lee, R. Fielding, L. Masinter, 18 November 1997.
Available at http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-01.txt. This is a work in progress that is expected to update [RFC1738] and [RFC1808].

[XML10]

"Extensible Markup Language (XML) 1.0" T. Bray, J. Paoli, C.M. Sperberg-McQueen, editors, 10 February 1998.
Available at http://www.w3.org/TR/REC-xml/.

I.2 Informative references

[DOM]

"Document Object Model Specification", L. Wood, A. Le Hors, 9 October 1997.
Available at http://www.w3.org/TR/WD-DOM/

[WAI-PAGEAUTH]

"WAI Accesibility Guidelines: Page Authoring" for designing accessible documents are available at:
http://www.w3.org/TR/WD-WAI-PAGEAUTH.

Property Index

This will contain the property index

Index

This will contain the index