W3C Working Draft

Scalable Vector Graphics (SVG) Tiny 1.2 Specification

W3C Working Draft *13 April 2005*

This version:
http://www.w3.org/TR/2005/WD-SVGMobile12-20050413/
Previous version:
http://www.w3.org/TR/2004/WD-SVGMobile12-20040813/
Latest SVG Mobile 1.2 version:
http://www.w3.org/TR/SVGMobile12/
Latest SVG Mobile Recommendation:
http://www.w3.org/TR/SVGMobile/
Editors:
Ola Andersson (Ikivo) <ola.andersson@ikivo.com>
Jon Ferraiolo (Adobe Systems) <jon.ferraiolo@adobe.com>
Vincent Hardy (Sun Microsystems, Inc.) <vincent.hardy@sun.com>
Dean Jackson (W3C) <dean@w3.org>
Antoine Quint (Invited Expert) <aq@fuchsia-design.com>
Authors:
See author list

Abstract

This specification defines the features and syntax for Scalable Vector Graphics (SVG) Tiny, Version 1.2, a modularized language for describing two-dimensional vector and mixed vector/raster graphics in XML. SVG Tiny 1.2 is the baseline profile of SVG, implementable on a range of devices from cellphones and PDAs to desktop and laptop computers, and is the core of SVG 1.2. Other SVG 1.2 specifications will extend this functionality to form supersets (for example, SVG 1.2 Full).

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a W3C Last Call Working Draft. If the feedback is positive, the SVG Working Group plans to submit this specification for consideration as a W3C Candidate Recommendation. Comments for this specification should have a subject starting with the prefix '[SVGMobile12]'. Please send them to www-svg@w3.org, the public email list for issues related to SVG. This list is archived and acceptance of this archiving policy is requested automatically upon first post. To subscribe to this list send an email to www-svg-request@w3.org with the word subscribe in the subject line. Comments are accepted until 20 May 2005.

Previous versions of this specification expressed the SVG Tiny 1.2 language as a profile; a set of pointers to portions of the SVG 1.1 and SVG 1.2 specifications plus a list of restrictions. Feedback from implementors and reviewers indicated that this was hard to follow and may have lacked precision. In this version, SVG Tiny 1.2 is described as a complete language specification, with no dependencies on other SVG specifications. This makes it a first class citizen, and is easier to read and to implement.

This document has been produced by the SVG Working Group as part of the W3C Graphics Activity, following the procedures set out for the W3C Process. The authors of this document are listed at the end in the Author List section.

The patent policy for this document is the 5 February 2004 W3C Patent Policy. Patent disclosures relevant to this specification may be found on the SVG Working Group's patent disclosure page. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy.

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

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.

Table of Contents

The authors of the SVG Tiny 1.2 specification are the people who participated in the SVG Working Group as members or alternates.

Authors:

- Ola Andersson, Ikivo
- Phil Armstrong, Corel Corporation
- Henric Axelsson, Ericsson AB
- Selim BalcÄ±soy, Nokia
- Robin Berjon, Expway
- BenoÃ®t BÃ©zaire, Itedo (formerly Corel Corporation)
- John Bowler, Microsoft Corporation
- Gordon Bowman, Corel Corporation
- Craig Brown, Canon Information Systems Research Australia
- Mike Bultrowicz, Savage Software
- Tolga Ã‡apin, Nokia
- Milt Capsimalis, Autodesk Inc.
- Mathias Larsson Carlander, Ericsson AB
- Jakob Cederquist, Ikivo
- Suresh Chitturi, Nokia
- Charilaos Christopoulos, Ericsson AB
- Richard Cohn, Adobe Systems Inc.
- Lee Cole, Quark
- Cyril Concolato, Groupe des Ecoles des TÃ©lÃ©communications (GET)
- Don Cone, America Online Inc.
- Alex Danilo, Canon Information Systems Research Australia
- Thomas DeWeese, Eastman Kodak
- David Dodds, Lexica
- Andrew Donoho, IBM
- David Duce, Oxford Brookes University
- Jean-Claude Dufourd, Streamezzo (formerly GET)
- Jerry Evans, Sun Microsystems
- Jon Ferraiolo, Adobe Systems Inc.
- Darryl Fuller, Schema Software
- è—¤æ²¢ æ·³ (FUJISAWA Jun), Canon
- Scott Furman, Netscape Communications Corporation
- Brent Getlin, Macromedia
- Christophe Gillette, Motorola (formerly BitFlash)
- Peter Graffagnino, Apple
- Rick Graham, BitFlash
- Vincent Hardy, Sun Microsystems Inc.
- ç«¯å±± è²´ä¹Ÿ (HAYAMA Takanari), KDDI Research Labs
- Scott Hayman, Research In Motion Limited
- Stephane Heintz, BitFlash
- Lofton Henderson, OASIS
- Ivan Herman, W3C
- Jan Christian Herlitz, Excosoft
- Alan Hester, Xerox Corporation
- Bob Hopgood, RAL (CCLRC)
- Bin Hu, Motorola
- Michael Ingrassia, Nokia
- çŸ³å· é›…åº· (ISHIKAWA Masayasu), W3C
- Dean Jackson, W3C (*W3C Team Contact*)
- Christophe Jolif, ILOG S.A.
- Lee Klosterman, Hewlett-Packard
- å°æž— äºœä»¤ (KOBAYASHI Arei), KDDI Research Labs
- Thierry Kormann, ILOG S.A.
- Yuri Khramov, Schema Software
- Kelvin Lawrence, IBM
- HÃ¥kon Lie, Opera
- Chris Lilley, W3C (*Working Group Chair*)
- Vincent Mahe, France Telecom
- Philip Mansfield, Schema Software
- Kevin McCluskey, Netscape Communications Corporation
- æ°´å£ å…… (MINAKUCHI Mitsuru), Sharp Corporation
- Luc Minnebo, Agfa-Gevaert N.V.
- Jean-Claude Moissinac, Groupe des Ecoles des TÃ©lÃ©communications (GET)
- Craig Northway, Canon Information Systems Research Australia
- Tuan Nguyen, Microsoft Corporation
- å°é‡Ž ä¿®ä¸€éƒŽ (ONO Shuichiro), Sharp Corporation
- Lars Piepel, Vodafone
- Antoine Quint, Fuchsia Design (formerly of ILOG)
- à¤£à¤¨à¥à¤¦à¤¨ à¤¨à¤¾à¤®à¤¨à¤¿ (Nandini Ramani), Sun Microsystems
- Bruno David SimÃµes Rodrigues, Vodafone

- ç›¸è‰¯ æ¯… (SAGARA Takeshi), KDDI Research Labs
- Troy Sandal, Visio Corporation
- Peter Santangeli, Macromedia
- Sebastian Schnitzenbaumer, SAP AG
- Haroon Sheikh, Corel Corporation
- Brad Sipes, Ikivo
- Andrew Sledd, Ikivo
- ÐŸÑ‘Ñ‚Ñ€ Ð¡Ð¾Ñ€Ð¾ÐºÐ¸Ð½ (Peter Sorotokin), Adobe Systems Inc.
- Gavriel State, Corel Corporation
- Robert Stevahn, Hewlett-Packard
- Timothy Thompson, Eastman Kodak
- ä¸Šç”° å®é«˜ (UEDA Hirotaka), Sharp Corporation
- Rick Yardumian, Canon Development Americas
- Charles Ying, Openwave Systems Inc.
- Shenxue Zhou, Quark

Acknowledgments

The SVG Working Group would like to acknowledge the many people outside of the SVG Working Group who help with the process of developing the SVG specification. These people are too numerous to list individually. They include but are not limited to the early implementers of the SVG languages (including viewers, authoring tools, and server-side transcoders), developers of SVG content, people who have contributed on the www-svg@w3.org and svg-developers@yahoogroups.com email lists, other Working Groups at the W3C, and the W3C Team. SVG is truly a cooperative effort between the SVG Working Group, the rest of the W3C, and the public and benefits greatly from the pioneering work of early implementers and content developers, feedback from the public.

1 Introduction

Contents

1.1 About SVG

SVG is a language for describing two-dimensional graphics in XML [XML10]. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), multimedia (such as raster images and video) and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects.

SVG drawings can be interactive and dynamic. Animations can be defined and triggered either declaratively (i.e., by embedding SVG animation elements in SVG content) or via scripting.

Sophisticated applications of SVG are possible by use of a supplemental scripting language which accesses the SVG Micro Document Object Model (uDOM), which provides complete access to all elements, attributes and properties. A rich set of event handlers 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 XHTML and SVG elements simultaneously within the same Web page.

SVG is a language for rich graphical content. For accessibility reasons, if there is an original source document containing higher-level structure and semantics, it is recommended that the higher-level information be made available somehow, either by making the original source document available, or making an alternative version available in a format which conveys the higher-level information, or by using SVG's facilities to include the higher-level information within the SVG content. For suggested techniques in achieving greater accessibility, see Accessibility.

It is believed that this specification is in conformance with the Web Architecture [AWWW].

1.2 SVG Tiny 1.2

Industry demand, overwhelming support in the SVG working group and requests from the SVG developer community established the need for some form of SVG suited to displaying vector graphics on small devices. Moreover, the mission statement of SVG 1.0 specifically addressed small devices as a target area for vector graphics display. In order to meet these demands the SVG Working Group created a profile specification that was suitable for use on mobile devices as well as on desktops. The SVG Mobile 1.1 specification addressed that requirement and defined two profiles to deal with the variety of mobile devices having different characteristics in terms of CPU speed, memory size, and color support. The SVG Mobile 1.1 specification defined SVG Tiny (SVGT) 1.1, suitable for highly restricted mobile devices; it also defined a second profile, SVG Basic (SVGB) 1.1, targeted for higher level mobile devices. The major difference between SVG Tiny 1.1 and SVG Basic 1.1 was the absence of scripting and styling in SVG 1.1 Tiny, and thus any requirement to maintain a Document Object Model (DOM). This saved a substantial amount of memory in most implementations.

Experience with SVG Tiny 1.1, which was widely adopted in the industry and shipped as standard on a variety of cellphones, indicated that the profile was a little too restrictive in some areas. Features from SVG 1.1 such as gradients and opacity, were seen to have substantial value for creating attractive content, and were shown to be implementable on cellphones. There was also considerable interest in adding audio and video capabilities, building on the SMIL support in SVG Tiny 1.1.

Advances such as DOM Level 3, which introduces namespace support and value normalization, prompted a second look at the use of programming languages and scripting with SVG Tiny. In conjunction with the Java JSR 226 group [JSR226], a lightweight interface called the microDOM, or uDOM, was developed. This could be, but need not be, implemented on top of DOM Level 3. With this advance, lightweight programatic control of SVG (for example, for games or user interfaces) and use with scripting languages, became feasible on the whole range of platforms from cellphones through to desktops. In consequence, there is only a single Mobile profile for SVG 1.2 - SVG Tiny 1.2

This specification defines the features and syntax for Scalable Vector Graphics (SVG) Tiny 1.2, the core specification and baseline profile of SVG 1.2. Other SVG specifications will extend this baseline functionality to create supersets (for example, SVG 1.2 Full). The SVG Tiny 1.2 specification adds to SVG Tiny 1.1 features requested by SVG authors, implementors and users; SVG Tiny 1.2 is a superset of SVG Tiny 1.1.

1.2.1 Modularization

This specification describes a collection of abstract modules that provide specific units of functionality. These modules may be combined with each other and with modules defined in other specifications (such as XHTML) to create SVG subset and extension document types that qualify as members of the SVG family of

document types. See Conformance for a description of SVG family documents, and [XHTMLplusMathMLplusSVG] for a profile that combines XHTML, MathML and SVG.

Each major section of the SVG specification produces a module named after that section, e.g. "Text Module" or "Structure Module".

1.2.2 Element and Attribute collections

Modules define a named collection of either elements or attributes. These collections are used as a shorthand when describing the set of attributes allowed on a particular element (eg. the "Style" attibute collection) or the set of elements allowed as children of a particular element (eg. the "Shape" element collection). All collections have names that begin with an uppercase character.

When defining a profile, it is assumed that all the element and attribute collections are defined to be empty. That way, a module can redefine the collection as it is included in the profile, adding elements or attributes to make them available within the profile. Therefore, it is not a mistake to refer to an element or attribute collection from a module that is not included in the profile, it simply means that collection is empty.

1.2.3 Profiling the SVG specification

The modularization of SVG 1.2 allows profiles to be described by listing the SVG modules they allow and possibly a small number of restrictions or extensions on the elements provided by those modules.

The "Tiny" profile of SVG 1.2 is the collection of all the modules listed in this specification.

When applied to conformance, the term "SVG Tiny" refers to the "Tiny" profile of SVG 1.2 defined by this specification. If an implementation does not implement the Tiny profile, it must state either the profile to which it conforms, or that it implements a subset of SVG Tiny.

1.3 Defining an SVG Tiny 1.2 document

SVG Tiny 1.2 is a backwards compatible upgrade to SVG Tiny 1.1 . A few key differences from SVG Tiny 1.1 should be noted:

- The value of the version attribute on the root-most svg element must be "1.2".
- There is no DTD for SVG 1.2, and therefore no need to specify the DOCTYPE for an SVG 1.2 document (unless it is desired to use the internal DTD subset, for purposes of entity definitions for example). Instead, identification is by the SVG namespace, plus the version and baseProfile attributes. In SVG Tiny 1.2, validation is provided by the SVG Tiny 1.2 RelaxNG schema.

The namespace for SVG Tiny 1.2 is the same as that of SVG 1.0 and 1.1, `http://www.w3.org/2000/svg`

Here is an example of an SVG 1.2 file:

Example: 01_01.svg

```
<?xml version="1.0"?>
<svg xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny"
    viewBox="0 0 30 30">
    <desc>Example SVG file</desc>
        <rect x="10" y="10" width="10" height="10" fill="red"/>
</svg>
```

Here is an example of defining an entity in the internal DTD subset. Note that in XML, there is no requirement to fetch the external DTD subset and so relying on an external subset reduces interoperability. Also note that the SVG Working Group does not provide a normative DTD for SVG Tiny 1.2 but instead provides a normative RelaxNG schema.

Example: entity.svg

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.2 Tiny//EN" [
    <!ENTITY Smile "
    <rect x='.5' y='.5' width='29' height='39' fill='black' stroke='red'/>
    <g transform='translate(0, 5)'>
          <circle cx='15' cy='15' r='10' fill='yellow'/>
          <circle cx='12' cy='12' r='1.5' fill='black'/>
          <circle cx='17' cy='12' r='1.5' fill='black'/>
          <path d='M 10 19 L 15 23 20 19' stroke='black' stroke-width='2'/>
    </g>
    ">
    ]>
<svg xmlns="http://www.w3.org/2000/svg">
   <desc>This example shows an entity from a DOCTYPE declaration being used.</desc>
   &Smile;
</svg>
```

1.4 SVG MIME type, file name extension and Macintosh file type

The MIME type for SVG is "`image/svg+xml`" (see Media Type registration for image/svg+xml).

It is recommended that SVG files have the extension "`.svg`" (all lowercase) on all platforms. It is recommended that gzip-compressed SVG files have the extension "`.svgz`" (all lowercase) on all platforms.

It is recommended that SVG files stored on Macintosh HFS file systems be given a file type of "`svg`" (all lowercase, with a space character as the fourth letter). It is recommended that gzip-compressed SVG files stored on Macintosh HFS file systems be given a file type of "`svgz`" (all lowercase).

1.5 Compatibility with Other Standards Efforts

SVG Tiny 1.2 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.

The following describes some of the ways in which SVG maintains compatibility with, leverages and integrates with other W3C efforts:

- SVG Tiny 1.2 is an application of XML and is compatible with the "Extensible Markup Language (XML) 1.0" Recommendation [XML10]
- SVG Tiny 1.2 is compatible with the "Namespaces in XML 1.1" Recommendation [XML-NS]
- SVG Tiny 1.2 utilizes "XML Linking Language (XLink)" [XLINK] for IRI referencing and requires support for base IRI specifications defined in "XML Base" [XML-BASE].

- SVG Tiny 1.2 uses the xml:id attribute [XMLID].
- SVG Tiny 1.2 content can be generated by XSLT (see "XSL Transformations (XSLT) Version 1.0" [XSLT]). (See Styling with XSL)
- SVG Tiny 1.2 supports formatting properties drawn from CSS and XSL. See SVG's styling properties.
- SVG Tiny 1.2 includes a compatible subset of the Document Object Model (DOM) and supports many of the facilities described in "Document Object Model (DOM) level 3" [DOM3], including namespace support and event handling.
- SVG Tiny 1.2 incorporates some features from "Synchronized Multimedia Integration Language (SMIL) 2.0 Specification" [SMIL2.0], including the 'switch' element, the **systemLanguage** attribute, animation features (see Animation) and the ability to reference audio and video media (see Multimedia). SVG's animation features incorporate and extend the general-purpose XML animation capabilities described in SMIL 2.0. In addition, SVG Tiny 1.2 has been designed to allow SMIL 2.0 to use animated or static SVG content as media components.
- SVG is compatible with W3C work on internationalization. References (W3C and otherwise) include: [UNICODE], [CHARMOD-RI] and [CHARMOD]. Also, see Internationalization Support.
- SVG is compatible with W3C work on Web Accessibility [WAI]. Also, see Accessibility Support.

In environments which support [DOM3] for other XML grammars (e.g., XHTML [XHTML]) and which also support SVG and the SVG DOM, a single scripting approach can be used simultaneously for both XML documents and SVG graphics, in which case interactive and dynamic effects will be possible on multiple XML namespaces using the same set of scripts.

1.6 Definitions

basic shape
Standard shapes which are predefined in SVG as a convenience for common graphical operations. Specifically: 'rect', 'circle', 'ellipse', 'line', 'polyline', 'polygon'.

canvas
A surface onto which graphics elements are drawn, which can be real physical media such as a display or paper or an abstract surface such as a allocated region of computer memory. See the discussion of the SVG canvas in the chapter on Coordinate Systems, Transformations and Units.

container element
An element which can have graphics elements and other container elements as child elements. Specifically: 'svg', 'g', 'defs', 'a' and 'switch'.

current SVG document fragment
The XML document sub-tree which starts with parent 'svg' element of a given SVG element, with the requirement that all container elements between the outermost 'svg' and this element are all elements in the SVG language.

current transformation matrix (CTM)
Transformation matrices define the mathematical mapping from one coordinate system into another using a 3x3 matrix using the equation **[x' y' 1] = [x y 1] * matrix**. The *current transformation matrix* (CTM) defines the mapping from the user coordinate system into the viewport coordinate system. See Coordinate system transformations.

fill
The operation of painting the interior of a shape or the interior of the character glyphs in a text string.

font
A font represents an organized collection of glyphs in which the various glyph representations will share a common look or styling such that, when a string of characters is rendered together, the result is highly legible, conveys a particular artistic style and provides consistent inter-character alignment and spacing.

glyph
A glyph represents a unit of rendered content within a font. Often, there is a one-to-one correspondence between characters to be drawn and corresponding glyphs (e.g., often, the character "A" is rendered using a single glyph), but other times multiple glyphs are used to render a single character (e.g., use of accents) or a single glyph can be used to render multiple characters (e.g., ligatures). Typically, a glyph is defined by one or more shapes such as a path, possibly with additional information such as rendering hints that help a font engine to produce legible text in small sizes.

graphics element
One of the element types that can cause graphics to be drawn onto the target canvas. Specifically: 'path', 'text', 'textArea', 'rect', 'circle', 'ellipse', 'line', 'polyline', 'polygon', 'image', 'use', 'animation' and 'video'.

graphics referencing element
A graphics element which uses a reference to a different document or element as the source of its graphical content. Specifically: 'use', 'image', 'animation' and 'video'.

in error
A value is 'in error' if it is specifically stated as being 'in error' or 'an error' in the prose of this specification. See Error Processing for more detail on handling errors.

IRI Reference
An International Resource Identifier [IRI] which serves as a reference to a resource or (with a fragment identifier) to a secondary resource. See References.

local IRI reference
An International Resource Identifier [IRI] that does not include an **<absoluteIRI>** or **<relativeIRI>** and thus represents a reference to an element within the current document. See References.

non-local IRI reference
An International Resource Identifier [IRI] that includes an **<absoluteIRI>** or **<relativeIRI>** and thus (usually) represents a reference to a different document or an element within a different document. See References and the 'defs' element.

paint
A paint represents a way of putting color values onto the canvas. A paint might consist of both color values and associated alpha values which control the blending of colors against already existing color values on the canvas. SVG supports two types of built-in paint: color and gradients.

presentation attribute
An XML attribute on an SVG element which specifies a value for a given property for that element. See Styling.

property
A parameter that helps specify how a document should be rendered. A complete list of SVG's properties can be found in Property Index. Properties are assigned to elements in the SVG language by presentation attributes. See Styling.

shape
A graphics element that is defined by some combination of straight lines and curves. Specifically: 'path', 'rect', 'circle', 'ellipse', 'line', 'polyline', 'polygon'.

stroke
The operation of painting the outline of a shape or the outline of character glyphs in a text string.

SVG canvas

The canvas onto which the SVG content is rendered. See the discussion of the SVG canvas in the chapter on Coordinate Systems, Transformations and Units.

SVG document fragment
The XML document sub-tree which starts with an 'svg' element. An SVG document fragment can consist of a stand-alone SVG document, or a fragment of a parent XML document enclosed by an 'svg' element. When an 'svg' element is a descendant of another 'svg' element, there are two SVG document fragments, one for each 'svg' element. (One SVG document fragment is contained within another SVG document fragment.)

SVG viewport
The viewport within the SVG canvas which defines the rectangular region into which SVG content is rendered. See the discussion of the SVG viewport in the chapter on Coordinate Systems, Transformations and Units.

text content element
One of SVG's elements that can define a text string that is to be rendered onto the canvas. SVG Tiny 1.2's text content elements are the following: 'text' and 'tspan'.

transformation
A modification of the current transformation matrix (CTM) by providing a supplemental transformation in the form of a set of simple transformations specifications (such as scaling, rotation or translation) and/or one or more transformation matrices. See Coordinate system transformations.

transformation matrix
Transformation matrices define the mathematical mapping from one coordinate system into another using a 3x3 matrix using the equation **[x' y' 1] = [x y 1] * matrix**. See current transformation matrix (CTM) and Coordinate system transformations.

unsupported value
An unsupported value is a value that does not conform to this specification, but is not specifically listed as 'in error'. See the Implementation Notes for more detail on processing unsupported values.

user agent
The general definition of a user agent is an application that retrieves and renders Web content, including text, graphics, sounds, video, images, and other content types. A user agent may require additional user agents that handle some types of content. For instance, a browser may run a separate program or plug-in to render sound or video. User agents include graphical desktop browsers, multimedia players, text browsers, voice browsers, and assistive technologies such as screen readers, screen magnifiers, speech synthesizers, onscreen keyboards, and voice input software.

A "user agent" may or may not have the ability to retrieve and render SVG content; however, an "SVG user agent" retrieves and renders SVG content.

user coordinate system
In general, a coordinate system defines locations and distances on the current canvas. The current user coordinate system is the coordinate system that is currently active and which is used to define how coordinates and lengths are located and computed, respectively, on the current canvas. See initial user coordinate system and Coordinate system transformations.

user space
A synonym for user coordinate system.

user units
A coordinate value or length expressed in user units represents a coordinate value or length in the current user coordinate system. Thus, 10 user units represents a length of 10 units in the current user coordinate system.

viewport
A rectangular region within the current canvas onto which graphics elements are to be rendered. See the discussion of the SVG viewport in the chapter on Coordinate Systems, Transformations and Units.

viewport coordinate system
In general, a coordinate system defines locations and distances on the current canvas. The viewport coordinate system is the coordinate system that is active at the start of processing of an 'svg' element, before processing the optional viewBox attribute. In the case of an SVG document fragment that is embedded within a parent document which uses CSS to manage its layout, then the viewport coordinate system will have the same orientation and lengths as in CSS, with the origin at the top-left on the viewport. See The initial viewport and Establishing a new viewport.

viewport space
A synonym for viewport coordinate system.

viewport units
A coordinate value or length expressed in viewport units represents a coordinate value or length in the viewport coordinate system. Thus, 10 viewport units represents a length of 10 units in the viewport coordinate system.

2 Concepts

Contents

2.1 Explaining the name: SVG

SVG stands for Scalable Vector Graphics, an XML grammar for 2D vector graphics, usable as an XML namespace.

2.1.1 Scalable

To be scalable means to increase or decrease uniformly. In terms of graphics, scalable means not being limited to a single, fixed, pixel size. On the Web, scalable means that a particular technology can grow to a large number of files, a large number of users, a wide variety of applications. SVG, being a graphics technology for the Web, is scalable in both senses of the word.

SVG graphics are scalable to different display resolutions, so that for example printed output uses the full resolution of the printer and can be displayed at the same size on screens of different resolutions. The same SVG graphic can be placed at different sizes on the same Web page, and re-used at different sizes on different pages. SVG graphics can be magnified to see fine detail, or to aid those with low vision.

SVG graphics are scalable because the same SVG content can be a stand-alone graphic or can be referenced or included inside other SVG graphics, thereby allowing a complex illustration to be built up in parts, perhaps by several people. The use and font capabilities promote re-use of graphical components, maximize the advantages of HTTP caching and avoid the need for a centralized registry of approved symbols.

2.1.2 Vector

Vector graphics contain geometric objects such as lines and curves. This gives greater flexibility compared to raster-only formats (such as PNG and JPEG) which have to store information for every pixel of the graphic. Typically, vector formats can also integrate raster images and can combine them with vector information to produce a complete illustration; SVG is no exception.

Since all modern displays are raster-oriented, the difference between raster-only and vector graphics comes down to where they are rasterized; client side in the case of vector graphics, as opposed to already rasterized on the server. SVG gives control over the rasterization process, for example to allow anti-aliased artwork without the ugly aliasing typical of low quality vector implementations.

2.1.3 Graphics

Most existing XML grammars represent either textual information, or represent raw data such as financial information. They typically provide only rudimentary graphical capabilities, often less capable than the HTML 'img' element. SVG fills a gap in the market by providing a rich, structured description of vector and mixed vector/raster graphics; it can be used stand-alone, or as an XML namespace with other grammars.

2.1.4 XML

XML, a W3C Recommendation for structured information exchange, has become extremely popular and is both widely and reliably implemented. By being written in XML, SVG builds on this strong foundation and gains many advantages such as a sound basis for internationalization, powerful structuring capability, an object model, and so on. By building on existing, cleanly-implemented specifications, XML-based grammars are open to implementation without a huge reverse engineering effort.

2.1.5 Namespace

It is certainly useful to have a stand-alone, SVG-only viewer. But SVG is also intended to be used as one component in a multi-namespace XML application. This multiplies the power of each of the namespaces used, to allow innovative new content to be created. For example, SVG graphics may be included in a document which uses any text-oriented XML namespace - including XHTML. A scientific document, for example, might also use MathML for mathematics in the document. The combination of SVG and SMIL leads to interesting, time based, graphically rich presentations.

SVG is a good, general-purpose component for any multi-namespace grammar that needs to use graphics.

2.1.6 Scriptable

The combination of scripting and the HTML DOM is often termed "Dynamic HTML" and is widely used for animation, interactivity and presentational effects. Similarly SVG allows the script-based manipulation of the document tree using a subset of the XML DOM and the SVG UDOM.

2.2 Important SVG concepts

2.2.1 Graphical Objects

With any XML grammar, consideration has to be given to what exactly is being modelled. For textual formats, modelling is typically at the level of paragraphs and phrases, rather than individual nouns, adverbs, or phonemes. Similarly, SVG models graphics at the level of graphical objects rather than individual points.

SVG provides a general path element, which can be used to create a huge variety of graphical objects, and also provides common basic shapes such as rectangles and ellipses. These are convenient for hand coding and may be used in the same ways as the more general path element. SVG provides fine control over the coordinate system in which graphical objects are defined and the transformations that will be applied during rendering.

2.2.2 Reuse

It would have been possible to define some standard, pre-defined graphics that all SVG implementations would provide. But which ones? There would always be additional symbols for electronics, cartography, flowcharts, etc., that people would need that were not provided until the "next version". SVG allows users to create, re-use and share their own graphical assets without requiring a centralized registry. Communities of users can create and refine the graphics that they need, without having to ask a committee. Designers can be sure exactly of the graphical appearance of the graphics they use and not have to worry about unsupported graphics.

Graphics may be re-used at different sizes and orientations.

2.2.3 Fonts

Graphically rich material is often highly dependent on the particular font used and the exact spacing of the glyphs. In many cases, designers convert text to outlines to avoid any font substitution problems. This means that the original text is not present and thus searchability and accessibility suffer. In response to feedback from designers, SVG includes font elements so that both text and graphical appearance are preserved.

2.2.4 Animation

Animation can be produced via script-based manipulation of the document, but scripts are difficult to edit and interchange between authoring tools is harder. Again in response to feedback from the design community, SVG includes declarative animation elements which were designed collaboratively by the SVG and SYMM Working Groups. This allows the animated effects common in existing Web graphics to be expressed in SVG.

2.3 Options for using SVG in Web pages

There are a variety of ways in which SVG content can be included within a Web page. Here are some of the options:

- **A stand-alone SVG Web page**

  In this case, an SVG document (i.e., a Web resource whose MIME type is "`image/svg+xml`") is loaded directly into a user agent such as a Web browser. The SVG document is the Web page that is presented to the user.
- **Embedding by reference**

  In this case, a parent Web page references a separately stored SVG document and specifies that the given SVG document should be embedded as a component of the parent Web page. For HTML or XHTML, here are three options:
  - The HTML/XHTML **'img'** element is the most common method for using graphics in HTML pages. For faster display, the width and height of the image can be given as attributes. One attribute that is required is **alt**, used to give an alternate textual string for people browsing with images off, or who cannot see the images. The string cannot contain any markup. A **longdesc** attribute lets you point to a longer description - often in HTML - which can have markup and richer formatting.
  - The HTML/XHTML **'object'** element can contain other elements nested within it, unlike **'img'**, which is empty. This means that several different formats can be offered, using nested **'object'** elements, with a final textual alternative (including markup, links, etc). The outermost element which can be displayed will be used.
  - The HTML/XHTML **'applet'** element which can invoke a Java applet to view SVG content within the given Web page. These applets can do many things, but a common task is to use them to display images, particularly ones in unusual formats or which need to be presented under the control of a program for some other reason.
- **Embedding inline**

  In this case, SVG content is embedded inline directly within the parent Web page. An example is an XHTML Web page with an SVG document fragment textually included within the XHTML.
- **External link, using the HTML 'a' element**

  This allows any stand-alone SVG viewer to be used, which can (but need not) be a different program to that used to display HTML. This option typically is used for unusual image formats.
- **Referenced from a CSS2 or XSL property**

  When a user agent supports CSS-styled XML content or XSL Formatting Objects and the user agent is a Conforming SVG Viewer, then that user agent must support the ability to reference SVG resources wherever CSS or XSL properties allow for the referencing of raster images, including the ability to tile SVG graphics wherever necessary and the ability to composite the SVG into the background if it has transparent portions. Examples include the **'background-image'** and **'list-style-image'** properties that are included in both CSS and XSL.

SVG Tiny 1.2 - 20050413 Top Contents Previous Next Elements Attributes

3 Rendering Model

Contents

3.1 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 shall match that described by this model.

The appendix 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 gamut of colors might be supported) and because of practical limitations in implementing a precise mathematical model (e.g. for realistic performance curves may be approximated by straight lines, the approximation need only be sufficiently precise to match the conformance requirements).

3.2 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. When the paint is not completely opaque the result on the output device is defined by the (mathematical) rules for compositing described under Alpha Blending.

3.3 Rendering Order

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

3.4 Types of graphics elements

SVG supports four fundamental types of graphics elements that can be rendered onto the canvas:

- Shapes, which represent some combination of straight line and curves
- Text, which represents some combination of character glyphs
- Raster images, which represent an array of values that specify the paint color and opacity (often termed alpha) at a series of points on a rectangular grid. (SVG requires support for specified raster image formats under conformance requirements.)
- Video, which represents a timed sequence of raster images.

3.4.1 Painting shapes and text

Shapes and text can be filled (i.e., apply paint to the interior of the shape) and stroked (i.e., apply paint along the outline of the shape). A stroke operation is centered on the outline of the object; thus, in effect, half of the paint falls on the interior of the shape and half of the paint falls outside of the shape.

The fill is painted first, then the stroke.

Each fill and stroke operation has its own opacity settings; thus, you can fill and/or stroke a shape with a semi-transparently drawn solid color, with different opacity values for the fill and stroke operations.

The fill and stroke operations are entirely independent painting operations; thus, if you both fill and stroke a shape, half of the stroke will be painted on top of part of the fill.

SVG Tiny supports the following built-in types of paint which can be used in fill and stroke operations:

- Solid color
- Gradients (linear and radial)

3.4.2 Painting raster images

When a raster 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.

3.4.3 Painting video

As a video stream is a timed sequence of raster images, rendering video has some similarity to painting raster images. However, given the processing required to decode a video stream, not all implementations may be able to transform the video output into SVG's userspace. Instead they may be limited to rendering in device space. More information can be found in the definition for video.

3.5 Parent Compositing

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

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

4 Basic Data Types and Color Keywords

Contents

4.1 Basic Data Types

The common data types for SVG's properties and attributes fall into the following categories:

- <integer>: An <integer> is specified as an optional sign character ('+' or '-') followed by one or more digits "0" to "9". If the sign character is not present, the number is non-negative.

  In SVG Tiny 1.2 the range for a <integer> is -32,767 to 32,767.

- <number> (real number value): A real number value is specified in either decimal notation or in scientific notation. A **<decimal-number>** consists of either an <integer>, or an optional sign character followed by zero or more digits followed by a dot (.) followed by one or more digits. A **<scientific-number>** consists of a <decimal-number> immediately followed by the letter "e" or "E" immediately followed by an <integer>.

  In SVG Tiny 1.2 the internal storage of a <number> may only have the capacity for a fixed point number in the range '-32,767.9999 to +32,767.9999'. It is recommended that higher precision floating point storage and computation be performed on operations such as coordinate system transformations to provide the best possible precision and to prevent round-off errors.

- <length>: A length is a distance measurement. The format of a <length> is a <number> optionally followed immediately by a unit identifier. If the <length> is expressed as a value without a unit identifier (e.g., **48**), then the <length> represents a distance in the current user coordinate system.

  SVG Tiny 1.2 only supports CSS units on the the 'width' and 'height' attributes on the outermost 'svg' element. These can specify values in any of the following CSS units: in, cm, mm, pt, pc, px and %. If one of the unit identifiers is provided (e.g., **12mm**), then the <length> is processed according to the description in Units.

  Percentage values (e.g., **10%**) on the width and height attributes of the svg element represent a percent of the viewport (refer to the section that discusses Units in general).

- <coordinate>: A <coordinate> represents a <length> in the user coordinate system that is the given distance from the origin of the user coordinate system along the relevant axis (the x-axis for X coordinates, the y-axis for Y coordinates).

- <list of xxx> (where xxx represents a value of some type): A list consists of a separated sequence of values. Unless explicitly described differently, lists can be either comma-separated, with optional white space before or after the comma, or white space-separated.

  White space in lists is defined as one or more of the following consecutive characters: "space" (Unicode code 32), "tab" (9), "line feed" (10), "carriage return" (13) and "form-feed" (12).

  Here is a description of the grammar for a <list of xxx>:

```
ListOfXXX:
    XXX
    | XXX comma-wsp ListOfXXX
comma-wsp:
    (wsp+ comma? wsp*) | (comma wsp*)
comma:
    ","
wsp:
    (#x20 | #x9 | #xD | #xA)
```

  where XXX represents a particular type of value.

- <color>: The basic type <color> is a CSS2-compatible specification for a color in the sRGB color space [sRGB]. <color> applies to SVG's use of the **'color'** property and is a component of the definitions of properties **'fill'**, **'stroke'** **'stop-color'** and **'solid-color'**.

  SVG supports all of the syntax alternatives for <color> defined in [CSS2-color-types], including a list of recognized color keywords names.

  A <color> is either a keyword (see Recognized color keyword names) or a numerical RGB specification.

  In addition to these color keywords, users may specify keywords that correspond to the colors used by objects in the user's environment. The normative definition of these keywords is [CSS2 system colors].

  The format of an RGB value in hexadecimal notation is a '#' immediately followed by either three or six hexadecimal characters. The three-digit RGB notation (#rgb) is converted into six-digit form (#rrggbb) by replicating digits, not by adding zeros. For example, #fb0 expands to #ffbb00. This ensures that white (#ffffff) can be specified with the short notation (#fff) and removes any dependencies on the color depth of the display. The format of an RGB value in the functional notation is 'rgb(' followed by a comma-separated list of three numerical values (either three integer values or three percentage values) followed by ')'. The integer value 255 corresponds to 100%, and to F or FF in the hexadecimal notation: rgb(255,255,255) = rgb(100%,100%,100%) = #FFF. White space characters are allowed around the numerical values. All RGB colors are specified in the sRGB color space (see [SRGB]). Using sRGB provides an unambiguous and objectively measurable definition of the color, which can be related to international standards.
- <paint> : The values for properties **'fill'** and **'stroke'** are specifications of the type of paint to use when filling or stroking a given graphics element. The available options and syntax for **<paint>** are described in Specifying paint.

- <percentage>: The format of a percentage value is a <number> immediately followed by a '%'. Percentage values are always relative to another value, for example a length. SVG Tiny 1.2 does not support percentage values except for the 'width' and 'height' attributes on the outermost 'svg' element. Each attribute or property that allows percentages also defines the reference distance measurement to which the percentage refers.

- <transform-list> : The detailed description of the possible values for a <transform-list> are detailed in Modifying the User Coordinate System: the transform attribute.

- <iri> The <iri> type holds a reference to an International Resource Identifier (see [IRI]): A IRI is the address of a resource on the Web. For the specification of IRI references in SVG, see IRI references.

- <time>: A time value is a <number> optionally followed by a time unit identifier. Time unit identifiers are:
  - **ms**: milliseconds
  - **s**: seconds

  Time values may not be negative. If a time value is specified without a time unit the time is interpreted as if seconds were specified.

4.2 Recognized color keyword names

The following sixteen color keywords can be used as a keyword value for data type <color>:

black	rgb(0, 0, 0)		**green**	rgb(0, 128, 0)
silver	rgb(192, 192, 192)		**lime**	rgb(0, 255, 0)
gray	rgb(128, 128, 128)		**olive**	rgb(128, 128, 0)
white	rgb(255, 255, 255)		**yellow**	rgb(255, 255, 0)
maroon	rgb(128, 0, 0)		**navy**	rgb(0, 0, 128)
red	rgb(255, 0, 0)		**blue**	rgb(0, 0, 255)
purple	rgb(128, 0, 128)		**teal**	rgb(0, 128, 128)
fuchsia	rgb(255, 0, 255)		**aqua**	rgb(0, 255, 255)

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

5 Document Structure

Contents

5.1 Defining an SVG document fragment: the 'svg' element

5.1.1 Overview

An SVG document fragment consists of any number of SVG elements contained within an 'svg' element.

An SVG document fragment can range from an empty fragment (i.e., no content inside of the 'svg' element), to a very simple SVG document fragment containing a single SVG graphics element such as a 'rect', to a complex, deeply nested collection of container elements and graphics elements.

An SVG document fragment can stand by itself as a self-contained file or resource, in which case the SVG document fragment is an SVG document, or it can be embedded inline as a fragment within a parent XML document.

The following example shows simple SVG content embedded inline as a fragment within a parent XML document. Note the use of XML namespaces to indicate that the 'svg' and 'ellipse' elements belong to the SVG namespace:

Example: 05_01.xml

```
<?xml version="1.0" standalone="yes"?>
<parent xmlns="http://example.org"
       xmlns:svg="http://www.w3.org/2000/svg">
   <!-- parent contents here -->
   <svg:svg width="4cm" height="8cm" version="1.2" baseProfile="tiny" viewBox="0 0 100 100">
      <svg:title>An ellipse</svg:title>
      <svg:ellipse cx="50" cy="50" rx="40" ry="20" />
   </svg:svg>
   <!-- ... -->
</parent>
```

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

Example: 05_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="4cm" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny" viewBox="0 0 100 100">
  <desc>Four separate rectangles
  </desc>
    <rect x="20" y="20" width="20" height="20"/>
    <rect x="50" y="20" width="30" height="15"/>
    <rect x="20" y="50" width="20" height="20"/>
    <rect x="50" y="50" width="20" height="40"/>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="98" height="98"
        fill="none" stroke="blue" stroke-width="2" />
</svg>
```

An SVG document fragment can only contain one single 'svg' element, this means that 'svg' elements cannot appear in the middle of SVG content.

In all cases, for compliance with the "Namespaces in XML 1.1" Recommendation [XML-NS], an SVG namespace declaration must be in scope on the 'svg' element, so that all SVG elements are identified as belonging to the SVG namespace.

For example, an xmlns attribute without a namespace prefix could be specified on an 'svg' element, which means that SVG is the default namespace for all elements within the scope of the element with the xmlns attribute:

Example: 05_03.svg

```
<svg xmlns="http://www.w3.org/2000/svg"...>
  <rect .../>
</svg>
```

If a namespace prefix is specified on the xmlns attribute (e.g., `xmlns:svg="http://www.w3.org/2000/svg"`), then the corresponding namespace is not the default namespace, so an explicit namespace prefix must be assigned to the elements:

Example: 05_04.svg

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg"...>
  <svg:rect .../>
</svg:svg>
```

Namespace declarations can also be specified on ancestor elements (illustrated in the above example). For more information, refer to the "Namespaces in XML" Recommendation [XML-NS].

5.1.2 The 'svg' element

Schema: svg

```
  <define name='svg'>
    <element name='svg'>
      <ref name='svg.AT'/>
      <zeroOrMore><ref name='svg.G.group'/></zeroOrMore>
    </element>
  </define>

  <define name='svg.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.Focus.attr'/>
    <ref name='svg.AnimateSync.attr'/>
```

```
    <ref name='svg.Core.attr'/>
    <ref name='svg.WH.attr'/>
    <ref name='svg.PAR.attr'/>
    <optional>
      <attribute name='viewBox' svg:animatable='true' svg:inheritable='false'>
        <ref name='ViewBoxSpec.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='zoomAndPan' a:defaultValue='magnify' svg:animatable='false' svg:inheritable='false'>
        <choice>
          <value>disable</value>
          <value>magnify</value>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='version' a:defaultValue='1.1' svg:animatable='false' svg:inheritable='false'>
        <choice>
          <value type='string'>1.0</value>
          <value type='string'>1.1</value>
          <value type='string'>1.2</value>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='baseProfile' a:defaultValue='none' svg:animatable='false' svg:inheritable='false'>
        <ref name='Text.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='contentScriptType' a:defaultValue='text/ecmascript' svg:animatable='false' svg:inheritable='false'>
        <ref name='ContentType.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='snapshotTime' svg:animatable='false' svg:inheritable='false'><text/></attribute>
    </optional>
    <optional>
      <attribute name='timelineBegin' a:defaultValue='onLoad' svg:animatable='false' svg:inheritable='false'>
        <choice>
          <value type='string'>onLoad</value>
          <value type='string'>onStart</value>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='playbackOrder' a:defaultValue='all' svg:animatable='false' svg:inheritable='false'>
        <choice>
          <value type='string'>all</value>
          <value type='string'>forwardOnly</value>
        </choice>
      </attribute>
    </optional>
  </define>
```

Attribute definitions:

`version` = " <number> "
Indicates the SVG language version to which this document fragment conforms.

In SVG 1.0 and SVG 1.1 this attribute had to the value "1.0" or "1.1" respectively. For SVG 1.2, the attribute should have the value "1.2". See rules for version processing for further instructions.

Animatable: no.

`baseProfile` = profile-name
Describes the minimum SVG language profile that the author believes is necessary to correctly render the content. See rules for baseProfile processing for further instructions.

If the attribute is not specified, the effect is as if a value of "none" were specified.

Animatable: no.

`width` = " <length> "
The intrinsic width of the SVG document fragment.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

If the attribute is not specified, the effect is as if a value of "100%" were specified.

Animatable: yes.

`height` = " <length> "
The intrinsic height of the SVG document fragment.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

If the attribute is not specified, the effect is as if a value of "100%" were specified.

Animatable: yes.

`viewBox` = <min-x> <min-y> <width> <height>,
Specifies a rectangular region of user space which should be mapped to the bounds of the viewport established by the 'svg' element, taking into account the value of the **preserveAspectRatio** attribute. (See The viewBox attribute)

Animatable: yes..

`preserveAspectRatio` = [defer] <align> [<meet>]
Indicates whether or not to force uniform scaling. (See The preserveAspectRatio attribute for the syntax of <align> and the interpretation of this attribute.)

Animatable: yes.

`snapshotTime` = " <time> "

Indicates a moment in time which is most relevant for a still-image of the animated svg content. This time can be used as a hint to the SVG User Agent for rendering a still-image of an animated SVG Document, such as a preview. See example below.

Animatable: no.

playbackOrder = " forwardOnly | all"

Indicates if the content can be seeked backwards or not. In earlier versions of SVG there has been no need to put restrictions on the direction of seeking but with the newly introduced facilities for long-running documents (e.g. the 'discard' element) there is sometimes a need to restrict this.

If playbackOrder is set to 'forwardOnly', the content will probably contain 'discard' elements or scripts that destroy resources, thus navigating backwards may result in missing content. If playbackOrder is 'forwardOnly', the content should not provide a way, through hyperlinking or script, of seeking backwards in the timeline. Similarly the UA should disable any controls it may provide in the UI for seeking backwards. Content with playbackOrder = "forwardOnly" that provides a mechanism for seeking backwards in time may result in undefined behavior or a document that is in error.

"forwardOnly"
This file is intended to be played only in the forward direction, sequentially, therefore seeking backwards should not be allowed.

"all"
Indicates that the document is authored appropriately for seeking in both directions.

The default value is 'all'.

Animatable: no.

timelineBegin = " onLoad | onStart "

Controls the initialization of the timeline for the document.

The svg element controls the global timeline, so for progressively loaded animations, the author would typically set this attribute to "onStart", thus allowing the nested timelines to begin as the document is loaded.

"onLoad"
The document's timeline starts the moment the SVGLoad event for the root element is triggered.

"onStart"
The document's timeline starts at the moment the root svg element's open tag is fully parsed and processed.

The default value is 'onLoad'.

Animatable: no.

contentScriptType = " *content-type* "

Identifies the default scripting language for the given document. This attribute sets the default scripting language all the instances of script in the document. The value ***content-type*** specifies a media type, per [RFC2045]. The default value is "application/ecmascript".

Animatable: no.

An SVG document should include a viewBox attribute on the 'svg' element of the referenced document. This describes the region of world coordinate space used by the graphic. This attribute thus provides a convenient way to design SVG documents to scale-to-fit into an arbitrary viewport.

The first example below has a fixed width and height in pixels. Content like this is often produced by illustration programs originally targetted at print. The second example is scalable, and has a viewBox rather than a width and height.

Example: width-height.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="300px" height="600px" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny">
  <desc>...</desc>
</svg>
```

Example: viewBox.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny" viewBox="0 0 300 600">
  <desc>...</desc>
</svg>
```

Below is an example of snapshotTime. A User Agent is displaying a number of SVG files in a directory by rendering a thumbnail image. It uses the snapshotTime as the time to render when generating the image, thus giving a more representative static view of the animation. The appearance of the thumbnail for a User Agent that honors the snapshotTime and for a User Agent that does not is shown below the example (UA with snapshot support at the left, without snapshot support at the right).

Example: 05_22.svg

```
<svg version="1.2" snapShotTime="3" baseProfile="tiny"
         xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
         viewBox="0 0 400 300">
  <title>Snapshot time example</title>
  <desc>This example shows the use of snapshot time on an animation of
     color
   </desc>
   <rect x="60" y="85" width="256" height="65" fill="none" stroke="rgb(60,126,220)" stroke-width="4"/>

   <text x="65" y="140" fill="rgb(60,126,220)" font-size="60">Hello SVG
      <animateColor attributeName="fill" begin="0" dur="3" from="white" to="rgb(60,126,220)"/>
   </text>
</svg>
```

5.2 Grouping: the 'g' element

5.2.1 Overview

The 'g' element is a container element for grouping together related graphics elements.

Grouping constructs, when used in conjunction with the 'desc' and 'title' elements, provide information about document structure and semantics. Documents that are rich in structure may be rendered graphically, as speech, or as braille, and thus promote accessibility.

A group of elements, as well as individual objects, can be given a name using the id attribute. Named groups are needed for several purposes such as animation and re-usable objects.

An example:

Example: 05_05.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="5cm" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny" viewBox="0 0 5 5">
  <desc>Two groups, each of two rectangles
  </desc>
  <g id="group1" fill="red" >
    <desc>First group of two red rectangles</desc>
    <rect x="1" y="1" width="1" height="1" />
    <rect x="3" y="1" width="1" height="1" />
  </g>
  <g id="group2" fill="blue" >
    <desc>Second group of two blue rectangles</desc>
    <rect x="1" y="3" width="1" height="1" />
    <rect x="3" y="3" width="1" height="1" />
  </g>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x=".01" y=".01" width="4.98" height="4.98"
        fill="none" stroke="blue" stroke-width=".02" />
</svg>
```

A 'g' element can contain other 'g' elements nested within it, to an arbitrary depth. Thus, the following is possible:

Example: 05_06.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="5cm" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny">
  <desc>Groups can nest
  </desc>
  <g>
     <g>
       <g>
       </g>
     </g>
   </g>
</svg>
```

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

5.2.2 The 'g' element

Schema: g

```
  <define name='g'>
    <element name='g'>
      <ref name='g.AT'/>
      <zeroOrMore><ref name='svg.G.group'/></zeroOrMore>
    </element>
  </define>

  <define name='g.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.Focus.attr'/>
    <ref name='svg.Transform.attr'/>
  </define>
```

5.3 The 'defs' element

The 'defs' element is a container element for referenced elements. For understandability and accessibility reasons, it is recommended that, whenever possible, referenced elements be defined inside of a 'defs'.

The content model for 'defs' is the same as for the 'g' element; thus, any element that can be a child of a 'g' can also be a child of a 'defs', and vice versa.

Elements that are descendants of a 'defs' are not rendered directly; they are prevented from becoming part of the rendering tree just as if the 'defs' element were a 'g' element and the 'display' property were set to **none**. Note, however, that the descendants of a 'defs' are always present in the source tree and can be referenced by other elements. The actual value of the 'display' property on the 'defs' element or any of its descendants does not change the rendering of these elements or

prevent these elements from being referenced.

Schema: defs

```
  <define name='defs'>
    <element name='defs'>
      <ref name='defs.AT'/>
      <zeroOrMore><ref name='svg.G.group'/></zeroOrMore>
    </element>
  </define>

  <define name='defs.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.Transform.attr'/>
  </define>
```

To provide some SVG user agents with an opportunity to implement efficient implementations in streaming environments, creators of SVG content are encouraged to place all elements which are targets of local IRI references within a **'defs'** element which is a direct child of one of the ancestors of the referencing element. For example:

Example: 05_10.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny" viewBox="0 0 8 3">
  <desc>Local URI references within ancestor's 'defs' element.</desc>
  <defs>
    <linearGradient id="Gradient01">
      <stop offset="20%" stop-color="#39F" />
      <stop offset="90%" stop-color="#F3F" />
    </linearGradient>
  </defs>
  <rect x="1" y="1" width="6" height="1"
        fill="url(#Gradient01)"  />
  <!-- Show outline of canvas using 'rect' element -->
  <rect x=".01" y=".01" width="7.98" height="2.98"
        fill="none" stroke="blue" stroke-width=".02" />
</svg>
```

In the document above, the linear gradient is defined within a **'defs'** element which is the direct child of the **'svg'** element, which in turn is an ancestor of the **'rect'** element which references the linear gradient. Thus, the above document conforms to the guideline.

5.4 The 'discard' element

The **'discard'** element allows authors to specify the time at which particular elements may be discarded, therefore reducing the resources required by an SVG UA. This is particularly useful for the SVG viewers to handle long-running documents. This element will not be processed by static SVG viewers.

The **'discard'** element can appear in the DOM tree in the same place as the 'animate' element.

Schema: discard

```
  <define name='discard'>
    <element name='discard'>
      <ref name='discard.AT'/>
      <ref name='discard.CM'/>
    </element>
  </define>

  <define name='discard.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.XLinkRequired.attr'/>
    <ref name='svg.AnimateBegin.attr'/>
  </define>

  <define name='discard.CM'>
    <zeroOrMore>
      <ref name='svg.Desc.group'/>
      <ref name='svg.Handler.group'/>
    </zeroOrMore>
  </define>
```

Attribute definitions:

xlink:href = " <iri> "
See the definition of target element.

Animatable: no.

begin = " offset-value | syncbase-value | event-value | accessKey-value "
The **'discard'** element has an implicit simple duration of 'indefinite'. As soon as the element's active duration starts, the user agent discards the element identified by the **'xlink:href'** attribute. The removal operation acts as if the method `removeChild` was called on the parent of the target element.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: no.

If the target element does not exist when the **'discard'** element gets active, then the **'discard'** element is ignored. An element gets discarded either when it is the target of a **'discard'** element and the value of the 'begin' attribute of this latter is reached or when one of its ancestors is discarded. The **'discard'** element itself can be discarded. It is discarded at the earliest: soon after its target element has been discarded or because it is the target of an other **'discard'** element.

When the **'discard'** element is used, the end user may see unexpected results when seeking backward because the seek will not re-insert the discarded elements.

So, authors are encouraged to set the 'playbackOrder' attribute to true when using the 'discard' element.

Example: discard01.svg

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
    width="352" height="240" sequentialContent="true">

    <ellipse cx="98.5" cy="17.5" rx="20.5" ry="17.5" fill="blue" stroke="black"
        transform="translate(9 252) translate(3 -296)">
        <animateTransform attributeName="transform" begin="0s" dur="2s" fill="remove"
            calcMode="linear" type="translate" additive="sum" from="0 0" to="-18 305"/>
        <discard begin="2s"/>
    </ellipse>

    <rect x="182" y="-39" width="39" height="30" transform="translate(30 301)" fill="red" stroke="black">
        <animateTransform attributeName="transform" begin="1s" dur="2s" fill="remove"
            calcMode="linear" type="translate" additive="sum" from="0 0" to="-26 -304"/>
        <discard begin="3s"/>
    </rect>

    <polygon points="-66,83.5814 -43,123.419 -89,123.419" fill="green" stroke="black"
        transform="matrix(1 0 0 1.1798 0 -18.6096)">
        <animateTransform attributeName="transform" begin="2s" dur="2s"
            fill="remove" calcMode="linear" type="translate" additive="sum" from="0 0"
            to="460 63.5699"/>
        <discard begin="4s"/>
    </polygon>
</svg>
```

5.5 The 'desc' and 'title' elements

Each container element or graphics element in an SVG drawing can supply a 'desc' and/or a 'title' description string where the description is text-only. When the current SVG document fragment is rendered as SVG on visual media, 'desc' and 'title' elements are not rendered as part of the graphics. User agents may, however, for example, display the 'title' element as a tooltip, as the pointing device moves over particular elements. Alternate presentations are possible, both visual and aural, which display the 'desc' and 'title' elements but do not display 'path' elements or other graphics elements. For deep hierarchies, and for following 'use' element references, it is sometimes desirable to allow the user to control how deep they drill down into descriptive text.

Schema: desc

```
  <define name='desc'>
    <element name='desc'>
      <ref name='DTM.AT'/>
      <ref name='DTM.CM'/>
    </element>
  </define>

  <define name='DTM.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
  </define>

  <define name='DTM.CM'>
    <text/>
  </define>
```

Schema: title

```
  <define name='desc'>
    <element name='desc'>
      <ref name='DTM.AT'/>
      <ref name='DTM.CM'/>
    </element>
  </define>
```

The following is an example. In typical operation, the SVG user agent would not render the 'desc' and 'title' elements but would render the remaining contents of the 'g' element.

Example: 05_11.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny">
<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>
```

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

Example: 05_12.svg

```
<?xml version="1.0" standalone="yes"?>
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
    version="1.2" baseProfile="tiny">
    <desc xmlns:mydoc="http://example.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>
```

Authors should always provide a 'title' child element to the 'svg' element within a stand-alone SVG document. The 'title' child element to an 'svg' element serves the purposes of identifying the content of the given SVG document fragment. Since users often consult documents out of context, authors should provide context-rich titles. Thus, instead of a title such as "Introduction", which doesnt provide much contextual background, authors should supply a title such as "Introduction to Medieval Bee-Keeping" instead. For reasons of accessibility, user agents should always make the content of the 'title' child element to the 'svg' element available to users. The mechanism for doing so depends on the user agent (e.g., as a caption, spoken).

It is strongly recommended that at most one 'desc' and at most one 'title' element appear as a child of any particular element, and that these elements appear before any other child elements (except possibly 'metadata' elements) or character data content. If user agents need to choose among multiple 'desc' or 'title' elements for processing (e.g., to decide which string to use for a tooltip), the user agent shall choose the first one.

5.6 The 'use' element

Any 'g' or graphics element is potentially a template object that can be re-used (i.e. "instanced") in the SVG document via a 'use' element. The 'use' element references another element and indicates that the graphical contents of that element is included/drawn at that given point in the document.

Unlike 'animation', the 'use' element cannot reference entire files.

Besides what is described about the 'use' element in this section important restrictions for 'use' can be found in the Reference Section.

The 'use' element has optional attributes x and y which are used to place the graphical contents of the referenced element into the current coordinate system.

The effect of a 'use' element is as if the contents of the referenced element were deeply cloned into a separate non-exposed DOM tree which had the 'use' element as its parent and all of the 'use' element's ancestors as its higher-level ancestors. Because the cloned DOM tree is non-exposed, the SVG Document Object Model (DOM) only contains the 'use' element and its attributes. The SVG DOM does not show the referenced element's contents as children of 'use' element.

Property inheritance works as if the referenced element had been textually included as a deeply cloned child of the 'use' element. The referenced element inherits properties from the 'use' element and the 'use' element's ancestors. An instance of a referenced element does not inherit properties from the referenced element's original parents.

If event attributes are assigned to referenced elements, then the actual target for the event will be the SVGElementInstance object within the "instance tree" corresponding to the given referenced element.

The event handling for the non-exposed tree works as if the referenced element had been textually included as a deeply cloned child of the 'use' element, except that events are dispatched to the SVGElementInstance objects. The event's target and currentTarget attributes are set to the SVGElementInstance that corresponds to the target and current target elements in the referenced subtree. An event propagates through the exposed and non-exposed portions of the tree in the same manner as it would in the regular document tree: first going to the target of the event, then bubbling back through non-exposed tree to the use element and then back through regular tree to the root element in the bubbling phase.

An element and all its corresponding SVGElementInstance objects share an event listener list. The currentTarget attribute of the event can be used to determine through which object an event listener was invoked.

The behavior of the 'visibility' property conforms to this model of property inheritance. Thus, specifying **visibility='hidden'** on a 'use' element does not guarantee that the referenced content will not be rendered. If the 'use' element specifies **visibility='hidden'** and the element it references specifies **visibility='hidden'** or **visibility='inherit'**, then that one element will be hidden. However, if the referenced element instead specifies **visibility='visible'**, then that element will be visible even if the 'use' element specifies **visibility='hidden'**.

Animations on a referenced element will cause the instances to also be animated.

As listed in the Reference Section the 'use' element is not allowed to reference an 'svg' element

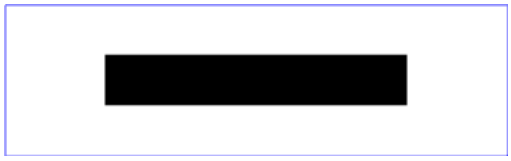
A 'use' element has the same visual effect as if the 'use' element were replaced by the following generated content:

- In the generated content, the 'use' will be replaced by 'g', where all attributes from the 'use' element except for x, y and xlink:href are transferred to the generated 'g' element. An additional transformation **translate(x,y)** is appended to the end (i.e., right-side) of the transform attribute on the generated 'g', where **x** and **y** represent the values of the x and y attributes on the 'use' element. The referenced object and its contents are deep-cloned into the generated tree.

Example Use01 below has a simple 'use' on a 'rect'.

Example: 05_13.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="10cm" height="3cm" viewBox="0 0 100 30" version="1.2"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     baseProfile="tiny">
  <desc>Example Use01 - Simple case of 'use' on a 'rect'</desc>
  <defs>
    <rect id="MyRect" width="60" height="10"/>
  </defs>
  <rect x=".1" y=".1" width="99.8" height="29.8"
        fill="none" stroke="blue" stroke-width=".2" />
  <use x="20" y="10" xlink:href="#MyRect" />
</svg>
```



The visual effect would be equivalent to the following document:

Example: 05_14.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" viewBox="0 0 100 30" version="1.2"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     baseProfile="tiny">
     <desc>Example Use01-GeneratedContent - Simple case of 'use' on a 'rect'</desc>
  <!-- 'defs' section left out -->
  <rect x=".1" y=".1" width="99.8" height="29.8"
        fill="none" stroke="blue" stroke-width=".2" />
  <!-- Start of generated content. Replaces 'use' -->
  <g transform="translate(20,10)">
    <rect width="60" height="10"/>
  </g>
  <!-- End of generated content -->
</svg>
```

Example Use03 illustrates what happens when a 'use' has a transform attribute.

Example: 05_17.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="10cm" height="3cm" viewBox="0 0 100 30" version="1.2"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     baseProfile="tiny">
  <desc>Example Use03 - 'use' with a 'transform' attribute</desc>
  <defs>
    <rect id="MyRect" x="0" y="0" width="60" height="10"/>
  </defs>
  <rect x=".1" y=".1" width="99.8" height="29.8"
        fill="none" stroke="blue" stroke-width=".2" />
  <use xlink:href="#MyRect"
       transform="translate(20,2.5) rotate(10)" />
</svg>
```

The visual effect would be equivalent to the following document:

Example: 05_18.svg

```
<?xml version="1.0" standalone="no"?>
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" viewBox="0 0 100 30" version="1.2"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     baseProfile="tiny">
  <desc>Example Use03-GeneratedContent - 'use' with a 'transform' attribute</desc>
  <!-- 'defs' section left out -->
  <rect x=".1" y=".1" width="99.8" height="29.8"
        fill="none" stroke="blue" stroke-width=".2" />
  <!-- Start of generated content. Replaces 'use' -->
  <g transform="translate(20,2.5) rotate(10)">
    <rect x="0" y="0" width="60" height="10"/>
  </g>
  <!-- End of generated content -->
</svg>
```

When a 'use' references another element which is another 'use' or whose content contains a 'use' element, then the deep cloning approach described above is recursive. However, a set of references that directly or indirectly reference a element to create a circular dependency is an error, as described in the References section.

Schema: use

```
  <define name='use'>
    <element name='use'>
      <ref name='use.AT'/>
      <ref name='use.CM'/>
    </element>
  </define>

  <define name='use.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.Transform.attr'/>
    <ref name='svg.XLinkEmbed.attr'/>
    <ref name='svg.Focus.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.XY.attr'/>
    <ref name='svg.Overflow.attr'/>
  </define>

  <define name='use.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='svg.Animate.group'/>
        <ref name='svg.Handler.group'/>
      </choice>
    </zeroOrMore>
  </define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the referenced element is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the referenced element is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

xlink:href = " <iri> "
A IRI reference to an element/fragment within an SVG document.

Animatable: yes.

5.7 The 'image' element

The 'image' element indicates that the contents of a complete file are to be rendered into a given rectangle within the current user coordinate system. In SVG Tiny 1.2, the 'image' must reference content that is a raster image format, such as PNG and JPG. SVG Tiny 1.2 does not allow SVG images to be referenced by the 'image' element; Instead, authors should use the 'animation' element. Conforming SVG viewers need to support at least PNG and JPEG format files.

The result of processing an 'image' is always a four-channel RGBA result. When an 'image' element references a raster image file such as PNG or JPEG files which only has three channels (RGB), then the effect is as if the object were converted into a 4-channel RGBA image with the alpha channel uniformly set to 1. For a single-channel raster image, the effect is as if the object were converted into a 4-channel RGBA image, where the single channel from the referenced object is used to compute the three color channels and the alpha channel is uniformly set to 1.

The 'image' element supports the 'opacity' property for controlling the image opacity.

An 'image' element establishes a new viewport for the referenced file as described in Establishing a new viewport. The bounds for the new viewport are defined by attributes x , y , width and height . The placement and scaling of the referenced image are controlled by the preserveAspectRatio attribute on the 'image' element.

The value of the ' viewBox ' attribute to use when evaluating the preserveAspectRatio attribute is defined by the referenced content. For content that clearly identifies a viewBox that value should be used. For most raster content (PNG, JPEG) the bounds of the image should be used (i.e. the 'image' element has an implicit 'viewBox' of "0 0 raster-image-width raster-image-height"). Where no value is readily available the preserveAspectRatio attribute is ignored, and only the translate due to the ' x ' & ' y ' attributes of the viewport is used to display the content.

For example, if the image element referenced a PNG or JPEG and **preserveAspectRatio="xMinYMin meet"**, then the aspect ratio of the raster would be preserved (which means that the scale factor from image's coordinates to current user space coordinates would be the same for both X and Y), the raster would be sized as large as possible while ensuring that the entire raster fits within the viewport, and the top/left of the raster would be aligned with the top/left of the viewport as defined by the attributes ' x ', ' y ', ' width ' and ' height ' on the 'image' element. If the value of preserveAspectRatio was 'none' then aspect ratio of the image would not be preserved. The image would be fitted such that the top/left corner of the raster exactly aligns with coordinate (x , y) and the bottom/right corner of the raster exactly aligns with coordinate (x + width , y + height).

The resource referenced by the 'image' element represents a separate document which generates its own parse tree and document object model (if the resource is XML). Thus, there is no inheritance of properties into the image.

The SVG specification does not specify when an image that is not being displayed should be loaded. A user agent is not required to load image data for an image that is not displayed (e.g. is is outside the initial document viewport). However, it should be noted that this may cause a delay when an image becomes visible for the first time. In the case where an author wants to suggest that the user agent load image data before it is displayed, they should use the 'prefetch' element.

Schema: image

```
<define name='image'>
  <element name='image'>
    <ref name='image.AT'/>
    <ref name='image.CM'/>
  </element>
</define>

<define name='image.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.XLinkEmbed.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.External.attr'/>
  <ref name='svg.Focus.attr'/>
  <ref name='svg.Transform.attr'/>
  <ref name='svg.Opacity.attr'/>
  <ref name='svg.XYWH.attr'/>
  <ref name='svg.PAR.attr'/>
  <ref name='svg.ContentType.attr'/>
</define>

<define name='image.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
      <ref name='svg.Animate.group'/>
      <ref name='svg.Handler.group'/>
      <ref name='svg.Discard.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the referenced document is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the referenced document is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
width = " <length> "
The width of the rectangular region into which the referenced document is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.
height = " <length> "
The height of the rectangular region into which the referenced document is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.
xlink:href = " <iri> "
A IRI reference.

Animatable: yes.
type = "<media/type>"
A hint about the expected Internet Media Type of the raster image. Implementations may choose not to fetch images of formats that they do not support.

Animatable: no.

An example:

Example: 05_21.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="100%" height="100%" version="1.2"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     baseProfile="tiny">
  <desc>This graphic links to an external image
  </desc>
  <image x="200" y="200" width="100px" height="100px"
         xlink:href="myimage.png">
    <title>My image</title>
  </image>
</svg>
```

5.8 Conditional processing

5.8.1 Conditional processing overview

SVG contains a 'switch' element along with attributes requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts to provide an ability to specify alternate viewing depending on the capabilities of a given user agent or the user's language.

Schema: conditional

```
<define name='svg.Conditional.attr' combine='interleave'>
  <optional>
    <attribute name='requiredFeatures' svg:animatable='false' svg:inheritable='false'>
      <ref name='FeatureList.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='requiredExtensions' svg:animatable='false' svg:inheritable='false'>
      <ref name='ExtensionList.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='requiredFormats' svg:animatable='false' svg:inheritable='false'>
      <ref name='FormatList.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='requiredFonts' svg:animatable='false' svg:inheritable='false'>
      <ref name='FontList.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='systemLanguage' svg:animatable='false' svg:inheritable='false'>
      <ref name='LanguageCodes.datatype'/>
    </attribute>
  </optional>
</define>
```

Attributes requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts act as tests and return either true or false results. The 'switch' renders the first of its children for which all of these attributes test true. If the given attribute is not specified, then a true value is assumed.

Similar to the 'display' property, conditional processing attributes only affect the direct rendering of elements and do not prevent elements from being successfully referenced by other elements (such as via a 'use').

In consequence:

- requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts attributes affect 'a', 'foreignObject' and 'tspan' elements.
- requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts attributes do not apply to the 'defs' and 'gradient' elements because they are not part of the rendering tree.
- requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts attributes affect 'animate', 'animateColor', 'animateMotion', 'animateTransform', and 'set' elements. If the conditional statement on these animation elements fails, the animation will never be triggered.

5.8.2 The 'switch' element

The 'switch' element evaluates the requiredFeatures , requiredExtensions , systemLanguage , requiredFormats and requiredFonts attributes on its direct child elements in order, and then processes and renders the first child for which these attributes evaluate to true. All others will be bypassed and therefore not rendered. If the child element is a container element such as a 'g' , then the entire subtree is either processed/rendered or bypassed/not rendered.

Note that the values of properties 'display' and 'visibility' have no effect on 'switch' element processing. In particular, setting 'display' to **none** on a child of a 'switch' element has no effect on true/false testing associated with 'switch' element processing.

Schema: switch

```
<element name='switch'>
  <ref name='switch.AT'/>
  <!-- the content model for switch is defined as part
       of the element that can contain it -->
</element>

<define name='switch.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.External.attr'/>
  <ref name='svg.Transform.attr'/>
</define>
```

For more information and an example, see Embedding foreign object types.

5.8.3 The requiredFeatures attribute

Definition of **requiredFeatures**:

requiredFeatures = ***list-of-features***
The value is a list of feature strings, with the individual values separated by white space. Determines whether all of the named *features* are supported by the user agent. Only feature strings defined in the Feature String appendix are allowed. If all of the given features are supported, then the attribute evaluates to true; otherwise, the current element and its children are skipped and thus will not be rendered.

Animatable: no.

If the attribute is not present, then its implicit return value is "true". If a null string or empty string value is given to attribute requiredFeatures, the attribute returns "false".

requiredFeatures is often used in conjunction with the 'switch' element. If the requiredFeatures is used in other situations, then it represents a simple switch on the given element whether to render the element or not.

5.8.4 The requiredExtensions attribute

The requiredExtensions attribute defines a list of required language extensions. Language extensions are capabilities within a user agent that go beyond the feature set defined in this specification. Each extension is identified by a IRI reference.

Definition of **requiredExtensions**:

requiredExtensions = ***list-of-extensions***
The value is a list of IRI references which identify the required extensions, with the individual values separated by white space. Determines whether all of the named *extensions* are supported by the user agent. If all of the given extensions are supported, then the attribute evaluates to true; otherwise, the current element and its children are skipped and thus will not be rendered.

Animatable: no.

If a given IRI reference contains white space within itself, that white space must be escaped.

If the attribute is not present, then its implicit return value is "true". If a null string or empty string value is given to attribute requiredExtensions, the attribute returns "false".

requiredExtensions is often used in conjunction with the 'switch' element. If the requiredExtensions is used in other situations, then it represents a simple switch on the given element whether to render the element or not.

The IRI names for the extension should include versioning information, such as "http://example.org/SVGExtensionXYZ/1.0", so that script writers can distinguish between different versions of a given extension.

5.8.5 The systemLanguage attribute

The attribute value is a comma-separated list of language names as defined in [RFC3066].

Evaluates to "true" if one of the languages indicated by user preferences exactly equals one of the languages given in the value of this parameter, or if one of the languages indicated by user preferences exactly equals a prefix of one of the languages given in the value of this parameter such that the first tag character following the prefix is "-".

Evaluates to "false" otherwise.

Note: This use of a prefix matching rule does not imply that language tags are assigned to languages in such a way that it is always true that if a user understands a language with a certain tag, then this user will also understand all languages with tags for which this tag is a prefix.

The prefix rule simply allows the use of prefix tags if this is the case.

Implementation note: When making the choice of linguistic preference available to the user, implementers should take into account the fact that users are not familiar with the details of language matching as described above, and should provide appropriate guidance. As an example, users may assume that on selecting "en-gb", they will be served any kind of English document if British English is not available. The user interface for setting user preferences should guide the user to add "en" to get the best matching behavior.

Multiple languages MAY be listed for content that is intended for multiple audiences. For example, content that is presented simultaneously in the original Maori and English versions, would call for:

```
<text systemLanguage="mi, en"><!-- content goes here --></text>
```

However, just because multiple languages are present within the object on which the **systemLanguage** test attribute is placed, this does not mean that it is intended for multiple linguistic audiences. An example would be a beginner's language primer, such as "A First Lesson in Latin," which is clearly intended to be used by an English-literate audience. In this case, the **systemLanguage** test attribute should only include "en".

Authoring note: Authors should realize that if several alternative language objects are enclosed in a 'switch', and none of them matches, this may lead to situations where no content is displayed. It is thus recommended to include a "catch-all" choice at the end of such a 'switch' which is acceptable in all cases.

For the **systemLanguage** attribute: *Animatable: no.*

If the attribute is not present, then its implicit return value is "true". If a null string or empty string value is given to attribute **systemLanguage**, the attribute returns "false".

systemLanguage is often used in conjunction with the 'switch' element. If the **systemLanguage** is used in other situations, then it represents a simple switch on the given element whether to render the element or not.

5.8.6 The **requiredFormats** attribute

Many resources, especially media such as audio and video, have a wide range of formats. As it is often not possible to require support for a particular format, due to legal or platform restrictions, it is often necessary to provide alternatives so that user agents can choose the format they support.

The requiredFormats attribute is a generic conditional processing attribute that can be used to enable or disable particular branches in the SVG document. It defines a list of resource formats. Each format is defined by the format definition with the syntax varied according to the specific type of resource. The User Agent must support all of the resource types for the attribute to evaluate to true.

Definition of **`requiredFormats`**:

`requiredFormats` = ***list-of-format-definitions***

Each format definition is separated by whitespace. A format definition can be a MIME-type beginning "image/", "video/" or "audio/", or must use one of the following formats:

image(<mime-type>):
Test the MIME-type as an image format.
video(<mime-type>):
Test the MIME-type as a video format.
audio(<mime-type>):
Test the MIME-type as an audio format.
font(<mime-type>):
Test the MIME-type as a font format.
script(<mime-type>):
Test the MIME-type as scripting language type.
foreignObject(namespaceIRI):
Test if the namespace is understood in the SVG foreignObject element.

For a list of MIME types for audio/video codecs, see the IANA registry and RFC2361.

Given that several important file formats are still not registered or not specific enough, the following format definitions are also understood:

- font(truetype)
- font(type1)
- font(opentype)

The following requiredFormats must always evaluate to true in compliant SVG viewers:

- font(image/svg+xml)
- image/png
- image/jpeg
- image/svg+xml,

Animatable: no.

If the attribute is not present, then its implicit return value is "true". If a null string or empty string value is given to attribute **requiredFormats**, the attribute returns "false". Format definitions that are not understood by the user agent return "false".

requiredFormats is often used in conjunction with the 'switch' element. If the **requiredFormats** is used in other situations, then it represents a simple switch on the given element whether to render the element or not.

5.8.7 The **requiredFonts** attribute

If the author wishes to have complete control over the appearance and location of text in the document then they must ensure that the correct font is used when rendering the text. This can be achieved by using SVG Fonts and embedding the font in the document. However, this is not practical in all cases, especially when the number of glyphs used is very large or if the licensing of the font forbids such embedding.

Definition of **`requiredFonts`**:

`requiredFonts` = ***list-of-font-names***
The requiredFonts attribute is a generic conditional processing attribute that can be used to enable or disable particular branches in the SVG document. It defines a list of fonts, separated by commas. The User Agent must have access to all of the fonts, either installed on the system or as an SVG font defined or embedded within the document, for the attribute to evaluate to true. requiredFonts uses same syntax as the **'font-family'** property, for example when processing quoted strings, multiple, leading and trailing spaces, and case sensitivity.

Animatable: no.

If the attribute is not present, then its implicit return value is "true". If a null string or empty string value is given to attribute **requiredFonts**, the attribute returns "false".

requiredFonts is often used in conjunction with the **'switch'** element. If the **requiredFonts** is used in other situations, then it represents a simple switch on the given element whether to render the element or not.

5.9 External Resources

5.9.1 Specifying whether external resources are required for proper rendering

Documents often reference and use the contents of other files (and other Web resources) as part of their rendering. In some cases, authors want to specify that particular resources are required for a document to be considered correct.

Attribute **externalResourcesRequired** is available on all container elements and to all elements which potentially can reference external resources. It specifies whether referenced resources that are not part of the current document are required for proper rendering of the given container element or graphics element.

Attribute definition:

`externalResourcesRequired` = "false | true"

false
(The default value.) Indicates that resources external to the current document are optional. Document rendering can proceed even if external resources are unavailable to the current element and its descendants.

true
Indicates that resources external to the current document are required. If an external resource is not available, progressive rendering is suspended, the document's SVGLoad event is not fired and the animation timeline does not begin until that resource and all other required resources become available, have been parsed and are ready to be rendered. If a timeout event occurs on a required resource, then the document goes into an error state (see Error processing). The document remains in an error state until all required resources become available.

Attribute **externalResourcesRequired** is not inheritable (from a sense of attribute value inheritance), but if set on a container element, its value will apply to all elements within the container.

Because setting **externalResourcesRequired="true"** on a container element can have the effect of disabling progressive display of the contents of that container, tools that generate SVG content are cautioned against using simply setting **externalResourcesRequired="true"** on the **'svg'** element on a universal basis. Instead, it is better to specify **externalResourcesRequired="true"** on those particular graphics elements or container elements which specifically need the availability of external resources in order to render properly.

For **externalResourcesRequired**: *Animatable: no.*

5.9.2 Progressive Rendering

When progressively downloading a document, a user agent conceptually builds a tree of nodes in various states. The possible states for these nodes are unresolved, resolved and error. This description of progressive rendering uses SAX events. However, user implementations are not required to implement SAX, any implementation producing the same result would be compliant.

The two SAX events referred to in the following prose are the startElement and endElement events. The startElement event is generally considered to be triggered when the Start-Tag or an Empty-Element Tag is read. The endElement event occurs either immediately preceding the statElement event in the case of an Empty-Element Tag or when the End-Tag is read.

When loading a document following the startElement event on a node, that node becomes part of the document tree in the unresolved state. If the node's dependencies are successfully resolved, then the node enters the resolved state or if the node's dependencies are found to be in error, then the node enters the error state.

Node dependencies include both children content (like the child elements on a g) and resources (e.g. images referenced by an image) referenced from that node or from its children. Children become resolved when the endElement event occurs on an element. Resources become resolved (or found in error) by a user agent specific mechanism.

A user agent implementing progressive rendering must render the current document tree with the following rules:

- The user agent updates the rendering following each startElement and/or endElement SAX event.
- The user agent renders the conceptual document tree nodes in document order up to, and not including the first node in the 'unresolved' state which has externalResourcesRequired set to true. Nodes in the 'resolved' state are always rendered. Nodes in the unresolved state but with externalResourcesRequired set to false are rendered in their current state. If the node has no rendering (e.g, an image pending a resource), then nothing is rendered for that node.
- If a node enters the error state then the document enters the error state and progressive rendering stops.

Fonts are an exception to the above rules: startElement and endElement events on font element children (font-face, hkern, missing-glyph, glyph) do not cause an update of the document rendering. However, the endElement event on the font element does cause a document rendering as for other node types.

Example

Example: progRend01.svg

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.2"
        baseProfile="tiny" id="svg-root" width="100%" height="100%" viewBox="0 0 480 360">
      <desc>externalResourcesRequired example.</desc>
      <g externalResourcesRequired="true">
         <rect id="rect_1" .... />
          ...
         <rect id="rect_1000" ..../>

         <image xlink:href="myImage.png" externalResourcesRequired="true" ... />
         <rect id="rect_1001" ..../>
      </g>
    </svg>
```

In this example, the g element rendering will start when the g closing tag has been parsed and processed and when all the resources needed by its children have been resolved. This means that the group's rendering will start when the group has been fully parsed and myImage.png has been successfully retrieved.

Forward reference of use element

Example: progRend02.svg

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.2"
```

```
        baseProfile="tiny" id="svg-root" width="100%" height="100%" viewBox="0 0 480 360">
      <desc>Forward reference of use element</desc>
      <use xlink:href="#myRect" x="200" fill="green"/>
      <circle cx="450" cy="50" r="50" fill="yellow" />

      <g fill="red">
         <rect id="myRect" width="100" height="100" />
      </g>
    </svg>
```

In this example, the various renderings will be (the rendering state follows the semi-colon):

1. use.startElement : empty
2. circle.startElement: yellow circle
3. g.startElement: no update
4. rect.startElement (use reference becomes resolved): green rect, yellow circle, red rect

Forward reference on use with externalResourcesRequired="true"

Example: progRend03.svg

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.2"
        baseProfile="tiny" id="svg-root" width="100%" height="100%" viewBox="0 0 480 360">
      <desc>Forward reference on use with eRR=true</desc>
      <use xlink:href="#myGroup" x="200" fill="green"
           externalResourcesRequired="true"/>
        <circle cx="450" cy="50" r="50" fill="yellow" />

        <g fill="red">
            <g id="myGroup">
                <rect id="myRect" width="100" height="100" />
                <use xlink:href="#myRect" x="50" fill="purple"/>
            </g>
        </g>

    </svg>
```

1. use.startElement : empty
2. circle.startElement: empty use is unresolved & externalResourcesRequired="true", rendering is stopped at the use)
3. g.startElement: no update
4. g.myGroup.startElement: no update (use is resolved but externalResourcesRequired="true" so rendering will no proceed until that reference enters the resolved state)
5. rect.startElement: no update
6. use.startElement: no update
7. g.myGroup.endElement (#myGroup reference becomes resolved, rendering can proceed): green rect, purple rect, yellow circle, red rect, purple rect.

Forward reference with use to an element under a container with externalResourcesRequired="true"

Example: progRend04.svg

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.2" baseProfile="tiny" id="svg-root" widt
      <desc>Forward Reference to a use under a container with eRR=true</desc>
      <use xlink:href="#myRect" x="200" fill="green"/>
        <circle cx="250" cy="50" r="50" fill="pink" />
        <g fill="red" externalResourcesRequired="true">
            <circle cx="450" cy="50" r="50" fill="yellow" />
            <rect id="myRect" width="100" height="100" />

        </g>
</svg>
```

1. use.startElement : empty
2. pink.circle.startElement: pink circle
3. g.startElement: no update (#myRect is resolved, but it has externalResourcesRequired set to true, so the referenced node is unresolved and rendering is stopped).
4. yellow.circle.startElement: no update (rendering suspended because of use)
5. myRect.rect.startElement: no update
6. g.endElement. Resources referenced by use become resolved and can be rendered. Rendering can proceed: green rect, pink circle, yellow circle, red rect

Font Resolution Example

Example: progRend05.svg

```
    <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.2"
         baseProfile="tiny" id="svg-root" width="100%" height="100%" viewBox="0 0 480 360">
     <desc>Font Resolution Example</desc>
     <text x="240" y="230" text-anchor="middle" font-size="120"
        font-family="fontC, fontB, fontA">A</text>

     <defs>
       <font id="fontA" horiz-adv-x="224" >
         <font-face
            font-family="fontA"
            units-per-em="1000"
            panose-1="0 0 0 0 0 0 0 0 0 0"
            ascent="917"
            descent="-250"
            alphabetic="0" />
       <missing-glyph horiz-adv-x="800" d="..." />
       <glyph unicode="A" glyph-name="A" ... />
     </font>

     <font id="fontB" horiz-adv-x="224">
       <font-face
           font-family="fontB"
           units-per-em="1000"
           panose-1="0 0 0 0 0 0 0 0 0 0"
```

```
          ascent="917"
          descent="-250"
          alphabetic="0" />

      <missing-glyph ... />
      <glyph unicode="A" glyph-name="B" ... />

    </font>

    <font id="fontC" horiz-adv-x="224" >
      <font-face
          font-family="fontC"
          units-per-em="1000"
          panose-1="0 0 0 0 0 0 0 0 0 0"
          ascent="917"
          descent="-250"
          alphabetic="0" />
        <missing-glyph ... />
        <glyph unicode="A" glyph-name="C" ... />
      </font>
    </defs>

  </svg>
```

Progressive rendering:

1. text.startElement: 'A' rendered with the default font
2. defs.startElement: no update
3. fontA.font.startElement: no update
4. fontA.font-face.startElement: no update
5. fontA.missingGlyph.startElement: no update
6. fontA.glyphA.startElement: no update
7. fontA.font.endElement: 'A' rendered with fontB (represents current document state rendering)
8. fontB.font.startElement: no update
9. fontB.font-face.startElement: no update
10. fontB.missingGlyph.startElement: no update
11. fontB.glyphA.startElement: no update
12. fontB.font.endElement: 'A' rendered with fontB (represents current document state rendering)
13. fontC.font.startElement: no update
14. fontC.font-face.startElement: no update
15. fontC.missingGlyph.startElement: no update
16. fontC.glyphA.startElement:
17. fontC.font.endElement: 'A' rendered with fontC (represents current document state rendering)

5.9.3 The 'prefetch' element

In SVG 1.1 it is not clear when an user agent should begin downloading references media, particularly when the media is not used in the initial document state (e.g. it is offscreen or hidden). SVG 1.2 does not require user agents to download referenced media that is not visual at the time the document is loaded. This means there may be a pause to download the file the first time a piece of media is displayed. More advanced user agents may wish to predict that particular media streams will be needed and therefore download them in anticipation.

SVG 1.2 also adds functionality (adapted from Section 4.4 of SMIL 2.0 - The PrefetchControl Module) to allow content developers to suggest fetching content from the server before it is needed to improve the rendering performance of the document.

The prefetch element will give a suggestion or hint to a user agent that media will be used in the future and the author would like part or all of it fetched ahead of time to make the document playback smoother. User-agents can ignore prefetch elements, though doing so may cause an interruption in the document playback when the resource is needed. It gives authoring tools and authors the ability to schedule retrieval of resources when they think that there is available bandwidth or time to do it.

When instead of referring to external media, prefetch refers to the same document it occurs in, then it can only reference a top level 'g' element. To enable smooth playback during progressive downloading in this scenario, it is recommended that each adjacent top level 'g' element contain adjacent chronological scenes in the animation. In this case the prefetch element must appear in a defs block before all defined 'g' elements in the document. In such cases, 'prefetch' is used to tell the user agent how much it needs to buffer in order to be able to play content back in a smooth and predictable manner.

None of the attributes on the prefetch element are animatable or inherited.

Schema: prefetch

```
<define name='prefetch'>
  <element name='prefetch'>
    <ref name='prefetch.AT'/>
    <ref name='prefetch.CM'/>
  </element>
</define>

<define name='prefetch.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.XLinkRequired.attr'/>
  <optional>
    <attribute name='mediaSize' svg:animatable='false' svg:inheritable='false'>
      <ref name='NumberOrPercentage.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='mediaTime' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='mediaCharacterEncoding' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='mediaContentEncodings' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='bandwidth' svg:animatable='false' svg:inheritable='false'>
      <ref name='NumberOrPercentage.datatype'/>
    </attribute>
  </optional>
</define>
```

```
  <define name='prefetch.CM'>
    <zeroOrMore>
      <ref name='svg.Desc.group'/>
    </zeroOrMore>
  </define>
```

Attribute definitions:

mediaSize = " <number> "
Defines how much of the media to fetch in bytes as a function of the file size of the media.

When prefetch refers to a resource in the same document (e.g. a 'g' element), the mediaSize attribute indicates the size in bytes of the g element and its children. That size corresponds to the encodings used when transmitting the document. If the document is encoded in UTF-8 and gzipped, then the size of the gzipped UTF-8 fragment applies. If that same document were decompressed and transcoded to UTF-16, the hints will become stale. Since streaming hints are to be used primarily in streaming scenarii, it is not expected that hint staleness will occur frequently.

Animatable: no.

mediaTime = " <time> "
Defines how much of the media to fetch as a function of the duration of the media. For discrete media (non-time based media like image/png) using this attribute causes the entire resource to be fetched.

When prefetch refers to a resource in the same document (e.g. a 'g' element), this is the active duration of the referenced 'g' element. In cases where the exact active duration can not be calculated before hand (e.g. end of an animation depends on user interaction), it is suggested that the content author estimate the minimum active duration for the referenced 'g'. This estimate, even if zero, will allow the user agent to calculate how much of the overall document to download before beginning playback in a streaming scenario.

Animatable: no.

bandwidth = " <number> "
Defines how much network bandwidth the user agent should use when doing the prefetch. Any attribute with a value of "0" is ignored and treated as if the attribute wasn't specified.

Animatable: no.

mediaCharacterEncoding = " <string>"
The mediaCharacterEncoding attribute indicates the XML character set encoding (UTF-8, ISO-8859-1, etc.) that the mediaSize attribute applies to. Tools that produce SVG must include this attribute if they specify the mediaSize attribute. The main use of this attribute is to know what character encoding was used when measuring mediaSize so that staleness of the hints may be easily detected.

Animatable: no.

mediaContentEncodings = " <list>"
The mediaContentEncodings attribute is a white space separated list of the content encodings (gzip, compress, etc.) that the mediaSize attribute applies to. The order of the list is the order in which the content encodings were applied to encode the data. Note that while situations in which multiple content codings are applied are currently rare, they are allowed by HTTP and thus that functionality is supported by SVG. Tools that produce SVG must include this attribute if they specify the mediaSize attribute. The main use of this attribute is to know what parameters were used when measuring mediaSize so that staleness of the hints may be easily detected.

Animatable: no.

xlink:href = " <iri> "
An IRI reference to the resource to prefetch.

Animatable: no.

When prefetch refers to external media, if both mediaSize and mediaTime are specified, then mediaSize is used and mediaTime is ignored.

When prefetch refers to a resource in the same document (e.g. a 'g' element), both the mediaSize and mediaTime attributes can be used together by a more advanced user agent to determine how much it needs to buffer in order to be able to play content back in a smooth manner.

Below is an example of the prefetch element when it refers to external media:

Example: prefetch01.svg

```
<svg width="400" height="300" version="1.2"
        xmlns="http://www.w3.org/2000/svg" baseProfile="tiny"
        xmlns:xlink="http://www.w3.org/1999/xlink">

     <desc>
        Prefetch the large images before starting the animation
        if possible.
     </desc>

     <defs>
       <prefetch id="pf1" xlink:href="http://www.example.com/images/huge1.png"/>
       <prefetch id="pf2" xlink:href="http://www.example.com/images/huge2.png"/>
       <prefetch id="pf3" xlink:href="http://www.example.com/images/huge3.png"/>
     </defs>

     <image x="0" y="0" width="400" height="300"
        xlink:href="http://www.example.com/images/huge1.png"
        display="none">

        <set attributeName="display" to="inline" begin="10s"/>

        <animate attributeName="xlink:href" values="
               http://www.example.com/images/huge1.png;
               http://www.example.com/images/huge2.png;
               http://www.example.com/images/huge3.png"
            begin="15s" dur="30s"/>
     </image>

   </svg>
```

Below is an example of the prefetch element when it refers to a resource (e.g. a 'g' element in the same document):

Example: prefetch02.svg

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<svg width="400" height="300" version="1.2"
        xmlns="http://www.w3.org/2000/svg" baseProfile="tiny"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        timelineBegin="onStart"
        playbackOrder="forwardOnly">

     <desc>
        Example of using SVGT 1.2 features for smooth playback
        during progressive downloading.
     </desc>

     <defs>

       <prefetch id="pf1" xlink:href="#scene1"
                 mediaCharacterEncoding="UTF-16"
                 mediaTime="5s" mediaSize="48" mediaEncodings="UTF-8"/>

       <prefetch id="pf2" xlink:href="#scene2"
                 mediaCharacterEncoding="UTF-16"
                 mediaTime="10s" mediaSize="1234" mediaEncodings="UTF-8"/>

       <prefetch id="pf3" xlink:href="#scene3"
                 mediaCharacterEncoding="UTF-16"
                 mediaTime="5s" mediaSize="62" mediaEncodings="UTF-8"/>

     </defs>

     <g id="scene1">
       <discard at="6s"/>
       <!-- graphics for scene 1 go here -->
     </g>

     <g id="scene2">
       <discard at="16s"/>
       <!-- graphics for scene 2 go here -->
     </g>

     <g id="scene3">
       <discard at="21s"/>
       <!-- graphics for scene 3 go here -->
     </g>

</svg>
```

5.10 Common attributes

5.10.1 Attributes common to all elements: **class**, **id**, **xml:id** and **xml:base**

The **class**, **id**, **xml:id** and **xml:base** attributes are available on all SVG elements:

Attribute definitions:

`class` = *list*
This attribute indicates membership in one or more sets. Multiple set names must be separated by white space characters.

Animatable: yes.

`id` = "name"
XML attribute for assigning a unique *name* to an element. Refer to the "Extensible Markup Language (XML) 1.0" Recommendation [XML10].

Animatable: no.

`xml:id` = "name"
Standard XML attribute for assigning a unique *name* to an element. Refer to the "xml:id Version 1.0" [xml:id]. Recommended for new content. Only a single attribute of type ID may be used on a given element; do not use both id and xml:id on the same element.

Animatable: no.

`xml:base` = " <iri> "
Specifies a base IRI other than the base IRI of the document or external entity. Refer to the "XML Base" specification [XML-BASE].

Animatable: no.

5.10.2 The **xml:lang** and **xml:space** attributes

Elements that might contain character data content have attributes **xml:lang** and **xml:space**:

Schema: langspace

```
<attribute name='xml:space' svg:animatable='false' svg:inheritable='false'>
   <choice>
      <value>default</value>
      <value>preserve</value>
   </choice>
</attribute>

<attribute name='xml:lang' svg:animatable='false' svg:inheritable='false'>
  <choice>
     <ref name='LanguageCode.datatype'/>
     <empty/>
  </choice>
</attribute>
```

Attribute definitions:

`xml:lang` = "languageID"
Standard XML attribute to specify the language (e.g., English) used in the contents and attribute values of particular elements. Refer to the "Extensible Markup Language (XML) 1.0" Recommendation [XML10].

Animatable: no.

`xml:space = "{default | preserve}"`
Standard XML attribute to specify whether white space is preserved in character data. The only possible values are *default* and *preserve*. Refer to the "Extensible Markup Language (XML) 1.0" Recommendation [XML10] and to the discussion white space handling in SVG.

Animatable: no.

5.11 Core Attribute Module

The Core Attribute Module contains the following attributes:

- id
- xml:base
- xml:lang
- xml:space

5.12 Structure Module

The Structure Module contains the following elements:

- svg
- g
- defs
- desc
- title
- metadata
- use

5.13 Conditional Processing Module

The Conditional Processing Module contains the following element:

- switch

5.14 Conditional Processing Attribute Module

The Conditional Processing Attribute Module contains the following attributes:

- requiredFeatures
- requiredFonts
- requiredFormats
- requiredExtensions
- systemLanguage

5.15 Image Module

The Image Module contains the following element:

- image

5.16 Prefetch Module

The Prefetch Module contains the following element:

- prefetch

5.17 ExternalResourcesRequired Attribute Module

The ExternalResourcesRequired Attribute Module contains the following attributes:

- externalResourcesRequired

SVG Tiny 1.2 - 20050413 Top | Contents | Previous | Next | Elements | Attributes

6 Styling

Contents

6.1 SVG's styling properties

SVG uses styling properties to describe many of its document parameters. Styling properties define how the graphics elements in the SVG content are to be rendered. SVG uses styling properties 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'**.
- Parameters which impact the way that graphical elements are rendered.

SVG shares many of its styling properties with CSS [CSS2] and XSL [XSL]. Except for any additional SVG-specific rules explicitly mentioned in this specification, the normative definition of properties that are shared with CSS and XSL is the definition of the property from the CSS2 specification [CSS2].

The following properties are shared between CSS2 and SVG. Apart from **'display'**, these properties are also defined in XSL:

- Font properties:
  - **'font-family'**
  - **'font-size'**
  - **'font-style'**
  - **'font-weight'**
- Other properties for visual media:
  - **'color'** is used to provide a potential indirect value (**currentColor**) for the **'fill'**, **'stroke'**, **'stop-color'** properties. (The SVG properties which support color allow a color specification which is extended from CSS2 to accommodate color definitions in arbitrary color spaces.
  - **'display'**
  - **'visibility'**

The following SVG properties are not defined in [CSS2]. The complete normative definitions for these properties are found in this specification:

- Gradient properties:
  - **'stop-color'**
  - **'stop-opacity'**
- Interactivity properties:
  - **'pointer-events'**
- Multimedia properties:
  - **'audio-level'**
- Color and Painting properties:
  - **'color-rendering'**
  - **'fill'**
  - **'fill-opacity'**
  - **'fill-rule'**
  - **'image-rendering'**
  - **'shape-rendering'**
  - **'solid-color'**
  - **'stroke'**
  - **'stroke-dasharray'**
  - **'stroke-dashoffset'**
  - **'stroke-linecap'**
  - **'stroke-linejoin'**
  - **'stroke-miterlimit'**
  - **'stroke-opacity'**
  - **'stroke-width'**
  - **'text-rendering'**
  - **'vector-effect'**
  - **'viewport-fill'**
  - **'viewport-fill-opacity'**
- Text properties:
  - **'line-increment'**
  - **'text-anchor'**

A table that lists and summarizes the styling properties can be found in the Property Index.

6.2 Usage scenarios for styling

SVG has many usage scenarios, each with different needs. Here are three common usage scenarios:

1. **SVG content used as an exchange format (style sheet language-independent)**:

   In some usage scenarios, reliable interoperability of SVG content across software tools is the main goal. Since support for a particular style sheet language is not guaranteed across all implementations, it is a requirement that SVG content can be fully specified without the use of a style sheet language.

2. **SVG content generated as the output from XSLT [XSLT]**:

   XSLT offers the ability to take a stream of arbitrary XML content as input, apply potentially complex transformations, and then generate SVG content as output. XSLT can be used to transform XML data extracted from databases into an SVG graphical representation of that data. It is a requirement that fully specified SVG content can be generated from XSLT.

3. **SVG content styled with CSS [CSS2]**:

   CSS is a widely implemented declarative language for assigning styling properties to XML content, including SVG. It represents a combination of features, simplicity and compactness that makes it very suitable for many applications of SVG. SVG Tiny 1.2 does not support CSS selectors applied to SVG content.

6.3 Specifying properties using the presentation attributes

For each styling property defined in this specification (see Property Index), there is a corresponding XML attribute (the presentation attribute) with the same name that is available on all relevant SVG elements. For example, SVG has a **'fill'** property that defines how to paint the interior of a shape. There is a corresponding presentation attribute with the same name (i.e., **fill**) that can be used to specify a value for the **'fill'** property on a given element.

The following example shows how the **'fill'** and **'stroke'** properties can be assigned to a rectangle using the **fill** and **stroke** presentation attributes. The rectangle will be filled with red and outlined with blue:

Example: 06_01.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100%" height="100%" viewBox="0 0 1000 500"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <rect x="200" y="100" width="600" height="300"
        fill="red" stroke="blue" stroke-width="3"/>
```

```
</svg>
```

The presentation attributes offer the following advantages:

- **Broad support**. All versions of Conforming SVG Interpreters and Conforming SVG Viewers are required to support the presentation attributes.
- **Simplicity**. Styling properties can be attached to elements by simply providing a value for the presentation attribute on the proper elements.
- **Restyling**. SVG content that uses the presentation attributes is highly compatible with downstream processing using XSLT [XSLT] or supplemental styling by adding CSS style rules to override some of the presentation attributes.
- **Convenient generation using XSLT [XSLT]**. In some cases, XSLT can be used to generate fully styled SVG content. The presentation attributes are compatible with convenient generation of SVG from XSLT.

In some situations, SVG content that uses the presentation attributes has potential limitations versus SVG content that is styled with a style sheet language such as CSS. In other situations, such as when an XSLT style sheet generates SVG content from semantically rich XML source files, the limitations below may not apply. Depending on the situation, some of the following potential limitations may or may not apply to the presentation attributes:

- **Styling attached to content**. The presentation attributes are attached directly to particular elements, thereby diminishing potential advantages that comes from abstracting styling from content, such as the ability to restyle documents for different uses and environments.
- **Flattened data model**. In and of themselves, the presentation attributes do not offer the higher level abstractions that you get with a styling system, such as the ability to define named collections of properties which are applied to particular categories of elements. The result is that, in many cases, important higher level semantic information can be lost, potentially making document reuse and restyling more difficult.
- **Potential increase in file size**. Many types of graphics use similar styling properties across multiple elements. For example, a company organization chart might assign one collection of styling properties to the boxes around temporary workers (e.g., dashed outlines, red fill), and a different collection of styling properties to permanent workers (e.g., solid outlines, blue fill). Styling systems such as CSS allow collections of properties to be defined once in a file. With the styling attributes, it might be necessary to specify presentation attributes on each different element.

An **!important** declaration within a presentation attribute definition is an error.

Animation of presentation attributes is equivalent to animating the corresponding property. Thus, for properties defined for SVG Tiny 1.2, the same effect occurs from animating the presentation attribute with **attributeType="XML"** as occurs with animating the corresponding property with **attributeType="CSS"**.

6.4 Styling with XSL

XSL style sheets (see [XSLT]) define how to transform XML content into something else, usually other XML. When XSLT is used in conjunction with SVG, sometimes SVG content will serve as both input and output for XSL style sheets. Other times, XSL style sheets will take non-SVG content as input and generate SVG content as output.

The following example uses an external XSL style sheet to transform SVG content into modified SVG content. The style sheet sets the **'fill'** and **'stroke'** properties on all rectangles to red and blue, respectively:

mystyle.xsl

Example: 06_02.xsl

```
<?xml version="1.0" standalone="no"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:svg="http://www.w3.org/2000/svg">
  <xsl:output
    method="xml"
    encoding="utf-8"
    doctype-public="-//W3C//DTD SVG 1.1//EN"
    doctype-system="http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"/>
  <!-- Add version to topmost 'svg' element -->
  <xsl:template match="/svg:svg">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:attribute name="version">1.1</xsl:attribute>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <!-- Add styling to all 'rect' elements -->
  <xsl:template match="svg:rect">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:attribute name="fill">red</xsl:attribute>
      <xsl:attribute name="stroke">blue</xsl:attribute>
      <xsl:attribute name="stroke-width">3</xsl:attribute>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

SVG file to be transformed by mystyle.xsl

Example: 06_03.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="10cm" height="5cm" viewBox="0 0 100 50" version="1.2"
     xmlns="http://www.w3.org/2000/svg" baseProfile="tiny">
  <rect x="20" y="10" width="60" height="30"/>
</svg>
```

SVG content after applying mystyle.xsl

Example: 06_04.svg

```
<?xml version="1.0" encoding="utf-8"?>
<svg width="10cm" height="5cm" viewBox="0 0 10 5" version="1.2"
     xmlns="http://www.w3.org/2000/svg" baseProfile="tiny">
  <rect x="2" y="1" width="6" height="3" fill="red" stroke="blue" stroke-width="3"/>
</svg>
```

6.5 Case sensitivity of property names and values

Property declarations via presentation attributes are expressed in XML [XML10], which is case-sensitive and must match the exact property name. When using a presentation attribute to specify a value for the **'fill'** property, the presentation attribute must be specified as **'fill'** and not **'FILL'** or **'Fill'**. Keyword values, such as "italic" in `font-style="italic"`, are also case-sensitive and must be specified using the exact case used in the specification which defines the given keyword. For example, the keyword "sRGB" must have lowercase "s" and uppercase "RGB".

6.6 Facilities from CSS and XSL used by SVG

SVG shares various relevant properties and approaches common to CSS and XSL, plus the semantics of many of the processing rules.

SVG shares the following facilities with CSS and XSL:

- Shared properties. Many of SVG's properties are shared between CSS2, XSL and SVG. (See list of shared properties).
- Syntax rules. (The normative references are [CSS2 syntax and basic data types] and [The grammar of CSS2].)
- Allowable data types. (The normative reference is [CSS2 syntax and basic data types]), with the exception that SVG allows <length> values without a unit identifier. See Units.)
- Inheritance rules.
- The color keywords from CSS2 that correspond to the colors used by objects in the user's environment. (The normative reference is [CSS2 system colors].)

6.7 Property inheritance

Property inheritance in SVG follows the property inheritance rules defined in the CSS2 specification. The normative definition for property inheritance is section 6.2 of the CSS2 specification (see Inheritance).

The definition of each property indicates whether the property can inherit the value of its parent.

In SVG, as in CSS2, most elements inherit computed values [CSS2-COMPUTED]. For cases where something other than computed values are inherited, the property definition will describe the inheritance rules. For specified values [CSS2-SPECIFIED] which are expressed in user units, in pixels (e.g., "20px") or in absolute values [CSS2-COMPUTED], the computed value equals the specified value. For specified values which use certain relative units (i.e., *em*, *ex* and percentages), the computed value will have the same units as the value to which it is relative. Thus, if the parent element has a **'font-size'** of "10pt" and the current element has a **'font-size'** of "120%", then the computed value for **'font-size'** on the current element will be "12pt". In cases where the referenced value for relative units is not expressed in any of the standard SVG units (i.e., CSS units or user units), such as when a percentage is used relative to the current viewport or an object bounding box, then the computed value will be in user units.

Note that SVG has some facilities wherein a property which is specified on an ancestor element might effect its descendant element, even if the descendant element has a different assigned value for that property. The key concept is that property assignment (with possible property inheritance) happens first. After properties values have been assigned to the various elements, then the user agent applies the semantics of each assigned property, which might result in the property assignment of an ancestor element affecting the rendering of its descendants.

SVG Tiny 1.2 - 20050413 Top Contents Previous Next Elements Attributes

7 Coordinate Systems, Transformations and Units

Contents

7.1 Introduction

For all media, the SVG canvas describes "the space where the SVG content 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 SVG viewport. For visual media [CSS2-VISUAL], the SVG viewport is the viewing area where the user sees the SVG content.

The size of the SVG 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 fragment and its parent (real or implicit). Once that negotiation process is completed, the SVG user agent is provided the following information:

- a number (usually an integer) that represents the width in "pixels" of the viewport
- a number (usually an integer) that represents the height in "pixels" of the viewport
- (highly desirable but not required) a real number value that indicates the size in real world units, such as millimeters, of a "pixel" (i.e., a *px* unit as defined in [CSS2 lengths])

Using the above information, the SVG user agent determines the viewport, an initial viewport coordinate system and an initial user coordinate system such that the two coordinates systems are identical. Both coordinates systems are established such that the origin matches the origin of the viewport (for the root viewport, the viewport origin is at the top/left corner), and one unit in the initial coordinate system equals one "pixel" in the viewport. (See Initial coordinate system.) The viewport coordinate system is also called viewport space and the user coordinate system is also called user space.

Lengths in SVG are specified as values in user space (e.g. "15") and in a couple of special cases with an additional absolute unit measure (e.g. "15mm"). See Units for details.

A new user space (i.e., a new current coordinate system) can be established at any place within an SVG document fragment 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 coordinate system transformations are fundamental operations to 2D graphics and represent the usual method of controlling the size, position, rotation and skew of graphic objects.

New viewports also can be established. By establishing a new viewport, you can 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.)

7.2 The initial viewport

The SVG user agent negotiates with its parent user agent to determine the viewport into which the SVG user agent can render the document. In some circumstances, SVG content will be embedded (by reference or inline) within a containing document. This containing document might include attributes, properties and/or other parameters (explicit or implicit) which specify or provide hints about the dimensions of the viewport for the SVG content. SVG content itself optionally can provide information about the appropriate viewport region for the content via the **width** and **height** XML attributes on the **'svg'** element. The negotiation process uses any information provided by the containing document and the SVG content itself to choose the viewport location and size.

The **width** attribute on the **'svg'** element establishes the viewport's width, unless the following conditions are met:

- the SVG content is a separately stored resource that is embedded by reference (such as the **'object'** element in [XHTML]), or the SVG content is embedded inline within a containing document;
- and the referencing element or containing document is styled using CSS [CSS2] or XSL [XSL];
- and there are CSS-compatible positioning properties [CSS2-POSN] specified on the referencing element (e.g., the **'object'** element) or on the containing document's **'svg'** element that are sufficient to establish the width of the viewport.

Under these conditions, the positioning properties establish the viewport's width.

Similarly, if there are positioning properties [CSS2-POSN] specified on the referencing element that are sufficient to establish the height of the viewport, then these positioning properties establish the viewport's height; otherwise, the **height** attribute on the **'svg'** element establishes the viewport's height.

If the **width** or **height** attributes on the **'svg'** element are in user units (i.e., no unit identifier has been provided), then the value is assumed to be equivalent to the same number of "px" units (see Units).

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

Example: 07_01.xml

```
<?xml version="1.0" standalone="yes"?>
<parent xmlns="http://some.url">
   <!-- SVG graphic -->
   <svg xmlns='http://www.w3.org/2000/svg'
      width="100px" height="200px" version="1.1">
      <path d="M100,100 Q200,400,300,100"/>
      <!-- rest of SVG graphic would go here -->
   </svg>
</parent>
```

7.3 The initial coordinate system

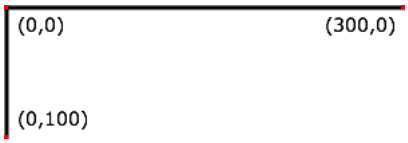
For the **'svg'** element, the SVG user agent determines an initial viewport coordinate system and an initial user coordinate system such that the two coordinates systems are identical. The origin of both coordinate systems is at the origin of the viewport, and one unit in the initial coordinate system equals one "pixel" (i.e., a *px* unit as defined in [CSS2 lengths]) in the viewport. In most cases, such as stand-alone SVG documents or SVG document fragments embedded (by reference or inline) within XML parent documents where the parent's layout is determined by CSS [CSS2] or XSL [XSL], the initial viewport coordinate system (and therefore the initial user coordinate system) has its origin at the top/left of the viewport, with the positive x-axis pointing towards the right, the positive y-axis pointing down, and text rendered with an "upright" orientation, which means glyphs are oriented such that Roman characters and full-size ideographic characters for Asian scripts have the top edge of the corresponding glyphs oriented upwards and the right edge of the corresponding glyphs oriented to the right.

If the SVG implementation is part of a user agent which supports styling XML documents using CSS2-compatible *px* units, then the SVG user agent should get its initial value for the size of a *px* unit in real world units to match the value used for other XML styling operations; otherwise, if the user agent can determine the size of a *px* unit from its environment, it should use that value; otherwise, it should choose an appropriate size for one *px* unit. In all cases, the size of a *px* must be in conformance with the rules described in [CSS2 lengths].

Example 07_02 below shows that the initial coordinate system has the origin at the top/left with the x-axis pointing to the right and the y-axis pointing down. The initial user coordinate system has one user unit equal to the parent (implicit or explicit) user agent's "pixel".

Example: 07_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="300px" height="100px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example InitialCoords - SVG's initial coordinate system</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <line x1="0" y1="1.5" x2="300" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="100" />
  </g>
  <g fill="red" stroke="none" >
    <rect x="0" y="0" width="3" height="3" />
    <rect x="297" y="0" width="3" height="3" />
    <rect x="0" y="97" width="3" height="3" />
  </g>
  <g font-size="14" font-family="Verdana" >
    <text x="10" y="20">(0,0)</text>
    <text x="240" y="20">(300,0)</text>
    <text x="10" y="90">(0,100)</text>
  </g>
</svg>
```



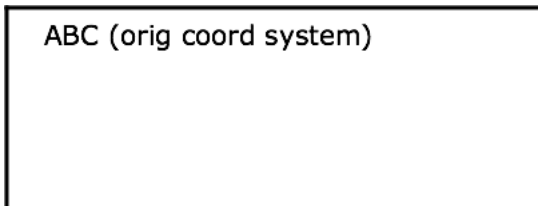
7.4 Coordinate system transformations

A new user space (i.e., a new current coordinate system) can be established by specifying transformations in the form of a **transform** attribute on a container element or graphics element or a **viewBox** attribute on the **'svg'** element. The **transform** and **viewBox** attributes transform user space coordinates and lengths on sibling attributes on the given element (see effect of the transform attribute on sibling attributes and effect of the viewBox attribute on sibling attributes) and all of its descendants. Transformations can be nested, in which case the effect of the transformations are cumulative.

Example 07_03 below shows a document without transformations. The text string is specified in the initial coordinate system.

Example: 07_03.svg

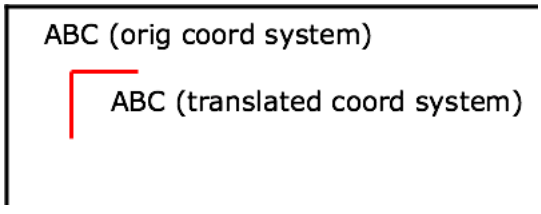
```
<?xml version="1.0" standalone="no"?>
<svg width="400px" height="150px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example OrigCoordSys - Simple transformations: original picture</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <!-- Draw the axes of the original coordinate system -->
    <line x1="0" y1="1.5" x2="400" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="150" />
  </g>
  <g>
    <text x="30" y="30" font-size="20" font-family="Verdana" >
      ABC (orig coord system)
    </text>
  </g>
</svg>
```



Example 07_04 establishes a new user coordinate system by specifying **transform="translate(50,50)"** on the third **'g'** element below. The new user coordinate system has its origin at location (50,50) in the original coordinate system. The result of this transformation is that the coordinate (30,30) in the new user coordinate system gets mapped to coordinate (80,80) in the original coordinate system (i.e., the coordinates have been translated by 50 units in X and 50 units in Y).

Example: 07_04.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="400px" height="150px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example NewCoordSys - New user coordinate system</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <!-- Draw the axes of the original coordinate system -->
    <line x1="0" y1="1.5" x2="400" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="150" />
  </g>
  <g>
    <text x="30" y="30" font-size="20" font-family="Verdana" >
      ABC (orig coord system)
    </text>
  </g>
  <!-- Establish a new coordinate system, which is
       shifted (i.e., translated) from the initial coordinate
       system by 50 user units along each axis. -->
  <g transform="translate(50,50)">
    <g fill="none" stroke="red" stroke-width="3" >
      <!-- Draw lines of length 50 user units along
           the axes of the new coordinate system -->
      <line x1="0" y1="0" x2="50" y2="0" stroke="red" />
      <line x1="0" y1="0" x2="0" y2="50" />
    </g>
    <text x="30" y="30" font-size="20" font-family="Verdana" >
      ABC (translated coord system)
    </text>
  </g>
</svg>
```

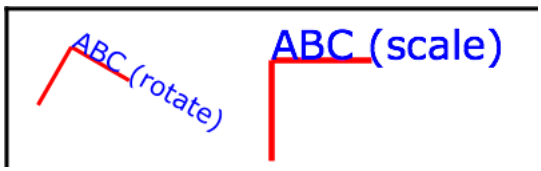


Example 07_05 illustrates simple **rotate** and **scale** transformations. The example defines two new coordinate systems:

- one which is the result of a translation by 50 units in X and 30 units in Y, followed by a rotation of 30 degrees
- another which is the result of a translation by 200 units in X and 40 units in Y, followed by a scale transformation of 1.5.

Example: 07_05.svg

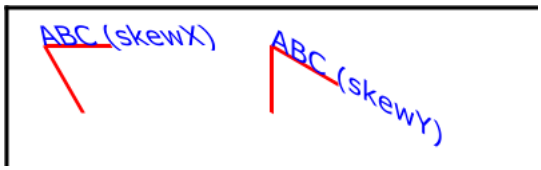
```
<?xml version="1.0" standalone="no"?>
<svg width="400px" height="120px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example RotateScale - Rotate and scale transforms</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <!-- Draw the axes of the original coordinate system -->
    <line x1="0" y1="1.5" x2="400" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="120" />
  </g>
  <!-- Establish a new coordinate system whose origin is at (50,30)
       in the initial coord. system and which is rotated by 30 degrees. -->
  <g transform="translate(50,30)">
    <g transform="rotate(30)">
      <g fill="none" stroke="red" stroke-width="3" >
        <line x1="0" y1="0" x2="50" y2="0" />
        <line x1="0" y1="0" x2="0" y2="50" />
      </g>
      <text x="0" y="0" font-size="20" font-family="Verdana" fill="blue" >
        ABC (rotate)
      </text>
    </g>
  </g>
  <!-- Establish a new coordinate system whose origin is at (200,40)
       in the initial coord. system and which is scaled by 1.5. -->
  <g transform="translate(200,40)">
    <g transform="scale(1.5)">
      <g fill="none" stroke="red" stroke-width="3" >
        <line x1="0" y1="0" x2="50" y2="0" />
        <line x1="0" y1="0" x2="0" y2="50" />
      </g>
      <text x="0" y="0" font-size="20" font-family="Verdana" fill="blue" >
        ABC (scale)
      </text>
    </g>
  </g>
</svg>
```



Example 07_06 defines two coordinate systems which are **skewed** relative to the origin coordinate system.

Example: 07_06.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="400px" height="120px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example Skew - Show effects of skewX and skewY</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <!-- Draw the axes of the original coordinate system -->
    <line x1="0" y1="1.5" x2="400" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="120" />
  </g>
  <!-- Establish a new coordinate system whose origin is at (30,30)
       in the initial coord. system and which is skewed in X by 30 degrees. -->
  <g transform="translate(30,30)">
    <g transform="skewX(30)">
      <g fill="none" stroke="red" stroke-width="3" >
        <line x1="0" y1="0" x2="50" y2="0" />
        <line x1="0" y1="0" x2="0" y2="50" />
      </g>
      <text x="0" y="0" font-size="20" font-family="Verdana" fill="blue" >
        ABC (skewX)
      </text>
    </g>
  </g>
  <!-- Establish a new coordinate system whose origin is at (200,30)
       in the initial coord. system and which is skewed in Y by 30 degrees. -->
  <g transform="translate(200,30)">
    <g transform="skewY(30)">
      <g fill="none" stroke="red" stroke-width="3" >
        <line x1="0" y1="0" x2="50" y2="0" />
        <line x1="0" y1="0" x2="0" y2="50" />
      </g>
      <text x="0" y="0" font-size="20" font-family="Verdana" fill="blue" >
        ABC (skewY)
      </text>
    </g>
  </g>
</svg>
```



Mathematically, all transformations can be represented as 3x3 transformation matrices of the following form:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

Since only six values are used in the above 3x3 matrix, a transformation matrix is also expressed as a vector: **[a b c d e f]**.

Transformations map coordinates and lengths from a new coordinate system into a previous coordinate system:

$$\begin{bmatrix} x_{prevCoordSys} \\ y_{prevCoordSys} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{newCoordSys} \\ y_{newCoordSys} \\ 1 \end{bmatrix}$$

Simple transformations are represented in matrix form as follows:

- Translation is equivalent to the matrix

  $$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

  or **[1 0 0 1 tx ty]**, where *tx* and *ty* are the distances to translate coordinates in *X* and *Y*, respectively.

- Scaling is equivalent to the matrix

  $$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  or **[sx 0 0 sy 0 0]**. One unit in the *X* and *Y* directions in the new coordinate system equals *sx* and *sy* units in the previous coordinate system, respectively.

- Rotation about the origin is equivalent to the matrix

  $$\begin{bmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  or **[cos(a) sin(a) -sin(a) cos(a) 0 0]**, which has the effect of rotating the coordinate system axes by angle *a*.

- A skew transformation along the x-axis is equivalent to the matrix

  $$\begin{bmatrix} 1 & tan(a) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  or **[1 0 tan(a) 1 0 0]**, which has the effect of skewing X coordinates by angle *a*.

- A skew transformation along the y-axis is equivalent to the matrix

  $$\begin{bmatrix} 1 & 0 & 0 \\ tan(a) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  or **[1 tan(a) 0 1 0 0]**, which has the effect of skewing Y coordinates by angle *a*.

7.5 Nested transformations

Transformations can be nested to any level. The effect of nested transformations is to post-multiply (i.e., concatenate) the subsequent transformation matrices onto previously defined transformations:

$$\begin{bmatrix} x_{prev} \\ y_{prev} \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & c_1 & e_1 \\ b_1 & d_1 & f_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_2 & c_2 & e_2 \\ b_2 & d_2 & f_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{curr} \\ y_{curr} \\ 1 \end{bmatrix}$$

For each given element, the accumulation of all transformations that have been defined on the given element and all of its ancestors up to and including the element that established the current viewport (usually, the 'svg' element which is the most immediate ancestor to the given element) is called the current transformation matrix or CTM. The CTM thus represents the mapping of current user coordinates to viewport coordinates:

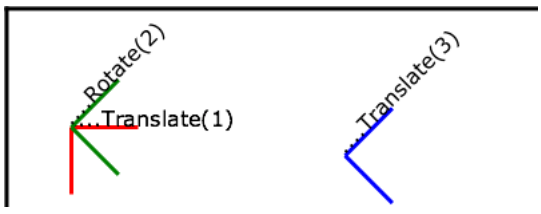
$$CTM = \begin{bmatrix} a_1 & c_1 & e_1 \\ b_1 & d_1 & f_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_2 & c_2 & e_2 \\ b_2 & d_2 & f_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \ldots \cdot \begin{bmatrix} a_n & c_n & e_n \\ b_n & d_n & f_n \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{viewport} \\ y_{viewport} \\ 1 \end{bmatrix} = CTM \cdot \begin{bmatrix} x_{userspace} \\ y_{userspace} \\ 1 \end{bmatrix}$$

Example 07_07 illustrates nested transformations.

Example: 07_07.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="400px" height="150px" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example Nested - Nested transformations</desc>
  <g fill="none" stroke="black" stroke-width="3" >
    <!-- Draw the axes of the original coordinate system -->
    <line x1="0" y1="1.5" x2="400" y2="1.5" />
    <line x1="1.5" y1="0" x2="1.5" y2="150" />
  </g>
  <!-- First, a translate -->
  <g transform="translate(50,90)">
    <g fill="none" stroke="red" stroke-width="3" >
      <line x1="0" y1="0" x2="50" y2="0" />
      <line x1="0" y1="0" x2="0" y2="50" />
    </g>
    <text x="0" y="0" font-size="16" font-family="Verdana" >
      ....Translate(1)
    </text>
    <!-- Second, a rotate -->
    <g transform="rotate(-45)">
      <g fill="none" stroke="green" stroke-width="3" >
        <line x1="0" y1="0" x2="50" y2="0" />
        <line x1="0" y1="0" x2="0" y2="50" />
      </g>
      <text x="0" y="0" font-size="16" font-family="Verdana" >
        ....Rotate(2)
      </text>
      <!-- Third, another translate -->
      <g transform="translate(130,160)">
        <g fill="none" stroke="blue" stroke-width="3" >
          <line x1="0" y1="0" x2="50" y2="0" />
          <line x1="0" y1="0" x2="0" y2="50" />
        </g>
        <text x="0" y="0" font-size="16" font-family="Verdana" >
          ....Translate(3)
        </text>
      </g>
    </g>
  </g>
</svg>
```



In the example above, the CTM within the third nested transformation (i.e., the **transform="translate(130,160)"**) consists of the concatenation of the three transformations, as follows:

$$CTM = \text{translate}(50,90), \text{rotate}(-45), \text{translate}(130,160)$$

$$= \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & 90 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} .707 & .707 & 0 \\ -.707 & .707 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 130 \\ 0 & 1 & 160 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} .707 & .707 & 255.03 \\ -.707 & .707 & 111.21 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{initial} \\ y_{initial} \\ 1 \end{bmatrix} = CTM \cdot \begin{bmatrix} x_{userspace} \\ y_{userspace} \\ 1 \end{bmatrix}$$

7.6 The **transform** attribute

transform = <transform-list> | ref(svg [, <x>, <y>])

For the **transform** attribute:

Animatable: *yes*.

See the **'animateTransform'** element for information on animating transformations.

The **<transform-list>** attribute type is defined below. The **'ref( )'** attribute type is defined in the Constrained Transformations section.

7.6.1 The TransformList value

A <transform-list> is defined as a list of transform definitions, which are applied in the order provided. The individual transform definitions are separated by whitespace and/or a comma. The available types of transform definitions include:

- **matrix(<a> <b> <c> <d> <e> <f>)**, which specifies a transformation in the form of a transformation matrix of six values. **matrix(a,b,c,d,e,f)** is equivalent to applying the transformation matrix **[a b c d e f]**.
- **translate(<tx> [<ty>])**, which specifies a translation by *tx* and *ty*. If <*ty*> is not provided, it is assumed to be zero.
- **scale(<sx> [<sy>])**, which specifies a scale operation by *sx* and *sy*. If <*sy*> is not provided, it is assumed to be equal to <*sx*>.
- **rotate(<rotate-angle> [<cx> <cy>])**, which specifies a rotation by **<rotate-angle>** degrees about a given point.

  If optional parameters **<cx>** and **<cy>** are not supplied, the rotate is about the origin of the current user coordinate system. The operation corresponds to the matrix **[cos(a) sin(a) -sin(a) cos(a) 0 0]**.

  If optional parameters **<cx>** and **<cy>** are supplied, the rotate is about the point **(<cx>, <cy>)**. The operation represents the equivalent of the following specification: **translate(<cx>, <cy>) rotate(<rotate-angle>) translate(-<cx>, -<cy>)**.
- **skewX(<skew-angle>)**, which specifies a skew transformation along the x-axis.
- **skewY(<skew-angle>)**, which specifies a skew transformation along the y-axis.

All numeric values are real <number>s.

If a list of transforms is provided, then the net effect is as if each transform had been specified separately in the order provided. For example,

Example: 07_08.svg

```
<g transform="translate(-10,-20) scale(2) rotate(45) translate(5,10)">
  <!-- graphics elements go here -->
</g>
```

is functionally equivalent to:

Example: 07_09.svg

```
<g transform="translate(-10,-20)">
  <g transform="scale(2)">
    <g transform="rotate(45)">
      <g transform="translate(5,10)">
        <!-- graphics elements go here -->
      </g>
    </g>
  </g>
</g>
```

The **transform** attribute is applied to an element before processing any other coordinate or length values supplied for that element. In the element

Example: 07_10.svg

```
<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. Attributes 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. Thus, the above **'rect'** element is functionally equivalent to:

Example: 07_11.svg

```
<g transform="scale(2)">
  <rect x="10" y="10" width="20" height="20"/>
</g>
```

The following is the Backus-Naur Form (BNF) for values for the **transform** attribute. The following notation is used:

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

```
transform-list:
    wsp* transforms? wsp*
transforms:
    transform
    | transform comma-wsp+ transforms
transform:
    matrix
    | translate
    | scale
    | rotate
    | skewX
    | skewY
matrix:
    "matrix" wsp* "(" wsp*
       number comma-wsp
       number comma-wsp
       number comma-wsp
       number comma-wsp
       number comma-wsp
       number wsp* ")"
translate:
    "translate" wsp* "(" wsp* number ( comma-wsp number )? wsp* ")"
scale:
    "scale" wsp* "(" wsp* number ( comma-wsp number )? wsp* ")"
rotate:
    "rotate" wsp* "(" wsp* number ( comma-wsp number comma-wsp number )? wsp* ")"
skewX:
    "skewX" wsp* "(" wsp* number wsp* ")"
skewY:
    "skewY" wsp* "(" wsp* number wsp* ")"
number:
    sign? integer-constant
    | sign? floating-point-constant
comma-wsp:
    (wsp+ comma? wsp*) | (comma wsp*)
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)
```

7.7 Constrained Transformations

SVG 1.2 extends the coordinate system transformations allowed on groups and elements to provide a method by which graphical objects can remain fixed in the viewport without being scaled.

The following summarizes the different transforms that are applied to a graphical object as it is rendered.

7.7.1 The user space transformation

The User Transform is the transformation that applies the user agent positioning controls to the coordinate system. This transform can be considered to be applied to a group that surrounds the outermost svg element of the document.

The user agent positioning controls consist of a translation (commonly referred to as the "pan"), a scale (commonly referred to as the "zoom") and a rotate.

```
US = User Scale (currentScale on SVGSVGElement)
UP = User Pan (currentTranslate on SVGSVGElement)
UR = User Rotate (currentRotate on SVGSVGElement)
```

The User Transform is the product of these component transformations.

```
U = User Transform
  = UP.US.UR
```

7.7.2 ViewBox to Viewport transformation

SVG elements, such as the root svg, create their own viewport. The viewBox to viewport transformation is the transformation on an svg element that adjusts the coordinate system to take the viewBox and preserveAspectRatio attributes into account.

We use the following notation for a viewBox to viewport transformation:

```
VB(svgId)
```

The 'svgId' parameter is the value of the id attribute on a given svg element.

7.7.3 Element Transform Stack

All elements in an SVG document have a transform stack. This is the list of transforms that manipulate the coordinate system between the element and its nearest ancestor svg element.

We use the following notation for the Element Transform stack on a given element:

```
TS(id)
```

The 'id' parameter is the value of the id attribute on a given element.

Example: Element transform stack

```
<svg id="root">
<g id="g" transform="scale(2)">
<rect id="r" transform="scale(4)"/>
<g id="g2">
<rect id="r2" transform="scale(0.5)" />
</g>
</g>
</svg>
```

In this example, the transforms are:

```
TS(g) = scale(2)
TS(r) = TS(g) . scale(4) = scale(8)
TS(g2) = TS(g) . scale(2) = scale(4)
TS(r2) = TS(g) . scale(0.5) = scale(1)
```

7.7.4 The Current Transform Matrix

Each element in the rendering tree has the concept of a Current Transform Matrix, or CTM. This is the product of all coordinate system transformations that apply to an element, effectively mapping the element into a coordinate system that is then transformed into device units by the user agent.

Consider the following example, with a rectangle having a set of ancestor g elements with ids "g-0" to "g-n" ("g-n" being the svg-root).

Example: Current transform matrix

```
<svg id="g-n">
...
<g id="g-n-1">
...
...
<g id="g-2">
...
<g id="g-1">
...
<g id="g-0">
...
<rect id="elt" .../>
</g>
</g>
</g>
</g>
</svg>
```

With the above definitions, the CTM for the rectangle with id "elt" is:

```
CTM(elt) = prod{i=0, i=n}(U[i].VB(g[i]).TS(g[i-1])).TS(elt)

Where prod{i=1, i=n}(f(i)) as:
prod{i=0, i=n}(f(i)) = f(n).f(n-1).f(n-2).[...].f(1).f(0)
```

In the above definition, g[n] refers to the element with the id "g-n".

The TS() of a non-existent element is the identity transform. And:

```
U[i] = Identity for i < n and U[n] = U.
```

Example: n=2

```
<svg id="g-2">
...
<g id="g-1">
...
<g id="g-0">
...
<rect id="elt" .../>
</g>
</g>
</svg>
```

This produces the following transformations:

```
CTM(elt) = U[2].VB(g[2]).TS(g[1])).U[1].VB(g[1]).TS(g[0]).U[0].VB(g[0]).TS(elt) =
= U.VB(g[2]).TS(g[1]).VB(g[1]).TS(g[0]).VB(g[0]).TS(elt)
```

Note the important relationship between an element's CTM and its parent CTM, for elements which do not define a viewport:

```
CTM(elt) = CTM(elt.parentElement).Txf(elt)
```

where Txf(elt) is the transform defined by the element's transform attribute.

7.7.5 The ref() transform value

By using the "ref()" attribute value on the transform attribute it is possible to provide simple constrained transformations.

The 'ref(svg, x, y)' transform evaluates to the inverse of the element's parent's CTM multiplied by the svg element's CTM but exclusive of the svg element's zoom/pan/rotate user transform, if any.

The x and y parameters are optional. If they are specified an additional translation is appended to the transform so that (0, 0) in the element's user space maps to (x, y) in the svg element's user space. If no x and y parameters are specified, no additional translation is applied.

Using the definitions provided above:

```
Inverse of the parent's CTM: inv(CTM(elt.parentElement))

The svg element's user transform, exclusive of zoom,
pan and rotate transforms:
   CTM(svg[0].parentElement).VB(svg[0])

CTM(svg[0].parentElement) evaluates to Identity since there
is no svg[0].parentElement element.
```

In addition, the T(x, y) translation is such that:

```
CTM(elt).(0, 0) = CTM(svg[0]).(x, y)
```

So the transform evaluates to:

```
Txf(elt) = inv(CTM(elt.parentElement)).CTM(svg[0].parentElement).VB(svg[0]).T(x, y)
```

The element's CTM is:

```
CTM(elt) = CTM(elt.parentElement).Txf(elt)
         = CTM(svg[0].parentElement).VB(svg[0]).T(x,y)
```

Example: ref() transform

A small rectangle initially marks the middle of a line. The user agent viewport is a square with sides of 200 units.

```
<svg id="root" viewBox="0 0 100 100">
<line x1="0" x2="100" y1="0" y2="100"/>
<rect id="r" transform="ref(svg)"
x="45" y="45" width="10" height="10"/>
</svg>
```

In this case:

```
Txf(r) = inv(CTM(r.parent)).CTM(root.parentElement).VB(root).T(x, y)

CTM(root.parentElement) evaluates to Identity.

T(x, y) evaluates to Identity because (x, y) is not specified

CTM(r) = CTM(r.parent).Txf(r)
= CTM(r.parent).inv(CTM(r.parent)).VB(root)
= VB(root)
= scale(2)
```

Consequently, regardless of the user transform (currentTranslate, currentScale, currentRotate) the rectangle's coordinates in viewport space will *always* be: (45, 45, 10, 10)*scale(2) = (90, 90, 20, 20). Initially, the line is from (0, 0) to (200, 200) in the viewport coordinate system. If we apply a user agent zoom of 3 (currentScale = 3), the rectangle is still (90, 90, 20, 20) but the line is (0, 0, 600, 600) and the marker no longer marks the middle of the line.

Example: ref() transform

A small rectangle always marks the middle of a line. Again, the user agent viewport is a square with sides of 200 units.

```
<svg id="root" baseProfile="tiny" viewBox="0 0 100 100">
<line x1="0" x2="100" y1="0" y2="100"/>
<g id="g" transform="ref(svg, 50, 50)">
<rect id="r" x="-5" y="-5" width="10" height="10"/>
</g>
</svg>
```

In this case:

```
Txf(g) = inv(CTM(g.parent)).CTM(root.parentElement).VB(root).T(x,y)

CTM(root.parentElement) evaluates to Identity.

CTM(g) = CTM(g.parent).Txf(r)
= CTM(g.parent).inv(CTM(g.parent)).VB(root).T(x,y)
= VB(root).T(x,y)
= scale(2).T(x,y)
```

Initially, (50, 50) in the svg user space is (100, 100) in viewport space. Therefore:

```
CTM(g).[0, 0] = CTM(root).[50, 50]
= scale(2).[50, 50]
= [100, 100]

and

scale(2).T(x,y) = [100, 100]

T(x,y) = translate(50, 50)
```

If the user agent pan was (50, 80) (modifying currentTranslate) then we now have (50, 50) in the svg element's user space located at (150, 180) in the viewport

space. This produces:

```
CTM(g).[0, 0] = CTM(root).[50, 50]
= translate(50, 80).scale(2).[50, 50]
= [150, 180]

and

scale(2).T(x,y) = [150, 180]

T(x, y) = translate(75, 90)
```

Therefore, regardless of the user transform, the rectangle will always overlap the middle of the line. Note that the rectangle will not rotate with the line (e.g., if currentRotate is set) and it will not scale either.

7.8 The **viewBox** attribute

It is often desirable to specify that a given set of graphics stretch to fit a particular container element. The **viewBox** attribute provides this capability. All elements that establish a new viewport (see elements that establish viewports) have attribute **viewBox**.

Attribute definition:

viewBox = "<list> | 'none'

<list>
A list of four numbers **<min-x>**, **<min-y>**, **<width>** and **<height>**, separated by whitespace and/or a comma, which specify a rectangle in user space which should be mapped to the bounds of the viewport established by the given element, taking into account attribute **preserveAspectRatio**. If specified, an additional transformation is applied to all descendants of the given element to achieve the specified effect.

'none'
Specifying a value of 'none' indicates that no viewBox should be used. This is exactly the same as not setting the viewBox at all.

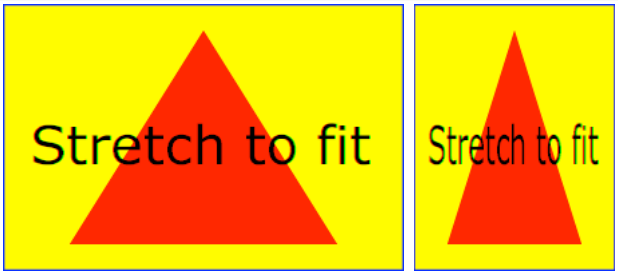
Animatable: yes.

A negative value for **<width>** or **<height>** is an error (see Error processing). A value of zero disables rendering of the element.

Example 07_12 illustrates the use of the **viewBox** attribute on the **'svg'** element to specify that the SVG content should stretch to fit bounds of the viewport.

Example: 07_12.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="300px" height="200px" version="1.2" baseProfile="tiny"
     viewBox="0 0 1500 1000" preserveAspectRatio="none"
     xmlns="http://www.w3.org/2000/svg">
  <desc>Example ViewBox - uses the viewBox
   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 (1500,1000) in user space.
       Because of the viewBox attribute above,
       the rectangle will end up filling the entire area
       reserved for the SVG content. -->
  <rect x="0" y="0" width="1500" height="1000"
        fill="yellow" stroke="blue" stroke-width="12"  />
  <!-- A large, red triangle -->
  <path fill="red"  d="M 750,100 L 250,900 L 1250,900 z"/>
  <!-- A text string that spans most of the viewport -->
  <text x="100" y="600" font-size="200" font-family="Verdana" >
    Stretch to fit
  </text>
</svg>
```



Rendered into viewport with width=300px, height=200px

width=150px, height=200px

The effect of the **viewBox** attribute is that the user agent automatically supplies the appropriate transformation matrix to map the specified rectangle in user space to the bounds of a designated region (often, the viewport). To achieve the effect of the example on the left, with viewport dimensions of 300 by 200 pixels, the user agent needs to automatically insert a transformation which scales both X and Y by 0.2. The effect is equivalent to having a viewport of size 300px by 200px and the following supplemental transformation in the document, as follows:

```
<svg width="300px" height="200px">

<g transform="scale(0.2)">
<!-- Rest of document goes here -->

</g>
</svg>
```

To achieve the effect of the example on the right, with viewport dimensions of 150 by 200 pixels, the user agent needs to automatically insert a transformation which scales X by 0.1 and Y by 0.2. The effect is equivalent to having a viewport of size 150px by 200px and the following supplemental transformation in the document, as follows:

```
<svg width="150px" height="200px">

<g transform="scale(0.1 0.2)">
<!-- Rest of document goes here -->

</g>
</svg>
```

(Note: in some cases the user agent will need to supply a **translate** transformation in addition to a **scale** transformation. For example, on an 'svg', a **translate** transformation will be needed if the **viewBox** attributes specifies values other than zero for **<min-x>** or **<min-y>**.)

Unlike the transform attribute (see effect of the transform on sibling attributes), the automatic transformation that is created due to a **viewBox** does not affect the **x**, **y**, **width** and **height** attributes on the element with the **viewBox** attribute. Thus, in the example above which shows an 'svg' element which has attributes **width**, **height** and **viewBox**, the **width** and **height** attributes represent values in the coordinate system that exists *before* the **viewBox** transformation is applied. On the other hand, like the transform attribute, it does establish a new coordinate system for all other attributes and for descendant elements.

7.9 The **preserveAspectRatio** attribute

In some cases, typically when using the viewBox attribute, it is desirable that the graphics stretch to fit non-uniformly to take up the entire viewport. In other cases, it is desirable that uniform scaling be used for the purposes of preserving the aspect ratio of the graphics.

Attribute **preserveAspectRatio="[defer] <align> [<meetOrSlice>]"**, which is available for all elements that establish a new viewport (see elements that establish viewports), indicates whether or not to force uniform scaling.

preserveAspectRatio only applies when a value has been provided for viewBox on the same element. Or, in some cases, if an implicit viewBox value can be established for the element (see each element description for details on this). If a viewBox value can not be determined then **preserveAspectRatio** is ignored.

If the value of **preserveAspectRatio** on an element that references data ('image', 'animation' and 'video') starts with 'defer' then the value of the **preserveAspectRatio** attribute on the referenced content if present should be used. If the referenced content lacks a value for **preserveAspectRatio** then the **preserveAspectRatio** attribute should be processed as normal (ignoring 'defer'). For **preserveAspectRatio** on all other elements the 'defer' portion of the attribute is ignored.

The **<align>** parameter indicates whether to force uniform scaling and, if so, the alignment method to use in case the aspect ratio of the viewBox doesn't match the aspect ratio of the viewport. The **<align>** parameter must be one of the following strings:

- **none** - Do not force uniform scaling. Scale the graphic content of the given element non-uniformly if necessary such that the element's bounding box exactly matches the viewport rectangle.
- **xMinYMin** - Force uniform scaling.

  Align the **<min-x>** of the element's viewBox with the smallest X value of the viewport.

  Align the **<min-y>** of the element's viewBox with the smallest Y value of the viewport.
- **xMidYMin** - Force uniform scaling.

  Align the midpoint X value of the element's viewBox with the midpoint X value of the viewport.

  Align the **<min-y>** of the element's viewBox with the smallest Y value of the viewport.
- **xMaxYMin** - Force uniform scaling.

  Align the **<min-x>+<width>** of the element's viewBox with the maximum X value of the viewport.

  Align the **<min-y>** of the element's viewBox with the smallest Y value of the viewport.
- **xMinYMid** - Force uniform scaling.

  Align the **<min-x>** of the element's viewBox with the smallest X value of the viewport.

  Align the midpoint Y value of the element's viewBox with the midpoint Y value of the viewport.
- **xMidYMid** (the default) - Force uniform scaling.

  Align the midpoint X value of the element's viewBox with the midpoint X value of the viewport.

  Align the midpoint Y value of the element's viewBox with the midpoint Y value of the viewport.
- **xMaxYMid** - Force uniform scaling.

  Align the **<min-x>+<width>** of the element's viewBox with the maximum X value of the viewport.

  Align the midpoint Y value of the element's viewBox with the midpoint Y value of the viewport.
- **xMinYMax** - Force uniform scaling.

  Align the **<min-x>** of the element's viewBox with the smallest X value of the viewport.

  Align the **<min-y>+<height>** of the element's viewBox with the maximum Y value of the viewport.
- **xMidYMax** - Force uniform scaling.

  Align the midpoint X value of the element's viewBox with the midpoint X value of the viewport.

  Align the **<min-y>+<height>** of the element's viewBox with the maximum Y value of the viewport.
- **xMaxYMax** - Force uniform scaling.

  Align the **<min-x>+<width>** of the element's viewBox with the maximum X value of the viewport.

  Align the **<min-y>+<height>** of the element's viewBox with the maximum Y value of the viewport.

The **<meetOrSlice>** parameter is optional and is only available due to historical reasons. The **<meetOrSlice>** is separated from the **<align>** value by one or more spaces and must equal the string **meet**. This is also the default value and therefore it is recommended that content do not specify this parameter since it adds no additional value.

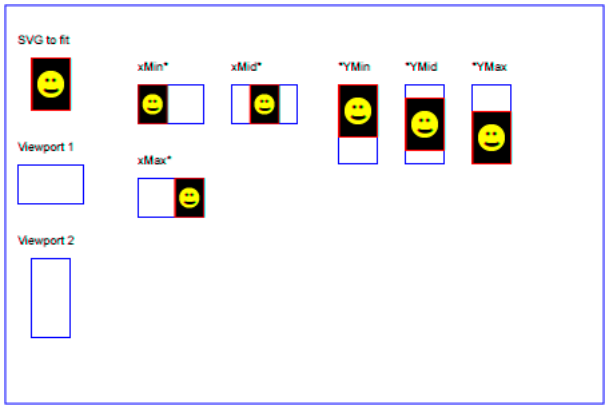
meet indicates to scale the graphic such that:

- aspect ratio is preserved
- the entire viewBox is visible within the viewport

- the **viewBox** 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 **viewBox** (i.e., the area into which the **viewBox** will draw will be smaller than the viewport).

Example PreserveAspectRatio illustrates the various options on **preserveAspectRatio**. The example creates several new viewports by including 'animation' elements (see Establishing a new viewport).



Example PreserveAspectRatio

For the **preserveAspectRatio** attribute:

Animatable: yes.

7.10 Establishing a new viewport

Some elements establish a new viewport. By establishing a new viewport, you implicitly establish a new viewport coordinate system and a new user coordinate system. Additionally, there is a new meaning for percentage units defined to be relative to the current viewport since a new viewport has been established (see Units)

'viewport-fill' and 'viewport-fill-opacity' properties can be applied on the new viewport.

The bounds of the new viewport are defined by the **x**, **y**, **width** and **height** attributes on the element establishing the new viewport, such as an 'animation' element. Both the new viewport coordinate system and the new user coordinate system have their origins at (**x**, **y**), where **x** and **y** represent the value of the corresponding attributes on the element establishing the viewport. The orientation of the new viewport coordinate system and the new user coordinate system correspond to the orientation of the current user coordinate system for the element establishing the viewport. A single unit in the new viewport coordinate system and the new user coordinate system are the same size as a single unit in the current user coordinate system for the element establishing the viewport.

For an extensive example of creating new viewports, see Example PreserveAspectRatio.

The following elements establish new viewports:

- The 'svg' element establishes the root viewport for the document.
- The 'animation' element.
- The 'image' element.
- The 'video' element.

7.11 Units

Besides the exceptions listed below all coordinates and lengths in SVG must be specified in user units, witch means that unit identifiers are not allowed.

Two exceptions exist:

- unit identifiers are allowed on the 'width' and 'height' attributes of the 'svg' element.
- Object bounding box units are allowed on 'linearGradient' and 'radialGradient' elements.

A user unit is a value in the current user coordinate system. For example:

Example: 07_17.svg

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

For the svg element's 'width'and 'height' attributes a coordinate or length value can be expressed as a number following by a unit identifier (e.g., "25cm" or "100%"). The list of unit identifiers in SVG are the following CSS unit identifiers: in, cm, mm, pt, pc, px and percentages (%). The following describes how the various unit identifiers are processed:

- One *px* unit is defined to be equal to one user unit. Thus, a length of "5px" is the same as a length of "5".
- The other absolute unit identifiers from CSS (i.e., pt, pc, cm, mm, in) are all defined as an appropriate multiple of one *px* unit (which, according to the previous item, is defined to be equal to one user unit), based on what the SVG user agent determines is the size of a *px* unit (possibly passed from the parent processor or environment at initialization time). For example, suppose that the user agent can determine from its environment that "1px" corresponds to "0.2822222mm" (i.e., 90dpi). Then, for all processing:
  - "1pt" equals "1.25px" (and therefore 1.25 user units)
  - "1pc" equals "15px" (and therefore 15 user units)
  - "1mm" would be "3.543307px" (3.543307 user units)
  - "1cm" equals "35.43307px" (and therefore 35.43307 user units)
  - "1in" equals "90px" (and therefore 90 user units)

Percentage values on 'width' and 'height' attributes mandates how much space the SVG viewport should take of the available initial viewport. See list below and initial viewport for details.

- For any width value expressed as a percentage of the viewport, the value to use is the specified percentage of the *actual-width* in user units for the nearest containing viewport, where *actual-width* is the width dimension of the viewport element within the user coordinate system for the viewport element.
- For any height value expressed as a percentage of the viewport, the value to use is the specified percentage of the *actual-height* in user units for the nearest containing viewport, where *actual-height* is the height dimension of the viewport element within the user coordinate system for the viewport element.

7.12 Object bounding box units

The following elements offer the option of expressing coordinate values and lengths as fractions of the bounding box (via keyword **objectBoundingBox**) on a given element:

Element	Attribute	Effect
'linearGradient'	gradientUnits ="objectBoundingBox"	Indicates that the attributes which specify the gradient vector (x1 , y1 , x2 , y2) represent fractions of the bounding box of the element to which the gradient is applied.
'radialGradient'	gradientUnits ="objectBoundingBox"	Indicates that the attributes which specify the center (cx , cy) and the radius (r) represent fractions of the bounding box of the element to which the gradient is applied.

In the discussion that follows, the term applicable element is the element to which the given effect applies. For gradients the applicable element is the graphics element which has its **'fill'** or **'stroke'** property referencing the given gradient. (See Inheritance of Painting Properties. For special rules concerning text elements, see the discussion of object bounding box units and text elements.)

When keyword **objectBoundingBox** is used, then the effect is as if a supplemental transformation matrix were inserted into the list of nested transformation matrices to create a new user coordinate system.

First, the (**minx**,**miny**) and (**maxx**,**maxy**) coordinates are determined for the applicable element and all of its descendants. The values **minx**, **miny**, **maxx** and **maxy** are determined by computing the maximum extent of the shape of the element in X and Y with respect to the user coordinate system for the applicable element. The bounding box is the tightest fitting rectangle aligned with the axes of the applicable element's user coordinate system that entirely encloses the applicable element and its descendants. The bounding box is computed exclusive of any values for opacity and stroke-width. For curved shapes, the bounding box encloses all portions of the shape, not just end points. For **'text'** elements, for the purposes of the bounding box calculation, each glyph is treated as a separate graphics element. The calculations assume that all glyphs occupy the full glyph cell. For example, for horizontal text, the calculations assume that each glyph extends vertically to the full ascent and descent values for the font.

Then, coordinate (0,0) in the new user coordinate system is mapped to the (minx,miny) corner of the tight bounding box within the user coordinate system of the applicable element and coordinate (1,1) in the new user coordinate system is mapped to the (maxx,maxy) corner of the tight bounding box of the applicable element. In most situations, the following transformation matrix produces the correct effect:

```
[ (maxx-minx) 0 0 (maxy-miny) minx miny ]
```

Any numeric value can be specified for values expressed as a fraction of object bounding box units. In particular, fractions less are zero or greater than one can be specified.

Keyword **objectBoundingBox** should not be used when the geometry of the applicable element has no width or no height, such as the case of a horizontal or vertical line, even when the line has actual thickness when viewed due to having a non-zero stroke width since stroke width is ignored for bounding box calculations. When the geometry of the applicable element has no width or height and **objectBoundingBox** is specified, then the given effect (e.g., a gradient) will be ignored.

7.13 Intrinsic Sizing Properties of the Viewport of SVG content

SVG must specify how to calculate some intrinsic sizing properties to enable inclusion within other languages. The intrinsic width and height of the viewport of SVG content is determined from the width and height attributes. If these are not present the default values of 100% should be used.

The intrinsic aspect ratio of the viewport of SVG content is necessary for example when including SVG from an object tag in XHTML styled with CSS. The intrinsic aspect ratio should be calculated based upon the following rules:

- If the width and height of the root SVG element are both specified in absolute units (in, mm, cm, pt, pc, px) then the aspect ratio is calculated from the width and height after resolving both values to the same units.
- If either/both of the width and height of the root svg are in percentage units, the aspect ratio is calculated from the width and height of the viewBox. If the viewBox is not present, or set to 'none', the intrinsic aspect ratio cannot be calculated and is unspecified.

Examples:

Example: Intrinsic Aspect Ratio 1

```
<svg width="10cm" height="5cm">
...
</svg>
```

In this example the intrinsic aspect ratio of the viewport is 2:1. The intrinsic width is 10cm and the intrinsic height is 5cm.

Example: Intrinsic Aspect Ratio 2

```
<svg width="100%" height="50%" viewBox="0 0 200 200">
...
</svg>
```

In this example the intrinsic aspect ratio of the outermost viewport is 1:1. An aspect ratio calculation in this case allows embedding in an object within a containing block that is only constrained in one direction.

Example: Intrinsic Aspect Ratio 3

```
<svg width="10cm" viewBox="0 0 200 200">
...
</svg>
```

In this case the intrinsic aspect ratio is 1:1.

Example: Intrinsic Aspect Ratio 4

```
<width="75%" height="10cm" viewbox="0 0 200 200">
...
</svg>
```

In this example, the intrinsic aspect ratio is 1:1.

7.14 Geographic Coordinate Systems

In order to allow interoperability between SVG content generators and user agents dealing with maps encoded in SVG, SVG encourages the use of a common metadata definition for describing the coordinate system used to generate SVG documents.

Such metadata should be added under the **'metadata'** element of the topmost **'svg'** element describing the map. They consist of an RDF description of the Coordinate Reference System definition used to generate the SVG map.

The definition should be conformant to the XML grammar described in the OpenGIS Recommendation on the Definition of Coordinate Reference System [OpenGIS Coordinate Systems]. In order to correctly map the 2-dimensional data used by SVG, the CRS must be of subtype ProjectedCRS or Geographic2dCRS. The first axis of the described CRS maps the SVG x-axis and the second axis maps the SVG y-axis. Optionally, an additional affine transformation may have been applied during this mapping. This additional transformation is described by an SVG **transform** attribute that can be added to the OpenGIS **'CoordinateReferenceSystem'** element. Note that the **transform** attribute on the **'CoordinateReferenceSystem'** does not indicate that a transformation should be applied to the data within the file, it simply describes the transformation that was applied to the data when being encoded in SVG.

There are three typical uses for the SVG transform attribute. These are described below and used in the examples.

- Most ProjectedCRS have the north direction represented by positive values of the second axis and conversely SVG has a y-down coordinate system. That's why, in order to follow the usual way to represent a map with the north at its top, it is recommended for that kind of ProjectedCRS to use the SVG transform attribute with a 'scale(1, -1)' value as in the third example below.
- Most Geographic2dCRS have the latitude as their first axis rather than the longitude, which means that the south-north axis would be represented by the x-axis in SVG instead of the usual y-axis. That's why, in order to follow the usual way to represent a map with the north at its top, it is recommended for that kind of Geographic2dCRS to use the SVG transform attribute with a 'rotate(-90)' value as in the first example (while also adding the scale(1, -1) as for ProjectedCRS).
- In addition, when converting for profiles which place restrictions on precision of real number values, it may be useful to add an additional scaling factor to retain good precision for a specific area. When generating an SVG document from WGS84 geographic coordinates (EPGS 4326), we recommend the use of an additional 100 times scaling factor corresponding to an SVG transform attribute with a 'rotate(-90) scale(100)' value (shown in the second example). Different scaling values may be required depending on the particular CRS.

The main purpose of such metadata is to indicate to the User Agent that two or more SVG documents can be overlayed or merged into a single document. Obviously, if two maps reference the same Coordinate Reference System definition and have the same SVG transform attribute value then they can be overlayed without reprojecting the data. If the maps reference different Coordinate Reference Systems and/or have different SVG transform attribute values, then a specialized cartographic User Agent may choose to transform the coordinate data to overlay the data. However, typical SVG user agents are not required to perform these types of transformations, or even recognize the metadata.

Below is a simple example of the coordinate metadata, which describes the coordinate system used by the document via a URI.

Example: 07_19.svg

```
<?xml version="1.0"?>
<svg width="100" height="100" viewBox="0 0 1000 1000" version="1.2"
     xmlns="http://www.w3.org/2000/svg" baseProfile="tiny">
     <desc>An example that references co-ordinate data.</desc>
    <metadata>
      <rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:crs = "http://www.ogc.org/crs"
               xmlns:svg="http://wwww.w3.org/2000/svg">
        <rdf:Description>
          <!-- The Coordinate Reference System is described
               through an URI. -->
          <crs:CoordinateReferenceSystem svg:transform="rotate(-90)"
                  rdf:resource=""http://www.example.org/srs/epsg.xml#4326"/>
        </rdf:Description>
      </rdf:RDF>
    </metadata>
    <!-- The actual map content -->
</svg>
```

The second example uses a well-known identifier to describe the coordinate system. Note that the coordinates used in the document have had the supplied transform applied.

Example: 07_20.svg

```
<?xml version="1.0"?>
<svg width="100" height="100" viewBox="0 0 1000 1000" version="1.2"
     xmlns="http://www.w3.org/2000/svg" baseProfile="tiny">
     <desc>Example using a well known co-ordinate system.</desc>
    <metadata>
      <rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:crs = "http://www.ogc.org/crs"
               xmlns:svg="http://wwww.w3.org/2000/svg">
        <rdf:Description>
          <!-- In case of a well-known Coordinate Reference System
               an 'Identifier' is enough to describe the CRS -->
          <crs:CoordinateReferenceSystem svg:transform="rotate(-90) scale(100, 100)">
            <crs:Identifier>
              <crs:code>4326</crs:code>
              <crs:codeSpace>EPSG</crs:codeSpace>
              <crs:edition>5.2</crs:edition>
            </crs:Identifier>
          </crs:CoordinateReferenceSystem>
```

```
        </rdf:Description>
      </rdf:RDF>
    </metadata>
    <!-- The actual map content -->
</svg>
```

The third example defines the coordinate system completely within the SVG document.

Example: 07_21.svg

```
<?xml version="1.0"?>
<svg width="100" height="100" viewBox="0 0 1000 1000" version="1.2"
    xmlns="http://www.w3.org/2000/svg" baseProfile="tiny">
    <desc>Co-ordinate Metadata defined within the SVG Document</desc>
    <metadata>
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:crs="http://www.ogc.org/crs"
               xmlns:svg="http://wwww.w3.org/2000/svg">
        <rdf:Description>
          <!-- For other CRS it should be entirely defined -->
          <crs:CoordinateReferenceSystem svg:transform="scale(1,-1)">
            <crs:NameSet>
              <crs:name>Mercator projection of WGS84</crs:name>
            </crs:NameSet>
            <crs:ProjectedCRS>
              <!-- The actual definition of the CRS -->
              <crs:CartesianCoordinateSystem>
                <crs:dimension>2</crs:dimension>
                <crs:CoordinateAxis>
                  <crs:axisDirection>north</crs:axisDirection>
                  <crs:AngularUnit>
                    <crs:Identifier>
                      <crs:code>9108</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                  </crs:AngularUnit>
                </crs:CoordinateAxis>
                <crs:CoordinateAxis>
                  <crs:axisDirection>east</crs:axisDirection>
                  <crs:AngularUnit>
                    <crs:Identifier>
                      <crs:code>9108</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                  </crs:AngularUnit>
                </crs:CoordinateAxis>
              </crs:CartesianCoordinateSystem>
              <crs:CoordinateReferenceSystem>
                <!-- the reference system of that projected system is
                          WGS84 which is EPSG 4326 in EPSG codeSpace -->
                <crs:NameSet>
                  <crs:name>WGS 84</crs:name>
                </crs:NameSet>
                <crs:Identifier>
                  <crs:code>4326</crs:code>
                  <crs:codeSpace>EPSG</crs:codeSpace>
                  <crs:edition>5.2</crs:edition>
                </crs:Identifier>
              </crs:CoordinateReferenceSystem>
              <crs:CoordinateTransformationDefinition>
                <crs:sourceDimensions>2</crs:sourceDimensions>
                <crs:targetDimensions>2</crs:targetDimensions>
                <crs:ParameterizedTransformation>
                  <crs:TransformationMethod>
                    <!-- the projection is a Mercator projection which is
                          EPSG 9805 in EPSG codeSpace -->
                    <crs:NameSet>
                      <crs:name>Mercator</crs:name>
                    </crs:NameSet>
                    <crs:Identifier>
                      <crs:code>9805</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                    <crs:description>Mercator (2SP)</crs:description>
                  </crs:TransformationMethod>
                  <crs:Parameter>
                    <crs:NameSet>
                      <crs:name>Latitude of 1st standart parallel</crs:name>
                    </crs:NameSet>
                    <crs:Identifier>
                      <crs:code>8823</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                    <crs:value>0</crs:value>
                  </crs:Parameter>
                  <crs:Parameter>
                    <crs:NameSet>
                      <crs:name>Longitude of natural origin</crs:name>
                    </crs:NameSet>
                    <crs:Identifier>
                      <crs:code>8802</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                    <crs:value>0</crs:value>
                  </crs:Parameter>
                  <crs:Parameter>
                    <crs:NameSet>
                      <crs:name>False Easting</crs:name>
```

```
                    </crs:NameSet>
                    <crs:Identifier>
                      <crs:code>8806</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                    <crs:value>0</crs:value>
                  </crs:Parameter>
                  <crs:Parameter>
                    <crs:NameSet>
                      <crs:name>False Northing</crs:name>
                    </crs:NameSet>
                    <crs:Identifier>
                      <crs:code>8807</crs:code>
                      <crs:codeSpace>EPSG</crs:codeSpace>
                      <crs:edition>5.2</crs:edition>
                    </crs:Identifier>
                    <crs:value>0</crs:value>
                  </crs:Parameter>
                </crs:ParameterizedTransformation>
              </crs:CoordinateTransformationDefinition>
            </crs:ProjectedCRS>
          </crs:CoordinateReferenceSystem>
        </rdf:Description>
      </rdf:RDF>
    </metadata>
    <!-- the actual map content -->
</svg>
```


SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

8 Paths

Contents

8.1 Introduction

Paths represent the outline of a shape which can be filled or stroked. (See Filling, Stroking and Paint Servers.)

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 the geometry of the outline of an object, defined in terms of *moveto* (set a new current point), *lineto* (draw a straight line), *curveto* (draw a curve using a cubic BÃ©zier) and *closepath* (close the current shape by drawing a line to the last *moveto*) elements. Compound paths (i.e., a path with multiple subpaths) are possible to allow effects such as "donut holes" in objects.

This chapter describes the syntax and behavior for SVG paths. Various implementation notes for SVG paths can be found in 'path' element implementation notes.

A path is defined in SVG using the 'path' element.

8.2 The 'path' element

Schema: path

```
<define name='path'>
  <element name='path'>
    <ref name='path.AT'/>
    <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
  </element>
</define>

<define name='path.AT' combine='interleave'>
  <ref name='svg.ShapeCommon.attr'/>
  <ref name='svg.D.attr'/>
  <optional>
    <attribute name='pathLength' svg:animatable='true' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
</define>
```

Attribute definitions:

d = "*path data*"
The definition of the outline of a shape. See Path data.

Animatable: yes. Path data animation is only possible when each path data specification within an animation specification has exactly the same list of path data commands as the **d** attribute. If an animation is specified and the list of path data commands is not the same, then the animation specification is in error

(see Error Processing). The animation engine interpolates each parameter to each path data command separately based on the attributes to the given animation element. Flags and booleans are interpolated as fractions between zero and one, with any non-zero value considered to be a value of one/true.

pathLength = "<number>"

The author's computation of the total length of the path, in user units. This value is used to calibrate the user agent's own distance-along-a-path calculations with that of the author. The user agent will scale all distance-along-a-path computations by the ratio of **pathLength** to the user agent's own computed value for total path length. **pathLength** potentially affects calculations for motion animation and various stroke operations.

A negative value is an error (see Error processing).

Animatable: yes.

8.3 Path data

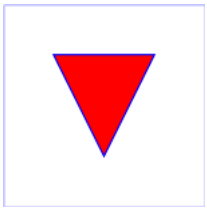
8.3.1 General information about path data

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 BÃ©ziers) and *closepath* instructions.

Example 08_01 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*).

Example: 08_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="4cm" height="4cm" viewBox="0 0 400 400"
     xmlns="http://www.w3.org/2000/svg" version="1.1" baseProfile="tiny">
  <title>Example triangle01- simple example of a 'path'</title>
  <desc>A path that draws a triangle</desc>
  <rect x="1" y="1" width="398" height="398"
        fill="none" stroke="blue" />
  <path d="M 100 100 L 300 100 L 200 300 z"
        fill="red" stroke="blue" stroke-width="3" />
</svg>
```



Path data can contain newline characters and thus can be broken up into multiple lines to improve readability. Because of line length limitations with certain related tools, it is recommended that SVG generators split long path data strings across multiple lines, with each line not exceeding 255 characters. Also note that newline characters are only allowed at certain places within path data.

The syntax of path data is concise 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 (uppercase means absolute coordinates, lowercase 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 Unicode [UNICODE] FULL STOP (".") character (also referred to in Unicode as PERIOD, dot and decimal point) 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, say: "13000.56".)

For the relative versions of the commands, all coordinate values are relative to the current point at the start of the command.

In the tables below, the following notation is used:

- (): grouping of parameters
- +: 1 or more of the given parameter(s) is required

The following sections list the commands.

8.3.2 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 (if there is one) must begin with a "moveto" command. 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.

8.3.3 The **"closepath"** command

The "closepath" (**Z** or **z**) ends the current subpath and causes an automatic straight line to be drawn from the current point to the initial point of the current subpath. If a "closepath" is followed immediately by a "moveto", then the "moveto" identifies the start point of the next subpath. If a "closepath" is followed immediately by any other command, then the next subpath starts at the same initial point as the current subpath.

When a subpath ends in a "closepath," it 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'. At the end of the command, the new current point is set to the initial point of the current subpath.

Command	Name	Parameters	Description
Z or **z**	closepath	(none)	Close the current subpath by drawing a straight line from the current point to current subpath's initial point.

8.3.4 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.

8.3.5 The curve commands

These groups of commands draw curves:

- Cubic BÃ©zier commands (**C**, **c**, **S** and **s**). A cubic BÃ©zier segment is defined by a start point, an end point, and two control points.
- Quadratic BÃ©zier commands (**Q**, **q**, **T** and **t**). A quadratic BÃ©zier segment is defined by a start point, an end point, and one control point.

8.3.6 The cubic BÃ©zier curve commands

The cubic BÃ©zier commands are as follows:

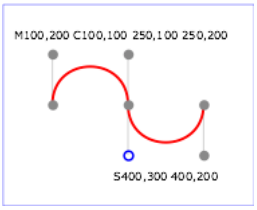
Command	Name	Parameters	Description
C (absolute) **c** (relative)	curveto	(x1 y1 x2 y2 x y)+	Draws a cubic BÃ©zier curve from the current point to (x,y) using (x1,y1) as the control point at the beginning of the curve and (x2,y2) as the control point at the end of the curve. **C** (uppercase) indicates that absolute coordinates will follow; **c** (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybÃ©zier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybÃ©zier.
S (absolute) **s** (relative)	shorthand/smooth curveto	(x2 y2 x y)+	Draws a cubic BÃ©zier 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 polybÃ©zier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybÃ©zier.

Example 08_02 shows some simple uses of cubic BÃ©zier commands within a path. The example uses an internal CSS style sheet to assign styling properties. Note that the control point for the "S" command is computed automatically as the reflection of the control point for the previous "C" command relative to the start point of the "S" command.

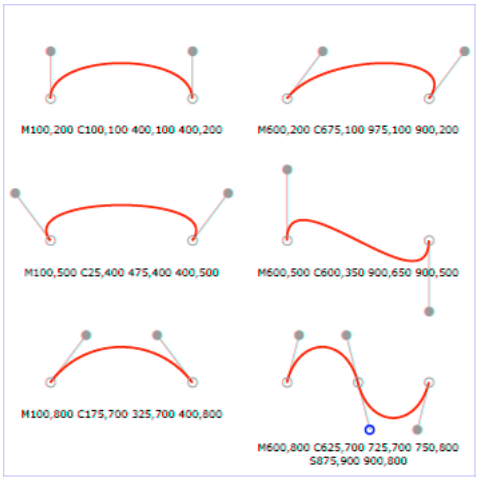
Example: 08_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="4cm" viewBox="0 0 500 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <title>Example cubic01- cubic BÃ©zier commands in path data</title>
  <desc>Picture showing a simple example of path data
        using both a "C" and an "S" command,
        along with annotations showing the control points
        and end points</desc>
  <rect fill="none" stroke="blue" stroke-width="1" x="1" y="1" width="498" height="398" />
  <polyline fill="none" stroke="#888888" stroke-width="1" points="100,200 100,100" />
  <polyline fill="none" stroke="#888888" stroke-width="1" points="250,100 250,200" />
  <polyline fill="none" stroke="#888888" stroke-width="1" points="250,200 250,300" />
  <polyline fill="none" stroke="#888888" stroke-width="1" points="400,300 400,200" />
  <path fill="none" stroke="red" stroke-width="5" d="M100,200 C100,100 250,100 250,200
                                 S400,300 400,200" />
  <circle fill="#888888" stroke="none" stroke-width="2" cx="100" cy="200" r="10" />
  <circle fill="#888888" stroke="none" stroke-width="2" cx="250" cy="200" r="10" />
  <circle fill="#888888" stroke="none" stroke-width="2" cx="400" cy="200" r="10" />
  <circle fill="#888888" stroke="none" cx="100" cy="100" r="10" />
```

```
  <circle fill="#888888" stroke="none" cx="250" cy="100" r="10" />
  <circle fill="#888888" stroke="none" cx="400" cy="300" r="10" />
  <circle fill="none" stroke="blue" stroke-width="4" cx="250" cy="300" r="9" />
  <text font-size="22" font-family="Verdana" x="25" y="70">M100,200 C100,100 250,100 250,200</text>
  <text font-size="22" font-family="Verdana" x="325" y="350"
        text-anchor="middle">S400,300 400,200</text>
</svg>
```



The following picture shows some how cubic BÃ©zier curves change their shape depending on the position of the control points. The first five examples illustrate a single cubic BÃ©zier path segment. The example at the lower right shows a "C" command followed by an "S" command.



View this example as SVG (SVG-enabled browsers only)

8.3.7 The quadratic BÃ©zier curve commands

The quadratic BÃ©zier commands are as follows:

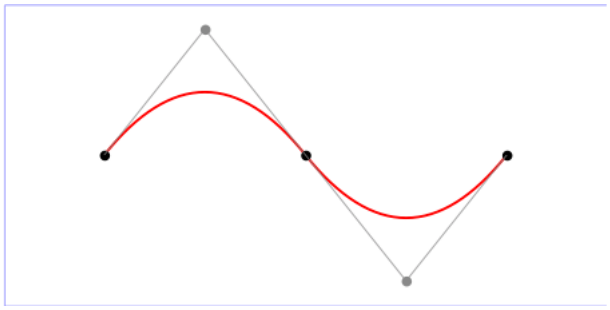
Command	Name	Parameters	Description
Q (absolute) **q** (relative)	quadratic BÃ©zier curveto	(x1 y1 x y)+	Draws a quadratic BÃ©zier curve from the current point to (x,y) using (x1,y1) as the control point. **Q** (uppercase) indicates that absolute coordinates will follow; **q** (lowercase) indicates that relative coordinates will follow. Multiple sets of coordinates may be specified to draw a polybÃ©zier. At the end of the command, the new current point becomes the final (x,y) coordinate pair used in the polybÃ©zier.
T (absolute) **t** (relative)	Shorthand/smooth quadratic BÃ©zier curveto	(x y)+	Draws a quadratic BÃ©zier 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 a 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 polybÃ©zier.

Example quad01 shows some simple uses of quadratic BÃ©zier commands within a path. Note that the control point for the "T" command is computed automatically as the reflection of the control point for the previous "Q" command relative to the start point of the "T" command.

Example: 08_03.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="12cm" height="6cm" viewBox="0 0 1200 600"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <title>Example quad01 - quadratic Bezier commands in path data</title>
  <desc>Picture showing a "Q" a "T" command,
        along with annotations showing the control points
        and end points</desc>
  <rect x="1" y="1" width="1198" height="598"
        fill="none" stroke="blue" stroke-width="1" />
  <path d="M200,300 Q400,50 600,300 T1000,300"
        fill="none" stroke="red" stroke-width="5"  />
  <!-- End points -->
  <g fill="black" >
    <circle cx="200" cy="300" r="10"/>
    <circle cx="600" cy="300" r="10"/>
    <circle cx="1000" cy="300" r="10"/>
  </g>
  <!-- Control points and lines from end points to control points -->
  <g fill="#888888" >
    <circle cx="400" cy="50" r="10"/>
```

```
    <circle cx="800" cy="550" r="10"/>
  </g>
  <path d="M200,300 L400,50 L600,300
           L800,550 L1000,300"
        fill="none" stroke="#888888" stroke-width="2" />
</svg>
```



8.3.8 The grammar for path data

The following notation is used in the Backus-Naur Form (BNF) description of the grammar for path data:

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

The following is the BNF for SVG paths.

```
svg-path:
    wsp* moveto-drawto-command-groups? wsp*
moveto-drawto-command-groups:
    moveto-drawto-command-group
    | moveto-drawto-command-group wsp* moveto-drawto-command-groups
moveto-drawto-command-group:
    moveto wsp* drawto-commands?
drawto-commands:
    drawto-command
    | drawto-command wsp* drawto-commands
drawto-command:
    closepath
    | lineto
    | horizontal-lineto
    | vertical-lineto
    | curveto
    | smooth-curveto
    | quadratic-bezier-curveto
    | smooth-quadratic-bezier-curveto
moveto:
    ( "M" | "m" ) wsp* moveto-argument-sequence
moveto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
closepath:
    ("Z" | "z")
lineto:
    ( "L" | "l" ) wsp* lineto-argument-sequence
lineto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
horizontal-lineto:
    ( "H" | "h" ) wsp* horizontal-lineto-argument-sequence
horizontal-lineto-argument-sequence:
    coordinate
    | coordinate comma-wsp? horizontal-lineto-argument-sequence
vertical-lineto:
    ( "V" | "v" ) wsp* vertical-lineto-argument-sequence
vertical-lineto-argument-sequence:
    coordinate
    | coordinate comma-wsp? vertical-lineto-argument-sequence
curveto:
    ( "C" | "c" ) wsp* curveto-argument-sequence
curveto-argument-sequence:
    curveto-argument
    | curveto-argument comma-wsp? curveto-argument-sequence
curveto-argument:
    coordinate-pair comma-wsp? coordinate-pair comma-wsp? coordinate-pair
smooth-curveto:
    ( "S" | "s" ) wsp* smooth-curveto-argument-sequence
smooth-curveto-argument-sequence:
    smooth-curveto-argument
    | smooth-curveto-argument comma-wsp? smooth-curveto-argument-sequence
smooth-curveto-argument:
    coordinate-pair comma-wsp? 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 comma-wsp?
       quadratic-bezier-curveto-argument-sequence
quadratic-bezier-curveto-argument:
    coordinate-pair comma-wsp? 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 comma-wsp? smooth-quadratic-bezier-curveto-argument-sequence
coordinate-pair:
    coordinate comma-wsp? coordinate
coordinate:
    number
nonnegative-number:
    integer-constant
    | floating-point-constant
number:
    sign? integer-constant
    | sign? floating-point-constant
flag:
    "0" | "1"
comma-wsp:
    (wsp+ comma? wsp*) | (comma wsp*)
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)
```

The processing of the BNF must consume as much of a given BNF production as possible, stopping at the point when a character is encountered which no longer satisfies the production. Thus, in the string "M 100-200", the first coordinate for the "moveto" consumes the characters "100" and stops upon encountering the minus sign because the minus sign cannot follow a digit in the production of a "coordinate". The result is that the first coordinate will be "100" and the second coordinate will be "-200".

Similarly, for the string "M 0.6.5", the first coordinate of the "moveto" consumes the characters "0.6" and stops upon encountering the second decimal point because the production of a "coordinate" only allows one decimal point. The result is that the first coordinate will be "0.6" and the second coordinate will be ".5".

Note that the BNF allows the path **'d'** attribute to be empty. This is not an error, instead it disables rendering of the path.

8.4 Distance along a path

Various operations, including motion animation and various stroke operations, require that the user agent compute the distance along the geometry of a graphics element, such as a **'path'**.

Exact mathematics exist for computing distance along a path, but the formulas are highly complex and require substantial computation. It is recommended that authoring products and user agents employ algorithms that produce as precise results as possible; however, to accommodate implementation differences and to help distance calculations produce results that approximate author intent, the **pathLength** attribute can be used to provide the author's computation of the total length of the path so that the user agent can scale distance-along-a-path computations by the ratio of **pathLength** to the user agent's own computed value for total path length.

A "moveto" operation within a **'path'** element is defined to have zero length. Only the various "lineto" and "curveto" commands contribute to path length calculations.

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

9 Basic Shapes

Contents

9.1 Introduction

SVG contains the following set of basic shape elements:

- **rectangles** (rectangle, including optional rounded corners)
- **circles**
- **ellipses**
- **lines**
- **polylines**
- **polygons**

Mathematically, these shape elements are equivalent to a **'path'** element that would construct the same shape. The basic shapes may be stroked, filled and used as clip paths. All of the properties available for **'path'** elements also apply to the basic shapes.

9.2 The 'rect' element

The 'rect' element defines a rectangle which is axis-aligned with the current user coordinate system. Rounded rectangles can be achieved by setting appropriate values for attributes rx and ry.

Schema: rect

```
  <define name='rect'>
    <element name='rect'>
      <ref name='rect.AT'/>
      <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='rect.AT' combine='interleave'>
    <ref name='svg.ShapeCommon.attr'/>
    <ref name='svg.XYWH.attr'/>
    <ref name='svg.RxRyCommon.attr'/>
  </define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of the side of the rectangle which has the smaller x-axis coordinate value in the current user coordinate system.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate> "
The y-axis coordinate of the side of the rectangle which has the smaller y-axis coordinate value in the current user coordinate system.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

width = " <length> "
The width of the rectangle.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

height = " <length> "
The height of the rectangle.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

rx = " <length> "
For rounded rectangles, the x-axis radius of the ellipse used to round off the corners of the rectangle.

A negative value is an error (see Error processing).

See the notes below about what happens if the attribute is not specified.

Animatable: yes.

ry = " <length> "
For rounded rectangles, the y-axis radius of the ellipse used to round off the corners of the rectangle.

A negative value is an error (see Error processing).

See the notes below about what happens if the attribute is not specified.

Animatable: yes.

If a properly specified value is provided for rx but not for ry, then the user agent processes the 'rect' element with the effective value for ry as equal to rx. If a properly specified value is provided for ry but not for rx, then the user agent processes the 'rect' element with the effective value for rx as equal to ry. If neither rx nor ry has a properly specified value, then the user agent processes the 'rect' element as if no rounding had been specified, resulting in square corners. If rx is greater than half of the width of the rectangle, then the user agent processes the 'rect' element with the effective value for rx as half of the width of the rectangle. If ry is greater than half of the height of the rectangle, then the user agent processes the 'rect' element with the effective value for ry as half of the height of the rectangle.

Mathematically, a 'rect' element can be mapped to an equivalent 'path' element as follows: (Note: all coordinate and length values are first converted into user space coordinates according to Units.)

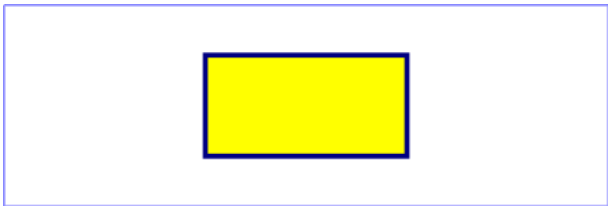
- perform an absolute moveto operation to location (*x*+*rx*,*y*), where *x* is the value of the 'rect' element's x attribute converted to user space, *rx* is the effective value of the rx attribute converted to user space and *y* is the value of the y attribute converted to user space
- perform an absolute horizontal lineto operation to location (*x*+*width*-*rx*,*y*), where *width* is the 'rect' element's width attribute converted to user space
- perform an absolute elliptical arc operation to coordinate (*x*+*width*,*y*+*ry*), where the effective values for the rx and ry attributes on the 'rect' element converted to user space are used as the *rx* and *ry* attributes on the elliptical arc command, respectively, the *x-axis-rotation* is set to zero, the *large-arc-flag* is set to zero, and the *sweep-flag* is set to one
- perform a absolute vertical *lineto* to location (*x*+*width*,*y*+*height*-*ry*), where *height* is the 'rect' element's height attribute converted to user space
- perform an absolute elliptical arc operation to coordinate (*x*+*width*-*rx*,*y*+*height*)
- perform an absolute horizontal *lineto* to location (*x*+*rx*,*y*+*height*)
- perform an absolute elliptical arc operation to coordinate (*x*,*y*+*height*-*ry*)
- perform an absolute absolute vertical *lineto* to location (*x*,*y*+*ry*)
- perform an absolute elliptical arc operation to coordinate (*x*+*rx*,*y*)

Example 09_01 shows a rectangle with sharp corners. The 'rect' element is filled with yellow and stroked with navy.

Example: 09_01.svg

```
<?xml version="1.0" standalone="no"?>
```

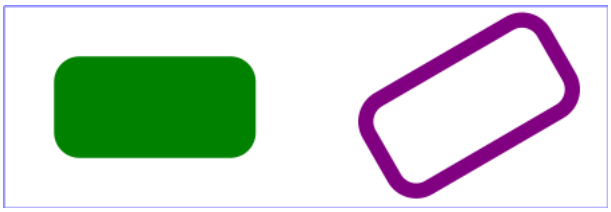
```
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example rect01 - rectangle with sharp corners</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2"/>
  <rect x="400" y="100" width="400" height="200"
        fill="yellow" stroke="navy" stroke-width="10"  />
</svg>
```



Example 09_02 shows two rounded rectangles. The rx specifies how to round the corners of the rectangles. Note that since no value has been specified for the ry attribute, it will be assigned the same value as the rx attribute.

Example: 09_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example rect02 - rounded rectangles</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2"/>
  <rect x="100" y="100" width="400" height="200" rx="50"
        fill="green" />
  <g transform="translate(700 210) rotate(-30)">
    <rect x="0" y="0" width="400" height="200" rx="50"
          fill="none" stroke="purple" stroke-width="30" />
  </g>
</svg>
```



9.3 The 'circle' element

The 'circle' element defines a circle based on a center point and a radius.

Schema: circle

```
  <define name='circle'>
    <element name='circle'>
      <ref name='circle.AT'/>
      <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='circle.AT' combine='interleave'>
    <ref name='svg.ShapeCommon.attr'/>
    <ref name='svg.CxCy.attr'/>
    <ref name='svg.R.attr'/>
  </define>
```

Attribute definitions:

cx = " <coordinate> **"**
The x-axis coordinate of the center of the circle.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

cy = " <coordinate> **"**
The y-axis coordinate of the center of the circle.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

r = " <length> **"**
The radius of the circle.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

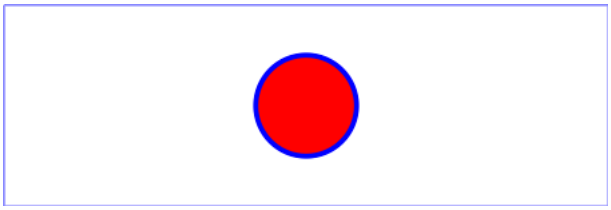
Animatable: yes.

The arc of a 'circle' element begins at the "3 o'clock" point on the radius and progresses towards the "9 o'clock" point. The starting point and direction of the arc are affected by the user space transform in the same manner as the geometry of the element.

Example circle01 consists of a 'circle' element that is filled with red and stroked with blue.

Example: 09_03.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example circle01 - circle filled with red and stroked with blue</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2"/>
  <circle cx="600" cy="200" r="100"
        fill="red" stroke="blue" stroke-width="10"  />
</svg>
```



9.4 The 'ellipse' element

The 'ellipse' element defines an ellipse which is axis-aligned with the current user coordinate system based on a center point and two radii.

Schema: ellipse

```
  <define name='ellipse'>
    <element name='ellipse'>
      <ref name='ellipse.AT'/>
      <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='ellipse.AT' combine='interleave'>
    <ref name='svg.ShapeCommon.attr'/>
    <ref name='svg.RxRyCommon.attr'/>
    <ref name='svg.CxCy.attr'/>
  </define>
```

Attribute definitions:

cx = " **<coordinate>** "
The x-axis coordinate of the center of the ellipse.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

cy = " **<coordinate>** "
The y-axis coordinate of the center of the ellipse.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

rx = " **<length>** "
The x-axis radius of the ellipse.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

ry = " **<length>** "
The y-axis radius of the ellipse.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

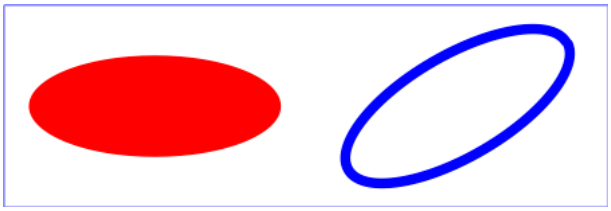
The arc of an 'ellipse' element begins at the "3 o'clock" point on the radius and progresses towards the "9 o'clock" point. The starting point and direction of the arc are affected by the user space transform in the same manner as the geometry of the element.

Example 09_04 below specifies the coordinates of the two ellipses in the user coordinate system established by the viewBox attribute on the 'svg' element and the transform attribute on the 'g' and 'ellipse' elements. Both ellipses use the default values of zero for the cx and cy attributes (the center of the ellipse). The second ellipse is rotated.

Example: 09_04.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
```

```
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example ellipse01 - examples of ellipses</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2" />
  <g transform="translate(300 200)">
    <ellipse rx="250" ry="100"
          fill="red"  />
  </g>
  <ellipse transform="translate(900 200) rotate(-30)"
        rx="250" ry="100"
        fill="none" stroke="blue" stroke-width="20"  />
</svg>
```



9.5 The 'line' element

The 'line' element defines a line segment that starts at one point and ends at another.

Schema: line

```
  <define name='line'>
    <element name='line'>
      <ref name='line.AT'/>
      <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='line.AT' combine='interleave'>
    <ref name='svg.ShapeCommon.attr'/>
    <ref name='svg.X12Y12.attr'/>
  </define>
```

Attribute definitions:

x1 = " <coordinate> "
The x-axis coordinate of the start of the line.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y1 = " <coordinate> "
The y-axis coordinate of the start of the line.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

x2 = " <coordinate> "
The x-axis coordinate of the end of the line.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y2 = " <coordinate> "
The y-axis coordinate of the end of the line.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

Mathematically, a 'line' element can be mapped to an equivalent 'path' element as follows: (Note: all coordinate and length values are first converted into user space coordinates according to Units.)

- perform an absolute moveto operation to absolute location (*x1*,*y1*), where *x1* and *y1* are the values of the 'line' element's x1 and y1 attributes converted to user space, respectively
- perform an absolute lineto operation to absolute location (*x2*,*y2*), where *x2* and *y2* are the values of the 'line' element's x2 and y2 attributes converted to user space, respectively

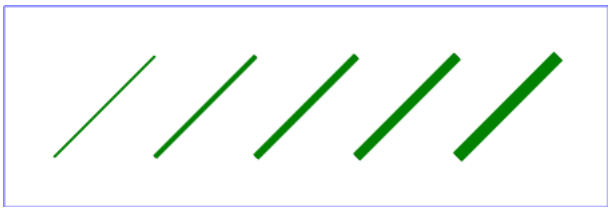
Because 'line' elements are single lines and thus are geometrically one-dimensional, they have no interior; thus, 'line' elements are never filled (see the **'fill'** property).

Example 09_05 below specifies the coordinates of the five lines in the user coordinate system established by the viewBox attribute on the 'svg' element. The lines have different thicknesses.

Example: 09_05.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example line01 - lines expressed in user coordinates</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
```

```
        fill="none" stroke="blue" stroke-width="2" />
  <g stroke="green" >
    <line x1="100" y1="300" x2="300" y2="100"
          stroke-width="5"  />
    <line x1="300" y1="300" x2="500" y2="100"
          stroke-width="10"  />
    <line x1="500" y1="300" x2="700" y2="100"
          stroke-width="15"  />
    <line x1="700" y1="300" x2="900" y2="100"
          stroke-width="20"  />
    <line x1="900" y1="300" x2="1100" y2="100"
          stroke-width="25"  />
  </g>
</svg>
```



9.6 The 'polyline' element

The 'polyline' element defines a set of connected straight line segments. Typically, 'polyline' elements define open shapes.

Schema: polyline

```
  <define name='polyline'>
    <element name='polyline'>
      <ref name='polyCommon.AT'/>
      <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='polyCommon.AT' combine='interleave'>
    <ref name='svg.ShapeCommon.attr'/>
    <attribute name='points' svg:animatable='true' svg:inheritable='false'>
      <ref name='Points.datatype'/>
    </attribute>
  </define>
```

Attribute definitions:

points = " <list-of-points> "
The points that make up the polyline. All coordinate values are in the user coordinate system.

Animatable: yes.

If an odd number of coordinates is provided, then the element is in error, with the same user agent behavior as occurs with an incorrectly specified 'path' element.

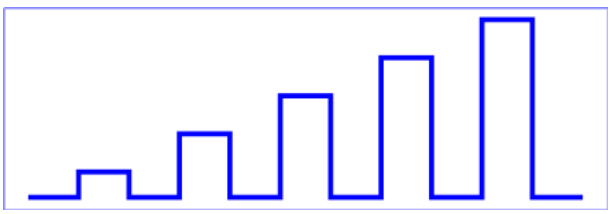
Mathematically, a 'polyline' element can be mapped to an equivalent 'path' element as follows:

- perform an absolute moveto operation to the first coordinate pair in the list of points
- for each subsequent coordinate pair, perform an absolute lineto operation to that coordinate pair.

Example 09_06 below specifies a polyline in the user coordinate system established by the viewBox attribute on the 'svg' element.

Example: 09_06.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example polyline01 - increasingly larger bars</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2" />
  <polyline fill="none" stroke="blue" stroke-width="10"
            points="50,375
                    150,375 150,325 250,325 250,375
                    350,375 350,250 450,250 450,375
                    550,375 550,175 650,175 650,375
                    750,375 750,100 850,100 850,375
                    950,375 950,25 1050,25 1050,375
                    1150,375" />
</svg>
```



9.7 The 'polygon' element

The 'polygon' element defines a closed shape consisting of a set of connected straight line segments.

Schema: polygon

```
<define name='polygon'>
  <element name='polygon'>
    <ref name='polyCommon.AT'/>
    <zeroOrMore><ref name='shapeCommon.CM'/></zeroOrMore>
  </element>
</define>
```

Attribute definitions:

points = " <list-of-points> "
The points that make up the polygon. All coordinate values are in the user coordinate system.

Animatable: yes.

If an odd number of coordinates is provided, then the element is in error, with the same user agent behavior as occurs with an incorrectly specified 'path' element.

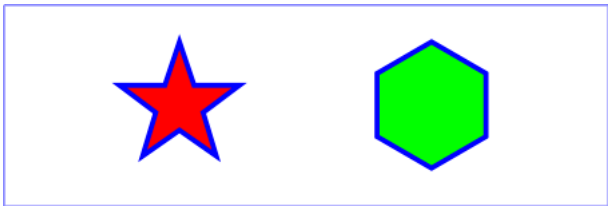
Mathematically, a 'polygon' element can be mapped to an equivalent 'path' element as follows:

- perform an absolute moveto operation to the first coordinate pair in the list of points
- for each subsequent coordinate pair, perform an absolute lineto operation to that coordinate pair
- perform a closepath command

Example 09_07 below specifies two polygons (a star and a hexagon) in the user coordinate system established by the viewBox attribute on the 'svg' element.

Example: 09_07.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example polygon01 - star and hexagon</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2" />
  <polygon fill="red" stroke="blue" stroke-width="10"
           points="350,75  379,161 469,161 397,215
                   423,301 350,250 277,301 303,215
                   231,161 321,161" />
  <polygon fill="lime" stroke="blue" stroke-width="10"
           points="850,75  958,137.5 958,262.5
                   850,325 742,262.6 742,137.5" />
</svg>
```



9.7.1 The grammar for points specifications in 'polyline' and 'polygon' elements

The following is the Backus-Naur Form (BNF) for points specifications in 'polyline' and 'polygon' elements. The following notation is used:

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

```
list-of-points:
    wsp* coordinate-pairs? wsp*
coordinate-pairs:
    coordinate-pair
    | coordinate-pair comma-wsp coordinate-pairs
coordinate-pair:
    coordinate comma-wsp coordinate
coordinate:
    number
number:
    sign? integer-constant
    | sign? floating-point-constant
comma-wsp:
    (wsp+ comma? wsp*) | (comma wsp*)
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)+
```

9.8 Shape Module

The Shape Module contains the following elements:

- path
- rect
- circle
- ellipse
- polygon
- polyline
- line

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

10 Text

Contents

10.1 Introduction

Text that is to be rendered as part of an SVG document fragment is specified using the **'text'** or **'textArea'** elements. The characters to be drawn are expressed as XML character data [XML10] inside the element.

SVG's **'text'** and **'textArea'** elements are rendered like other graphics elements. Thus, coordinate system transformations and painting features apply to text elements in the same way as they apply to shapes such as paths and rectangles.

Each **'text'** element causes a single string of text to be rendered. The **'text'** element performs no automatic line breaking or word wrapping. To achieve the effect of multiple lines of text, use one of the following methods:

- The **'textArea'** element
- The author or authoring package needs to pre-compute the line breaks and use multiple **'text'** elements (one for each line of text).
- Express the text to be rendered in another XML namespace such as XHTML [XHTML] embedded inline within a **'foreignObject'** element. (Note: the exact semantics of this approach are not completely defined at this time.)

The text strings within **'text'** elements can be rendered in a straight line. SVG supports the following international text processing features for straight line text:

- left-to-right or bidirectional text (i.e., languages which intermix right-to-left and left-to-right text, such as Arabic and Hebrew)
- when SVG fonts are used, automatic selection of the correct glyph corresponding to the current form for Arabic and Han text

The layout rules for straight line text are described in Text layout.

Because SVG text is packaged as XML character data [XML10]:

- Text data in SVG content is readily accessible to the visually impaired (see Accessibility Support)
- In many viewing scenarios, the user will be able to search for and select text strings and copy selected text strings to the system clipboard (see Text selection and clipboard operations)
- XML-compatible Web search engines will find text strings in SVG content with no additional effort over what they need to do to find text strings in other XML documents

Multi-language SVG content is possible by substituting different text strings based on the user's preferred language.

For accessibility reasons, it is recommended that text which is included in a document have appropriate semantic markup to indicate its function. See SVG accessibility guidelines for more information.

10.2 Characters and their corresponding glyphs

In XML [XML10], textual content is defined in terms of a sequence of XML characters, where each character is defined by a particular Unicode code point [UNICODE]. Fonts, on the other hand, consists of a collection of glyphs and other associated information, such as font tables. A glyph is a presentable form of one or more characters (or a part of a character in some cases). Each glyph consists of some sort of identifier (in some cases a string, in other cases a number) along with drawing instructions for rendering that particular glyph.

In many cases, there is a one-to-one mapping of Unicode characters (i.e., Unicode code points) to glyphs in a font. For example, it is common for a font designed for Latin languages (where the term *Latin* is used for European languages such as English with alphabets similar to and/or derivative to the Latin language) to contain a single glyph for each of the standard ASCII characters (i.e., A-to-Z, a-to-z, 0-to-9, plus the various punctuation characters found in ASCII). Thus, in most situations, the string "XML", which consists of three Unicode characters, would be rendered by the three glyphs corresponding to "X", "M" and "L", respectively.

In various other cases, however, there is not a strict one-to-one mapping of Unicode characters to glyphs. Some of the circumstances when the mapping is not one-to-one:

- Ligatures - For best looking typesetting, it is often desirable that particular sequences of characters are rendered as a single glyph. An example is the word "office". Many fonts will define an "ffi" ligature. When the word "office" is rendered, sometimes the user agent will render the glyph for the "ffi" ligature instead of rendering distinct glyphs (i.e., "f", "f" and "i") for each of the three characters. Thus, for ligatures, multiple Unicode characters map to a single glyph. (Note that for proper rendering of some languages, ligatures are required for certain character combinations.)
- Composite characters - In various situations, commonly used adornments such as diacritical marks will be stored once in a font as a particular glyph and then composed with one or more other glyphs to result in the desired character. For example, it is possible that a font engine might render the Ã© character by first rendering the glyph for **e** and then rendering the glyph for Â´ (the accent mark) such that the accent mark will appear over the **e**. In this situation, a single Unicode character maps to multiple glyphs.
- Glyph substitution - Some typography systems examine the nature of the textual content and utilize different glyphs in different circumstances. For example, in Arabic, the same Unicode character might render as any of four different glyphs, depending on such factors as whether the character appears at the start, the end or the middle of a sequence of cursively joined characters. Different glyphs might be used for a punctuation character depending on inline-progression-direction (e.g., horizontal vs. vertical). In these situations, a single Unicode character might map to one of several alternative glyphs.
- In some languages, particular sequences of characters will be converted into multiple glyphs such that parts of a particular character are in one glyph and the remainder of that character is in another glyph.

In many situations, the algorithms for mapping from characters to glyphs are system-dependent, resulting in the possibility that the rendering of text might be (usually slightly) different when viewed in different user environments. If the author of SVG content requires precise selection of fonts and glyphs, then the recommendation is that the necessary fonts (potentially subsetted to include only the glyphs needed for the given document) be available either as SVG fonts embedded within the SVG content or as WebFonts posted at the same Web location as the SVG content.

Throughout this chapter, the term character shall be equivalent to the definition of a character in XML [XML10].

10.3 Fonts, font tables and baselines

A font consists of a collection of glyphs together with the information (the font tables) necessary to use those glyphs to present characters on some medium. The combination of the collection of glyphs and the font tables is called the *font data*. The font tables include the information necessary to map characters to glyphs, to determine the size of glyph areas and to position the glyph area. Each font table consists of one or more font characteristics, such as the font-weight and font-style.

The geometric font characteristics are expressed in a coordinate system based on the EM box. (The EM is a relative measure of the height of the glyphs in the font; see CSS2 em square.) The box 1 EM high and 1 EM wide is called the *design space*. This space is given a geometric coordinates by sub-dividing the EM into a number of units-per-em.

Note: Units-per-em is a font characteristic. A typical value for units-per-EM is 1000 or 2048.

The coordinate space of the EM box is called the *design space coordinate system*. For scalable fonts, the curves and lines that are used to draw a glyph are represented using this coordinate system.

Note: Most often, the (0,0) point in this coordinate system is positioned on the left edge of the EM box, but not at the bottom left corner. The Y coordinate of the bottom of a roman capital letter is usually zero. And the descenders on lowercase roman letters have negative coordinate values.

SVG assumes that the font tables will provide at least three font characteristics: an ascent, a descent and a set of baseline-tables. The ascent is the distance to the top of the EM box from the (0,0) point of the font; the descent is the distance to the bottom of the EM box from the (0.0) point of the font. The baseline-table is explained below.

Note: Within an OpenType font, for horizontal writing-modes, the ascent and descent are given by the sTypoAscender and sTypoDescender entries in the OS/2 table. For vertical writing-modes, the descent (the distance, in this case from the (0,0) point to the left edge of the glyph) is normally zero because the (0,0) point is on the left edge. The ascent for vertical writing-modes is either 1 em or is specified by the ideographic top baseline value in the OpenType Base table for vertical writing-modes.

In horizontal writing-modes, the glyphs of a given script are positioned so that a particular point on each glyph, the *alignment-point* , is aligned with the alignment-points of the other glyphs in that script. The glyphs of different scripts, for example, Western, Northern Indic and Far-Eastern scripts, are typically aligned at different points on the glyph. For example, Western glyphs are aligned on the bottoms of the capital letters, northern indic glyphs are aligned at the top of a horizontal stroke near the top of the glyphs and far-eastern glyphs are aligned either at the bottom or center of the glyph. Within a script and within a line of text having a single font-size, the sequence of alignment-points defines, in the inline- progression-direction, a geometric line called a *baseline*. Western and most other alphabetic and syllabic glyphs are aligned to an "alphabetic" baseline, the northern indic glyphs are aligned to a "hanging" baseline and the far-eastern glyphs are aligned to an "ideographic" baseline.

A *baseline-table* specifies the position of one or more baselines in the design space coordinate system. The function of the baseline table is to facilitate the alignment of different scripts with respect to each other when they are mixed on the same text line. Because the desired relative alignments may depend on which script is dominant in a line (or block), there may be a different baseline table for each script. In addition, different alignment positions are needed for horizontal and vertical writing modes. Therefore, the font may have a set of baseline tables: typically, one or more for horizontal writing-modes and zero or more for vertical writing-modes.

Note: Some fonts may not have values for the baseline tables. Heuristics are suggested for approximating the baseline tables when a given font does not supply baseline tables.

SVG further assumes that for each glyph in the font data for a font, there is a width value, an alignment-baseline and an alignment-point for horizontal writing-mode. (Vertical writing-mode is not supported in SVGT1.2).

In addition to the font characteristics required above, a font may also supply substitution and positioning tables that can be used by a formatter to re-order, combine and position a sequence of glyphs to make one or more composite glyphs. The combination may be as simple as a ligature, or as complex as an indic syllable which combines, usually with some re-ordering, multiple consonants and vowel glyphs.

10.4 The 'text' element

The 'text' element defines a graphics element consisting of text. The XML [XML10] character data within the 'text' element, along with relevant attributes and properties and character-to-glyph mapping tables within the font itself, define the glyphs to be rendered. (See Characters and their corresponding glyphs.) The attributes and properties on the 'text' element indicate such things as the writing direction, font specification and painting attributes which describe how exactly to render the characters. Subsequent sections of this chapter describe the relevant text-specific attributes and properties, particular text layout and bidirectionality.

Since 'text' elements are rendered using the same rendering methods as other graphics elements, all of the same coordinate system transformations and painting features that apply to shapes such as paths and rectangles also apply to 'text' elements.

It is possible to apply a gradient to text. When this facility is applied to text then the object bounding box units are computed relative to the entire 'text' element in all cases, even when different effects are applied to different 'tspan' elements within the same 'text' element.

The 'text' element renders its first glyph (after bidirectionality reordering) at the initial current text position, which is established by the x and y attributes on the 'text' element (with possible adjustments due to the value of the **'text-anchor'** property). After the glyph(s) corresponding to the given character is (are) rendered, the current text position is updated for the next character. In the simplest case, the new current text position is the previous current text position plus the glyphs' advance value. See text layout for a description of glyph placement and glyph advance.

Schema: text

```
<define name='text'>
  <element name='text'>
    <ref name='text.AT'/>
    <zeroOrMore><ref name='svg.TextCommon.group'/></zeroOrMore>
  </element>
</define>

<define name='text.AT' combine='interleave'>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.Core.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.Editable.attr'/>
  <ref name='svg.Focus.attr'/>
  <ref name='svg.Transform.attr'/>
  <optional>
    <attribute name='x' svg:animatable='true' svg:inheritable='false'>
      <ref name='Coordinates.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='y' svg:animatable='true' svg:inheritable='false'>
      <ref name='Coordinates.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='rotate' svg:animatable='true' svg:inheritable='false'>
      <ref name='Numbers.datatype'/>
    </attribute>
  </optional>
</define>

<define name='svg.TextCommon.group' combine='choice'>
  <choice>
    <ref name='svg.Desc.group'/>
    <ref name='svg.Animate.group'/>
    <ref name='svg.Handler.group'/>
    <ref name='svg.Discard.group'/>
    <ref name='tspan'/>
    <text/>
    <element name='switch'>
      <ref name='switch.AT'/>
      <zeroOrMore><ref name='svg.TextCommon.group'/></zeroOrMore>
    </element>
    <element name='a'>
      <ref name='a.AT'/>
      <zeroOrMore><ref name='svg.TextCommon-noA.group'/></zeroOrMore>
    </element>
  </choice>
</define>
```

Attribute definitions:

x = " <coordinate>+"
If a single <coordinate> is provided, then the value represents the new absolute X coordinate for the current text position for rendering the glyphs that correspond to the first character within this element or any of its descendants.

If a comma- or space-separated list of <n> <coordinate>s is provided, then the values represent new absolute X coordinates for the current text position for rendering the glyphs corresponding to each of the first <n> characters within this element or any of its descendants.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate>+"
The corresponding list of absolute Y coordinates for the glyphs corresponding to the characters within this element. The processing rules for the 'y' attribute parallel the processing rules for the 'x' attribute.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

editable = "true | false"
Whether the given element represents can be edited by the user. If the attribute is not specified, the effect is as if a value of "false" were specified.

Animatable: no.

rotate = " <number>+"
The supplemental rotation about the alignment-point that will be applied to all of the glyphs corresponding to each character within this element.

If a comma- or space-separated list of <number>s is provided, then the first <number> represents the supplemental rotation for the glyphs corresponding to

the first character within this element or any of its descendants, the second <number> represents the supplemental rotation for the glyphs that correspond to the second character, and so on.

If more <number>s are provided than there are characters, then the extra <number>s will be ignored.

If more characters are provided than <number>s, then for each of these extra characters: (a) if an ancestor 'text' element specifies a supplemental rotation for the given character via a **rotate** attribute, then the given supplemental rotation is applied to the given character, else (b) no supplemental rotation occurs.

If the attribute is not specified: (a) if an ancestor 'text' element specifies a supplemental rotation for a given character via a **rotate** attribute, then the given supplemental rotation is applied to the given character (nearest ancestor has precedence), else (b) no supplemental rotation occurs.

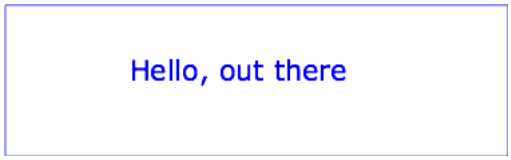
This supplemental rotation has no impact on the rules by which current text position is modified as glyphs get rendered.

Animatable: yes (non-additive, 'set' and 'animate' elements only).

Example text01 below contains the text string "Hello, out there" which will be rendered onto the canvas using the Verdana font family with the glyphs filled with the color blue.

Example: 10_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="10cm" height="3cm" viewBox="0 0 1000 300"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example text01 - 'Hello, out there' in blue</desc>
  <text x="250" y="150"
        font-family="Verdana" font-size="55" fill="blue" >
    Hello, out there
  </text>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="998" height="298"
        fill="none" stroke="blue" stroke-width="2" />
</svg>
```



10.5 The 'tspan' element

Within a 'text' or 'textArea' element, graphic and font properties can be adjusted by including a 'tspan' element.

Schema: tspan

```
  <define name='tspan'>
    <element name='tspan'>
      <ref name='tspan.AT'/>
      <zeroOrMore><ref name='svg.TextCommon.group'/></zeroOrMore>
    </element>
  </define>

  <define name='tspan.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.Focus.attr'/>
  </define>
```

The following examples show basic use of the 'tspan' element.

Example tspan01 uses a 'tspan' element to indicate that the word "not" is to use a bold font and have red fill.

Example: 10_03.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="10cm" height="3cm" viewBox="0 0 1000 300"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example tspan01 - using tspan to change visual attributes</desc>
  <g font-family="Verdana" font-size="45" >
    <text x="200" y="150" fill="blue" >
      You are
        <tspan font-weight="bold" fill="red" >not</tspan>
      a banana.
    </text>
  </g>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="998" height="298"
        fill="none" stroke="blue" stroke-width="2" />
</svg>
```

Within a **'text'** or **'textArea'** element, graphic and font properties can be adjusted by including a **'tspan'** element. Positional attributes such as x, y, dx, dy and rotate are not available on **'tspan'** in SVG Tiny 1.2.

10.6 Text layout

10.6.1 Text layout introduction

This section describes the text layout features supported by SVG. **Many of the features here go beyond the functionality provided by the elements in this profile of the SVG specification (for example, vertical text). However, they are described here to provide a single reference point for layout information.** The content here includes support for various international writing directions, such as left-to-right (e.g., Latin scripts) and bidirectional (e.g., Hebrew or Arabic) and vertical (e.g., Asian scripts). The descriptions in this section assume straight line text (i.e., text that is either strictly horizontal or vertical with respect to the current user coordinate system).

For each **'text'** and **'textArea'** element, the SVG user agent determines the current reference orientation. The reference orientation is the vector pointing towards negative infinity in Y within the current user coordinate system. (Note: in the initial coordinate system, the reference orientation is up.)

Based on the reference orientation the SVG user agent determines the current inline-progression-direction. For left-to-right text, the inline-progression-direction points 90 degrees clockwise from the reference orientation vector. For right-to-left text, the inline progression points 90 degrees counter-clockwise from the reference orientation vector. For top-to-bottom text, the inline-progression-direction points 180 degrees from the reference orientation vector.

Based on the reference orientation the SVG user agent determines the current block-progression-direction. For left-to-right and right-to-left text, the block-progression-direction points 180 degrees from the reference orientation vector because the only available horizontal writing modes are **lr-tb** and **rl-tb**. For top-to-bottom text, the block-progression-direction always points 90 degrees counter-clockwise from the reference orientation vector because the only available top-to-bottom writing mode is **tb-rl**.

In processing a given **'text'** or **'textArea'** element, the SVG user agent keeps track of the current text position. The initial current text position is established by the **x** and **y** attributes on the **'text'** or **'textArea'** element.

The current text position is adjusted after each glyph to establish a new current text position at which the next glyph shall be rendered. The adjustment to the current text position is based on the current inline-progression-direction, glyph-specific advance values corresponding to the glyph orientation of the glyph just rendered, kerning tables in the font and the current values of various attributes and properties, such as the spacing properties and any **x** and **y** attributes on **'text'** and **'textArea'** elements. If a glyph does not provide explicit advance values corresponding to the current glyph orientation, then an appropriate approximation should be used. For vertical text, a suggested approximation is the sum of the ascent and descent values for the glyph. Another suggested approximation for an advance value for both horizontal and vertical text is the size of an *em* (see units-per-em).

For each glyph to be rendered, the SVG user agent determines an appropriate alignment-point on the glyph which will be placed exactly at the current text position. The alignment-point is determined based on glyph cell metrics in the glyph itself, the current inline-progression-direction and the glyph orientation relative to the inline-progression-direction. For most uses of Latin text the alignment-point in the glyph will be the intersection of left edge of the glyph cell (or some other glyph-specific x-axis coordinate indicating a left-side origin point) with the Latin baseline of the glyph. For many cases with top-to-bottom vertical text layout, the reference point will be either a glyph-specific origin point based on the set of vertical baselines for the font or the intersection of the center of the glyph with its *top line* (see [CSS2-topline] for a definition of *top line*). If a glyph does not provide explicit origin points corresponding to the current glyph orientation, then an appropriate approximation should be used, such as the intersection of the left edge of the glyph with the appropriate horizontal baseline for the glyph or intersection of the top edge of the glyph with the appropriate vertical baseline. If baseline tables are not available, user agents should establish baseline tables that reflect common practice.

Adjustments to the current text position are either absolute position adjustments or relative position adjustments. An absolute position adjustment occurs in the following circumstances:

- At the start of a **'text'** element
- At the start of a **'textArea'** element
- For each character within a **'text'** element which has an **x** or **y** attribute value assigned to it explicitly

All other position adjustments to the current text position are relative position adjustments.

Each absolute position adjustment defines a new text chunk. Absolute position adjustments impact text layout in the following ways:

- Ligatures only occur when a set of characters which might map to a ligature are all in the same text chunk.
- Each text chunk represents a separate block of text for alignment due to **'text-anchor'** property values.
- Reordering of characters due to bidirectionality only occurs within a text chunk. Reordering does *not* happen across text chunks.

The following additional rules apply to ligature formation:

- As in [CSS2-spacing], when the resultant space between two characters is not the same as the default space, user agents should not use ligatures
- Ligature formation should not be enabled for the glyphs corresponding to characters within different DOM text nodes; thus, characters separated by markup should not use ligatures.
- As mentioned above, ligature formation should not be enabled for the glyphs corresponding to characters within different text chunks.

10.6.2 Relationship with bidirectionality

The characters in certain scripts are written from right to left. In some documents, in particular those written with the Arabic or Hebrew script, and in some mixed-language contexts, text in a single line may appear with mixed directionality. This phenomenon is called bidirectionality, or "bidi" for short.

The Unicode standard ([UNICODE], section 3.11) defines a complex algorithm for determining the proper directionality of text. The algorithm consists of an implicit part based on character properties, as well as explicit controls for embeddings and overrides. The SVG user agent applies this bidirectional algorithm when determining the layout of characters within a **'text'** or **'textArea'** element.

A more complete discussion of bidirectionality can be found in the "Cascading Style Sheets (CSS) level 2" specification [CSS2-direction].

The processing model for bidirectional text is as follows. The user agent processes the characters which are provided in logical order (i.e. the order the characters appear in the original document). The user agent determines the set of independent blocks within each of which it should apply the Unicode bidirectional algorithm. Each text chunk represents an independent block of text. While kerning or ligature processing might be font-specific, the preferred model is that kerning and ligature processing occurs between combinations of characters or glyphs after the characters have been re-ordered.

10.7 Text rendering order

The glyphs associated with the characters within **'text'** and **'textArea'** elements are rendered in the logical order of the characters in the original document, independent of any re-ordering necessary to implement bidirectionality. Thus, for text that goes right-to-left visually, the glyphs associated with the rightmost character are rendered before the glyphs associated with the other characters.

Additionally, each distinct glyph is rendered in its entirety (i.e., it is filled and stroked as specified by the **'fill'** and **'stroke'** properties) before the next glyph gets rendered.

10.8 Alignment properties

10.8.1 Text alignment properties

The **'text-anchor'** property is used to align (start-, middle- or end-alignment) a string of text relative to a given point.

The **'text-anchor'** property is applied to each individual text chunk within a given 'text' element. Each text chunk has an initial current text position, which represents the point in the user coordinate system resulting from (depending on context) application of the x and y attributes on the 'text' element assigned explicitly to the first rendered character in a text chunk.

'text-anchor'
Value: start | middle | end | inherit
Initial: start
Applies to: 'text' Element
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

Values have the following meanings:

start
The rendered characters are aligned such that the start of the text string is at the initial current text position. For Latin or Arabic, which is usually rendered horizontally, this is comparable to left alignment. For Asian text with a vertical primary text direction, this is comparable to top alignment.
middle
The rendered characters are aligned such that the middle of the text string is at the current text position.
end
The rendered characters are aligned such that the end of the text string is at the initial current text position. For Latin text in its usual orientation, this is comparable to right alignment.

10.9 Font selection properties

SVG uses the following font specification properties. Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-fonts]. Any SVG-specific notes about these properties are contained in the descriptions below.

'font-family'
Value: [[<family-name> |
<generic-family>],]* [<family-name> |
<generic-family>] | inherit
Initial: depends on user agent
Applies to: text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

This property indicates which font family is to be used to render the text, specified as a prioritized list of font family names and/or generic family names. Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-font-family]. The rules for expressing the syntax of CSS property values can be found at [CSS2-propdef].

'font-style'
Value: normal | italic | oblique | inherit
Initial: normal
Applies to: text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

This property specifies whether the text is to be rendered using a normal, italic or oblique face. Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-font-style].

'font-variant'
Value: normal | small-caps | inherit
Initial: normal
Applies to: text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

This property indicates whether the text is to be rendered using the normal glyphs for lowercase characters or using small-caps glyphs for lowercase characters. Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-font-variant].

'font-weight'
Value: normal | bold | bolder | lighter | 100 | 200 | 300
| 400 | 500 | 600 | 700 | 800 | 900 | inherit
Initial: normal
Applies to: text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

This property refers to the boldness or lightness of the glyphs used to render the text, relative to other fonts in the same font family. Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-font-weight].

'font-size'

Value: <absolute-size> | <relative-size> |

<length> | inherit
Initial: medium
Applies to: text content elements
Inherited: yes, the computed value is inherited
Percentages: N/A
Media: visual
Animatable: yes

This property refers to the size of the font from baseline to baseline when multiple lines of text are set solid in a multiline layout environment. The SVG user agent processes the **<length>** as a height value in the current user coordinate system.

Except for any additional information provided in this specification, the normative definition of the property is in [CSS2-font-size].

10.10 White space handling

SVG supports the standard XML attribute **xml:space** to specify the handling of white space characters within a given **'text'** element's character data. The SVG user agent has special processing rules associated with this attribute as described below. These are behaviors that occur subsequent to XML parsing [XML10] and any construction of a Document Object Model [DOM2].

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 using a copy of the original character data content. First, it will remove all newline characters. Then it will convert all tab characters into space characters. Then, it will strip off all leading and trailing space characters. Then, all contiguous space characters will be consolidated.
- **preserve** - When `xml:space="preserve"`, the SVG user agent will do the following using a copy of the original character data content. It will convert all newline 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 `"ab"` (three spaces between "a" and "b") will produce a larger separation between "a" and "b" than `"ab"` (one space between "a" and "b").

The following example illustrates that line indentation can be important when using `xml:space="default"`. The fragment below show two pairs of similar **'text'** elements, with both **'text'** elements using **xml:space='default'**. For these examples, there is no extra white space at the end of any of the lines (i.e., the line break occurs immediately after the last visible character).

```
[01]  <text xml:space='default'>
[02]    WS example
[03]    indented lines
[04]  </text>
[05]  <text xml:space='preserve'>WS example indented lines</text>
[06]
[07]  <text xml:space='default'>
[08]WS example
[09]non-indented lines
[10]  </text>
[11]  <text xml:space='preserve'>WS examplenon-indented lines</text>
```

The first pair of **'text'** elements above show the effect of indented character data. The attribute **xml:space='default'** in the first **'text'** element instructs the user agent to:

- convert all tabs (if any) to space characters,
- strip out all line breaks (i.e., strip out the line breaks at the end of lines [01], [02] and [03]),
- strip out all leading space characters (i.e., strip out space characters before "WS example" on line [02]),
- strip out all trailing space characters (i.e., strip out space characters before "</text>" on line [04]),
- consolidate all intermediate space characters (i.e., the space characters before "indented lines" on line [03]) into a single space character.

The second pair of **'text'** elements above show the effect of non-indented character data. The attribute **xml:space='default'** in the third **'text'** element instructs the user agent to:

- convert all tabs (if any) to space characters,
- strip out all line breaks (i.e., strip out the line breaks at the end of lines [07], [08] and [09]),
- strip out all leading space characters (there are no leading space characters in this example),
- strip out all trailing space characters (i.e., strip out space characters before "</text>" on line [10]),
- consolidate all intermediate space characters into a single space character (in this example, there are no intermediate space characters).

Note that XML parsers are required to convert the standard representations for a newline indicator (e.g., the literal two-character sequence "#xD#xA" or the stand-alone literals #xD or #xA) into the single character #xA before passing character data to the application. Thus, each newline in SVG will be represented by the single character #xA, no matter what representation for newlines might have been used in the original resource. (See XML end-of-line handling.)

Any features in the SVG language or the SVG DOM that are based on character position number are based on character position after applying the white space handling rules described here. In particular, if `xml:space="default"`, it is often the case that white space characters are removed as part of processing. Character position numbers index into the text string after the white space characters have been removed per the rules in this section.

The **xml:space** attribute is:

Animatable: no.

10.11 Text in an area

10.11.1 Intoduction to text in an area

The **'textArea'** element allows wrapping text content within a given shape or (for some profiles of the SVG language) a sequence of shapes using a simplistic wrapping algorithm.

Text wrapping via the **'textArea'** element is available as a lightweight and convenient facility for simple text wrapping when invocation of a complete box model layout engine is not required. For cases where box model layout is required, it is suggested that a **'foreignObject'** element be used which invokes a box model facility such as CSS layout.

The layout of wrapped text is user agent dependent; thus, content developers need to be aware that there might be different results, particularly with regard to where line breaks occur. The user agent may choose to implement a simplistic text wrapping algorithm as described in this specification or invoke a sophisticated layout engine (e.g., an engine that supports the CSS or XSL-FO box models). The simplistic wrapping algorithm described in this specification is upwardly compatible with the XSL-FO box model.

The minimal layout facilities required for text in an area are described in text in an area layout rules.

10.11.2 The 'textArea' element

Schema: textArea

```
<define name='textArea'>
  <element name='textArea'>
    <ref name='textArea.AT'/>
    <zeroOrMore><ref name='svg.TextCommon.group'/></zeroOrMore>
  </element>
</define>

<define name='textArea.AT' combine='interleave'>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.Core.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.Focus.attr'/>
  <ref name='svg.Transform.attr'/>
  <ref name='svg.XYWH.attr'/>
  <ref name='svg.Editable.attr'/>
</define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the text content will be placed. If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the text content will be placed. If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

width = "auto | <coordinate> "
The width of the rectangular region into which the text content will be placed. A value of "auto" indicates that the width of the rectangular region is infinite. If the attribute is not specified, the effect is as if a value of "auto" were specified.

Animatable: yes.

height = "auto | <coordinate> "
The height of the rectangular region into which the text content will be placed. A value of "auto" indicates that the width of the rectangular region is infinite. If the attribute is not specified, the effect is as if a value of "auto" were specified.

Animatable: yes.

editable = "true | false"
Whether the given element represents can be edited by the user. If the attribute is not specified, the effect is as if a value of "false" were specified.

Animatable: no.

10.11.3 The 'tBreak' element

The 'tBreak' element is an empty element that forcibly breaks the current line of text.

Schema: tBreak

```
<define name='tBreak'>
  <element name='tBreak'>
    <ref name='tBreak.AT'/>
    <empty/>
  </element>
</define>

<define name='tBreak.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
</define>
```

No Attributes

10.11.4 The 'line-increment' property

The 'line-increment' property provides limited control over the size of each line in the block-progression-direction. It is a proper subset of the CSS 'line-height' property so that user agents which have a CSS box model engine can use that engine to provide text wrapping.

'line-increment'

Value:	auto | <number> | inherit
Initial:	auto
Applies to:	text content elements
Inherited:	yes
Percentages:	none
Media:	visual
Animatable:	yes

Values for the property have the following meaning:

auto
Subsequent lines are offset from the previous line by the maximum font-size for any glyphs drawn within that line.

<number>
Subsequent lines are offset from the previous line by this amount (in user units).

10.11.5 The 'display-align' property

The 'display-align' property specifies the alignment, in the block-progression-direction, of the text content of the 'textArea' element.

'display-align'

Value:	autol before I center I after I inherit
Initial:	start
Applies to:	text content elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

Values for the property have the following meaning:

auto
For SVG, 'auto' is equivalent to 'before'.
before
The before-edge of the first line is aligned with the before-edge of the first region.
center
The lines are centered in the block-progression-direction.
after
The after-edge of the last line is aligned with the after-edge of the last region.

Below is a recommended algorithm for implementing 'display-align'.

1. A temporary delta variable, D, is initialized to the height of the textArea bounding box in the text progression direction (the distance between the co-ordinate of the shape closest to the 'after' edge and the co-ordinate of the shape closest to the 'before' edge.) A starting point is chosen as the co-ordinate of the shape closest to the 'before' edge.
2. The flowed text is laid out inside the textArea.
3. At the completion of layout, the space remaining in the before direction, B, (the distance from the first line to the co-ordinate of the shape closest to the 'before' edge) is calculated.
4. The space remaining in the after direction, A (the distance between the last line and the co-ordinate of the shape closest to the 'after' edge) is calculated.
5. A measure of the difference from exact centering, C, is calculated as the absolute value of A - B.
6. If C is less than 1px in non-transformed user space for the textArea layout terminates.
7. If C is greater than D then layout terminates. (This will not happen on the first iteration, but may on subsequent iterations if text is flowed into irregular shapes).
8. The temporary delta variable, D, is set to C.
9. A new starting point is chosen, computed as B + 0.5(A - B) and execution returns to step 2.

10.11.6 Text in an area layout rules

Text in an area layout is defined as a post processing step to the standard text layout model of SVG.

A conformant user agent can implement a simplistic layout algorithm which consists simply of inserting line breaks whenever the content explicitly specifies a line break with a 'tBreak' element or when the current line cannot fit all of the remaining glyphs. Any lines of glyphs that do not completely fit within the region(s) are not rendered.

User agents should implement a line-breaking algorithm that supports at a minimum the features described below as a post processing step to SVG's standard text layout model.

1. The text is processed in logical order to determine line breaking opportunities between characters, according to Unicode Standard Annex No. 14
2. Text layout is performed as normal, on one infinitely long line, soft hyphens are included in the line. The result is a set of positioned Glyphs.
3. The first line is positioned such that its before edge is flush against the region's before edge relative to the block-progression-direction.
4. Glyphs represent a character or characters within a word. Each glyph is associated with the word that contains its respective characters. In cases where characters from multiple words contribute to the same glyph the words are merged and all the glyphs are treated as part of the earliest word in logical order.
5. The glyphs from a word are collapsed into Glyph Groups. A Glyph Group is comprised of all consecutive glyphs from the same word. In most cases each word generates one glyph group however in some cases the interaction between BIDI and special markup may cause glyphs from one word to have glyphs from other words embedded in it.
6. Each Glyph Group has two extents calculated: it's normal extent, and it's last in text area extent. It's normal extent is the sum of the advances of all glyphs in the group except soft hyphens. The normal extent is the extent used when a Glyph Group from a later word is in the same text area. The last in text area extent includes the advance of a trailing soft hyphens but does not include the advance of trailing whitespace or combining marks. The last in text region extent is used when this glyph group is from the last word (in logical order) in this text area. (If the entire line consists of a single word which is not breakable, the User Agent may choose to force a break in the line so that at least some text will appear for the given line.)
7. Words are added to the current line in logical order. All the Glyph Groups from a word must be in the same line and all the glyphs from a Glyph Group must be in the same Text Area.
8. If line-increment is a number, then each line will be sized in the block-progression-direction to the value of line-increment. If line-increment is "auto", then the maximum font-size for any glyph in the line will determine the size of the line in the block-progression-direction. When a word is added the line increment may increase, it can never decrease from the first word. An increase in the line increment can only reduce the space available for text placement in the span. The span will have the maximum possible number of words. The position of the dominant baseline for a given line is determined by first computing the line-increment value for that line and then choosing a position for the dominant baseline using the position where the given baseline would appear for the font that will be used to render the first character and an assumed font-size equal to the line-increment value.
9. The Glyphs from the Glyph Groups are then collapsed into the text regions by placing the first selected glyph (in display order) at the start of the text area and each subsequent glyph at the location of the glyph following the preceding selected glyph (in display order).
10. The next word is selected and the next line location is determined. The next line is positioned such that its before edge is flush against the after edge of the previous line relative to the block-progression-direction. Goto Step 7.
11. Any lines which extend outside of the area(s) in the block-progression-direction are not rendered.

10.12 Editable Text Fields

SVG Tiny 1.2 allows text elements to be edited. Although simple text editing can be implemented directly in script, implementing an intuitive and well internationalized text input system which works on a variety of platforms is complex.Therefore, this functionality is provided by the SVG user agent, which has access to system text libraries. Content authors can build higher level widgets, such as form entry fields, on top of the editable text functionality.

10.12.1 The editable attribute

The text and textArea elements have an editable attribute which specifies whether the contents of the elements can be edited in place.

Schema: editable

```
<define name='svg.Editable.attr' combine='interleave'>
  <optional>
    <attribute name='editable' a:defaultValue='false' svg:animatable='false' svg:inheritable='false'>
      <ref name='Boolean.datatype'/>
    </attribute>
```

```
    </optional>
  </define>
```

editable = "true" | "false"

If set to "false" the contents of the text or textArea elements are not editable in place through the user agent. If set to "true", the user agent must provide a way for the user to edit the content of the text or textArea elements and all contained subelements which are not hidden (with visibility="hidden") or disabled (through the switch element or display="none"). The user agent must also, (if a clipboard is supported by the platform), provide a way to cut or copy the selected text from the element to the clipboard, and to paste text from the clipboard into the element . If no value is given for this attribute, the default value is "false".

Animatable: Yes.

SVG Tiny 1.2 user agents should allow for the inline WYSIWYG editing of text. However, editing with a modal editing dialog is an alternate possibility, and may be the only option on some platforms. The current editing position should be indicated, for example with a caret. SVG Tiny 1.2 user agents must also support system functions such as copy/paste and drag/drop if they are available to applications on the pletform.

To start editing , the current animated value of the editable attribute must be "true", the text or textArea elements must have focus, and it must then be activated. When editing text in a text field, all DOM 3 Text and key events [DOM3Events] events are dispatched to the user agent, which processes the events for proper handling of text entry and cancels propagation of those events while editing is taking place.

If a text or textArea element is editable, then the SVG user agent must render whitespace conforming to the SVG language defined behavior for xml:space="preserve" even if the given element has 'xml:space' with a value of "default".

For WYSIWYG editing the following functionality must be made available:

- movement to the next/previous character (in logical order), for example with Left/Right arrows
- in textArea elements, movement to the next/previous line, for example with the Down/Up keys
- movement to the beginning of the line, for example with the Home key
- movement to the end of the line, for example with the End key
- copy/cut/paste, if a clipboard is supported, for example with Copy and Paste keys

The functionality should use the normal key bindings that are used for those tasks on the given platform. For devices without keyboard access, the equivalent system input methods should be used wherever possible to provide the functionality described above.

When doing WYSIWYG editing in place, the content of the DOM nodes that are being edited should be live at all times and reflect the current state of the edited string as it being edited. When using a modal editing dialog, the content of the DOM nodes will only change once the user commits the edit (for example, by using an Enter key or clicking an 'OK' button, or whatever the platform normally does).

If an Input Method Editor (IME) is used (for example, to input Kanji text, or to input Latin text using number keys on mobile phones), the text events returned by DOM 3 correspond to the actual text entered (eg the Kanji character, or the Latin character) and not to the keyboard or mouse gestures needed to produce it (such as the sequence of kana characters, or the number of sequential presses of a numeric key).

10.13 Text selection and clipboard operations

Conforming SVG viewers which support text selection and copy/paste operations are required to support:

- user selection of text strings in SVG content
- the ability to copy selected text strings to the system clipboard

A text selection operation starts when all of the following occur:

- the user positions the pointing device or caret over a glyph that has been rendered as part of a text or textArea element, initiates a *select* operation (e.g., pressing the standard system mouse button for select operations) and then moves the current text position while continuing the *select* operation (e.g., continuing to press the standard system mouse button for select operations)
- no other visible graphics element has been painted above the glyph at the point at which the pointing device was clicked
- no links or events have been assigned to the **'text'** or **'textArea'** element (or their ancestors) associated with the given glyph.

As the text selection operation proceeds (e.g., the user continues to press the given mouse button), all associated events with other graphics elements are ignored (i.e., the text selection operation is modal) and the SVG user agent shall dynamically indicate which characters are selected by an appropriate highlighting technique, such as redrawing the selected glyphs with inverse colors. As the current text position is moved during the text selection process, the end glyph for the text selection operation is the glyph within the same **'text'** element whose glyph cell is closest to the pointer. All characters within the **'text'** element whose position within the **'text'** element is between the start of selection and end of selection shall be highlighted, regardless of position on the canvas and regardless of any graphics elements that might be above the end of selection point.

Once the text selection operation ends (e.g., the user releases the given mouse button), the selected text will stay highlighted until an event occurs which cancels text selection, such as a pointer device activation event (e.g., pressing a mouse button).

Detailed rules for determining which characters to highlight during a text selection operation are provided in Text selection implementation notes.

For systems which have system clipboards, the SVG user agent is required to provide a user interface for initiating a copy of the currently selected text to the system clipboard. It is sufficient for the SVG user agent to post the selected text string in the system's appropriate clipboard format for plain text, but it is preferable if the SVG user agent also posts a rich text alternative which captures the various font properties associated with the given text string.

For bidirectional text, the user agent must support text selection in logical order, which will result in discontinuous highlighting of glyphs due to the bidirectional reordering of characters. User agents can also optionally provide an alternative ability to select bidirectional text in visual rendering order (i.e., after bidirectional text layout algorithms have been applied), with the result that selected character data might be discontinuous logically. In this case, if the user requests that bidirectional text be copied to the clipboard, then the user agent is required to make appropriate adjustments to copy only the visually selected characters to the clipboard.

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

10.14 Text Module

The Text Module contains the following elements:

- text
- tspan
- textArea
- tBreak

11 Painting: Filling, Stroking, Colors and Paint Servers

Contents

11.1 Introduction

'path' elements, 'text' elements and basic shapes 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 solid color with opacity
- a gradient (linear or radial)

SVG uses the general notion of a **paint server**. Paint servers are specified using a IRI reference on a **'fill'** or **'stroke'** property (the IRI reference must be local as described in the IRI Reference Section). Gradients and solid colors are just specific types of paint servers.

11.2 Specifying paint

Properties **'fill'** and **'stroke'** take on a value of type **<paint>**, which is specified as follows:

<paint>:**none |**

currentColor |

<color> |

<iri> |

inherit

none
Indicates that no paint is applied.

currentColor
Indicates that painting is done using 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>
<color> is the explicit color (in the sRGB [SRGB] color space) to be used to paint the current object. SVG supports all of the syntax alternatives for <color> defined in [CSS2-color-types] including the list of recognized color keywords names.

<iri>
The <iri> is how you identify a paint server such as a gradient. The <iri> provides the ID of the paint server (e.g., a gradient or solid color) to be used to paint the current object. The IRI reference must be local as described in the IRI Reference Section. If the IRI reference 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 document is in error (see Error processing).

11.3 Fill Properties

'fill'

Value:	<paint> (See Specifying paint)
Initial:	black
Applies to:	shapes and text content elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

The **'fill'** property paints the interior of the given graphical element. The area to be painted consists of any areas inside the outline of the shape. To determine the inside of the shape, all subpaths are considered, and the interior is determined according to the rules associated with the current value of the **'fill-rule'** property. The zero-width geometric outline of a shape is included in the area to be painted.

The fill operation fills open subpaths by performing the fill operation as if an additional "closepath" command were added to the path to connect the last point of the subpath with the first point of the subpath. Thus, fill operations apply to both open subpaths within 'path' elements (i.e., subpaths without a closepath command) and 'polyline' elements.

'fill-rule'

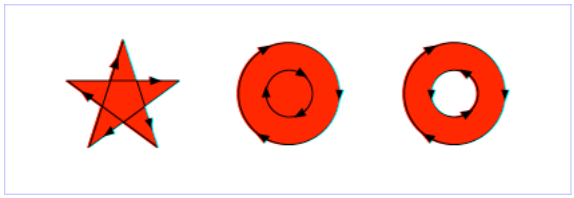
Value: nonzero | evenodd | inherit
Initial: nonzero
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

The **'fill-rule'** property indicates the algorithm which is to be used to determine what parts of the canvas are included inside the shape. For a simple, non-intersecting path, it is intuitively clear what region lies "inside"; however, for a more complex path, such as a path that intersects itself or where one subpath encloses another, the interpretation of "inside" is not so obvious.

The **'fill-rule'** property provides two options for how the inside of a shape is determined:

nonzero

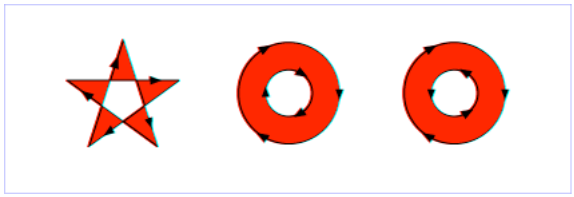
This rule determines the "insideness" of a point on the canvas by drawing a ray from that point to infinity in any direction and then examining the places where a segment of the shape crosses the ray. Starting with a count of zero, add one each time a path segment crosses the ray from left to right and subtract one each time a path segment crosses the ray from right to left. After counting the crossings, if the result is zero then the point is *outside* the path. Otherwise, it is *inside*. The following drawing illustrates the **nonzero** rule:



View this example as SVG (SVG-enabled browsers only)

evenodd

This rule determines the "insideness" of a point on the canvas by drawing a ray from that point to infinity in any direction and counting the number of path segments from the given shape that the ray crosses. If this number is odd, the point is inside; if even, the point is outside. The following drawing illustrates the **evenodd** rule:



View this example as SVG (SVG-enabled browsers only)

(Note: the above explanations do not specify what to do if a path segment coincides with or is tangent to the ray. Since any ray will do, one may simply choose a different ray that does not have such problem intersections.)

'fill-opacity'

Value: <opacity-value> | inherit
Initial: 1
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

'fill-opacity' specifies the opacity of the painting operation used to paint the interior the current object. (See Painting shapes and text.)

<opacity-value>

The opacity of the painting operation used to fill the current object. Any values outside the range 0.0 (fully transparent) to 1.0 (fully opaque) will be clamped to this range. (See Clamping values which are restricted to a particular range.)

Related property: **'stroke-opacity'**.

11.4 Stroke Properties

The following are the properties which affect how an element is stroked.

In all cases, all stroking properties which are affected by directionality, such as those having to do with dash patterns, must be rendered such that the stroke operation starts at the same point at which the graphics element starts. In particular, for 'path' elements, the start of the path is the first point of the initial "moveto" command.

For stroking properties such as dash patterns whose computations are dependent on progress along the outline of the graphics element, distance calculations are required to utilize the SVG user agent's standard Distance along a path algorithms.

When stroking is performed using a complex paint server, such as a gradient, the stroke operation must be identical to the result that would have occurred if the geometric shape defined by the geometry of the current graphics element and its associated stroking properties were converted to an equivalent 'path' element and

then filled using the given paint server.

'stroke'
Value: <paint> (See Specifying paint)
Initial: none
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

The **'stroke'** property paints along the outline of the given graphical element.

A subpath (see Paths) consisting of a single moveto is not stroked. A subpath consisting of a moveto and lineto to the same exact location or a subpath consisting of a moveto and a closepath will be stroked only if the **'stroke-linecap'** property is set to "round", producing a circle centered at the given point.

'stroke-width'
Value: <length> | inherit
Initial: 1
Applies to: shapes and text content elements
Inherited: yes
Percentages: No
Media: visual
Animatable: yes

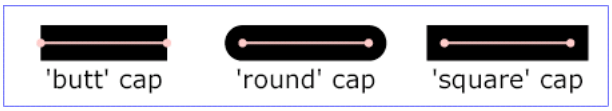
<length>
The width of the stroke on the current object.

A zero value causes no stroke to be painted. A negative value is an error (see Error processing).

'stroke-linecap'
Value: butt | round | square | inherit
Initial: butt
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

'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.



View this example as SVG (SVG-enabled browsers only)

'stroke-linejoin'
Value: miter | round | bevel | inherit
Initial: miter
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

'stroke-linejoin' specifies the shape to be used at the corners of paths or basic shapes when they are stroked.

miter
See drawing below.
round
See drawing below.
bevel
See drawing below.

View this example as SVG (SVG-enabled browsers only)

'stroke-miterlimit'
Value: <miterlimit> | inherit
Initial: 4

Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

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-width'**. When the limit is exceeded, the join is converted from a miter to a bevel.

<miterlimit>
The limit on the ratio of the miter length to the **'stroke-width'**. The value of **<miterlimit>** must be a number greater than or equal to 1. Any other value is an error (see Error processing).

The ratio of miter length (distance between the outer tip and the inner corner of the miter) to **'stroke-width'** is directly related to the angle (theta) between the segments in user space by the formula:

```
miterLength / stroke-width = 1 / sin ( theta / 2 )
```

For example, a miter limit of 1.414 converts miters to bevels for theta less than 90 degrees, a limit of 4.0 converts them for theta less than approximately 29 degrees, and a limit of 10.0 converts them for theta less than approximately 11.5 degrees.

'stroke-dasharray'
Value: none | <dasharray> | inherit
Initial: none
Applies to: shapes and text content elements
Inherited: yes
Percentages: no
Media: visual
Animatable: yes (non-additive)

'stroke-dasharray' controls the pattern of dashes and gaps used to stroke paths. **<dasharray>** contains a list of comma-separated (with optional white space) <length>s that specify the lengths of alternating dashes and gaps. If an odd number of values is provided, then the list of values is repeated to yield an even number of values. Thus, **stroke-dasharray: 5,3,2** is equivalent to **stroke-dasharray: 5,3,2,5,3,2**.

none
Indicates that no dashing is used. If stroked, the line is drawn solid.
<dasharray>
A list of comma-separated <length>'s (with optional white space). A negative <length> value is an error (see Error processing). If the sum of the <length>'s is zero, then the stroke is rendered as if a value of **none** were specified.

'stroke-dashoffset'
Value: <length> | inherit
Initial: 0
Applies to: shapes and text content elements
Inherited: yes
Percentages: no
Media: visual
Animatable: yes

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

<length>
Values can be negative.

'stroke-opacity'
Value: <opacity-value> | inherit
Initial: 1
Applies to: shapes and text content elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

'stroke-opacity' specifies the opacity of the painting operation used to stroke the current object. (See Painting shapes and text.)

<opacity-value>
The opacity of the painting operation used to stroke the current object. Any values outside the range 0.0 (fully transparent) to 1.0 (fully opaque) will be clamped to this range. (See Clamping values which are restricted to a particular range.)

Related property: **'fill-opacity'**.

11.5 Non-Scaling Stroke

Sometimes it is of interest to let the outline of an object keep its original width no matter which transforms are applied to it. E.g. in a map with a 2px wide line representing roads it is of interest to keep the roads 2px wide even when the user zooms into the map.

To achieve this SVGT1.2 introduces the **'vector-effect'** property. Future versions of the SVG language will allow for more powerful vector effects through this property but this version restricts it to the non-scaling stroke.

'vector-effect'
Value: non-scaling-stroke | none
Initial: none
Applies to: graphical elements
Inherited: no
Percentages: N/A
Media: visual

Animatable: yes

The value 'none' corresponds to not having a vector effect, i.e. the default rendering behaviour from SVG 1.1 is used which is to first fill the geometry of a shape with a specified paint, then stroke the outline with a specified paint.

The value 'non-scaling-stroke' modifies the way object is stroked. Normally stroking involves calculating stroke outline of the shape's path in current user space and filling that outline with the stroke paint (color or gradient). With non-scaling-stroke vector effect, stroke outline is calculated in the "host" coordinate space instead of user coordinate space. More precisely: a user agent establishes a host coordinate space which in SVG 1.2 is always the same as "screen coordinate space". The stroke outline is calculated in the following manner: first, the shape's path is transformed into the host coordinate space. Stroke outline is calculated in the host coordinate space. Resulting outline is transformed back to the user coordinate space. (Stroke outline is always filled with stroke paint in the current user space). The resulting visual effect of this modification is that stroke width is not dependant on the transformations of the element (including non-uniform scaling and shear transformations) and zoom level.

Note: Future versions of SVG may allow ways to control host coordinate system.

11.6 Simple alpha compositing

Graphics elements are blended into the elements already rendered on the canvas using simple alpha compositing, in which the resulting color and opacity at any given pixel on the canvas is the result of the following formulas (all color values use premultiplied alpha):

```
Er, Eg, Eb    - Element color value
     Ea            - Element alpha value
     Cr, Cg, Cb    - Canvas color value (before blending)
     Ca            - Canvas alpha value (before blending)
     Cr', Cg', Cb' - Canvas color value (after blending)
     Ca'           - Canvas alpha value (after blending)
     Ca' = 1 - (1 - Ea) * (1 - Ca)
     Cr' = (1 - Ea) * Cr + Er
     Cg' = (1 - Ea) * Cg + Eg
     Cb' = (1 - Ea) * Cb + Eb
```

The following rendering properties, which provide information about the color space in which to perform the compositing operations, apply to compositing operation:

- **'color-rendering'**

11.7 The 'viewport-fill' Property

SVG enables the author to specify a paint server which will be used to fill the viewport of any element that creates a viewport, such as the root svg element. The referenced paint server is restricted to being a solid color.

The 'viewport-fill' property defines the paint used to fill the viewport created by a particular element.

'viewport-fill'
Value: <paint>
Initial: none
Applies to: viewport-creating elements
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

It is an error to reference a fill that is not a solid color operation. Below is an example of 'viewport-fill'.

Example: 11_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.2"
      viewport-fill="red" baseProfile="tiny">
      <desc>
      Everything here has a red background.
      The rectangle is not filled, so the red background will show through
      </desc>
      <rect x="20" y="20" width="100" height="100" fill="none" stroke="black"/>
</svg>
```

The filling of the viewport is the first operation in the rendering chain of an element. Therefore:

- The viewport fill operation happens before filling and stroking.
- The viewport fill operation occurs before compositing, and thus is part of the input to the compositing operations.
- The viewport fill operation renders into the element's conceptual offscreen buffer, and thus opacity applies as usual.
- Viewport fill is not affected by the 'fill' or 'fill-opacity' properties.

11.8 The 'viewport-fill-opacity' Property

The 'viewport-fill-opacity' property specifies the opacity of the 'viewport-fill'.

'viewport-fill-opacity'
Value: <float>
Initial: 1.0
Applies to: viewport-creating elements
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

11.9 The 'overflow' Property

'overflow'
Value: visible
Initial: see prose
Applies to: 'animation' element
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

The **'overflow'** property only applies to the 'animation' element in SVGT1.2. It's only possible value is 'visible' which implies that clipping shall not occur.

The initial value for **'overflow'** as defined in [CSS2-overflow] is 'visible'; however, SVG traditionally overrides this initial value and set the **'overflow'** property to the value 'hidden'. 'hidden' implies clipping but this is not a supported value for SVGT1.2 which means that clipping never needs to be performed.

11.10 Controlling visibility

SVG uses two properties, **'display'** and **'visibility'**, to control the visibility of graphical elements or (in the case of the **'display'** property) container elements.

The differences between the two properties are as follows:

- When applied to a container element, setting **'display'** to **none** causes the container and all of its children to be invisible; thus, it acts on groups of elements as a group. **'visibility'**, however, only applies to individual graphics elements. Setting **'visibility'** to **hidden** on a 'g' will make its children invisible as long as the children do not specify their own **'visibility'** properties as **visible**. Note that **'visibility'** is *not* an inheritable property.
- When the **'display'** property is set to **none**, then the given element does not become part of the rendering tree. With **'visibility'** set to **hidden**, however, processing occurs as if the element were part of the rendering tree and still taking up space, but not actually rendered onto the canvas. This distinction has implications for the 'tspan' element, event processing, and for bounding box calculations. If **'display'** is set to **none** on a 'tspan' then the text string is ignored for the purposes of text layout; however, if **'visibility'** is set to **hidden**, the text string is used for text layout (i.e., it takes up space) even though it is not rendered on the canvas. Regarding events, if **'display'** is set to **none**, the element receives no events; however, if **'visibility'** is set to **hidden**, the element might still receive events, depending on the value of property **'pointer-events'**. The geometry of a graphics element with **'display'** set to **none** is not included in bounding box calculations; however, even if **'visibility'** is to **hidden**, the geometry of the graphics element still contributes to bounding box calculations.

'display'
Value: inline | block | list-item |

run-in | compact | marker |

table | inline-table | table-row-group | table-header-group |

table-footer-group | table-row | table-column-group | table-column |

table-cell | table-caption | none | inherit
Initial: inline
Applies to: 'svg', 'g', 'switch', 'a', 'foreignObject', graphics elements (including the 'text' element) and text sub-elements (i.e., 'tspan' and 'a')
Inherited: no
Percentages: N/A
Media: all
Animatable: yes

A value of **display: none** indicates that the given element and its children shall not be rendered directly (i.e., those elements are not present in the rendering tree). Any value other than **none** or **inherit** indicates that the given element shall be rendered by the SVG user agent.

The **'display'** property only affects the direct rendering of a given element, whereas it does not prevent elements from being referenced by other elements.

Elements with **display: none** do not take up space in text layout operations, do not receive events, and do not contribute to bounding box calculations.

Except for any additional information provided in this specification, the normative definition is the CSS2 definition of the 'display' property.

'visibility'
Value: visible | hidden | collapse | inherit
Initial: visible
Applies to: graphics elements (including the 'text' element) and text sub-elements (i.e., 'tspan' and 'a')
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

visible
The current graphics element is visible.
hidden or collapse
The current graphics element is invisible (i.e., nothing is painted on the canvas).

Note that if the **'visibility'** property is set to **hidden** on a 'tspan' element, then the text is invisible but still takes up space in text layout calculations.

Depending on the value of property **'pointer-events'**, graphics elements which have their **'visibility'** property set to **hidden** still might receive events.

Except for any additional information provided in this specification, the normative definition is the CSS2 definition of the 'visibility' property.

11.11 Rendering hints

11.11.1 The **'color-rendering'** property

The creator of SVG content might want to provide a hint to the implementation about how to make speed vs. quality tradeoffs as it performs color interpolation and compositing. The **'color-rendering'** property provides a hint to the SVG user agent about how to optimize its color interpolation and compositing operations.

'color-rendering'
Value: auto | optimizeSpeed | optimizeQuality | inherit
Initial: auto
Applies to: container elements, graphics elements and 'animateColor'
Inherited: yes

Percentages: N/A
Media: visual
Animatable: yes

auto
Indicates that the user agent shall make appropriate tradeoffs to balance speed and quality, but quality shall be given more importance than speed.
optimizeSpeed
Indicates that the user agent shall emphasize rendering speed over quality. For RGB display devices, this option will sometimes cause the user agent to perform color interpolation and compositing in the device RGB color space.
optimizeQuality
Indicates that the user agent shall emphasize quality over rendering speed.

11.11.2 The **'shape-rendering'** property

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

'shape-rendering'
Value: auto | optimizeSpeed | crispEdges | geometricPrecision | inherit
Initial: auto
Applies to: shapes
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

auto
Indicates that the user agent shall make appropriate tradeoffs to balance speed, crisp edges and geometric precision, but with geometric precision given more importance than speed and crisp edges.
optimizeSpeed
Indicates that the user agent shall emphasize rendering speed over geometric precision and crisp edges. This option will sometimes cause the user agent to turn off shape anti-aliasing.
crispEdges
Indicates that the user agent shall 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 and curves 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.
geometricPrecision
Indicates that the user agent shall emphasize geometric precision over speed and crisp edges.

11.11.3 The **'text-rendering'** property

The creator of SVG content 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: auto | optimizeSpeed | optimizeLegibility | geometricPrecision | inherit
Initial: auto
Applies to: 'text' elements
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

auto
Indicates that the user agent shall make appropriate tradeoffs to balance speed, legibility and geometric precision, but with legibility given more importance than speed and geometric precision.
optimizeSpeed
Indicates that the user agent shall emphasize rendering speed over legibility and geometric precision. This option will sometimes cause the user agent to turn off text anti-aliasing.
optimizeLegibility
Indicates that the user agent shall 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.
geometricPrecision
Indicates that the user agent shall 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.

11.11.4 The **'image-rendering'** property

The creator of SVG content 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: auto | optimizeSpeed | optimizeQuality | inherit
Initial: auto
Applies to: images
Inherited: yes
Percentages: N/A
Media: visual
Animatable: yes

auto
Indicates that the user agent shall make appropriate tradeoffs to balance speed and quality, but quality shall be given more importance than speed. The user agent shall employ a resampling algorithm at least as good as nearest neighbor resampling, but bilinear resampling is strongly preferred. For Conforming High-Quality SVG Viewers, the user agent shall employ a resampling algorithm at least as good as bilinear resampling.
optimizeQuality
Indicates that the user agent shall emphasize quality over rendering speed. The user agent shall employ a resampling algorithm at least as good as bilinear resampling.

optimizeSpeed
Indicates that the user agent shall emphasize rendering speed over quality. The user agent should use a resampling algorithm which achieves the goal of fast rendering, with the requirement that the resampling algorithm shall be at least as good as nearest neighbor resampling. If performance goals can be achieved with higher quality algorithms, then the user agent should use the higher quality algorithms instead of nearest neighbor resampling.

In all cases, resampling must be done in a truecolor (e.g., 24-bit) color space even if the original data and/or the target device is indexed color.

11.12 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 graphics element individually, never at the container element (e.g., a 'g') level. Thus, for the following SVG, even though the gradient fill is specified on the 'g' , the gradient is simply inherited through the 'g' element down into each rectangle, each of which is rendered such that its interior is painted with the gradient.

Example Inheritance

Example: 11_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="7cm" height="2cm" viewBox="0 0 700 200"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Gradients apply to leaf nodes
  </desc>
  <g>
    <defs>
      <linearGradient id="MyGradient" gradientUnits="objectBoundingBox">
        <stop offset="0%" stop-color="#F60" />
        <stop offset="100%" stop-color="#FF6" />
      </linearGradient>
    </defs>
    <rect x="1" y="1" width="698" height="198"
          fill="none" stroke="blue" stroke-width="2" />
    <g fill="url(#MyGradient)" >
      <rect x="100" y="50" width="200" height="100"/>
      <rect x="400" y="50" width="200" height="100"/>
   </g>
  </g>
</svg>
```

Any painting properties defined in terms of the object's bounding box use the bounding box of the graphics element to which the operation applies. Note that text elements are defined such that any painting operations defined in terms of the object's bounding box use the bounding box of the entire 'text' element. (See the discussion of object bounding box units and text elements.)

11.13 Object and group opacity: the 'opacity' property

There are several opacity properties within SVG:

- Fill opacity
- Stroke opacity
- Gradient stop opacity
- Object/group opacity (described here)

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 offscreen image, the object/group opacity setting specifies how to blend the offscreen image into the current background.

Object/group opacity can, if applied to container elements, be a resource intensive operation. Therefore this version of SVG restricts this property to only apply to the image element.

'opacity'
Value: <opacity-value> | inherit
Initial: 1
Applies to: image element
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

<opacity-value>
The uniform opacity setting to be applied across an entire object. Any values outside the range 0.0 (fully transparent) to 1.0 (fully opaque) will be clamped to this range. (See Clamping values which are restricted to a particular range.)

11.14 Color

All SVG colors are specified in the sRGB color space (see [SRGB]). SVG user agents must conform to the color behavior requirements specified in the color units section and the minimal gamma correction rules defined in the CSS2 specification.

11.14.1 The solidColor Element

The solidColor element is a paint server that provides a single color with opacity. It can be referenced like the other paint servers (i.e. gradients).

Schema: solidColor

```
<define name='solidColor'>
  <element name='solidColor'>
    <ref name='solidColor.AT'/>
```

```
      <ref name='solidColor.CM'/>
    </element>
  </define>

  <define name='solidColor.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='svg.Animate.group'/>
        <ref name='svg.Handler.group'/>
        <ref name='svg.Discard.group'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='solidColor.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
  </define>

  <define name='svg.Properties.attr' combine='interleave'>
    <optional>
      <attribute name='solid-color' svg:animatable='true' svg:inheritable='false'>
        <choice>
          <value>inherit</value>
          <ref name='SVGColor.datatype'/>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='solid-opacity' svg:animatable='true' svg:inheritable='false'>
        <choice>
          <value>inherit</value>
          <ref name='OpacityValue.datatype'/>
        </choice>
      </attribute>
    </optional>
  </define>
```

The solid-color property indicates what color to use for this solidColor element. The keyword currentColor can be specified in the same manner as within a <paint> specification for the fill and stroke properties.

solid-color

Value: currentColor | <color> | inherit
Initial: black
Applies to: solidColor elements
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

The solid-opacity property defines the opacity of a given solid color.

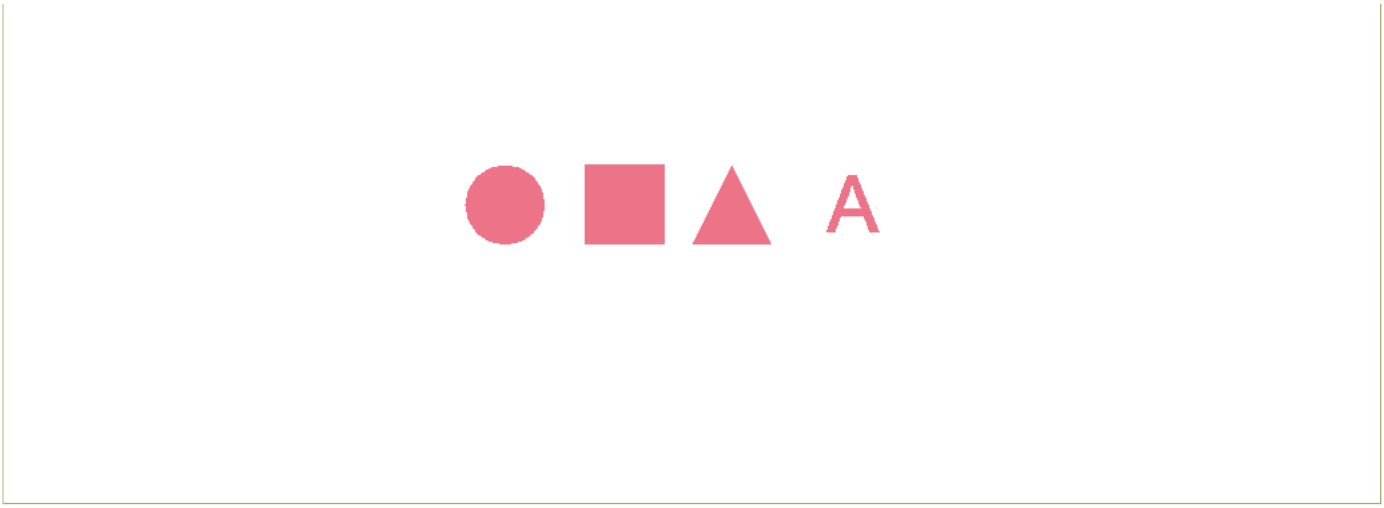
solid-opacity

Value: <opacity-value> | inherit
Initial: 1
Applies to: solidColor elements
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

Below is an example of the solidColor element:

Example: solidcolor.svg

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="tiny" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" id="svg-root" widt
    <title>&lt;solidColor&gt; Example</title>

    <defs>
        <solidColor id="solidCrimson" solid-color="crimson" solid-opacity="0.7"/>
    </defs>

    <g>
        <circle transform="translate(100, 150)" fill="url(#solidCrimson)n" r="30" />
        <rect fill="url(#solidCrimson)" transform="translate(190, 150)" x="-30" y="-30" width="60" height="60" />
        <path fill="url(#solidCrimson)" transform="translate(270, 150)"  d="M 0 -30 L 30 30 L -30 30 Z" />
        <text fill="url(#solidCrimson)" transform="translate(340, 150)" y="21" font-weight="bold" font-size="60">A</text>
    </g>
</svg>
```



11.14.2 The 'color' property

The **'color'** property is used to provide a potential indirect value (**currentColor**) for the **'fill'**, **'stroke'**, **'stop-color'** properties.

'color'

Value:	<color> I inherit
Initial:	depends on user agent
Applies to:	elements to which properties **'fill'**, **'stroke'**, **'stop-color'** apply
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

Except for any additional information provided in this specification, the normative definition of the property is in [CSS2].

11.15 Paint Servers

With SVG, you can fill (i.e., paint the interior) or stroke (i.e., paint the outline) of shapes and text using one of the following:

- color (using <color> or the **'solidColor'** element)
- gradients (linear or radial)

SVG uses the general notion of a **paint server**. Gradients and patterns are just specific types of built-in paint servers. The **'solidColor'** element is another built-in paint server, described in Color.

Paint servers are referenced using an IRI reference on a **'fill'** or **'stroke'** property.

11.16 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**.

Once defined, gradients are then referenced using **'fill'** or **'stroke'** properties on a given graphics element to indicate that the given element shall be filled or stroked with the referenced gradient.

11.16.1 Linear gradients

Linear gradients are defined by a **'linearGradient'** element.

Schema: linearGradient

```
<define name='linearGradient'>
  <element name='linearGradient'>
    <ref name='linearGradient.AT'/>
    <ref name='GradientCommon.CM'/>
  </element>
</define>

<define name='linearGradient.AT' combine='interleave'>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.Core.attr'/>
  <ref name='svg.X12Y12.attr'/>
</define>

<define name='svg.GradientCommon.attr' combine='interleave'>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.Core.attr'/>
  <optional>
    <attribute name='gradientUnits' svg:animatable='true' svg:inheritable='false'>
      <choice>
        <value>userSpaceOnUse</value>
        <value>objectBoundingBox</value>
      </choice>
    </attribute>
  </optional>
</define>

<define name='GradientCommon.CM'>
```

```
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='svg.Animate.group'/>
        <ref name='svg.Discard.group'/>
        <ref name='stop'/>
      </choice>
    </zeroOrMore>
  </define>
```

Attribute definitions:

`gradientUnits = "`*userSpaceOnUse* | *objectBoundingBox*`"`
Defines the coordinate system for attributes x1, y1, x2, y2.

If **gradientUnits="userSpaceOnUse"**, **x1, y1, x2, y2** represent values in the coordinate system that results from taking the current user coordinate system in place at the time when the gradient element is referenced (i.e., the user coordinate system for the element referencing the gradient element via a **'fill'** or **'stroke'** property).

If **gradientUnits="objectBoundingBox"**, the user coordinate system for attributes **x1, y1, x2, y2** is established using the bounding box of the element to which the gradient is applied (see Object bounding box units).

When **gradientUnits="objectBoundingBox"** the stripes of the linear gradient are perpendicular to the gradient vector in object bounding box space (i.e., the abstract coordinate system where (0,0) is at the top/left of the object bounding box and (1,0) is at the top/right of the object bounding box). When the object's bounding box is not square, the stripes that are conceptually perpendicular to the gradient vector within object bounding box space will render non-perpendicular relative to the gradient vector in user space due to application of the non-uniform scaling transformation from bounding box space to user space.

If attribute **gradientUnits** is not specified, then the effect is as if a value of **objectBoundingBox** were specified.

Animatable: yes.
`x1 = "` **<coordinate>** **`"`**
x1, y1, x2, y2 define a *gradient vector* for the linear gradient. This *gradient vector* provides starting and ending points onto which the gradient stops are mapped. The values of **x1, y1, x2, y2** can only be numbers.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
`y1 = "` **<coordinate>** **`"`**
See x1.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
`x2 = "` **<coordinate>** **`"`**
See x1.

If the attribute is not specified, the effect is as if a value of "1" were specified.

Animatable: yes.
`y2 = "` **<coordinate>** **`"`**
See x1.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

If **x1** = **x2** and **y1** = **y2**, then the area to be painted will be painted as a single color using the color and opacity of the last gradient stop.

If the gradient starts or ends inside the bounds of the *target rectangle* the terminal colors of the gradient are used to fill the remainder of the target region.

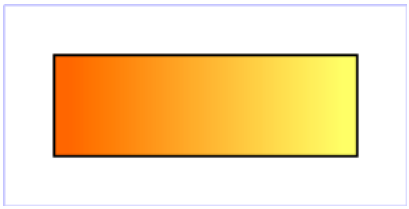
Properties inherit into the **'linearGradient'** element from its ancestors; properties do *not* inherit from the element referencing the **'linearGradient'** element.

'linearGradient' elements are never rendered directly; their only usage is as something that can be referenced using the **'fill'** and **'stroke'** properties. The **'display'** property does not apply to the **'linearGradient'** element; thus, **'linearGradient'** elements are not directly rendered even if the **'display'** property is set to a value other than **none**, and **'linearGradient'** elements are available for referencing even when the **'display'** property on the **'linearGradient'** element or any of its ancestors is set to **none**.

Example lingrad01 shows how to fill a rectangle by referencing a linear gradient paint server.

Example: 13_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="8cm" height="4cm" viewBox="0 0 800 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example lingrad01 - fill a rectangle using a
          linear gradient paint server</desc>
  <g>
    <defs>
      <linearGradient id="MyGradient">
        <stop offset="5%" stop-color="#F60" />
        <stop offset="95%" stop-color="#FF6" />
      </linearGradient>
    </defs>
    <!-- Outline the drawing area in blue -->
    <rect fill="none" stroke="blue"
          x="1" y="1" width="798" height="398"/>
    <!-- The rectangle is filled using a linear gradient paint server -->
    <rect fill="url(#MyGradient)" stroke="black" stroke-width="5"
          x="100" y="100" width="600" height="200"/>
  </g>
</svg>
```



11.16.2 Radial gradients

Radial gradients are defined by a **'radialGradient'** element.

Schema: radialGradient

```
<define name='radialGradient'>
  <element name='radialGradient'>
    <ref name='radialGradient.AT'/>
    <ref name='GradientCommon.CM'/>
  </element>
</define>

<define name='radialGradient.AT' combine='interleave'>
  <ref name='svg.Properties.attr'/>
  <ref name='svg.Core.attr'/>
  <ref name='svg.CxCy.attr'/>
  <ref name='svg.R.attr'/>
</define>
```

Attribute definitions:

`gradientUnits` = "***userSpaceOnUse*** | ***objectBoundingBox***"
Defines the coordinate system for attributes cx, cy, r.

If **gradientUnits="userSpaceOnUse"**, **cx, cy, r** represent values in the coordinate system that results from taking the current user coordinate system in place at the time when the gradient element is referenced (i.e., the user coordinate system for the element referencing the gradient element via a **'fill'** or **'stroke'** property).

If **gradientUnits="objectBoundingBox"**, the user coordinate system for attributes **cx, cy, r** is established using the bounding box of the element to which the gradient is applied (see Object bounding box units).

When **gradientUnits="objectBoundingBox"** the rings of the radial gradient are circular with respect to the object bounding box space (i.e., the abstract coordinate system where (0,0) is at the top/left of the object bounding box and (1,1) is at the bottom/right of the object bounding box). When the object's bounding box is not square, the rings that are conceptually circular within object bounding box space will render as elliptical due to application of the non-uniform scaling transformation from bounding box space to user space.

If attribute **gradientUnits** is not specified, then the effect is as if a value of **objectBoundingBox** were specified.

Animatable: yes.

`cx` = " **<coordinate>** "
cx, cy, r define the largest (i.e., outermost) circle for the radial gradient. The gradient will be drawn such that the "1" gradient stop is mapped to the perimeter of this largest (i.e., outermost) circle.

If the attribute is not specified, the effect is as if a value of "0.5" were specified.

The gradient will be drawn such that the "0" gradient stop is mapped to (cx, cy).

Animatable: yes.

`cy` = " **<coordinate>** "
See cx.

If the attribute is not specified, the effect is as if a value of "0.5" were specified.

Animatable: yes.

`r` = " **<length>** "
See **cx**.

A negative value is an error (see Error processing). A value of zero will cause the area to be painted as a single color using the color and opacity of the last gradient stop.

If the attribute is not specified, the effect is as if a value of "0.5" were specified.

Animatable: yes.

If the gradient starts or ends inside the bounds of the object(s) being painted by the gradient the terminal colors of the gradient are used to fill the remainder of the target region.

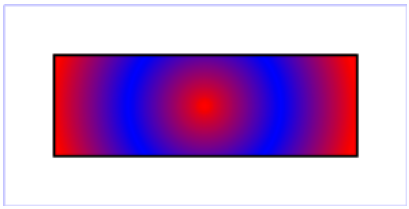
Properties inherit into the **'radialGradient'** element from its ancestors; properties do *not* inherit from the element referencing the **'radialGradient'** element.

'radialGradient' elements are never rendered directly; their only usage is as something that can be referenced using the **'fill'** and **'stroke'** properties. The **'display'** property does not apply to the **'radialGradient'** element; thus, **'radialGradient'** elements are not directly rendered even if the **'display'** property is set to a value other than **none**, and **'radialGradient'** elements are available for referencing even when the **'display'** property on the **'radialGradient'** element or any of its ancestors is set to **none**.

Example radgrad01 shows how to fill a rectangle by referencing a radial gradient paint server.

Example: 13_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="8cm" height="4cm" viewBox="0 0 800 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example radgrad01 - fill a rectangle by referencing a
          radial gradient paint server</desc>
  <g>
    <defs>
      <radialGradient id="MyGradient" gradientUnits="userSpaceOnUse"
                      cx="400" cy="200" r="300" fx="400" fy="200">
        <stop offset="0%" stop-color="red" />
        <stop offset="50%" stop-color="blue" />
        <stop offset="100%" stop-color="red" />
      </radialGradient>
    </defs>
    <!-- Outline the drawing area in blue -->
    <rect fill="none" stroke="blue"
          x="1" y="1" width="798" height="398"/>
    <!-- The rectangle is filled using a radial gradient paint server -->
    <rect fill="url(#MyGradient)" stroke="black" stroke-width="5"
          x="100" y="100" width="600" height="200"/>
  </g>
</svg>
```



11.16.3 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.

Schema: stop

```
  <define name='stop'>
    <element name='stop'>
      <ref name='stop.AT'/>
      <ref name='stop.CM'/>
    </element>
  </define>

  <define name='stop.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='svg.Animate.group'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='stop.AT' combine='interleave'>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Core.attr'/>
    <attribute name='offset' svg:animatable='true' svg:inheritable='false'>
      <ref name='GradientOffset.datatype'/>
    </attribute>
  </define>
```

Attribute definitions:

`offset` = " ***<number>*** "
The **offset** attribute is a <number> (usually ranging from 0 to 1) which indicates where the gradient stop is placed. For linear gradients, the **offset** attribute represents a location along the *gradient vector*. For radial gradients, it represents a relative distance from (cx,cy) to the edge of the outermost/largest circle.

Animatable: yes.

The **'stop-color'** property indicates what color to use at that gradient stop. The keyword **currentColor** can be specified in the same manner as within a <paint> specification for the **'fill'** and **'stroke'** properties.

'stop-color'

Value:	currentColor | <color> | inherit
Initial:	black
Applies to:	'stop' elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The **'stop-opacity'** property defines the opacity of a given gradient stop.

'stop-opacity'

Value: <opacity-value> | inherit
Initial: 1
Applies to: 'stop' elements
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

Some notes on gradients:

- Gradient offset values less than 0 are rounded up to 0. Gradient offset values greater than 1 are rounded down to 1.
- It is necessary that at least two stops defined to have a gradient effect. If no stops are defined, then painting shall 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 is required to 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. In particular:

```
<stop offset="0" stop-color="white"/>
<stop offset=".2" stop-color="red"/>
<stop offset=".2" stop-color="blue"/>
<stop offset="1" stop-color="black"/>
```

will have approximately the same effect as:

```
<stop offset="0" stop-color="white"/>
<stop offset=".1999999999" stop-color="red"/>
<stop offset=".2" stop-color="blue"/>
<stop offset="1" stop-color="black"/>
```

which is a gradient that goes smoothly from white to red, then abruptly shifts from red to blue, and then goes smoothly from blue to black.

11.17 Paint Attribute Module

The Paint Attribute Module contains the following attributes:

- color
- fill
- fill-rule
- stroke
- stroke-dash-array
- stroke-dash-offset
- stroke-linecap
- stroke-linejoin
- stroke-miterlimit
- stroke-width

11.18 Opacity Attribute Module

The Opacity Attribute Module contains the following attributes:

- fill-opacity
- stroke-opacity

11.19 Graphics Attribute Module

The Graphics Attribute Module contains the following attributes:

- display
- image-rendering
- pointer-events
- shape-rendering
- text-rendering
- visibility

11.20 Gradient Module

The Gradient Module contains the following elements:

- linearGradient
- radialGradient
- stop

11.21 Solid Color Module

The Solid Color Module contains the following element:

- solidColor

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

12 Multimedia

Contents

12.1 Media elements

SVG supports media elements similar to the SMIL Media Elements. Media elements define their own timelines within their time container. All SVG Media elements support the SVG Timing attributes and run time synchronization.

The following elements are media elements:

- **audio**
- **video**
- **animation**

12.2 The 'audio' element

The **audio** element specifies an audio file which is to be rendered to provide synchronized audio. The usual SMIL timing features are used to start and stop the audio at the appropriate times. An xlink:href is used to link to the audio content. No visual representation is produced. However, content authors can if desired create graphical representations of control panels to start, stop, pause, rewind, or adjust the volume of audio content.

Schema: audio

```
<define name='audio'>
  <element name='audio'>
    <ref name='audio.AT'/>
    <ref name='audio.CM'/>
  </element>
</define>

<define name='audio.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.XLinkEmbed.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.External.attr'/>
  <ref name='svg.AnimateTimingNoFillNoMinMax.attr'/>
  <ref name='svg.AnimateSync.attr'/>
  <ref name='svg.Media.attr'/>
  <ref name='svg.ContentType.attr'/>
</define>

<define name='audio.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
      <ref name='svg.Animate.group'/>
      <ref name='svg.Handler.group'/>
      <ref name='svg.Discard.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

`xlink:href` = " <iri> "
An IRI reference.

Animatable: yes.
`type` = "<media/type>"
The audio format.

Animatable: no.

This element also supports all **Run-time synchronization attributes** and SVG Timing attributes.

The following example illustrates the use of the audio element. When the button is pushed, the audio file is played three times.

Example: media01.svg

```
<svg width="100%" height="100%" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">

  <desc>SVG audio example</desc>

  <audio xlink:href="ouch.ogg" volume="0.7" type="application/ogg"
       begin="mybutton.click" repeatCount="3"/>

  <g id="mybutton">
    <rect width="150" height="50" x="20" y="20" rx="10"
      fill="#ffd" stroke="#933" stroke-width="5"/>
    <text x="95" y="55" text-anchor="middle" font-size="30"
      fill="#933">Press Me</text>
  </g>

  <rect x="0" y="0" width="190" height="90" fill="none" stroke="#777"/>

</svg>
```

When rendered, this looks as follows:

This specification does not mandate support for any particular audio format. Content can check for a particular audio codec with the requiredFormats test attribute.

12.3 The 'video' element

The **video** element specifies a video file which is to be rendered to provide synchronized video. It should not point to other types of media. The usual SMIL timing features are used to start and stop the video at the appropriate times. An **xlink:href** is used to link to the video content. It is assumed that the video content may also include an audio stream, since this is the usual way that video content is produced, and thus the audio is controlled by the **video** element's media attributes.

The **video** element produces a rendered result, and thus has **width**, **height**, **x** and **y** attributes.

A **'video'** element establishes a new viewport for the referenced file as described in Establishing a new viewport. Therefore the **video** element supports the **'viewport-fill'** and **'viewport-fill-opacity'** properties. The bounds for the new viewport are defined by attributes **x**, **y**, **width** and **height**. The placement and scaling of the referenced video is controlled by the **preserveAspectRatio** attribute on the **'video'** element.

The value of the '**viewBox**' attribute to use when evaluating the **preserveAspectRatio** attribute is defined by the referenced content. For content that clearly identifies a viewBox that value should be used. For most video content the bounds of the video should be used (i.e. the **'video'** element has an implicit 'viewBox' of "0 0 video-width video-height"). Where no value is readily available the **preserveAspectRatio** attribute is ignored.

Schema: video

```
<define name='video'>
  <element name='video'>
    <ref name='video.AT'/>
    <ref name='video.CM'/>
  </element>
</define>

<define name='video.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.Media.attr'/>
  <ref name='svg.XLinkEmbed.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.External.attr'/>
  <ref name='svg.AnimateTimingNoFillNoMinMax.attr'/>
  <ref name='svg.AnimateSync.attr'/>
  <ref name='svg.Focus.attr'/>
  <ref name='svg.Transform.attr'/>
  <ref name='svg.XYWH.attr'/>
  <ref name='svg.PAR.attr'/>
  <ref name='svg.ContentType.attr'/>
  <optional>
    <attribute name='transformBehavior' a:defaultValue='geometric' svg:animatable='no' svg:inheritable='false'>
      <choice>
        <value>geometric</value>
        <value>pinned</value>
      </choice>
    </attribute>
  </optional>
  <optional>
    <attribute name='overlay' a:defaultValue='none' svg:animatable='no' svg:inheritable='false'>
      <choice>
        <value>none</value>
        <value>top</value>
      </choice>
    </attribute>
  </optional>
</define>

<define name='video.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
      <ref name='svg.Animate.group'/>
      <ref name='svg.Handler.group'/>
      <ref name='svg.Discard.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the video is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the video is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

width = " <length> "
The width of the rectangular region into which the video is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

height = " <length> "

The height of the rectangular region into which the video is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

xlink:href = " <iri> "

An IRI reference.

Animatable: yes.

preserveAspectRatio = [defer] <align> [<meetOrSlice>]

Indicates whether or not to force uniform scaling. (See The preserveAspectRatio attribute for the syntax of <align> and the interpretation of this attribute.)

Animatable: yes.

type = "<media/type>"

The video format.

Animatable: no.

transformBehavior = "geometric | pinned"

See the transformBehavior attribute.

overlay = "top | none"

See the overlay attribute.

This element also supports all **Run-time synchronization attributes** and SVG Timing attributes.

The following example illustrates the use of the **'video'** element. The video content is partially obscured by other graphics elements. This example shows the **video** element being rendered into an offscreen buffer and then transformed and composited in the normal way, so that it behaves like any other graphical primitive such as an image or a rectangle. Like this, the video element may be scaled, rotated, skewed, displayed at various sizes, and animated.

Example: media02.svg

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="420" height="340" viewBox="0 0 420 340">
  <desc>SVG 1.2 video example</desc>
    <g>
    <circle cx="0" cy="0" r="170" fill="#da4" fill-opacity="0.3"/>
    <video xlink:href="noonoo.avi" volume=".8" type="video/x-msvideo"
         width="320" height="240" x="50" y="50" repeatCount="indefinite"/>
    <circle cx="420" cy="340" r="170" fill="#927" fill-opacity="0.3"/>
    <rect x="1" y="1" width="418" height="338" fill="none"
       stroke="#777" stroke-width="1"/>
    </g>
</svg>
```

Show this example of the **video** element (requires an SVG 1.2 viewer and support for a Windows AVI using Motion JPEG. This is a 3.7M video file).

When rendered, this looks as follows:

This specification does not mandate support for any particular video format. Content can check for a particular video codec with the requiredFormats test attribute.

The content creator should be aware that video is a feature that may not be available on all target devices. In order to create interoperable content the content creator should provide a fall-back alternative by using the **'switch'** element. The following feature string is defined for checking for video support: http://www.w3.org/TR/SVG12/feature#Video. Video may not be completely supported on a resource limited device. SVGT 1.2 introduces more granular video rendering control to provide reproducible results in all environments. This control is documented in the two following sections.

12.3.1 Restricting the transformation of the 'video' element

Transforming video is an expensive operation that should be used with caution, especially on content targeted for mobile devices. Using transformed video may also lead to non-interoperable content since not all devices will support this feature. To give the content creator control over video transformation SVGT 1.2 introduces the **transformBehavior** attribute and a corresponding feature string: http://www.w3.org/TR/SVG12/feature#TransformedVideo. A viewer supporting video transformation must treat the **'video'** element like any other element regarding transformations. A viewer not supporting video transformation treats the video as a point (given by x and y attributes). The width and height attributes are ignored if present and instead the width and height (in device pixels) are taken from the media itself. The video is displayed with its center aligned with the origin of the local coordinate system.

A content creator can use the **transformBehavior** attribute to explicitly choose the transform behavior on a viewer supporting transformed video. This might be of interest to increase the performance of content targeting restricted devices.

Attribute definition:

`transformBehavior` = "geometric" | "pinned"
Define whether a video is transformed/resampled (in essence treated as a geometric rectangle) or pinned/unresampled (ie, treated as a pin point for a non-geometric blit region).

The attribute value can be either of the following:

"geometric"
the media is treated as a geometric rectangle in the local coordinate system, defined by x y width and height attributes. the media is resampled to fill the rectangle and is subject to transformation. This is the default **transformBehavior**
"pinned"
The media is treated as a point, defined by x and y attributes. This point is transformed to the nearest actual device pixel. Video at the native resolution given by the media is then painted on a region whose center is the pin point and whose width and height are defined by the media. The pixels are aligned to the device pixel grid and no resampling will be done.

Animatable: no.

12.3.2 Restricting compositing of the 'video' element

For the same reasons as restricting transformations the content creator might need to restrict the compositing of video with other elements. Not all devices support compositing of the video element with other content. In that case it is necessary to render the video on top of all other svg content. SVGT1.2 therefore introduces the **overlay** attribute and a corresponding feature string: http://www.w3.org/TR/SVG12/feature#ComposedVideo. A viewer supporting compositing of video must render the video element according to the svg painters model, graphical elements might be rendered on top of the video. A viewer not supporting video compositing will always render the video on top of all other svg elements.

A content creator can use the `overlay` attribute to explicitly choose the compositing behavior on a viewer supporting composited video. This may increase the performance of content that is targeted at restricted devices.

Attribute definition:

`overlay` = "top" | "none"
Define whether a 'video' is rendered according to the svg painters model or if it should be positioned on top of all other svg elements.

The attribute value can be either of the following:

"top"
When a 'video' element has the **overlay** set to 'top' it is not composited to the background as usual. Instead a temporary video canvas is set aside and drawn last in the whole document's compositing process.
"none"
The 'video' is rendered according to the svg painters model. This is the default **overlay** behavior.

Animatable: no.

If multiple 'video' elements have overlay='top', the drawing order between those 'video' elements is document order.

12.3.3 Example

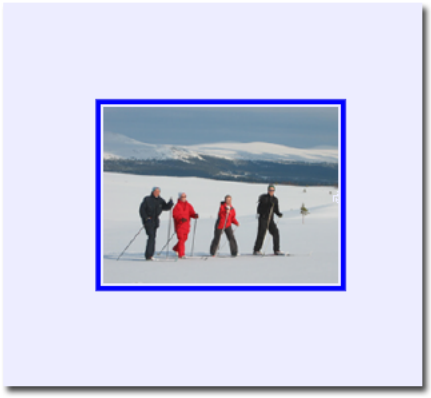
The following example illustrates the use of the TransformedVideo feature string. A switch element is wrapped around two groups, the first group will render a scaled and rotated video sequence on a viewer supporting video transformations while the second group will render the untransformed video on viewers that doesn't support video transformations.

Example: media04.svg

```
<svg version="1.2" baseProfile="tiny" xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%" viewBox="0 0 400 300">
     <desc>Example of switching on the TransformedVideo feature string</desc>
     <switch>

       <!-- Transformed video group -->
       <g requiredFeatures="http://www.w3.org/TR/SVG12/feature#TransformedVideo"
            transform="translate(-21,-34) scale(1.24) rotate(-30)">
         <rect x="6" y="166" width="184" height="140" fill="none" stroke="blue"
                  stroke-width="4" />
         <video xlink:href="ski.avi" volume=".8" type="video/x-msvideo"
                   x="10" y="170" width="176" height="132"/>
       </g>

       <!-- Untransformed video group -->
       <g>
         <rect x="6" y="166" width="184" height="140" fill="none" stroke="blue"
                  stroke-width="4"/>
         <video xlink:href="ski.avi" volume=".8" type="video/x-msvideo"
                   x="10" y="170"/>
       </g>
     </switch>
</svg>
```

Screenshot of the example above rendered on two viewers, the first one supporting transformed video, the second one not:

SVGT 1.2 viewer supporting transformed video.



SVGT 1.2 viewer not supporting transformed video.

12.4 The 'animation' element

The **animation** elements specifies an SVG document or an SVG document fragment providing synchronized animated vector graphics. Like **video**, the **animation** element is a graphical object with size determined by its x, y, width and height attributes.

While somewhat similar (both reference svg content), the behaviour of the **'animation'** element is different from the **'use'** element. The **'use'** element makes a conceptual reference to the original content while the **'animation'** element makes a duplicate of the original content. Furthermore the **'animation'** element supports timing and synchronisation attributes which allows multiple animations to run with independent timelines in the same svg document.

Besides what is described about the **'animation'** element in this section, important restrictions for **'animation'** can be found in the Reference Section.

An **'animation'** element referencing an SVG Document establishes a new viewport for the referenced file as described in Establishing a new viewport. The bounds for the new viewport are defined by attributes **'x'**, **'y'**, **'width'** and **'height'**. The placement and scaling of the referenced image are controlled by the **'preserveAspectRatio'** attribute on the **'animation'** element.

An **'animation'** element referencing an SVG document fragment do not establish a new viewport, and the **'width'** and **'height'** attributes must be ignored.

When an **'animation'** element references an SVG Document the **'preserveAspectRatio'** attribute on the root element in the referenced SVG Document is ignored (in the same manner as the x, y, width and height attributes are ignored). Instead, the **'preserveAspectRatio'** attribute on the referencing **'animation'** element defines how the SVG content is fitted into the viewport. The same rule applies for the **'viewport-fill'** and **'viewport-fill-opacity'** properties.

The value of the **'viewBox'** attribute to use when evaluating the **'preserveAspectRatio'** attribute is defined by the referenced content. For **'animation'** elements that references an SVG Document the 'viewBox' value of that document is used. When no value is available the **'preserveAspectRatio'** attribute is ignored, and only the translate due to the **'x'** & **'y'** attributes of the viewport is used to display the content.

The usual SMIL timing features are used to start and stop the animation at the appropriate times. The xlink:href attribute refers to the vector graphics content that is to be animated. The restrictions listed in the Reference section need to be met, otherwise the document is in error.

In the case where the animation element references an external SVG document the external document represents a separate document which generates its own parse tree and document object model (if the resource is XML). Thus, there is no inheritance of properties into the referenced animation. For details, see Externally referenced documents.

In the case where the animation element references an SVG document fragment, the conceptual processing model is that the entire document containing the fragment is processed as a complete SVG document instance (coordinate systems transformations and stylesheets are applied; scripts are executed; etc.). The document is conceptually rendered to an invisible (offscreen) canvas except for the referenced fragment which is rendered directly to the screen canvas. When the animation element references a document fragment the 'preserveAspectRatio', 'width' and 'height' attributes have no meaning. The rules for applying 'x' and 'y' are the same set of rules that apply for the **'use'** element.

The animation element supports the 'overflow' property.

Schema: animation

```
<define name='animation'>
  <element name='animation'>
    <ref name='animation.AT'/>
    <ref name='animation.CM'/>
  </element>
</define>

<define name='animation.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.Media.attr'/>
  <ref name='svg.Conditional.attr'/>
  <ref name='svg.External.attr'/>
  <ref name='svg.XLinkEmbed.attr'/>
  <ref name='svg.Focus.attr'/>
  <ref name='svg.AnimateTimingNoMinMax.attr'/>
  <ref name='svg.AnimateSync.attr'/>
  <ref name='svg.XYWH.attr'/>
  <ref name='svg.PAR.attr'/>
  <ref name='svg.Overflow.attr'/>
</define>

<define name='animation.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
      <ref name='svg.Animate.group'/>
      <ref name='svg.Discard.group'/>
      <ref name='svg.Handler.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the animation is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the animation is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.
width = " <length> "
The width of the rectangular region into which the animation is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.
height = " <length> "
The height of the rectangular region into which the animation is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.
xlink:href = " <iri> "
An IRI reference.

Animatable: yes.
preserveAspectRatio = [defer] <align> [<meetOrSlice>]
Indicates whether or not to force uniform scaling. (See The preserveAspectRatio attribute for the syntax of <align> and the interpretation of this attribute.)

Animatable: yes.

This element also supports all **Run-time synchronization attributes** and SVG Timing attributes.

Example: media03.svg

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     version="1.2" baseProfile="tiny">

  <desc>Example of the animation element.</desc>

  <defs>
    <rect id="movieClip" x="-50" y="-50" width="100" height="100"
          fill="purple" stroke="black">

      <animate attributeName="fill" values="purple;yellow;purple"
               keyTimes="0;.5;1" begin="0" dur="2" fill="freeze"/>

   </rect>
  </defs>


  <animation begin="1" dur="3" repeatCount="1.5" fill="freeze"
             x="100 y="100" xlink:href="#movieClip"/>

  <animation begin="2" x="100" y="300" xlink:href="#movieClip"/>

</svg>
```

12.5 The audio-level property

The **audio-level** property can be applied to the **'audio'**, **'video'** and **'animation'** elements described above, plus container elements such as the **'g'** element.

audio-level
Value: <float>
Initial: 1
Applies to: **audio**, **video**, **animation** and container elements
Inherited: no
Percentages: N/A
Media: visual, audible
Animatable: yes

The **audio-level** property specifies a value between 0 and 1 that is used to calculate the volume of a particular element. Values above 1 and below 0 are clipped.

An element's volume is the product of its **audio-level** property and the element volume of its parent. The exception to this rule is the root element, where the element volume is only the value of its **audio-level** property. Therefore, element volume is calculated in a similar way to opacity.

If the audio-level of an element to a value less than 1.0, all children elements will be quieter. It is not possible to increase the volume on an element.

The output signal level is calculated using the logarithmic scale described below (where vol is the value of the element volume):

```
dB change in signal level = 20 * log10(vol)
```

If the element has an element volume of 0, then the output signal will be inaudible. If the element has an element volume of 1, then the output signal will be at the system volume level. Neither the **audio-level** property or the element volume override the system volume setting.

12.6 Attributes for run-time synchronization

SVG supports the four attributes listed below from SMIL 2 to control run-time synchronization of different time containers. In SVGT1.2 these attributes apply to <video>, <audio> and <animation>.

- syncBehavior: (canSlip | locked | independent | default)
- syncBehaviorDefault: (canSlip | locked | independent | inherit)
- syncTolerance: (Clock-value | default)
- syncToleranceDefault: (Clock-value | inherit)

The normative definition for these attributes is the SMIL 2 specification.

Animatable: no

12.7 Audio Module

The Audio Module contains the following element:

- audio

12.8 Video Module

The Video Module contains the following element:

- video

12.9 Animation Module

The Animation Module contains the following element:

- animation

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

13 Interactivity

Contents

13.1 Introduction

SVG content can be interactive (i.e., responsive to user-initiated events) by utilizing the following features in the SVG language:

- User-initiated actions such as a key-press can cause animations to start or stop, scripts to execute or listeners elements to trigger handler elements.
- The user can initiate hyperlinks to new Web pages (see Links out of SVG content: the 'a' element) by actions such as a stylus click on a particular graphics element.
- In many cases, depending on the value of the **zoomAndPan** attribute on the **'svg'** element and on the characteristics of the user agent, users are able to zoom into and pan around SVG content.

This chapter describes:

- information about events, including under which circumstances events are triggered
- how to indicate whether a given document can be zoomed and panned
- Element focus and navigation.

Related information can be found in other chapters:

- hyperlinks are discussed in Links
- scripting and handler elements are discussed in Scripting
- animation is discussed in Animation

13.2 Complete list of supported events

The following aspects of SVG are affected by events:

- The SVG uDOM enables a script to register event listeners so that the script can be invoked when a given event occurs.
- The ev:event attribute on the 'handler' element specifies for which event the handler should trigger.
- SVG's SMIL animation elements and media elements can be defined to begin or end based on events.

The following table lists all of the events which are recognized and supported in SVG.

Event Identifier {event-namespace, event-localname}	Description	DOM3 event category	Animation event name
{"http://www.w3.org/2001/xml-	Occurs when an element receives focus.	UIEvent	focusin

events", "DOMFocusIn"} SVG 1.2 alias: {"http://www.w3.org/2001/xml-events", "focusin"} (see Notes below).			
{"http://www.w3.org/2001/xml-events", "DOMFocusOut"} SVG 1.2 alias: {"http://www.w3.org/2001/xml-events", "focusout"} (see Notes below).	Occurs when an element loses focus.	UIEvent	focusout
{"http://www.w3.org/2001/xml-events", "DOMActivate"} SVG 1.2 alias: {"http://www.w3.org/2001/xml-events", "activate"} (see Notes below).	Occurs when an element is activated, for instance, thru a mouse click or a keypress. A numerical argument is provided to give an indication of the type of activation that occurs: 1 for a simple activation (e.g. a simple click or Enter), 2 for hyperactivation (for instance a double click or Shift Enter).	UIEvent	activate
{"http://www.w3.org/2001/xml-events", "click"}	Occurs when the pointing device button is clicked over an element. A click is defined as a mousedown and mouseup over the same screen location. The sequence of these events is: `mousedown`, `mouseup`, `click`. If multiple clicks occur at the same screen location, the sequence repeats with the `detail` attribute incrementing with each repetition.	MouseEvent	click
{"http://www.w3.org/2001/xml-events", "mousedown"}	Occurs when the pointing device button is pressed over an element.	MouseEvent	mousedown
{"http://www.w3.org/2001/xml-events", "mouseup"}	Occurs when the pointing device button is released over an element.	MouseEvent	mouseup
{"http://www.w3.org/2001/xml-events", "mouseover"}	Occurs when the pointing device is moved onto an element.	MouseEvent	mouseover
{"http://www.w3.org/2001/xml-events", "mousemove"}	Occurs when the pointing device is moved while it is over an element.	MouseEvent	mousemove
{"http://www.w3.org/2001/xml-events", "mouseout"}	Occurs when the pointing device is moved away from an element.	MouseEvent	mouseout
{"http://www.w3.org/2001/xml-events", "textInput"}	One or more characters have been entered.	TextEvent	none
{"http://www.w3.org/2001/xml-events", "keydown"}	A key is pressed down. (The normative definition of this event is the description in the DOM3 Events specification.)	KeyboardEvent	none
{"http://www.w3.org/2001/xml-events", "keyup"}	A key is released. (The normative definition of this event is the description in the DOM3 Events specification.)	KeyboardEvent	none
{"http://www.w3.org/2001/xml-events", "load"} Deprecated backwards-compatibility alias: {"http://www.w3.org/2001/xml-events", "SVGLoad"} (see Notes below).	The event is triggered at the point at which the user agent has fully parsed the element and its descendants and is ready to act appropriately upon that element, such as being ready to render the element to the target device. Referenced external resources that are required must be loaded, parsed and ready to render before the event is triggered. Optional external resources are not required to be ready for the event to be triggered.	HTMLEvent	load
{"http://www.w3.org/2001/xml-events", "resize"} Deprecated backwards-compatibility alias: SVGResize (see Notes below).	Occurs when a document view is being resized. This event is only applicable to outermost 'svg' elements and is dispatched after the resize operation has taken place. The target of the event is the 'svg' element.	HTMLEvent	resize
{"http://www.w3.org/2001/xml-events", "scroll"} Deprecated backwards-compatibility alias: SVGScroll (see Notes below).	Occurs when a document view is being shifted along the X or Y or both axis, either through a direct user interaction or any change on the **'currentTranslate'** property available on SVGSVGElement interface. This event is only applicable to outermost 'svg' elements and is dispatched after the shift modification has taken place. The target of the event is the 'svg' element.	HTMLEvent	scroll
{"http://www.w3.org/2001/xml-events", "zoom"} Deprecated backwards-compatibility alias: {"http://www.w3.org/2001/xml-events", "SVGZoom"} (see Notes below).	Occurs when the zoom level of a document view is being changed, either through a direct user interaction or any change to the **'currentScale'** property available on SVGSVGElement interface. This event is only applicable to outermost 'svg' elements and is dispatched after the zoom level modification has taken place. The target of the event is the 'svg' element.	DOM3's SVG Events	zoom
{"http://www.w3.org/2001/xml-events", "beginEvent"}	Occurs when an animation element begins. For details, see the description of Interface TimeEvent in the SMIL Animation specification.	DOM3's Timing Events	begin
{"http://www.w3.org/2001/xml-events", "endEvent"}.	Occurs when an animation element ends. For details, see the description of Interface TimeEvent in the SMIL Animation specification.	DOM3's Timing Events	end
{"http://www.w3.org/2001/xml-events", "repeatEvent"}	Occurs when an animation element repeats. It is raised each time the element repeats, after the first iteration. For details, see the description of Interface TimeEvent in the SMIL Animation specification.	DOM3's Timing Events	repeat
{"http://www.w3.org/2001/xml-	Occurs when a rotational input device has been activated.	UIEvent	none

events", "wheel"}			
{"http://www.w3.org/2000/svg", "connectionData"}	Occurs when data arrives.	none	none
{"http://www.w3.org/2000/svg", "preload"}	A load operation has begun.	none	none
{"http://www.w3.org/2000/svg", "loadprogress"}	Progress has occurred in loading a given resource.	none	none
{"http://www.w3.org/2000/svg", "postload"}	A load operation has completed.	none	none

Notes:

- SVG 1.1 names are all assumed to be in the "http://www.w3.org/2001/xml-events" namespace. This allows SVG 1.1 content (which did not have a notion of namespaced events) to be upwardly compatible with SVG 1.2 (which adds a notion of namespaced events). Therefore, the SVG 1.1 "SVGZoom" event becomes the {"http://www.w3.org/2001/xml-events", "SVGZoom"} event in SVG 1.2.
- In order to unify event names with other W3C languages, SVG 1.2 deprecates some of the SVG 1.1 event names. (The term "deprecate" in this case means that user agents which are compatible with both SVG 1.1 and SVG 1.2 must support both the old deprecated event names and the new event names, but an SVG 1.2-only user agent should support only the new event names. Content creators who are making content that targets SVG 1.2 should use the new event names, not the deprecated event names.) Here are the specifics:
  - "SVGLoad" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "load"}
  - "SVGResize" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "resize"}
  - "SVGScroll" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "scroll"}
  - "SVGZoom" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "zoom"}

  In cases where the event name from SVG 1.1 differs from the DOM3 event name, but the SVG event name is more user-friendly (e.g., "focusin" is more user friendly than "DOMFocusIn"), the SVG 1.1 event name is not deprecated but instead retained as an alias for the DOM3 event name. Therefore:

  - {"http://www.w3.org/2001/xml-events", "focusin"} is equivalent to {"http://www.w3.org/2001/xml-events", "DOMFocusIn"}
  - {"http://www.w3.org/2001/xml-events", "focusout"} is equivalent to {"http://www.w3.org/2001/xml-events", "DOMFocusOut"}
  - {"http://www.w3.org/2001/xml-events", "activate"} is equivalent to {"http://www.w3.org/2001/xml-events", "DOMActivate"}
- In almost all cases, the event identifier for SMIL timing attributes (e.g., the 'begin' and 'end' attributes on animation elements) is the localname for the event, where the assumed namespace for the event is "http://www.w3.org/2001/xml-events". For example, to indicate that an animation should begin with a "click" event, then the begin attribute would be specified as `begin="click"`. Some exceptions to this rule is necessary for SVG 1.1 backwards compatibility and to make it easier to integrate SVG with other W3C languages. The following events use a different string than their localname as their event identifier for SMIL timing attributes:
  - The DOMFocusIn event uses the string "focusin"
  - The DOMFocusOut event uses the string "focusout"
  - The DOMActivate event uses the string "activate"

  For backwards-compatibility and to make it easier to integrate SVG with other W3C languages, with SVG 1.2 SMIL timing attributes accept the following aliases for event identifiers:
  - "load" and "SVGLoad" are aliases for each other
  - "resize" and "SVGResize" are aliases for each other
  - "scroll" and "SVGScroll" are aliases for each other
  - "zoom" and "SVGZoom" are aliases for each other

A **SVGLoad** event is dispatched only to the element to which the event applies; it is not dispatched to its ancestors. For example, if an 'image' element and its parent 'g' element both have event listeners for **SVGLoad** events, when the 'image' element has been loaded, only its event listener will be invoked. (The 'g' element's event listener will indeed get invoked, but the invocation will happen when the 'g' itself has been loaded.)

Details on the parameters passed to event listeners for the event types from DOM2 can be found in the DOM2 specification. For other event types, the parameters passed to event listeners are described elsewhere in this specification.

13.3 User interface events

On user agents which support interactivity, it is common for authors to define SVG documents such that they are responsive to user interface events. Among the set of possible user events are pointer events, keyboard events, and document events.

In response to user interface (UI) events, the author might start an animation, perform a hyperlink to another Web page, highlight part of the document (e.g. change the color of the graphics elements which are under the pointer), initiate a "roll-over" (e.g., cause some previously hidden graphics elements to appear near the pointer) or launch a script which communicates with a remote database.

For all UI event-related features defined as part of the SVG language via XML Events or animation, the event model corresponds to the *event bubbling* model described in DOM2 [DOM2-EVBUBBLE]. The *event capture* model from DOM2 is not supported.

13.4 Pointer events

User interface events that occur because of user actions performed on a pointer device are called pointer events.

Many systems support pointer devices such as a mouse or trackball. On systems which use a mouse, pointer events consist of actions such as mouse movements and mouse clicks. On systems with a different pointer device, the pointing device often emulates the behavior of the mouse by providing a mechanism for equivalent user actions, such as a button to press which is equivalent to a mouse click.

For each pointer event, the SVG user agent determines the *target element* of a given pointer event. The target element is the topmost graphics element whose relevant graphical content is under the pointer at the time of the event. (See property **'pointer-events'** for a description of how to determine whether an element's relevant graphical content is under the pointer, and thus in which circumstances that graphic element can be the target element for a pointer event.) When an element is not displayed (i.e., when the **'display'** property on that element or one of its ancestors has a value of **none**), that element cannot be the target of pointer events.

The event is either initially dispatched to the *target element*, to one of the target element's ancestors, or not dispatched, depending on the following:

- If there are no graphics elements whose relevant graphics content is under the pointer (i.e., there is no target element), the event is not dispatched.
- Otherwise, if the target element has an appropriate event handler for the given event, the event is dispatched to the target element.
- Otherwise, each ancestor of the target element (starting with its immediate parent) is checked to see if it has an appropriate event handler. If an ancestor is found with an appropriate event handler, the event is dispatched to that ancestor element.
- Otherwise, the event is discarded.

When event bubbling [DOM2-EVBUBBLE] is active, bubbling occurs up to all direct ancestors of the target element. Descendant elements receive events before their ancestors. Thus, if a 'path' element is a child of a 'g' element and they both have event listeners for **click** events, then the event will be dispatched to the 'path' element before the 'g' element.

After an event is initially dispatched to a particular element, the event will be passed to the appropriate event handlers (if any) for that element's ancestors (in the case of event bubbling) for further processing.

13.5 Processing order for user interface events

The processing order for user interface events is as follows:

- Event handlers assigned to the topmost graphics element under the pointer (and the various ancestors of that graphics element via potential event bubbling) receive the event first. If none of the activation event handlers take an explicit action to prevent further processing of the given event, then the event is passed on for:
- (For those user interface events which invoke hyperlinks, such as mouse clicks in some user agents) Link processing. If a hyperlink is invoked in response to a user interface event, the hyperlink typically will disable further activation event processing (e.g., often, the link will define a hyperlink to another Web page). If link processing does not disable further processing of the given event, then the event is passed on for:
- (For those user interface events which can select text, such as mouse clicks and drags on 'text' elements) Text selection processing. When a text selection operation occurs, typically it will disable further processing of the given event; otherwise, the event is passed on for:
- Document-wide event processing, such as user agent facilities to allow zooming and panning of an SVG document fragment.

Various elements in SVG create shadow content which can be the target of user interface events. The following is a list of elements that can create shadow content which can be the target of user interface events:

- The use element
- The animation element

For these situations, user interface events within the shadow content participate in the processing of user interface events in the same manner as if the shadow content were part of the main document. In other words, if shadow content contains a graphics element that renders above other content at the current pointer location, then it represents the topmost graphics element and will receive the pointer events before other elements. In this case, the user interface events bubble up through the target's ancestors, and then across the document border into the referencing element, and then through the ancestors of the referencing element. This process continues as necessary if there are multiple levels of nested shadow trees.

13.6 The **'pointer-events'** property

In different circumstances, authors may want to control under what circumstances particular graphic elements can become the target of pointer events. For example, the author might want a given element to receive pointer events only when the pointer is over the stroked perimeter of a given shape. In other cases, the author might want a given element to ignore pointer events under all circumstances so that graphical elements underneath the given element will become the target of pointer events.

For example, suppose a circle with a **'stroke'** of **red** (i.e., the outline is solid red) and a **'fill'** of **none** (i.e., the interior is not painted) is rendered directly on top of a rectangle with a **'fill'** of **blue**. The author might want the circle to be the target of pointer events only when the pointer is over the perimeter of the circle. When the pointer is over the interior of the circle, the author might want the underlying rectangle to be the target element of pointer events.

The **'pointer-events'** property specifies under what circumstances a given graphics element can be the target element for a pointer event. It affects the circumstances under which the following are processed:

- user interface events such as key press.
- hyperlinks (see Links out of SVG content: the 'a' element)

'pointer-events'

Value:	visiblePainted | visibleFill | visibleStroke | visible | painted | fill | stroke | all | none | inherit
Initial:	visiblePainted
Applies to:	graphics elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

visiblePainted
The given element can be the target element for pointer events when the **'visibility'** property is set to **visible** and when the pointer is over a "painted" area. The pointer is over a painted area if it is over the interior (i.e., fill) of the element and the **'fill'** property is set to a value other than 'none' or it is over the perimeter (i.e., stroke) of the element and the **'stroke'** property is set to a value other than 'none'.

visibleFill
The given element can be the target element for pointer events when the **'visibility'** property is set to **visible** and when the pointer is over the interior (i.e., fill) of the element. The value of the **'fill'** property does not effect event processing.

visibleStroke
The given element can be the target element for pointer events when the **'visibility'** property is set to **visible** and when the pointer is over the perimeter (i.e., stroke) of the element. The value of the **'stroke'** property does not effect event processing.

visible
The given element can be the target element for pointer events when the **'visibility'** property is set to **visible** and the pointer is over either the interior (i.e., fill) or the perimeter (i.e., stroke) of the element. The values of the **'fill'** and **'stroke'** do not effect event processing.

painted
The given element can be the target element for pointer events when the pointer is over a "painted" area. The pointer is over a painted area if it is over the interior (i.e., fill) of the element and the **'fill'** property is set to a value other than 'none' or it is over the perimeter (i.e., stroke) of the element and the **'stroke'** property is set to a value other than 'none'. The value of the **'visibility'** property does not effect event processing.

fill
The given element can be the target element for pointer events when the pointer is over the interior (i.e., fill) of the element. The values of the **'fill'** and **'visibility'** properties do not effect event processing.

stroke
The given element can be the target element for pointer events when the pointer is over the perimeter (i.e., stroke) of the element. The values of the **'stroke'** and **'visibility'** properties do not effect event processing.

all
The given element can be the target element for pointer events whenever the pointer is over either the interior (i.e., fill) or the perimeter (i.e., stroke) of the element. The values of the **'fill'**, **'stroke'** and **'visibility'** properties do not effect event processing.

none
The given element does not receive pointer events.

For text elements, hit detection is performed on a character cell basis:

- The value **visiblePainted** means that the text string can receive events anywhere within the character cell if either the **'fill'** property is set to a value other than **none** or the **'stroke'** property is set to a value other than **none**, with the additional requirement that the **'visibility'** property is set to **visible**.
- The values **visibleFill**, **visibleStroke** and **visible** are equivalent and indicate that the text string can receive events anywhere within the character cell if the **'visibility'** property is set to **visible**. The values of the **'fill'** and **'stroke'** properties do not effect event processing.
- The value **painted** means that the text string can receive events anywhere within the character cell if either the **'fill'** property is set to a value other than **none** or the **'stroke'** property is set to a value other than **none**. The value of the **'visibility'** property does not effect event processing.

- The values **fill**, **stroke** and **all** are equivalent and indicate that the text string can receive events anywhere within the character cell. The values of the **'fill'**, **'stroke'** and **'visibility'** properties do not effect event processing.
- The value **none** indicates that the given text does not receive pointer events.

For raster images, hit detection is either performed on a whole-image basis (i.e., the rectangular area for the image is one of the determinants for whether the image receives the event) or on a per-pixel basic (i.e., the alpha values for pixels under the pointer help determine whether the image receives the event):

- The value **visiblePainted** means that the raster image can receive events anywhere within the bounds of the image if any pixel from the raster image which is under the pointer is not fully transparent, with the additional requirement that the **'visibility'** property is set to **visible**.
- The values **visibleFill**, **visibleStroke** and **visible** are equivalent and indicate that the image can receive events anywhere within the rectangular area for the image if the **'visibility'** property is set to **visible**.
- The value **painted** means that the raster image can receive events anywhere within the bounds of the image if any pixel from the raster image which is under the pointer is not fully transparent. The value of the **'visibility'** property does not effect event processing.
- The values **fill**, **stroke** and **all** are equivalent and indicate that the image can receive events anywhere within the rectangular area for the image. The value of the **'visibility'** property does not effect event processing.
- The value **none** indicates that the image does not receive pointer events.

Note that for raster images, the values of properties **'fill-opacity'**, **'stroke-opacity'**, **'fill'** and **'stroke'** do not effect event processing.

13.7 Magnification and panning

Magnification represents a complete, uniform transformation on an SVG document fragment, where the magnify operation scales all graphical elements by the same amount. A magnify operation has the effect of a supplemental scale and translate transformation placed at the outermost level on the SVG document fragment (i.e., outside the **'svg'** element).

Panning represents a translation (i.e., a shift) transformation on an SVG document fragment in response to a user interface action.

SVG user agents that operate in interaction-capable user environments are required to support the ability to magnify and pan.

The **'svg'** element in an SVG document fragment has attribute **zoomAndPan**, which takes the possible values of *disable* and *magnify*, with the default being *magnify*.

If *disable*, the user agent shall disable any magnification and panning controls and not allow the user to magnify or pan on the given document fragment.

If *magnify*, in environments that support user interactivity, the user agent shall provide controls to allow the user to perform a "magnify" operation on the document fragment.

Animatable: no.

13.8 Element focus

13.8.1 The focusable attribute

In many cases, such as text editing, the user is required to place focus on a particular element, ensuring that input events, such as keyboard input, are sent to that element.

All renderable elements can be focusable, including both container elements and graphics elements,. It is possible for a focusable container element to have focusable descendants.

Attribute definition:

`focusable` = "true" | "false" | "auto"
Defines if an element can get keyboard focus (i.e. receive keyboard events) and be a target for field-to-field navigation actions. (Note: in some environments, field-to-field navigation can be accomplished with the tab key.)

The attribute value can be either of the following:

"true"
The element is keyboard-aware and should be treated as any other UI component that can get focus.
"false"
The element is not focusable.
"auto"
The default value. Equivalent to "false", except that it acts like "true" for the following cases:

- the a element
- the text and textArea elements with editable set to "true"

Animatable: Yes

13.9 Navigation

13.9.1 Navigation behavior

System-dependent input facilities (e.g., the tab key on most desktop computers) should be supported to allow navigation between elements that can obtain focus (ie. elements for which the value of the focusable property evaluates to "true").

The document has the concept of a focus ring, which is the order in which elements obtain focus. By default the focus ring is obtained using document order. All focusable elements are part of the default focus ring. It is possible to override the default focus ring (see the Specifying navigation section below).

The SVG language supports a flattened notion of field navigation between focusable elements where it is possible to define field navigation between any two focusable elements defined within a given SVG document without regard to document hierarchy. For example:

```
<rect id="r1" focusable="true".../>
<g> id="g1" focusable="true">
  <circle id="c1" focusable="true".../>
</g>
```

In the above example, it is possible to specify field-to-field navigation such that it is possible to navigate directly from any of the three elements. Thus, assuming a desktop computer which uses the tab key for field navigation, it is possible to specify focus navigation order such that the tab key takes the user from r1 to c1 to g1.

When navigating to an element that is not visible on the canvas the following rules apply:

- It **is not possible** to navigate to an element which has `display='none'`. (An element which has `display='none'` is not focusable.)
- It **is possible** to navigate to an element which is not visible (i.e. which has a 100% transparency or which is hidden by another element).
- It **is possible** to navigate to an element which is located outside of the current viewport. In this case it is recommended that the UA SHOULD change the current viewport so that the focused element becomes visible.

SVG's flattened notion of field navigation extends to referenced content and shadow trees as follows:

- Focusable elements within the content referenced by a use element participate in field navigation operations using the flattened focus model. (Note: If a referenced group contains a focusable element, and that group is referenced by two use elements, then the document will have two separate focusable fields, not just one.)
- If an animation element references an SVG document, then all of the focusable fields defined within the referenced SVG document participate in field navigation operations using the flattened focus model.

Focus navigation behaves as specified:

1. When the document is loaded the focus is first offered to the User Agent.
2. Once the User Agent releases focus, then focus passes to the entity that first matches the following criteria:
    1. the root 'svg' element if it is focusable,
    2. the element referenced by the 'focusNext' attribute on the root 'svg' element, if the attribute is present,
    3. the first focusable element in the document starting from the root 'svg' element,
    4. the User Agent
3. If the focus is held by an element in the document, then the next element in navigation order is the entity that first matches the following criteria:
    1. the element referenced by the 'focusNext' attribute on the focused element,
    2. the next focusable element in document order,
    3. the User Agent
4. If the focus is held by an element in the document, then the previous element in navigation order is the entity that first matches the following criteria:
    1. the element referenced by the 'focusPrev' attribute on the focused element,
    2. the previous focusable element in document order,
    3. the User Agent

For stand-alone SVG documents, the User Agent must always have a currently focused object. If focus is not held by any elements in the document, the User Agent just give focus to the SVGDocument object.

For SVG documents which are referenced by a non-SVG host document (e.g., XHTML), the SVG document might participate within the host document's focus ring, which would allow direct navigation from an SVG focusable element onto a focusable element within the host document.

13.9.2 Specifying navigation

Navigation order can be specified using the focus attributes.

- **focusNext**: references the next element in the focus ring
- **focusPrev**: references the previous element in the focus ring
- **focusNorth**: references the element to be focused if a navigation event in the North direction is fired
- **focusNorthEast**: references the element to be focused if a navigation event in the North-East direction is fired
- **focusEast**: references the element to be focused if a navigation event in the East direction is fired
- **focusSouthEast**: references the element to be focused if a navigation event in the South-East direction is fired
- **focusSouth**: references the element to be focused if a navigation event in the South direction is fired
- **focusSouthWest**: references the element to be focused if a navigation event in the South-West direction is fired
- **focusWest**: references the element to be focused if a navigation event in the West direction is fired
- **focusNorthWest**: references the element to be focused if a navigation event in the North-West direction is fired

The allowed values for the focus attributes are either an IDREF or the string 'auto'. An IDREF value specifies the element that should receive focus if the corresponding navigation is triggered. The value 'auto' on **focusNext** and **focusPrev** effectively means the behavior is as if the attribute was not specified (navigation follows the rules specified above). The behaviour of 'auto' on the other focus attributes is left up to the User Agent.

13.9.3 Obtaining and listening to focus programmatically

When the user agent gives an element focus it receives a {"http://www.w3.org/2001/xml-events", "DOMFocusIn"} event which has the the new focused object as the event target and a {"http://www.w3.org/2001/xml-events", "DOMFocusOut"} event which has the previously focused object as the event target.

The SVGSVGElement interface has a setFocus(DOMObject object) method that puts the focus on the requested object. Calling setFocus with an element that is not focusable causes focus to stay on the currently focused object.

The SVGSVGElement interface has a moveFocus(short motionType) which moves current focus to a different object based on the value of motionType.

User agents which support pointer devices such as a mouse MUST allow users to put focus onto focusable elements. For example, it should be possible to click on a focusable element in order to give focus.

Empty text fields in SVG theoretically take up no space, but they have a point or zero-width line segment that represents the location of the empty text field. User agents should allow users with pointer devices to put focus into empty text fields by initiating a select action (e.g., a mouse click) at the location of the empty text field.

It is possible to change the field navigation order in script by catching the input event related to the navigation and cancelling the event.

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

14 Linking

Contents

14.1 References

14.1.1 Overview

SVG makes extensive use of IRI references [IRI] to other objects. For example, to fill a rectangle with a linear gradient, you first define a **'linearGradient'** element and give it an ID, as in:

Example: 05_07.xml

```
<linearGradient id="MyGradient">...</linearGradient>
```

You then reference the linear gradient as the value of the 'fill' property for the rectangle, as in:

Example: 05_08.xml

```
<rect fill="url(#MyGradient)"/>
```

IRI references are defined in the following form:

```
<iri-reference> = [ <absoluteIRI> | <relativeIRI> ] [ "#" <elementID> ]
```

where **<elementID>** is the ID of the referenced element.

(Note that the bare name form above (i.e., **#<elementID>**) is formulated in a syntax compatible with "XPointer Framework" [XPTRFW].

SVG supports two types of IRI references:

- local IRI references, where the IRI reference does not contain an **<absoluteIRI>** or **<relativeIRI>** and thus only contains a fragment identifier (i.e., **#<elementID>**))
- non-local IRI references, where the IRI reference does contain an **<absoluteIRI>** or **<relativeIRI>**

The following rules apply to the processing of IRI references:

- IRI references to nodes that do not exist shall be treated as invalid references.
- IRI references to elements which are inappropriate targets for the given reference shall be treated as invalid references (see list below for appropriate targets). For example, the **'fill'** property refering to a 'rect' element (**fill="url(#MyRectId)"**) would be invalid.
- IRI references that directly or indirectly reference themselves are treated as invalid circular references.

The following list describes the elements and properties that allow IRI references and the valid target types for those references:

- the **'a'** element can reference any local or non-local resource.
- the **'animate'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'animateColor'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'animateMotion'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'animateTransform'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'animation'** element can reference any local or non-local resource with the following restrictions:
  - When an **'animation'** element references external SVG documents (or document fragments) the referenced document/fragment MUST NOT contain scripting, hyperlinking to animations or any externally referenced 'use' or 'animation' elements.
  - An **'animation'** element referencing a document fragment is not allowed to reference an **'svg'** element.
- the **'audio'** element can reference any local or non-local resource.
- the **'discard'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'fill'** property can reference a local resource as defined in Specifying paint.
- the **'font-face-URI'** element can reference any local or non-local resource.
- the **'foreignObject'** element can reference any local or non-local resource.
- the **'handler'** element can reference any local or non-local resource.
- the **'image'** element can reference any local or non-local resource.
- the **'mpath'** element can reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'prefetch'** element can reference any local or non-local resource.
- the **'script'** element can reference an external resource that provides the script content.
- the **'stroke'** property can reference a local resource as defined in Specifying paint.
- the **'set'** element can only reference a local resource (see Identifying the target element for an animation for reference rules).
- the **'use'** element can reference any local or non-local resource with the following restrictions:
  - When a **'use'** element references an external element (an external document fragment), the referenced fragment MUST NOT contain scripting, hyperlinking to animations or any externally referenced 'use' or 'animation' elements.
  - A **'use'** element is not allowed to reference an **'svg'** element.
- the **'video'** element can reference any local or non-local resource.

The following rules apply to the processing of invalid IRI references:

- An invalid local IRI reference (i.e., an invalid references to a node within the current document) represents an error (see Error processing), apart from the **xlink:href** attribute on the **'a'** element and the properties that allow for backup values in the case where the IRI reference is invalid (see **'fill'** and **'stroke'**).
- An invalid circular IRI reference represents an error (see Error processing).
- When attribute **externalResourcesRequired** has been set to **true** on the referencing element or one of its ancestors, then an unresolved external IRI reference (i.e., a resource that cannot be located) represents an error (see Error processing).

It is recommended that, wherever possible, referenced elements be defined inside of a **'defs'** element. Among the elements that are always referenced are **'linearGradient'** and **'radialGradient'** . Defining these elements inside of a **'defs'** element promotes understandability of the SVG content and thus promotes accessibility.

14.1.2 IRIs and URIs

Internationalized Resource Identifiers (IRIs) are a more generalized complement to Uniform Resource Identifiers (URIs). An IRI is a sequence of characters from the Universal Character Set [Unicode40]. A URI is constructed from a much more restricted set of characters. All URIs are already conformant IRIs. A mapping from IRIs to URIs is defined by the IRI specification, which means that IRIs can be used instead of URIs in XML documents, to identify resources. IRIs can be converted to URIs for resolution on a network, if the protocol does not support IRIs directly.

Previous versions of SVG, following XLink, defined a [IRI Reference] type as a URI *or as a sequence of characters which must result in a URI reference after a*

particular escaping procedure was applied. The escaping procedure was repeated in the XLink specification [XLink], and in the W3C XML Schema Part 2: Datatypes specification [Schema2]. This copying introduced the possibility of error and divergence, but was done because the IRI specification was not yet standardized.

In this specification, the correct term "IRI' is used for this 'URI or sequence of characters plus an algorithm' and the escaping method is defined by reference to the IRI specification [RFC3987], which has since become an IETF Proposed Standard. Other W3C specifications are expected to be revised over time to remove these duplicate descriptions of the escaping procedure and to refer to IRI directly.

14.1.3 IRI reference attributes

A IRI reference is specified within an **href** attribute in the XLink [XLINK] namespace. For example, if the prefix of 'xlink' is used for attributes in the XLink namespace, then the attribute will be specified as **xlink:href**. The value of this attribute is a IRI reference for the desired resource (or secondary resource, if there is a fragment identifier).

The value of the **href** attribute must be a IRI reference as defined in [RFC3987].

If the protocol, such as HTTP, does not support IRIs directly, the IRI is converted to a URI by the SVG implementation, as described in section 3.1 of the IRI specification.

Because it is impractical for any application to check that a value is an IRI reference, this specification follows the lead of [RFC3986] in this matter and imposes no such conformance testing requirement on SVG applications.

If the IRI reference is relative, its absolute version must be computed by the method of [XML-Base] before use.

Additional XLink attributes can be specified that provide supplemental information regarding the referenced resource.

Schema: xlinkatt

```
<define name='svg.XLinkBase.attr' combine='interleave'>
  <optional>
    <attribute name='xlink:type' svg:animatable='true' svg:inheritable='false'>
      <value>simple</value>
    </attribute>
  </optional>
  <optional>
    <attribute name='xlink:role' svg:animatable='false' svg:inheritable='false'>
      <ref name='IRI.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='xlink:arcrole' svg:animatable='false' svg:inheritable='false'>
      <ref name='IRI.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='xlink:title' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
</define>

<define name='svg.XLinkHrefRequired.attr' combine='interleave'>
  <attribute name='xlink:href' svg:animatable='true' svg:inheritable='false'>
    <ref name='IRI.datatype'/>
  </attribute>
</define>

<define name='svg.XLinkBaseRequired.attr' combine='interleave'>
  <ref name='svg.XLinkBase.attr'/>
  <ref name='svg.XLinkHrefRequired.attr'/>
</define>


<define name='svg.XLinkActuateOnLoad.attr' combine='interleave'>
  <optional>
    <attribute name='xlink:actuate' svg:animatable='false' svg:inheritable='false'>
      <value>onLoad</value>
    </attribute>
  </optional>
</define>

<define name='svg.XLinkShowOther.attr' combine='interleave'>
  <optional>
    <attribute name='xlink:show' svg:animatable='false' svg:inheritable='false'>
      <value>other</value>
    </attribute>
  </optional>
</define>

<define name='svg.XLinkEmbed.attr' combine='interleave'>
  <optional>
    <attribute name='xlink:show' svg:animatable='false' svg:inheritable='false'>
      <value>embed</value>
    </attribute>
  </optional>
  <ref name='svg.XLinkActuateOnLoad.attr'/>
  <ref name='svg.XLinkBaseRequired.attr'/>
</define>


<define name='svg.XLinkRequired.attr' combine='interleave'>
  <ref name='svg.XLinkShowOther.attr'/>
  <ref name='svg.XLinkActuateOnLoad.attr'/>
  <ref name='svg.XLinkBaseRequired.attr'/>
</define>


<define name='svg.XLinkReplace.attr' combine='interleave'>
  <optional>
    <attribute name='xlink:show' a:defaultValue='replace' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>new</value>
        <value>replace</value>
```

```
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='xlink:actuate' svg:animatable='false' svg:inheritable='false'>
        <value>onRequest</value>
      </attribute>
    </optional>
    <ref name='svg.XLinkBaseRequired.attr'/>
  </define>

  <define name='svg.XLink.attr' combine='interleave'>
    <optional>
      <ref name='svg.XLinkHrefRequired.attr'/>
    </optional>
    <ref name='svg.XLinkShowOther.attr'/>
    <ref name='svg.XLinkActuateOnLoad.attr'/>
    <ref name='svg.XLinkBase.attr'/>
  </define>
```

xlink:type = 'simple'
Identifies the type of XLink being used. In SVG Tiny 1.2, only simple links are available. All links are simple links by default, so the attribute **xlink:type**="simple" is optional and need not be explicitly stated. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:role = ' <iri> '
An optional IRI reference that identifies some resource that describes the intended property. The value must be a IRI reference as defined in [RFC2396], except that if the IRI scheme used is allowed to have absolute and relative forms, the IRI portion must be absolute. When no value is supplied, no particular role value is to be inferred. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:arcrole = ' <iri> '
An optional IRI reference that identifies some resource that describes the intended property. The value must be a IRI reference as defined in [RFC2396], except that if the IRI scheme used is allowed to have absolute and relative forms, the IRI portion must be absolute. When no value is supplied, no particular role value is to be inferred. The arcrole attribute corresponds to the [RDF] notion of a property, where the role can be interpreted as stating that "starting-resource HAS arc-role ending-resource." This contextual role can differ from the meaning of an ending resource when taken outside the context of this particular arc. For example, a resource might generically represent a "person," but in the context of a particular arc it might have the role of "mother" and in the context of a different arc it might have the role of "daughter." Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:title = '<string>'
The title attribute is used to describe the meaning of a link or resource in a human-readable fashion, along the same lines as the role or arcrole attribute. A value is optional; if a value is supplied, it should contain a string that describes the resource. In general it is preferable to use a **'title'** child element rather than a **'title'** attribute. The use of this information is highly dependent on the type of processing being done. It may be used, for example, to make titles available to applications used by visually impaired users, or to create a table of links, or to present help text that appears when a user lets a mouse pointer hover over a starting resource. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:show = 'new' | 'replace' | 'embed' | 'other' | 'none'
This attribute is provided for backwards compatibility with SVG 1.1. It provides documentation to XLink-aware processors. In case of a conflict, the target attribute has priority, since it can express a wider range of values. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:actuate = 'onLoad'
This attribute is provided for backwards compatibility with SVG 1.1. It provides documentation to XLink-aware processors. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

In all cases, for compliance with the "Namespaces in XML 1.1" Recommendation [XML-NS], an explicit XLink namespace declaration must be provided whenever one of the above XLink attributes is used within SVG content. One simple way to provide such an XLink namespace declaration is to include an **xmlns** attribute for the XLink namespace on the **'svg'** element for content that uses XLink attributes. For example:

Example: 05_09.svg

```
<svg xmlns:xlink="http://www.w3.org/1999/xlink"...>
  <image xlink:href="foo.png" .../>
</svg>
```

14.1.4 Externally referenced documents

In this section, the term "primary document" refers to an outermost SVG document or SVG document fragment. If an SVG document is loaded directly by a web browser (e.g., the browser views file foo.svg), then it is the primary document. If an SVG document as a whole is referenced for inclusion by a parent document, such as using the HTML object or SVG animation elements, then that document itself is also a primary document. If an SVG document fragment is embedded inline within a non-SVG document, then the outermost svg element defines the root element for a subtree which acts as a primary document.

Implementations are free to optimize for the case of large resource library loaded into multiple primary documents, but logically each primary document represents its own separate, self-contained document instance. In particular, the conceptual processing model is that events (e.g., the document load event) are fired; scripts are executed in normal fashion and resource documents are modifiable by scripts; coordinate systems transformations are applied; timelines are initiated and animations execute; etc. For example, if document A.svg includes an svg:animation element which refers to B.svg, then both are primary documents, and both represent separate, self-contained documents with their own scripting contexts and animation timelines.

The term "resource document" refers to a complete, self-contained SVG document which has at least one of its elements referenced as a resource by a primary document. For example, suppose document A.svg is loaded into a browser for viewing, and this document refers to a gradient element within B.svg via a IRI reference. In this case, A.svg is a primary document and B.svg is a resource document.

Each primary document has an associated dictionary that maps all IRIs for resource documents it references; initially it is populated only with the primary document itself. Each resource or subresource loaded directly or indirectly is resolved through that dictionary with resource documents downloaded as needed.

The conceptual model is that each resource document is loaded only once; if the same resource document is referenced multiple times directly or indirectly by the same primary document, that resource document is only retrieved and processed one time.

The conceptual processing model for resource documents is that the document is processed as a complete and separate SVG document instance. The only

difference between a resource document and a primary document is that the primary document is rendered directly to the canvas, whereas all resource documents are conceptually rendered to an invisible (offscreen) canvas. In particular, the conceptual processing model is that events (e.g., the document load event) are fired; scripts are executed in normal fashion and resource documents are modifiable by scripts; coordinate systems transformations are applied; stylesheets are applied and the CSS cascade is run (not relevant to SVGT1.2); timelines are initiated and animations execute; sXBL transformations are applied (not relevant to SVGT1.2); etc.

Because of HTTP redirects, the same source document might be retrieved from multiple different source IRIs. The rule for SVG is that documents are considered to be unique based on string comparisons of the full IRI after resolving relative IRIs into absolute IRIs and after taking into account HTTP redirects (i.e., use the post-redirect IRI instead of the original source IRI).

14.2 Links out of SVG content: the 'a' element

SVG provides an 'a' element, analogous to HTML's 'a' element, to indicate links (also known as *hyperlinks* or *Web links*). SVG uses XLink ([XLink]) for all link definitions.

SVG Tiny 1.2 only requires that user agents support XLink's notion of simple links. Each simple link associates exactly two resources, one local and one remote, with an arc going from the former to the latter.

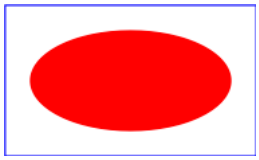
A simple link is defined for each separate rendered element contained within the 'a' element; thus, if the 'a' element contains three 'circle' elements, a link is created for each circle. For each rendered element within an 'a' element, the given rendered element is the local resource (the source anchor for the link).

The remote resource (the destination for the link) is defined by a IRI specified by the XLink href attribute on the 'a' element. The remote resource may be any Web resource (e.g., an image, a video clip, a sound bite, a program, another SVG document, an HTML document, etc.). By activating these links (by clicking with the mouse, through keyboard input, voice commands, etc.), users may traverse hyperlinks to these resources.

Example link01 assigns a link to an ellipse.

Example: 17_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="3cm" viewBox="0 0 5 3" version="1.2" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
  <desc>Example link01 - a link on an ellipse
  </desc>
  <rect x=".01" y=".01" width="4.98" height="2.98"
        fill="none" stroke="blue"  stroke-width=".03"/>
  <a xlink:href="http://www.w3.org">
    <ellipse cx="2.5" cy="1.5" rx="2" ry="1"
             fill="red" />
  </a>
</svg>
```



If the above SVG file is viewed by a user agent that supports both SVG and HTML, then clicking on the ellipse will cause the current window or frame to be replaced by the W3C home page.

Schema: a

```
  <element name='a'>
      <ref name='a.AT'/>
      <!-- the content model of 'a' is defined in the
           context of its containing element -->
  </element>


  <define name='a.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.Focus.attr'/>
    <ref name='svg.Transform.attr'/>
    <ref name='svg.XLinkReplace.attr'/>
    <optional>
      <attribute name='target' svg:animatable='true' svg:inheritable='false'>
        <ref name='LinkTarget.datatype'/>
      </attribute>
    </optional>
  </define>
```

Attribute definitions:

`xlink:type` = 'simple'
Optional. (See generic description of xlink:type attribute.)
`xlink:role` = ' <iri> '
Optional. (See generic description of xlink:role attribute.)
`xlink:arcrole` = ' <iri> '
Optional. (See generic description of xlink:arcrole attribute.)
`xlink:title` = '<string>'
(See generic description of xlink:title attribute.)
`xlink:show` = 'new' | 'replace'
This optional attribute is provided for backwards compatibility with SVG 1.1. It provides documentation to XLink-aware processors. If target="_blank" then use xlink:show="new" else use "replace". In case of a conflict, the target attribute has priority, since it can express a wider range of values. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:actuate = 'onRequest'

This optional attribute is provided for backwards compatibility with SVG 1.1. It provides documentation to XLink-aware processors that an application should traverse from the starting resource to the ending resource only on a post-loading event triggered for the purpose of traversal. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: no.

xlink:href = " <iri> "

The location of the referenced object, expressed as a IRI reference. Refer to the "XML Linking Language (XLink)" [XLink].

Animatable: yes.

target = '_replace' | '_self' | '_parent' | '_top' | '_blank' | "<frame-target>"

This attribute is used when there are multiple possible targets for the ending resource, such as when the parent document is a multi-frame HTML or XHTML document. This attribute specifies the name or portion of the target window, frame, pane, tab, or other relevant presentation context (e.g., an HTML or XHTML frame) into which a document is to be opened when the link is activated. The values and semantics of this attribute are the same as the WebCGM Picture Behavior values [WebCGM]:

_replace
The current SVG image is replaced by the linked content in the same rectangular area in the same frame as the current SVG image. This is the default value, if the target attribute is not specified.

_self
The current SVG image is replaced by the linked content in the same frame as the current SVG image.

_parent
The immediate frameset parent of the SVG image is replaced by the linked content.

_top
The content of the full window or tab, including any frames, is replaced by the linked content

_blank
A new un-named window or tab is requested for the display of the linked content. If this fails, the result is the same as _top

"<frame-target>"
Specified the name of the frame, pane, or other relevant presentation context for display of the linked content. If this already exists, it is re-used, replacing the existing content. if it does not exist, it is created (the same as _blank, except that it now has a name). frame-target must be an XML name [XML] so it must start with a name start alphabetic character (a-zA-Z).

Note: The value _new is *not* a legal value for target (use _blank).

Animatable: yes.

14.3 Linking into SVG content: IRI fragments and SVG views

14.3.1 Introduction: IRI fragments and SVG views

On the Internet, resources are identified using IRIs (International Resource Identifiers) [RFC3987]. For example, an SVG file called MyDrawing.svg located at http://example.com might have the following IRI:

```
http://example.com/MyDrawing.svg
```

An IRI can also address a particular element within an XML document by including a IRI fragment identifier as part of the IRI. An IRI which includes a IRI fragment identifier consists of an optional base IRI, followed by a "#" character, followed by the IRI fragment identifier. For example, the following IRI can be used to specify the element whose ID is "Lamppost" within file MyDrawing.svg:

```
http://example.com/MyDrawing.svg#Lamppost
```

Because SVG content often represents a picture or drawing of something, a common need is to link into a particular view of the document, where a view indicates the initial transformations so as to present a closeup of a particular section of the document.

14.3.2 SVG fragment identifiers

To link into a particular view of an SVG document, the IRI fragment identifier needs to be a correctly formed SVG fragment identifier. An SVG fragment identifier defines the meaning of the "selector" or "fragment identifier" portion of IRIs that locate resources of MIME media type "image/svg+xml".

An SVG fragment identifier can come in two forms:

1. Shorthand *bare name* form of addressing (e.g., **MyDrawing.svg#MyView**). This form of addressing, which allows addressing an SVG element by its ID, is compatible with the fragment addressing mechanism for older versions of HTML and the shorthand bare name formulation in "XPointer Framework" [XPTRFW].
2. SVG view specification (e.g., **MyDrawing.svg#svgView(transform(scale(2))**). This form of addressing specifies the desired view of the document (e.g., the region of the document to view, the initial zoom level) completely within the SVG fragment specification. The contents of the SVG view specification is transform(...) whose parameters have the same meaning as the corresponding attribute has on a 'g' element).

An SVG fragment identifier is defined as follows:

```
SVGFragmentIdentifier ::= BareName |
                          SVGViewSpec

BareName ::= XML_Name
SVGViewSpec ::= 'svgView(' SVGViewAttributes ')'
SVGViewAttributes ::= SVGViewAttribute |
                      SVGViewAttribute ';' SVGViewAttributes

SVGViewAttribute ::= transformSpec
transformSpec ::= 'transform(' TransformParams ')'
```

where:

- **XML_Name** conforms to the XML production for 'XML Name'
- **TransformParams** corresponds to the parameter values for the **transform** attribute that is available on many elements. For example, **transform(scale(5))**.

Spaces are not allowed in fragment specifications; thus, commas are used to separate numeric values within an SVG view specification (e.g., **#svgView(transform(scale(5),translate(100,100)))**) and semicolons are used to separate attributes (e.g., **#svgView(scale(2);rotate(45))**).

Note: characters in fragment identifiers that are outside the repertoire of US-ASCII must be encoded using UTF-8 and %HH escaping if the value of the XLink href attribute is converted to a URI for resolution. [RFC3987]

When a source document performs a link into an SVG document via an HTML [HTML4] anchor element (i.e., **<a href=...>** element in HTML) or an XLink specification [XLINK], then the SVG fragment identifier specifies the initial view into the SVG document, as follows:

- If no SVG fragment identifier is provided (e.g, the specified IRI did not contain a "#" character, such as **MyDrawing.svg**), then the initial view into the SVG document is established using the view specification attributes (i.e., viewBox, etc.) on the outermost **'svg'** element.
- If the SVG fragment identifier addresses a specific SVG view (e.g., **MyDrawing.svg#svgView(transform(translate(0,200,1000,1000),scale(3))**), then the document fragment defined by the root **'svg'** element is displayed in the viewport using the SVG view specification provided by the SVG fragment identifier.
- If the SVG fragment identifier addresses any element , such as **MyDrawing.svg#rectId**), then the document defined by the root **'svg'** element is displayed in the viewport using the view specification attributes on the root **'svg'** element.

14.4 Hyperlinking Module

The Hyperlinking Module contains the following element:

- a

14.5 XLink Attribute Module

The XLink Attribute Module contains the following attributes:

- xlink:type
- xlink:href
- xlink:role
- xlink:arcrole
- xlink:title
- xlink:show
- xlink:actuate

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

15 Scripting

Contents

15.1 Specifying the scripting language

15.1.1 Specifying the default scripting language

The **'contentScriptType'** attribute on the **'svg'** element specifies the default scripting language for the given document fragment.

15.1.2 Local declaration of a scripting language

It is also possible to specify the scripting language for each individual **'script'** or **'handler'** elements by specifying a type attribute on the **'script'** / **'handler'** elements.

15.2 The 'script' element

A script element may either contain or point to executable content (e.g., ECMAScript or Java JAR file). Executable content can come either in the form of a script (textual code) or in the form of compiled code. If the code is textual, it can either be placed inline in the script element (as character data) or as an external resource, referenced through xlink:href attribute. Compiled code must be an external resource.

Scripting languages (such as ECMAScript) that have a notion of a "global scope" or a "global object" must have a single global object associated with the document (unique for each DOM Document node). This object is shared by all elements. Thus, an ECMAScript function defined within any script is in the "global" scope of the entire document in which the script occured. The global object must have all the methods and attributes from the SVGGlobal interface. It can be obtained from an SVG document through the SVGDocument::global attribute. Event listeners attached through event attributes and handler elements are also evaluated using the global scope of the document in which such a listener is defined.

For compiled languages (such as Java) that don't have a notion of "global scope", each script element, in effect, provides a separate scope object per script element. This scope object performs an initialization and serves as event listener factory for the handler element.

Script execution happens only at load time. Removing, inserting or altering script elements once the document has been loaded has no effect.

Exact details on how this element works depend on the executable content's type (either the resource MIME type or specified as an attribute). SVG does not require the support of any particular scripting language. However, SVG defines the behavior for two specific script types in the case where an implementation supports the type:

application/ecmascript

> This type of executable content must be source code for the ECMAScript programming language (either of ECMA-262 3rd Edition, ECMA-327, or ECMA-357). This code is executed in the context of this element's owner document's global scope as explained above.
>
> Also, the script global object must implement the EventListenerInitializer2 interface by providing an initializeEventListeners function that is called for every script element in the document and a createEventListener function that is called for every handler element that has a script content type of "application/ecmascript".
>
> SVG implementations that load external resources through protocols such as HTTP that support content coding must accept external script files that have been encoded using gzip compression (flagged using "Content-Encoding: gzip" for HTTP).

application/java-archive

> This type of executable content must be an external resource that contains a Java JAR. The manifest file in the JAR archive must have an entry named SVG-Handler-Class. The entry's value must be a fully-qualified Java class name for a class contained in this archive. The user agent must instantiate the class from the JAR file and cast it to the EventListenerInitializer2 interface. Then the initializeEventListeners method must be called with the script element itself as a parameter. If a class listed in SVG-Handler-Class do not implement EventListenerInitializer2, it is an error.
>
> Note that the user agent may reuse classes loaded from the same URL, so the code must not assume that every script element or every document will create its own separate class object. Thus, one cannot assume, for instance, that static fields in the class are private to a document.

Implementations may also accept the script type "text/ecmascript" for backwards compatibility with SVG 1.1. However, this type is deprecated and should not be used by content authors.

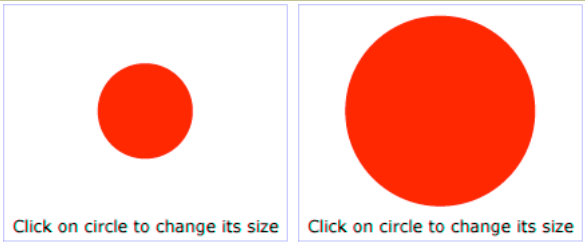
Other language bindings are encouraged to adopt a similar approach to either of the two described above.

Example 18_01 defines a function `circle_click` which is called when the 'circle' element is being clicked. The drawing below on the left is the initial image. The drawing below on the right shows the result after clicking on the circle. The example uses the 'handler' element which is described further down in this chapter.

Note that this example demonstrates the use of the 'click' event for explanatory purposes. The example presupposes the presence of an input device with the same behavioral characteristics as a mouse, which will not always be the case. To support the widest range of users, the 'activate' event attribute should be used instead of the 'click' event attribute.

Example: 18_01.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="6cm" height="5cm" viewBox="0 0 600 500"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
     xmlns:ev="http://www.w3.org/2001/xml-events">
  <desc>Example: invoke an ECMAScript function from an click event
  </desc>
  <!-- ECMAScript to change the radius with each click -->
  <script type="application/ecmascript"> <![CDATA[
    function circle_click(evt) {
      var circle = evt.target;
      var currentRadius = circle.getFloatTrait("r");
      if (currentRadius == 100)
        circle.setFloatTrait("r", currentRadius*2);
      else
        circle.setFloatTrait("r", currentRadius*0.5);
    }
  ]]> </script>

  <!-- Outline the drawing area with a blue line -->
  <rect x="1" y="1" width="598" height="498" fill="none" stroke="blue"/>
  <!-- Act on each click event -->
  <circle onclick="circle_click(evt)" cx="300" cy="225" r="100" fill="red">
  <handler type="application/ecmascript" ev:event="click">
        circle_click(evt);
    </handler>
  </circle>

  <text x="300" y="480" font-family="Verdana" font-size="35" text-anchor="middle">
    Click on circle to change its size
  </text>
</svg>
```



Schema: script

```
  <define name='script'>
    <element name='script'>
      <ref name='script.AT'/>
      <ref name='script.CM'/>
    </element>
  </define>

  <define name='script.AT' combine='interleave'>
    <ref name='svg.CorePreserve.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.ContentType.attr'/>
  </define>

  <define name='script.CM'>
    <choice>
      <group>
        <ref name='svg.XLinkRequired.attr'/>
      </group>
      <text/>
    </choice>
  </define>
```

Attribute definitions:

`type` = " *content-type* "
Identifies the scripting language for the given 'script' element. The value ***content-type*** specifies a media type, per [RFC2045]. If a 'type' is not provided, the value of 'contentScriptType' on the 'svg' element shall be used.

Animatable: no.

15.3 XML Events

XML Events is an XML syntax for integrating event listeners and handlers with DOM 2 or DOM 3 Event interfaces. Declarative event handling in SVG 1.1 was hardwired into the language, in that the developer was required to embed the event handler in the element syntax (e.g. an element has an onclick attribute). SVG Tiny does not support the event attributes (onload, onclick, onactivate, etc.). Instead SVG Tiny uses XML Events to provide the ability to listen to custom events, as well as specifying the event listener separately from the graphical content.

SVG Tiny 1.2 makes the following modifications to XML Events:

1. **Namespaced events**: As SVG 1.2 supports DOM Level 3 Events, which are namespaced, XML Events in SVG 1.2 must allow namespaced events.
2. **IRIREFs instead of IDREFs**: the observer and target attributes from XML Events are currently IDREFs. Since SVG 1.2 requires a declarative syntax for event handling in more than one document, it uses IRIREFs for those attributes, with the following restriction: only documents that are declaratively referenced as part of the current document, via the **'use'** and **'animation'** elements, can be referred to. Referring to any other arbitrary external document is an error.
3. **No capture phase**: SVG Tiny does not have an event capture phase, only a bubble phase.

The list of events supported by SVG Tiny 1.2 is given in the Interactivity chapter.

The following is an example of an SVG file using XML Events:

Example: handler.svg

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
     xmlns:ev="http://www.w3.org/2001/xml-events">

  <desc>An example of the handler element.</desc>

  <rect id="myRect" x="10" y="20" width="200" height="300"
        fill="red"/>

  <!-- register a listener for a myRect.click event -->
  <ev:listener ev:event="click" ev:observer="myRect"
               ev:handler="#myClickHandler" />

  <handler id="myClickHandler" type="application/ecmascript">
    var myRect = document.getElementById("myRect");
    var width = myRect.getFloatTrait("width"));
    myRect.setFloatTrait("width", (width+10));
  </handler>

</svg>
```

In the above example, the ev:listener element registers that the myClickHandler element should be invoked whenever a "click" event happens on "myRect".

The combination of the XML Events syntax and the new handler element allows event handling to be more easily processed in a compiled language. Below is an example of an event handler using the Java language:

Example: javahandler.svg

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
     xmlns:ev="http://www.w3.org/2001/xml-events"
     xmlns:foo="http://www.example.com/foo">

  <desc>Example of a Java handler</desc>

  <rect id="myRect" x="10" y="20" width="200" height="300"
        fill="red"/>

  <!-- reference a jar containing an EventListenerInitializer2 object -->
  <script type="application/java-archive" id="init" xlink:href="http://example.com/myJar.jar"/>

  <!-- register a listener for a myRect.click event -->
  <ev:listener ev:event="click" ev:observer="myRect"
                    ev:handler="#myClickHandler" />

  <handler id="myClickHandler" type="application/java-archive" xlink:href="#init" foo:offset="10"/>

</svg>
```

In this case, the **'handler'** element specifies a reference to the script element that specificies the location of compiled code that conforms to the EventListenerInitializer2 interface. The user agent invokes the `createEventListener` method within the targeted interface.

In this case, the MyEventListenerInitializer2 referenced by the SVG-Handler-Class entry of the myJar.jar manifest has the following definition:

MyEventListenerInitializer2

```
package com.example;

import org.w3c.svg.EventListenerInitializer2;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.events.Event;
import org.w3c.dom.events.EventListener;

public class MyEventListenerInitializer2 implements EventListenerInitializer2 {

   Document document;

    public void initializeEventListeners(Element scriptElement) {
        document = scriptElement.getOwnerDocument();
    }
```

```
    public EventListener createEventListener(final Element handlerElement) {
        return new EventListener() {
            public void handleEvent(Event event) {
                Element myRect = document.getElementById("myRect");
                float width = Float.parseFloat(myRect.getAttribute("width"));
                float offest = Float.parseFloat(handlerElement.getAttributeNS("http://www.example.com/foo", "offset");
                myRect.setAttribute(width, "" + (width + 10));
            }
        };
    }
}
```

The EventListenerInitializer2 interface is currently defined in the SVG package/namespace. Future specifications may move this package.

15.4 The handler element

The **'handler'** element is similar to the **script** element: its contents, either included inline or referenced, are code that is to be executed by the scripting engine(s) used by user agent.

However, where the script element evaluates its contents when the document is loaded, the **'handler'** element evaluates its contents in response to an event. This means that SVGT1.2 use **'handler'** to get the equivalent functionality as the SVG Full event attributes.

For example, consider the following SVG 1.1 document:

Example: SVG 1.1 scripting

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
<rect id="myRect" x="10" y="20" width="200" height="300" fill="red"
onclick="var width =  parseFloat(
document.getElementById('myRect').getAttribute('width'));
document.setAttribute('myRect', 'width', (width+10));"/>
</svg>
```

The above example should be rewritten to use the **'handler'** element and XML Events (described below) as shown:

Example: handler2.svg

```
<?xml version="1.0" encoding="utf-8"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.2"
     xmlns:ev="http://www.w3.org/2001/xml-events">
  <desc>handler element example</desc>
  <rect id="myRect" x="10" y="20" width="200" height="300" fill="red">

    <handler type="application/ecmascript" ev:event="click">
        var myRect = evt.target;
        var width = myRect.getFloatTrait("width"));
        myRect.setFloatTrait("width", (width+10));
    </handler>

  </rect>

</svg>
```

In ECMAScript, the contents of the **'handler'** element behave as if they are the contents of a new function object, created as shown:

```
function(evt) {
//contents of handler
}
```

Other interpreted languages should behave in a similar manner.

The 'evt' parameter shown above is an Event object corresponding to the event that has triggered the handler.

Schema: handler

```
<define name='handler'>
  <element name='handler'>
    <ref name='handler.AT'/>
    <ref name='handler.CM'/>
  </element>
</define>

<define name='handler.AT' combine='interleave'>
  <ref name='svg.CorePreserve.attr'/>
  <ref name='svg.External.attr'/>
  <attribute name='ev:event' svg:animatable='false' svg:inheritable='false'>
    <ref name='XSLT-QName.datatype'/>
  </attribute>
  <ref name='svg.ContentType.attr'/>
</define>

<define name='handler.CM'>
  <choice>
    <group>
      <ref name='svg.XLinkRequired.attr'/>
    </group>
    <text/>
  </choice>
</define>
```

Attribute definitions:

`type` = "content-type"
Identifies the language used for the handler element. The value specifies a media type, per RFC2045. If a 'type' is not provided, the value of **'contentScriptType'** on the **'svg'** element shall be used.

Animatable: no.

xlink:href = "<iri>"
If this attribute is present, then the script content of the handler element should be loaded from this resource.

Animatable: no.

ev:event = "<string>"
The name of the event to handle. See event list for a list of all supported events.

Animatable: no.

For compiled languages, the xlink:href attribute must reference a script element that itself reference a jar file with a manifest with an SVG-Handler-Class entry pointing to an EventListenerInitializer2 implementation.

15.4.1 Parameters to handler elements

In many situations, the script author uses the **'handler'** as a template for calling other functions, using the content of the **'handler'** element to pass parameters. However, for compiled languages the **'handler'** element does not have any executable content.

In this case, the author should embed the parameters into the **'handler'** as custom content. The execution of the **'handler'** code retrieves the contents of the parameters using the DOM.

Below is an example of using parameters on the **'handler'** element:

Example: handlerparam.svg

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.2"
     xmlns:ev="http://www.w3.org/2001/xml-events"
     xmlns:foo="http://www.example.com/foo">
  <desc>An example of parameters on the handler element.</desc>

  <rect id="myRect" x="10" y="20" width="200" height="300"
        fill="red"/>

  <!-- reference a jar containing an EventListenerInitializer2 object -->
  <script type="application/java-archive" id="init" xlink:href="http://example.com/myJar.jar"/>

  <!-- register a listener for a myRect.click event -->
  <ev:listener ev:event="click" ev:observer="myRect"
               ev:handler="#myClickHandler" />

  <handler id="myClickHandler" type="application/java-archive"
           xlink:href="#init">
      <foo:offset value="10"/>
      <foo:person>
         <foo:name>Victor Vector</foo:name>
         <foo:age>42</foo:age>
      </foo:person>
  </handler>

</svg>
```

In this case, the object referenced by the SVG-Handler-Class entry of the myJar.jar manifest has its createEventListener method called and the returning EventListener registered. Whenever a 'click' event on the 'myRect' object is observed, the handleEvent method of the listener is called. The object can then retrieve the children of the **'handler'** element in order to obtain the contents of the elements from the "foo" namespace.

15.5 Event handling

Events can cause scripts to execute when either of the following has occurred:

- Events are assigned to particular elements and connected with script through the 'handler' element. The script is executed when the given event occurs.
- Event listeners as described in "Document Object Model Events" [DOM3-EVENTS] are defined which are invoked when a given event happens on a given object

Related sections of the spec:

- User interface events describes how an SVG user agent handles events such as pointer movements events (e.g., mouse movement) and activation events (e.g., mouse click).

15.6 Script Module

The Script Module contains the following element:

- script

15.7 Handler Module

The Handler Module contains the following element:

- handler

SVG Tiny 1.2 - 20050413 Top Contents Previous Next Elements Attributes

16 Animation

Contents

16.1 Introduction

SVG supports the ability to change vector graphics over time. SVG content can be animated in the following ways:

- Using SVG's animation elements. SVG document fragments can describe time-based modifications to the document's elements. Using the various animation elements, you can define motion paths, fade-in or fade-out effects, and objects that grow, shrink, spin or change color.
- Using the SVG uDOM. The SVG uDOM conforms to key aspects of the Document Object Model (DOM) Level 1, 2 and 3 specifications (DOM1, DOM2, DOM3). Some of the SVG attributes are accessible to scripting and by manipulating them many kind of animations can be achieved. The timer facilities in scripting languages such as ECMAScript can be used to start up and control the animations. (See example below.)
- SVG has been designed to allow future versions of SMIL [SMIL1] to use animated or static SVG content as media components.
- In the future, it is expected that SMIL will be modularized and that components of it could be used in conjunction with SVG and other XML grammars to achieve animation effects.

16.2 Animation elements

16.2.1 Overview

SVG's animation elements were developed in collaboration with the W3C Synchronized Multimedia (SYMM) Working Group, developers of the Synchronized Multimedia Integration Language (SMIL) 1.0 Specification [SMIL1].

The SYMM Working Group, in collaboration with the SVG Working Group, has authored the SMIL Animation specification [SMILANIM], which represents a general-purpose XML animation feature set. SVG incorporates the animation features defined in the SMIL Animation specification and provides some SVG-specific extensions.

For an introduction to the approach and features available in any language that supports SMIL Animation, see SMIL Animation overview and SMIL Animation animation model. For the list of animation features which go beyond SMIL Animation, see SVG extensions to SMIL Animation.

16.2.2 Relationship to SMIL Animation

SVG is a host language in terms of SMIL Animation and therefore introduces additional constraints and features as permitted by that specification. Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for SVG's animation elements and attributes is the SMIL Animation [SMILANIM] specification.

SVG supports the following four animation elements which are defined in the SMIL Animation specification:

'animate'	allows scalar attributes and properties to be assigned different values over time
'set'	a convenient shorthand for **'animate'**, which is useful for assigning animation values to non-numeric attributes and properties, such as the **'visibility'** property
'animateMotion'	moves an element along a motion path
'animateColor'	modifies the color value of particular attributes or properties over time

Additionally, SVG includes the following compatible extensions to SMIL Animation:

'animateTransform'	modifies one of SVG's transformation attributes over time, such as the **transform** attribute
path attribute	SVG allows any feature from SVG's path data syntax to be specified in a **path** attribute to the **'animateMotion'** element (SMIL Animation only allows a subset of SVG's path data syntax within a **path** attribute)
'mpath' element	SVG allows an **'animateMotion'** element to contain a child **'mpath'** element which references an SVG **'path'** element as the definition of the motion path
keyPoints attribute	SVG adds a **keyPoints** attribute to the **'animateMotion'** to provide precise control of the velocity of motion path animations
rotate attribute	SVG adds a **rotate** attribute to the **'animateMotion'** to control whether an object is automatically rotated so that its x-axis points in the same direction (or opposite direction) as the directional tangent vector of the motion path
discard element	SVG adds a discard element which removes the target element from the DOM tree at a specified time

For compatibility with other aspects of the language, SVG uses IRI references via an **xlink:href** attribute to identify the elements which are to be targets of the animations.

SMIL Animation requires that the host language define the meaning for document begin and the document end. Since an **'svg'** is sometimes the root of the XML document tree and other times can be a component of a parent XML grammar, the *document begin* for a given SVG document fragment is defined to be the exact time at which the **'svg'** element's load event is triggered. The *document end* of an SVG document fragment is the point at which the document fragment has been released and is no longer being processed by the user agent.

For SVG, the term presentation time indicates the position in the timeline relative to the *document begin* of a given document fragment.

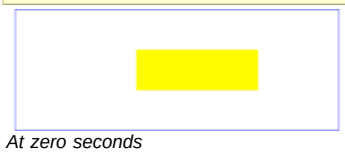
SVG defines more constrained error processing than is defined in the SMIL Animation [SMILANIM] specification. SMIL Animation defines error processing behavior where the document continues to run in certain error situations, whereas all animations within an SVG document fragment will stop in the event of any error within the document (see Error processing).

16.2.3 Animation elements example

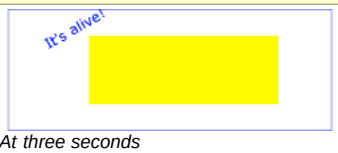
Example anim01 below demonstrates each of SVG's five animation elements.

Example: 19_01.svg

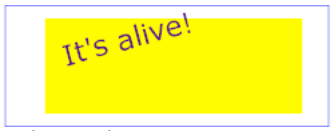
```
<?xml version="1.0" standalone="no"?>
<svg width="8cm" height="3cm"  viewBox="0 0 800 300"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example anim01 - demonstrate animation elements</desc>
  <rect x="1" y="1" width="798" height="298"
        fill="none" stroke="blue" stroke-width="2" />
  <!-- The following illustrates the use of the 'animate' element
        to animate a rectangles x, y, and width attributes so that
        the rectangle grows to ultimately fill the viewport. -->
  <rect id="RectElement" x="300" y="100" width="300" height="100"
        fill="rgb(255,255,0)"  >
    <animate attributeName="x"
             begin="0s" dur="9s" fill="freeze" from="300" to="0" />
    <animate attributeName="y"
             begin="0s" dur="9s" fill="freeze" from="100" to="0" />
    <animate attributeName="width"
             begin="0s" dur="9s" fill="freeze" from="300" to="800" />
    <animate attributeName="height"
             begin="0s" dur="9s" fill="freeze" from="100" to="300" />
  </rect>
  <!-- Set up a new user coordinate system so that
        the text string's origin is at (0,0), allowing
        rotation and scale relative to the new origin -->
  <g transform="translate(100,100)" >
    <!-- The following illustrates the use of the 'set', 'animateMotion',
         'animateColor' and 'animateTransform' elements. The 'text' element
         below starts off hidden (i.e., invisible). At 3 seconds, it:
           * becomes visible
           * continuously moves diagonally across the viewport
           * changes color from blue to dark red
           * rotates from -30 to zero degrees
           * scales by a factor of three. -->
    <text id="TextElement" x="0" y="0"
          font-family="Verdana" font-size="35.27" visibility="hidden"  >
      It's alive!
      <set attributeName="visibility" to="visible"
           begin="3s" dur="6s" fill="freeze" />
      <animateMotion path="M 0 0 L 100 100"
           begin="3s" dur="6s" fill="freeze" />
      <animateColor attributeName="fill"
           from="rgb(0,0,255)" to="rgb(128,0,0)"
           begin="3s" dur="6s" fill="freeze" />
      <animateTransform attributeName="transform"
           type="rotate" from="-30" to="0"
           begin="3s" dur="6s" fill="freeze" />
      <animateTransform attributeName="transform"
           type="scale" from="1" to="3" additive="sum"
           begin="3s" dur="6s" fill="freeze" />
    </text>
  </g>
</svg>
```



At zero seconds



At three seconds



At six seconds

At nine seconds

The sections below describe the various animation attributes and elements.

16.2.4 Attributes to identify the target element for an animation

The following attributes are common to all animation elements and identify the target element for the animation.

Schema: animatecommon

```
  <define name='svg.AnimateCommon.attr'>
    <ref name='svg.XLink.attr'/>
  </define>
```

Attribute definitions:

`xlink:href = " <iri> "`

An IRI reference to the element which is the target of this animation and which therefore will be modified over time.

The target element must be part of the current SVG document fragment.

<iri> must point to exactly one target element which is capable of being the target of the given animation. If <iri> points to multiple target elements, if the given target element is not capable of being a target of the given animation, or if the given target element is not part of the current SVG document fragment, then the document is in error (see Error processing).

If the **xlink:href** attribute is not provided, then the target element will be the immediate parent element of the current animation element.

Refer to the descriptions of the individual animation elements for any restrictions on what types of elements can be targets of particular types of animations.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: Specifying the animation target.

Animatable: no.

16.2.5 Attributes to identify the target attribute or property for an animation

The following attributes identify the target attribute or property for the given target element whose value changes over time.

Schema: animateattribute

```
<define name='svg.AnimateAttributeCommon.attr'>
  <attribute name='attributeName'  svg:animatable='false' svg:inheritable='false'><text/></attribute>
  <optional>
    <attribute name='attributeType' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>XML</value>
        <value>CSS</value>
        <value>auto</value>
      </choice>
    </attribute>
  </optional>
</define>
```

Attribute definitions:

attributeName = <attributeName>
Specifies the name of the target attribute. An XMLNS prefix may be used to indicate the XML namespace for the attribute. The prefix will be interpreted in the scope of the current (i.e., the referencing) animation element.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: Specifying the animation target.

Animatable: no.

attributeType = "CSS | XML | auto"
Specifies the namespace in which the target attribute and its associated values are defined. The attribute value is one of the following (values are case-sensitive):

"CSS"
This specifies that the value of "attributeName" is the name of a CSS property defined as animatable in this specification.

"XML"
This specifies that the value of "attributeName" is the name of an XML attribute defined in the default XML namespace for the target element. If the value for `attributeName` has an XMLNS prefix, the implementation must use the associated namespace as defined in the scope of the target element. The attribute must be defined as animatable in this specification.

"auto"
The implementation should match the `attributeName` to an attribute for the target element. The implementation must first search through the list of CSS properties for a matching property name, and if none is found, search the default XML namespace for the element.

The default value is "auto".

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: Specifying the animation target.

Paced animation and complex types

Paced animations assume a notion of distance between the various animation values defined by the to, from, by and values attributes. The following table explains how the distance between values of different types should be computed.

Distance is defined for types which can be expressed as a list of values, where each value is a vector of scalars in an n dimentional space. For example, an angle value is a list of one value in a 1 dimensional space and a color is a list of 1 value in a 3 dimensional space.

The following table uses the following notation to describe two values for which a distance should be computed:

Va = $\{v_{a0}, v_{a1}, ..., v_{an}\}$

Vb = $\{v_{b0}, v_{b1}, ..., v_{bn}\}$

Value Type	Description	Distance	Examples
angle, integer, length, percentage, coordinate	single 1 dimensional value. $v_{a0}[0]$ = scalarA $v_{b0}[0]$ = scalarB	||VaVb|| = abs(scalarA- scalarB)	x attribute on <rect> stroke-width on <circle>
color	single 3 dimensional value. $v_{a0}[0]$ = colorA $v_{b0}[0]$ = colorB	||VaVb|| = sqrt((colorA.getRed() - colorB.getRed())^2 + (colorA.getGreen() - colorB.getGreen())^2 + (colorA.getBlue() - colorB.getBlue())^2)	fill attribute on <ellipse>

list of length	n 1 dimensional values.	llVaVbll = sum(for i = 1 to n, abs(v_{ai}[0] - v_{bi}[0])) / n	stroke-dasharray on <path>
list of points	n 2 dimensional values	llVaVbll = sum(for i = 1 to n, dist(v_{ai},v_{bi})) / n dist(v_{ai},v_{bi}) = sqrt((v_{ai}[0] - v_{bi}[0])^2 + (v_{ai}[1] - v_{bi}[1])^2))	points on <polygon>
path	n 2 dimensional values where each value is a control point in the path definition.	llVaVbll = sum(for i = 1 to n, dist(v_{ai},v_{bi})) / n dist(v_{ai},v_{bi}) = sqrt((v_{ai}[0] - v_{bi}[0])^2 + (v_{ai}[1] - v_{bi}[1])^2))	d on <path>
transform list	***type: translate*** one 2 dimensional value v_{a0}[0] = txa v_{a0}[1] = tya v_{b0}[0] = txb v_{b0}[1] = tyb ***type: rotate*** one 1 dimensional value and 1 2 dimensional value v_{a0}[0] = angleA v_{a1}[0] = cxa v_{a1}[1] = cya v_{b0}[0] = angleB v_{b1}[0] = cxb v_{b1}[1] = cyb ***type: scale*** two 1 dimensional values v_{a0}[0] = scaleXa v_{a1}[0] = scaleYa v_{b0}[0] = scaleXb v_{b1}[0] = scaleYb ***type: skewX, skewY*** single 1 dimension value v_{a0}[0] = skewXorYa v_{b0}[0] = skewXorYb	***type: translate*** llVaVbll = dist(v_a0 ,v_b0) = sqrt((v_a0 [0] - v_b0 [0])^2 + (v_a0 [1] - v_b0 [1])^2)) ***type: rotate*** llVaVbll = (abs(angleA - angleB) + sqrt((v_a1 [0] - v_b1 [0])^2 + (v_a1 [1] - v_b1 [1])^2))) / 2 ***type: scale*** llVaVbll = (abs(scaleXa- scaleXb) + abs(scaleYa - scaleYb)) / 2 ***type: skewX, skewY*** llVaVbll = abs(skewXorYa- skewXorYb)	transform attribute on <g> using <animateTransform>

16.2.6 Attributes to control the timing of the animation

The following attributes are common to all animation elements and control the timing of the animation, including what causes the animation to start and end, whether the animation runs repeatedly, and whether to retain the end state the animation once the animation ends.

The timing attributes also applies to Media Elements.

Schema: animatetiming

```
<define name='svg.AnimateBegin.attr' combine='interleave'>
  <optional><attribute name='begin' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
</define>

<define name='svg.AnimateTimingNoFillNoMinMax.attr' combine='interleave'>
  <ref name='svg.AnimateBegin.attr'/>
  <optional><attribute name='dur' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='end' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='repeatCount' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='repeatDur' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional>
    <attribute name='restart' a:defaultValue='always' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>always</value>
```

```
          <value>never</value>
          <value>whenNotActive</value>
        </choice>
      </attribute>
    </optional>
  </define>

  <define name='svg.AnimateTimingNoMinMax.attr' combine='interleave'>
    <ref name='svg.AnimateTimingNoFillNoMinMax.attr'/>
    <optional>
      <attribute name='fill' a:defaultValue='remove' svg:animatable='false' svg:inheritable='false'>
        <choice>
          <value>remove</value>
          <value>freeze</value>
        </choice>
      </attribute>
    </optional>
  </define>

  <define name='svg.AnimateTiming.attr' combine='interleave'>
    <ref name='svg.AnimateTimingNoMinMax.attr'/>
    <optional><attribute name='min' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
    <optional><attribute name='max' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  </define>
```

In the syntax specifications that follow, optional white space is indicated as "S", defined as follows:

```
S ::= (#x20 | #x9 | #xD | #xA)*
```

Attribute definitions:

begin : begin-value-list
Defines when the element should begin (i.e. become active).

The attribute value is a semicolon separated list of values.

begin-value-list ::= begin-value (S? ";" S? begin-value-list)?
A semicolon separated list of begin values. The interpretation of a list of begin times is detailed in SMIL Animation's section on "Evaluation of begin and end time lists".
begin-value : (offset-value | syncbase-value | event-value | repeat-value | accessKey-value | "indefinite")
Describes the element begin.
offset-value ::= (S? "+" | "-" S?)? (Clock-value)
For SMIL Animation, this describes the element begin as an offset from an implicit syncbase. For SVG, the implicit syncbase begin is defined to be relative to the document begin. Negative begin times are entirely valid and easy to compute, as long as there is a resolved document begin time.
syncbase-value ::= (Id-value "." ("begin" | "end")) (S? ("+"|"-") S? Clock-value)?
Describes a syncbase and an optional offset from that syncbase. The element begin is defined relative to the begin or active end of another animation. A syncbase consists of an ID reference to another animation element followed by either `begin` or `end` to identify whether to synchronize with the beginning or active end of the referenced animation element.
event-value ::= (Id-value ".")? (event-ref) (S? ("+"|"-") S? Clock-value)?
Describes an event and an optional offset that determine the element begin. The animation begin is defined relative to the time that the event is raised. The list of event-symbols available for a given event-base element is listed in the 'Animation event name' column in Complete list of supported events. Details of event-based timing are described in SMIL Animation: Unifying Event-based and Scheduled Timing.
repeat-value ::= (Id-value ".")? "repeat(" integer ")" (S? ("+"|"-") S? Clock-value)?
Describes a qualified repeat event. The element begin is defined relative to the time that the repeat event is raised with the specified iteration value.
accessKey-value ::= "accessKey(" character ")" (S? ("+"|"-") S? Clock-value)?
Describes an accessKey that determines the element begin. The element begin is defined relative to the time of the keydown event corresponding to the specified key. From a formal processing model perspective, accessKey is a keydown event listener on the document which behaves as if stopPropagation() and preventDefault() have both been invoked. The "character" value can be any of the keyboard event identifier strings listed in Appendix A.2 of the DOM3 Events specification [DOM3Events]. user.
"indefinite"
The begin of the animation will be determined by a "beginElement()" method call or a hyperlink targeted to the element.

The animation UDOM methods are described in the ElementTimeControl interface.

Hyperlink-based timing is described in SMIL Animation: Hyperlinks and timing.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'begin' attribute.

Animatable: no.
dur : Clock-value | "media" | "indefinite"
Specifies the simple duration.

The attribute value can be either of the following:

Clock-value
Specifies the length of the simple duration in presentation time. Value must be greater than 0.
"media"
Specifies the simple duration as the intrinsic media duration. This is only valid for elements that define media.

(For SVG's animation elements, if **"media"** is specified, the attribute will be ignored.)
"indefinite"
Specifies the simple duration as indefinite.

If the animation does not have a **dur** attribute, the simple duration is indefinite. Note that interpolation will not work if the simple duration is indefinite (although this may still be useful for 'set' elements). Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'dur' attribute.

Animatable: no.
end : end-value-list
Defines an end value for the animation that can constrain the active duration. The attribute value is a semicolon separated list of values.

end-value-list ::= end-value (S? ";" S? end-value-list)?
A semicolon separated list of end values. The interpretation of a list of end times is detailed below.

end-value : (offset-value | syncbase-value | event-value | repeat-value | accessKey-value | "indefinite")
Describes the active end of the animation.

A value of **"indefinite"** specifies that the end of the animation will be determined by a "endElement()" method call (the animation UDOM methods are described in ElementTimeControl interface.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see description of SMIL Animation: 'end' attribute.

Animatable: no.

min : Clock-value | "media"
Specifies the minimum value of the active duration.

The attribute value can be either of the following:

Clock-value
Specifies the length of the minimum value of the active duration, measured in local time.

Value must be greater than 0.

"media"
Specifies the minimum value of the active duration as the intrinsic media duration. This is only valid for elements that define media. (For SVG's animation elements, if **"media"** is specified, the attribute will be ignored.)

The default value for **min** is "0". This does not constrain the active duration at all.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'min' attribute.

Animatable: no.

max : Clock-value | "media"
Specifies the maximum value of the active duration.

The attribute value can be either of the following:

Clock-value
Specifies the length of the maximum value of the active duration, measured in local time.

Value must be greater than 0.

"media"
Specifies the maximum value of the active duration as the intrinsic media duration. This is only valid for elements that define media. (For SVG's animation elements, if **"media"** is specified, the attribute will be ignored.)

There is no default value for **max**. This does not constrain the active duration at all.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'max' attribute.

Animatable: no.

restart : "always" | "whenNotActive" | "never"

always
The animation can be restarted at any time.

This is the default value.

whenNotActive
The animation can only be restarted when it is not active (i.e. after the active end). Attempts to restart the animation during its active duration are ignored.

never
The element cannot be restarted for the remainder of the current simple duration of the parent time container. (In the case of SVG, since the parent time container is the SVG document fragment, then the animation cannot be restarted for the remainder of the document duration.)

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'restart' attribute.

Animatable: no.

repeatCount : numeric value | "indefinite"
Specifies the number of iterations of the animation function. It can have the following attribute values:

numeric value
This is a (base 10) "floating point" numeric value that specifies the number of iterations. It can include partial iterations expressed as fraction values. A fractional value describes a portion of the simple duration. Values must be greater than 0.

"indefinite"
The animation is defined to repeat indefinitely (i.e. until the document ends).

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'repeatCount' attribute.

Animatable: no.

repeatDur : Clock-value | "indefinite"
Specifies the total duration for repeat. It can have the following attribute values:

Clock-value
Specifies the duration in presentation time to repeat the animation function `f(t)` .

"indefinite"
The animation is defined to repeat indefinitely (i.e. until the document ends).

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'repeatDur' attribute.

Animatable: no.

fill : "freeze" | "remove"

This attribute can have the following values:

freeze
The animation effect F(t) is defined to freeze the effect value at the last value of the active duration. The animation effect is "frozen" for the remainder of the document duration (or until the animation is restarted - see SMIL Animation: Restarting animation).

remove
The animation effect is removed (no longer applied) when the active duration of the animation is over. After the active end of the animation, the animation no longer affects the target (unless the animation is restarted - see SMIL Animation: Restarting animation).

This is the default value.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'fill' attribute.

Animatable: no.

The SMIL Animation [SMILANIM] specification defines the detailed processing rules associated with the above attributes. Except for any SVG-specific rules explicitly mentioned in this specification, the SMIL Animation [SMILANIM] specification is the normative definition of the processing rules for the above attributes.

Clock values

Clock values have a subsetted syntax of the clock values syntax in SMIL Animation [SMILANIM]:

```
Clock-val          ::= Timecount-val
Timecount-val      ::= Timecount ("." Fraction)? (Metric)?
Metric             ::= "s" | "ms"
Fraction           ::= DIGIT+
Timecount          ::= DIGIT+
DIGIT              ::= [0-9]
```

For Timecount values, the default metric suffix is "s" (for seconds). No embedded white space is allowed in clock values, although leading and trailing white space characters will be ignored.

Clock values describe presentation time.

The following are examples of legal clock values:

- Timecount values:

  `30s` = 30 seconds

  `5ms` = 5 milliseconds

  `12.467` = 12 seconds and 467 milliseconds

Fractional values are just (base 10) floating point definitions of seconds. Thus:

```
00.5s = 500 milliseconds
```

16.2.7 Attributes that define animation values over time

The following attributes are common to elements 'animate' , 'animateMotion' , 'animateColor' and 'animateTransform' . These attributes define the values that are assigned to the target attribute or property over time. The attributes below provide control over the relative timing of keyframes and the interpolation method between discrete values.

Schema: animatevalue

```
<define name='svg.AnimateToCommon.attr' combine='interleave'>
  <optional><attribute name='to' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
</define>

<define name='svg.AnimateValueCommon.attr'>
  <ref name='svg.AnimateToCommon.attr'/>
  <optional>
    <attribute name='calcMode' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>discrete</value>
        <value>linear</value>
        <value>paced</value>
        <value>spline</value>
      </choice>
    </attribute>
  </optional>
  <optional><attribute name='values' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='keyTimes' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='keySplines' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='from' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='by' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
</define>
```

Attribute definitions:

`calcMode` = "discrete | linear | paced | spline"
Specifies the interpolation mode for the animation. This can take any of the following values. The default mode is "linear", however if the attribute does not support linear interpolation (e.g. for strings), the `calcMode` attribute is ignored and discrete interpolation is used.

`discrete`
This specifies that the animation function will jump from one value to the next without any interpolation.

linear
Simple linear interpolation between values is used to calculate the animation function. Except for 'animateMotion', this is the default `calcMode`.
paced
Defines interpolation to produce an even pace of change across the animation. This is only supported for values that define a linear numeric range, and for which some notion of "distance" between points can be calculated (e.g. position, width, height, etc.). If "`paced`" is specified, any `keyTimes` or `keySplines` will be ignored. For 'animateMotion', this is the default `calcMode`.
spline
Interpolates from one value in the `values` list to the next according to a time function defined by a cubic Bzier spline. The points of the spline are defined in the `keyTimes` attribute, and the control points for each interval are defined in the `keySplines` attribute.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'calcMode' attribute.

Animatable: no.

values = "<list>"
A semicolon-separated list of one or more values. Vector-valued attributes are supported using the vector syntax of the `attributeType` domain. Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'values' attribute.

Animatable: no.

keyTimes = "<list>"
A semicolon-separated list of time values used to control the pacing of the animation. Each time in the list corresponds to a value in the `values` attribute list, and defines when the value is used in the animation function. Each time value in the `keyTimes` list is specified as a floating point value between 0 and 1 (inclusive), representing a proportional offset into the simple duration of the animation element.

If a list of `keyTimes` is specified, there must be exactly as many values in the `keyTimes` list as in the `values` list.

Each successive time value must be greater than or equal to the preceding time value.

The `keyTimes` list semantics depends upon the interpolation mode:

- For linear and spline animation,the first time value in the list must be 0, and the last time value in the list must be 1. The `keyTime` associated with each value defines when the value is set; values are interpolated between the `keyTimes`.
- For discrete animation, the first time value in the list must be 0. The time associated with each value defines when the value is set; the animation function uses that value until the next time defined in `keyTimes`.

If the interpolation mode is "paced", the `keyTimes` attribute is ignored.

If there are any errors in the `keyTimes` specification (bad values, too many or too few values), the document fragment is in error (see error processing).

If the simple duration is indefinite, any `keyTimes` specification will be ignored.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'keyTimes' attribute.

Animatable: no.

keySplines = "<list>"
A set of Bzier control points associated with the `keyTimes` list, defining a cubic Bzier function that controls interval pacing. The attribute value is a semicolon separated list of control point descriptions. Each control point description is a set of four values: `x1 y1 x2 y2`, describing the Bzier control points for one time segment. The `keyTimes` values that define the associated segment are the Bzier "anchor points", and the `keySplines` values are the control points. Thus, there must be one fewer sets of control points than there are `keyTimes`.

The values must all be in the range 0 to 1.

This attribute is ignored unless the `calcMode` is set to "spline".

If there are any errors in the `keySplines` specification (bad values, too many or too few values), the document fragment is in error (see error processing).

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'keySplines' attribute.

Animatable: no.

from = "<value>"
Specifies the starting value of the animation.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'from' attribute.

Animatable: no.

to = "<value>"
Specifies the ending value of the animation.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'to' attribute.

Animatable: no.

by = "<value>"
Specifies a relative offset value for the animation.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'by' attribute.

Animatable: no.

The SMIL Animation [SMILANIM] specification defines the detailed processing rules associated with the above attributes. Except for any SVG-specific rules explicitly mentioned in this specification, the SMIL Animation [SMILANIM] specification is the normative definition of the processing rules for the above attributes.

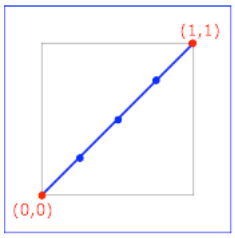
The animation values specified in the animation element must be legal values for the specified attribute. Leading and trailing white space, and white space before and after semicolon separators, will be ignored.

All values specified must be legal values for the specified attribute (as defined in the associated namespace). If any values are not legal, the document fragment is in error (see error processing).

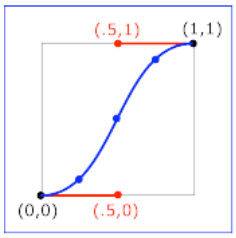
If a list of values is used, the animation will apply the values in order over the course of the animation. If a list of *values* is specified, any *from*, *to* and *by* attribute values are ignored.

The processing rules for the variants of *from/by/to* animations are described in Animation function values.

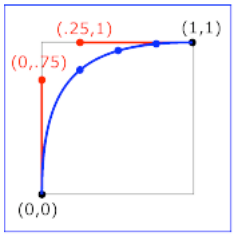
The following figure illustrates the interpretation of the `keySplines` attribute. Each diagram illustrates the effect of `keySplines` settings for a single interval (i.e. between the associated pairs of values in the `keyTimes` and `values` lists.). The horizontal axis can be thought of as the input value for the *unit progress* of interpolation within the interval - i.e. the pace with which interpolation proceeds along the given interval. The vertical axis is the resulting value for the *unit progress*, yielded by the `keySplines` function. Another way of describing this is that the horizontal axis is the input *unit time* for the interval, and the vertical axis is the output *unit time*. See also the section Timing and real-world clock times.



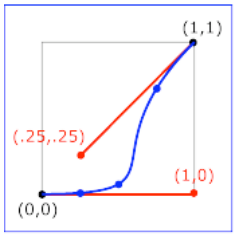
keySplines="0011" (the default)



keySplines=".50.51"



keySplines="0.75.251"



keySplines="10.25.25"

Examples of keySplines

To illustrate the calculations, consider the simple example:

```
<animate dur="4s" values="10; 20" keyTimes="0; 1"
     calcMode="spline" keySplines={as in table} />
```

Using the keySplines values for each of the four cases above, the approximate interpolated values as the animation proceeds are:

keySplines values	Initial value	After 1s	After 2s	After 3s	Final value
0 0 1 1	10.0	12.5	15.0	17.5	20.0
.5 0 .5 1	10.0	11.0	15.0	19.0	20.0
0 .75 .25 1	10.0	18.0	19.3	19.8	20.0
1 0 .25 .25	10.0	10.1	10.6	16.9	20.0

For a formal definition of Bzier spline calculation, see [FOLEY-VANDAM].

16.2.8 Attributes that control whether animations are additive

It is frequently useful to define animation as an offset or delta to an attribute's value, rather than as absolute values. A simple "grow" animation can increase the width of an object by 10 pixels:

```
<rect width="20px" ...>
  <animate attributeName="width" from="0px" to="10px" dur="10s"
           additive="sum"/>
</rect>
```

It is frequently useful for repeated animations to build upon the previous results, accumulating with each interation. The following example causes the rectangle to continue to grow with each repeat of the animation:

```
<rect width="20px" ...>
  <animate attributeName="width" from="0px" to="10px" dur="10s"
           additive="sum" accumulate="sum" repeatCount="5"/>
</rect>
```

At the end of the first repetition, the rectangle has a width of 30 pixels. At the end of the second repetition, the rectangle has a width of 40 pixels. At the end of the fifth repetition, the rectangle has a width of 70 pixels.

For more information about additive animations, see SMIL Animation: Additive animation. For more information on cumulative animations, see SMIL Animation: Controlling behavior of repeating animation - Cumulative animation.

The following attributes are common to elements 'animate' , 'animateMotion' , 'animateColor' and 'animateTransform' .

Schema: animateaddition

```
<define name='svg.AnimateAdditionCommon.attr'>
  <optional>
    <attribute name='additive' a:defaultValue='replace' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>replace</value>
        <value>sum</value>
      </choice>
    </attribute>
  </optional>
  <optional>
    <attribute name='accumulate' a:defaultValue='none' svg:animatable='false' svg:inheritable='false'>
      <choice>
        <value>none</value>
        <value>sum</value>
      </choice>
    </attribute>
  </optional>
</define>
```

Attribute definitions:

additive = "replace | sum"
Controls whether or not the animation is additive.

sum
Specifies that the animation will add to the underlying value of the attribute and other lower priority animations.
replace
Specifies that the animation will override the underlying value of the attribute and other lower priority animations. This is the default, however the behavior is also affected by the animation value attributes **by** and **to** , as described in SMIL Animation: How from, to and by attributes affect additive behavior.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'additive' attribute.

Animatable: no.
accumulate = "none | sum"
Controls whether or not the animation is cumulative.

sum
Specifies that each repeat iteration after the first builds upon the last value of the previous iteration.
none
Specifies that repeat iterations are not cumulative. This is the default.

This attribute is ignored if the target attribute value does not support addition, or if the animation element does not repeat.

Cumulative animation is not defined for "*to animation*".

This attribute will be ignored if the animation function is specified with only the `to` attribute.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this attribute is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'accumulate' attribute.

Animatable: no.

16.2.9 Inheritance

SVG allows both attributes and properties to be animated. If a given attribute or property is inheritable by descendants, then animations on a parent element such as a 'g' element has the effect of propagating the attribute or property animation values to descendant elements as the animation proceeds; thus, descendant elements

can inherit animated attributes and properties from their ancestors.

16.2.10 The 'animate' element

The 'animate' element is used to animate a single attribute or property over time. For example, to make a rectangle repeatedly fade away over 5 seconds, you can specify:

```
<rect>
  <animate attributeType="CSS" attributeName="opacity"
         from="1" to="0" dur="5s" repeatCount="indefinite" />
</rect>
```

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this element is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'animate' element.

Schema: animate

```
<define name='animate'>
  <element name='animate'>
    <ref name='animate.AT'/>
    <zeroOrMore><ref name='animateCommon.CM'/></zeroOrMore>
  </element>
</define>

<define name='animate.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.AnimateCommon.attr'/>
  <ref name='svg.AnimateAttributeCommon.attr'/>
  <ref name='svg.AnimateTiming.attr'/>
  <ref name='svg.AnimateValueCommon.attr'/>
  <ref name='svg.AnimateAdditionCommon.attr'/>
</define>
```

For a list of attributes and properties that can be animated using the 'animate' element, see Attributes and properties that can be animated.

16.2.11 The 'set' element

The 'set' element provides a simple means of just setting the value of an attribute for a specified duration. It supports all attribute types, including those that cannot reasonably be interpolated, such as string and boolean values. The 'set' element is non-additive. The additive and accumulate attributes are not allowed, and will be ignored if specified.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this element is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'set' element.

Schema: set

```
<define name='set'>
  <element name='set'>
    <ref name='set.AT'/>
    <zeroOrMore><ref name='animateCommon.CM'/></zeroOrMore>
  </element>
</define>

<define name='set.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.AnimateCommon.attr'/>
  <ref name='svg.AnimateAttributeCommon.attr'/>
  <ref name='svg.AnimateTiming.attr'/>
  <ref name='svg.AnimateToCommon.attr'/>
</define>
```

Attribute definitions:

`to` = "<value>"
Specifies the value for the attribute during the duration of the 'set' element. The argument value must match the attribute type.

Animatable: no.

For a list of attributes and properties that can be animated using the 'set' element, see Attributes and properties that can be animated.

16.2.12 The 'animateMotion' element

The 'animateMotion' element causes a referenced element to move along a motion path.

The following lists all of the elements which can be animated by the 'animateMotion' element:

- 'g'
- 'defs'
- 'use'
- 'image'
- 'switch'
- 'path'

- 'rect'
- 'circle'
- 'ellipse'
- 'line'
- 'polyline'
- 'polygon'
- 'text'
- 'animation'
- 'video'
- 'a'
- 'foreignObject'

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this element is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'animateMotion' element.

Schema: animateMotion

```
<define name='animateMotion'>
  <element name='animateMotion'>
    <ref name='animateMotion.AT'/>
    <zeroOrMore>
      <ref name='animateCommon.CM'/>
    </zeroOrMore>
    <optional>
      <ref name='mpath'/>
    </optional>
    <zeroOrMore>
      <ref name='animateCommon.CM'/>
    </zeroOrMore>
  </element>
</define>

<define name='animateMotion.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.AnimateCommon.attr'/>
  <ref name='svg.AnimateTiming.attr'/>
  <ref name='svg.AnimateAdditionCommon.attr'/>
  <ref name='svg.AnimateValueCommon.attr'/>
  <ref name='svg.AnimateTypeCommon.attr'/>
  <optional><attribute name='path' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='keyPoints' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='rotate' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
  <optional><attribute name='origin' svg:animatable='false' svg:inheritable='false'><text/></attribute></optional>
</define>
```

Attribute definitions:

`calcMode` = "discrete | linear | paced | spline"
Specifies the interpolation mode for the animation. Refer to general description of the calcMode attribute above. The only difference is that the default value for the **calcMode** for **'animateMotion'** is **paced**. See SMIL Animation: 'calcMode' attribute for 'animateMotion'.

Animatable: no.

`path` = "<path-data>"
The motion path, expressed in the same format and interpreted the same way as the d= attribute on the 'path' element. The effect of a motion path animation is to add a supplemental transformation matrix onto the CTM for the referenced object which causes a translation along the x- and y-axes of the current user coordinate system by the computed X and Y values computed over time.

Animatable: no.

`keyPoints` = "<list-of-numbers>"
keyPoints takes a semicolon-separated list of floating point values between 0 and 1 and indicates how far along the motion path the object shall move at the moment in time specified by corresponding **keyTimes** value. Distance calculations use the user agent's distance along the path algorithm. Each progress value in the list corresponds to a value in the `keyTimes` attribute list.

If a list of `keyPoints` is specified, there must be exactly as many values in the `keyPoints` list as in the `keyTimes` list.

If there are any errors in the `keyPoints` specification (bad values, too many or too few values), then the document is in error (see Error processing).

Animatable: no.

`rotate` = "<angle> | auto | auto-reverse"
auto indicates that the object is rotated over time by the angle of the direction (i.e., directional tangent vector) of the motion path. **auto-reverse** indicates that the object is rotated over time by the angle of the direction (i.e., directional tangent vector) of the motion path plus 180 degrees. An actual angle value can also be given, which represents an angle relative to the x-axis of current user coordinate system. The **rotate** attribute adds a supplemental transformation matrix onto the CTM to apply a rotation transformation about the origin of the current user coordinate system. The rotation transformation is applied after the supplemental translation transformation that is computed due to the path attribute. The default value is 0.

Animatable: no.

`origin` = "default"
The **origin** attribute is defined in the SMIL Animation specification [SMILANIM-ATTR-ORIGIN]. It has no effect in SVG.

Animatable: no.

For **'animateMotion'**, the specified values for from , by , to and values consists of x, y coordinate pairs, with a single comma and/or white space separating the x coordinate from the y coordinate. For example, **from="33,15"** specifies an x coordinate value of **33** and a y coordinate value of **15**.

If provided, the values attribute must consists of a list of x, y coordinate pairs. Coordinate values are separated by at least one white space character or a comma. Additional white space around the separator is allowed. For example, **values="10,20;30,20;30,40"** or **values="10mm,20mm;30mm,20mm;30mm,40mm"**. Each coordinate represents a length. Attributes from , by , to and values specify a shape on the current canvas which represents the motion path.

Two options are available which allow definition of a motion path using any of SVG's path data commands:

- the path attribute defines a motion path directly on 'animateMotion' element using any of SVG's path data commands.
- the 'mpath' sub-element provides the ability to reference an external 'path' element as the definition of the motion path.

Note that SVG's path data commands can only contain values in user space, whereas from , by , to and values can specify coordinates in user space or using unit identifiers. See Units.

The various (x,y) points of the shape provide a supplemental transformation matrix onto the CTM for the referenced object which causes a translation along the x- and y-axes of the current user coordinate system by the (x,y) values of the shape computed over time. Thus, the referenced object is translated over time by the offset of the motion path relative to the origin of the current user coordinate system. The supplemental transformation is applied on top of any transformations due to the target element's transform attribute or any animations on that attribute due to 'animateTransform' elements on the target element.

The additive and accumulate attributes apply to 'animateMotion' elements. Multiple 'animateMotion' elements all simultaneously referencing the same target element can be additive with respect to each other; however, the transformations which result from the 'animateMotion' elements are always supplemental to any transformations due to the target element's transform attribute or any 'animateTransform' elements.

The default calculation mode (calcMode) for animateMotion is "paced". This will produce constant velocity motion along the specified path. Note that while animateMotion elements can be additive, it is important to observe that the addition of two or more "paced" (constant velocity) animations might not result in a combined motion animation with constant velocity.

When a path is combined with "discrete", "linear" or "spline" calcMode settings, and if attribute keyPoints is not provided, the number of values is defined to be the number of points defined by the path, unless there are "move to" commands within the path. A "move to" command within the path (i.e. other than at the beginning of the path description) A "move to" command does not count as an additional point when dividing up the duration, or when associating `keyTimes`, `keySplines` and `keyPoints` values. When a path is combined with a "paced" calcMode setting, all "move to" commands are considered to have 0 length (i.e. they always happen instantaneously), and is not considered in computing the pacing.

For more flexibility in controlling the velocity along the motion path, the keyPoints attribute provides the ability to specify the progress along the motion path for each of the keyTimes specified values. If specified, keyPoints causes keyTimes to apply to the values in keyPoints rather than the points specified in the values attribute array or the points on the path attribute.

The override rules for 'animateMotion are as follows. Regarding the definition of the motion path, the 'mpath' element overrides the the path attribute, which overrides values, which overrides from/by/to. Regarding determining the points which correspond to the keyTimes attributes, the keyPoints attribute overrides path, which overrides values, which overrides from/by/to.

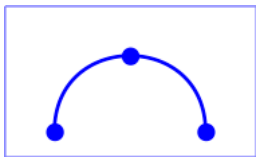
At any time *t* within a motion path animation of duration *dur*, the computed coordinate (x,y) along the motion path is determined by finding the point (x,y) which is *t/dur* distance along the motion path using the user agent's distance along the path algorithm.

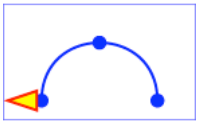
The following example demonstrates the supplemental transformation matrices that are computed during a motion path animation.

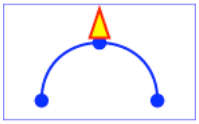
Example animMotion01 shows a triangle moving along a motion path.

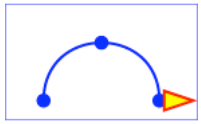
Example: 19_02.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="5cm" height="3cm"  viewBox="0 0 500 300"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny"
     xmlns:xlink="http://www.w3.org/1999/xlink" >
  <desc>Example animMotion01 - demonstrate motion animation computations</desc>
  <rect x="1" y="1" width="498" height="298"
        fill="none" stroke="blue" stroke-width="2" />
  <!-- Draw the outline of the motion path in blue, along
         with three small circles at the start, middle and end. -->
  <path id="path1" d="M100,250 C 100,50 400,50 400,250"
        fill="none" stroke="blue" stroke-width="7.06"  />
  <circle cx="100" cy="250" r="17.64" fill="blue"  />
  <circle cx="250" cy="100" r="17.64" fill="blue"  />
  <circle cx="400" cy="250" r="17.64" fill="blue"  />
  <!-- Here is a triangle which will be moved about the motion path.
       It is defined with an upright orientation with the base of
       the triangle centered horizontally just above the origin. -->
  <path d="M-25,-12.5 L25,-12.5 L 0,-87.5 z"
        fill="yellow" stroke="red" stroke-width="7.06"  >
    <!-- Define the motion path animation -->
    <animateMotion dur="6s" repeatCount="indefinite" rotate="auto" >
       <mpath xlink:href="#path1"/>
    </animateMotion>
  </path>
</svg>
```




At zero seconds


At three seconds


At six seconds

The following table shows the supplemental transformation matrices that are applied to achieve the effect of the motion path animation.

	After 0s	After 3s	After 6s
Supplemental transform due to movement along motion path	translate(100,250)	translate(250,100)	translate(400,250)
Supplemental transform due to	rotate(-90)	rotate(0)	rotate(90)

rotate="auto"			

16.2.13 The 'mpath' element

The 'mpath' element is a sub element to the 'animateMotion' element (its only place in the document tree is as a child of an 'animateMotion'). 'mpath' reference an external 'path' element that will serve as the definition of the motion path.

Example:

Example: mpath01.svg

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.2" baseProfile="tiny" xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        width="100%" height="100%" viewBox="0 0 80 60">
        <desc>mpath example</desc>
        <path id="mpathRef" d="M15,43 C15,43 36,20 65,33" fill="none" stroke="black" stroke-width="1"/>
        <animateMotion begin="0s" dur="6s" calcMode="linear" fill="freeze">
                <mpath xlink:href="#mpathRef"/>
        </animateMotion>
</svg>
```

Schema: mpath

```
  <define name='mpath'>
    <element name='mpath'>
      <ref name='mpath.AT'/>
      <zeroOrMore><ref name='svg.Desc.group'/></zeroOrMore>
    </element>
  </define>

  <define name='mpath.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.XLinkRequired.attr'/>
  </define>
```

Attribute definitions:

xlink:href = " <iri> "
A URI reference to the 'path' element which defines the motion path.

Animatable: no.

16.2.14 The 'animateColor' element

The 'animateColor' element specifies a color transformation over time.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for this element is the SMIL Animation [SMILANIM] specification. In particular, see SMIL Animation: 'animateColor' element.

Schema: animateColor

```
  <define name='animateColor'>
    <element name='animateColor'>
      <ref name='animateColor.AT'/>
      <zeroOrMore><ref name='animateCommon.CM'/></zeroOrMore>
    </element>
  </define>

  <define name='animateColor.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.AnimateCommon.attr'/>
    <ref name='svg.AnimateAttributeCommon.attr'/>
    <ref name='svg.AnimateTiming.attr'/>
    <ref name='svg.AnimateValueCommon.attr'/>
    <ref name='svg.AnimateAdditionCommon.attr'/>
  </define>
```

The from , by and to attributes take color values, where each color value is expressed using the following syntax (the same syntax as used in SVG's properties that can take color values):

<color>

The values attribute for the 'animateColor' element consists of a semicolon-separated list of color values, with each color value expressed in the above syntax.

Out of range color values can be provided, but user agent processing will be implementation dependent. User agents should clamp color values to allow color range values as late as possible, but note that system differences might preclude consistent behavior across different systems.

For a list of attributes and properties that can be animated using the 'animateColor' element, see Attributes and properties that can be animated.

16.2.15 The 'animateTransform' element

The 'animateTransform' element animates a transformation attribute on a target element, thereby allowing animations to control translation, scaling, rotation and/or skewing.

Schema: animateTransform

```
<define name='animateTransform'>
  <element name='animateTransform'>
    <ref name='animateTransform.AT'/>
    <zeroOrMore><ref name='animateCommon.CM'/></zeroOrMore>
  </element>
</define>

<define name='animateTransform.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.AnimateCommon.attr'/>
  <ref name='svg.AnimateAttributeCommon.attr'/>
  <ref name='svg.AnimateTiming.attr'/>
  <ref name='svg.AnimateValueCommon.attr'/>
  <ref name='svg.AnimateAdditionCommon.attr'/>
  <ref name='svg.AnimateTypeCommon.attr'/>
</define>
```

Attribute definitions:

`type` = "translate | scale | rotate | skewX | skewY"
Indicates the type of transformation which is to have its values change over time.

Animatable: no.

The **from**, **by** and **to** attributes take a value expressed using the same syntax that is available for the given transformation type:

- For a **type="translate"**, each individual value is expressed as **<tx> [,<ty>]**.
- For a **type="scale"**, each individual value is expressed as **<sx> [,<sy>]**.
- For a **type="rotate"**, each individual value is expressed as **<rotate-angle> [<cx> <cy>]**.
- For a **type="skewX"** and **type="skewY"**, each individual value is expressed as **<skew-angle>**.

(See The transform attribute.)

The **values** attribute for the 'animateTransform' element consists of a semicolon-separated list of values, where each individual value is expressed as described above for **from**, **by** and **to**.

If **calcMode** has the value **paced**, then the "distance" for the transformation is calculated consisting of the sum of the absolute values of the differences between each pair of values as further described in Paced animations and complex types.

When an animation is active, the effect of non-additive **'animateTransform'** (i.e., **additive="replace"**) is to replace the given attribute's value with the transformation defined by the **'animateTransform'**. The effect of additive (i.e., **additive="sum"**) is to post-multiply the transformation matrix corresponding to the transformation defined by this **'animateTransform'**. To illustrate:

```
<rect transform="skewX(30)"...>
  <animateTransform attributeName="transform" attributeType="XML"
                    type="rotate" from="0" to="90" dur="5s"
                    additive="replace" fill="freeze"/>
  <animateTransform attributeName="transform" attributeType="XML"
                    type="scale" from="1" to="2" dur="5s"
                    additive="replace" fill="freeze"/>
</rect>
```

In the code snippet above, because the both animations have **additive="replace"**, the first animation overrides the transformation on the rectangle itself and the second animation overrides the transformation from the first animation; therefore, at time 5 seconds, the visual result of the above two animations would be equivalent to the following static rectangle:

```
<rect transform="scale(2)" ... />
```

whereas in the following example:

```
<rect transform="skewX(30)"...>
  <animateTransform attributeName="transform" attributeType="XML"
                    type="rotate" from="0" to="90" dur="5s"
                    additive="sum" fill="freeze"/>
  <animateTransform attributeName="transform" attributeType="XML"
                    type="scale" from="1" to="2" dur="5s"
                    additive="sum" fill="freeze"/>
</rect>
```

In this code snippet, because the both animations have **additive="sum"**, the first animation post-multiplies its transformation to any transformations on the rectangle itself and the second animation post-multiplies its transformation to any transformation from the first animation; therefore, at time 5 seconds, the visual result of the above two animations would be equivalent to the following static rectangle:

```
<rect transform="skewX(30) rotate(90) scale(2)" ... />
```

For a list of attributes and properties that can be animated using the **'animateTransform'** element, see Attributes and properties that can be animated.

16.2.16 Attributes and properties that can be animated

Each attribute or property within this specification indicates whether or not it can be animated by SVG's animation elements. Animatable attributes and properties are designated as follows:

Animatable: yes.

whereas attributes and properties that cannot be animated are designated:

Animatable: no.

SVG has a defined set of basic data types for its various supported attributes and properties. For those attributes and properties that can be animated, the following table indicates which animation elements can be used to animate each of the basic data types. If a given attribute or property can take values of keywords (which are not additive) or numeric values (which are additive), then additive animations are possible if the subsequent animation uses a numeric value even if the base animation uses a keyword value; however, if the subsequent animation uses a keyword value, additive animation is not possible.

Data type	Additive?	'animate'	'set'	'animateColor'	'animateTransform'	Notes
<color>	yes	yes	yes	yes	no	Only RGB color values are additive.
<coordinate>	yes	yes	yes	no	no	
<integer>	yes	yes	yes	no	no	
<length>	yes	yes	yes	no	no	
<list of xxx>	no	yes	yes	no	no	
<number>	yes	yes	yes	no	no	
<paint>	yes	yes	yes	yes	no	Only RGB color values are additive.
<percentage>	yes	yes	yes	no	no	
<time>	no	no	no	no	no	
<transform-list>	yes	no	no	no	yes	Additive means that a transformation is post-multiplied to the base set of transformations.
<iri>	no	yes	yes	no	no	
All other data types used in animatable attributes and properties	no	yes	yes	no	no	

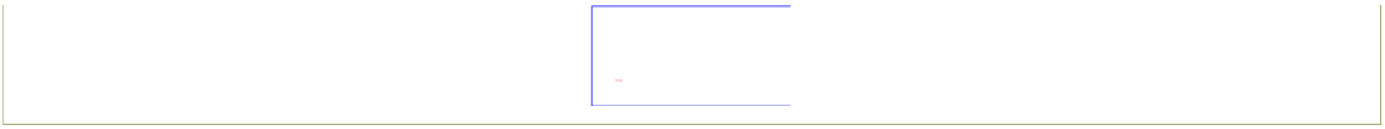
Any deviation from the above table or other special note about the animation capabilities of a particular attribute or property is included in the section of the specification where the given attribute or property is defined.

16.3 Animation using the SVG DOM

Example dom01 shows a simple animation using the DOM.

Example: 19_03.svg

```
<?xml version="1.0" standalone="no"?>
<svg width="4cm" height="2cm" viewBox="0 0 400 200" id="root"
     xmlns:ev="http://www.w3.org/2001/xml-events"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>A simple animation using the DOM.</desc>
  <script type="application/ecmascript"><![CDATA[
    var timeValue = 0;
    var timerIncrement = 50;
    var maxTime = 5000;
    var textElement;
    var svgRoot;

    function Init() {
      textElement = document.getElementById("svgtext");
      svgRoot = document.getElementById("root");
      ShowAndGrowElement(0);
    }

    function ShowAndGrowElement(repeatCount) {
      timeValue = (repeatCount + 1) * timerIncrement;
      // Scale the text string gradually until it is 20 times larger
      scalefactor = (timeValue * 20.) / maxTime;
      var matrix = createSVGMatrixComponents(scalefactor, 0, 0, scalefactor, 0, 0);
      textElement.setMatrixTrait("transform", matrix);
      // Make the string more opaque
      opacityfactor = timeValue / maxTime;
      textElement.setFloatTrait("fill-opacity", opacityfactor);
    }
  ]]></script>

  <handler type="application/ecmascript" ev:event="load">
        Init();
  </handler>

  <rect x="1" y="1" width="398" height="198"
        fill="none" stroke="blue" stroke-width="2"/>
  <g transform="translate(50,150)" font-size="7" stroke="none">
    <text fill="red" fill-opacity="1" id="svgtext">SVG</text>
  </g>

  <rect x="1000" y="1000" width="1" height="1" display="none">
     <animate id="timer" attributeName="display" from="none" to="none"
              begin="0" dur="50" repeatCount="99">
        <handler type="application/ecmascript" ev:event="repeatEvent">
           ShowAndGrowElement(evt.detail);
        </handler>
     </animate>
  </rect>

</svg>
```




At zero seconds

At 2.5 seconds

At five seconds

The above SVG file contains a text element that says "SVG". The animation loops for 5 seconds. The text string starts out small and transparent and grows to be large and opaque. Here is an explanation of how this example works:

- This is the first 'handler' in the example:

```
<handler type="application/ecmascript" ev:event="load">
Init();
</handler>
```

  Once the document has been fully loaded and processed, this 'handler' invokes the ECMAScript function `Init`.
- The 'script' element defines the ECMAScript which makes the animation happen. The `Init()` function is only called once to give a value to global variables `textElement` and `svgRoot` and to make the initial call to `ShowAndGrowElement`. `ShowAndGrowElement()` sets the `transform` and `fill-opacity` attributes on the text element to new values each time it is called.
- `ShowAndGrowElement()` is called every 50 milliseconds from the second 'handler' in the example.

```
<handler type="application/ecmascript" ev:event="repeat">
ShowAndGrowElement(evt.detail);
</handler>
```

  This 'handler' is placed on an animation that 'does nothing', its sole purpose is to emulate timer functionality. For each repeat (every 50ms) the 'handler' calls `ShowAndGrowElement`, passing the repeat number as a parameter.

If an attribute/property value is modified while an animation element is animating the same attribute/property, the animations are required to adjust dynamically to the new value.

16.4 Timed Animation Module

The Timed Animation Module contains the following elements:

- animate
- animateColor
- animateMotion
- animateTransform
- mpath
- set

SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

17 Fonts

Contents

17.1 Introduction

Reliable delivery of fonts is a requirement for SVG. Designers need to create SVG content with arbitrary fonts and know that the same graphical result will appear when the content is viewed by all end users, even when end users do not have the necessary fonts installed on their computers. This parallels the print world, where the designer uses a given font when authoring a drawing for print, and the graphical content appears exactly the same in the printed version as it appeared on the designer's authoring system.

Historically, one approach has been to convert all text to paths representing the glyphs used. This preserves the visual look, but means that the text is lost; it cannot be dynamically updated and is not accessible. Another approach is to hope that a font with a given name is available to all renderers. this assumption does not work well on the desktop and works very badly in a heterogenous mobile environment. SVG solves this problem by allowing the text to be converted to paths, but storing those paths as an SVG font. The text is retained and remains dynamicly modifiable and accessible.

17.1.1 Describing fonts available to SVG

SVG utilizes an XML version of the WebFonts facility defined in the "Cascading Style Sheets (CSS) level 2" specification [CSS2] to reference fonts. Described fonts may be in a variety of different formats.By describing key details of the font such as its family name, weight, whether it is italic and so on, text can continue to use

the font properties without having to explicitly indicate the font that is to be used for each span of text.

17.1.2 Defining fonts in SVG

One disadvantage to the WebFont facility to date is that specifications such as [CSS2] do not require support of particular font formats. The result is that different implementations support different Web font formats, thereby making it difficult for Web site creators to post a single Web site using WebFonts that work across all user agents.

To provide a common font format for SVG that is guaranteed to be supported by all conforming SVG viewers, SVG also defines a font format, in SVG, which uses the same geometric mechanism for glyphs as is used by the SVG path element. This facility is called SVG fonts. SVG implementations must support the SVG font format, and may also support other formats.WebFonts may be contained in the SVG document which uses them, or stored in a separate document (for example, to allow sharing of the same font by multiple SVG documents).

Taken together, these two mechanisms ensure reliable delivery of font data to end users, preserving graphical richness and enabling accessible access to the textual data. . In a common scenario, SVG authoring applications generate compressed, subsetted WebFonts for all text elements used by a given SVG document fragment. SVG fonts can improve the semantic richness of graphics that represent text. For example, many company logos consist of the company name drawn artistically. In some cases, accessibility may be enhanced by expressing the logo as a series of glyphs in an SVG font and then rendering the logo as a 'text' element which references this font.

17.2 Overview of SVG fonts

An SVG font is a font defined using SVG's 'font' element.

The purpose of SVG fonts is to allow for delivery of glyph outlines in display-only environments. SVG fonts that accompany Web pages must be supported only in browsing and viewing situations. Graphics editing applications or file translation tools must not attempt to convert SVG fonts into system fonts. The intent is that SVG files be interchangeable between two content creators, but not the SVG fonts that might accompany these SVG files. Instead, each content creator will need to license the given font before being able to successfully edit the SVG file. The font-face-name element indicates the name of licensed font to use for editing.

SVG fonts contain unhinted font outlines. Because of this, on many implementations there will be limitations regarding the quality and legibility of text in small font sizes. For increased quality and legibility in small font sizes, content creators may want to use an alternate font technology, such as fonts that ship with operating systems or an alternate WebFont format.

Because SVG fonts are expressed using SVG elements and attributes, in some cases the SVG font will take up more space than if the font were expressed in a different format which was especially designed for compact expression of font data. For the fastest delivery of Web pages, content creators may want to use an alternate font technology as a first choice, with a fallback to an SVG fornt for interoperability.

A key value of SVG fonts is guaranteed availability in SVG user agents. In some situations, it might be appropriate for an SVG font to be the first choice for rendering some text. In other situations, the SVG font might be an alternate, back-up font in case the first choice font (perhaps a hinted system font) is not available to a given user.

The characteristics and attributes of SVG fonts correspond closely to the font characteristics and parameters described in the "Fonts" chapter of the "Cascading Style Sheets (CSS) level 2" specification [CSS2]. In this model, various font metrics, such as advance values and baseline locations, and the glyph outlines themselves, are expressed in units that are relative to an abstract square whose height is the intended distance between lines of type in the same type size. This square is called the em square and it is the design grid on which the glyph outlines are defined. The value of the units-per-em attribute on the 'font' element specifies how many units the em square is divided into. Common values for other font types are, for example, 250 (Intellifont), 1000 (Type 1) and 2048 (TrueType, TrueType GX and Open-Type). Unlike standard graphics in SVG, where the initial coordinate system has the y-axis pointing downward (see The initial coordinate system), the design grid for SVG fonts, along with the initial coordinate system for the glyphs, has the y-axis pointing upward for consistency with accepted industry practice for many popular font formats.

SVG fonts and their associated glyphs do not specify bounding box information. Because the glyph outlines are expressed as SVG path elements, the implementation has the option to render the glyphs either using standard graphics calls or by using special-purpose font rendering technology, in which case any necessary maximum bounding box and overhang calculations can be performed from analysis of the path elements contained within the glyph outlines.

An SVG font can be either embedded within the same document that uses the font or saved as part of an external resource.

Here is an example of how you might embed an SVG font inside of an SVG document.

Example: 20_01.svg

```
<?xml version="1.0" standalone="yes"?>
<svg width="400px" height="300px" version="1.2" baseProfile="tiny"
  xmlns = 'http://www.w3.org/2000/svg'>
  <defs>
    <font id="Font1" horiz-adv-x="1000">
      <font-face font-family="Super Sans" font-weight="bold" font-style="normal"
          units-per-em="1000" cap-height="600" x-height="400"
          ascent="700" descent="300" alphabetic="0" mathematical="350" ideographic="400" hanging="500">
        <font-face-src>
          <font-face-name name="Super Sans Bold"/>
        </font-face-src>
      </font-face>
      <missing-glyph><path d="M0,0h200v200h-200z"/></missing-glyph>
      <glyph unicode="!" horiz-adv-x="300" d="--Outline of exclam. pt. glyph--"/>
      <glyph unicode="@" d="--Outline of @ glyph--"/>
      <!-- more glyphs -->
    </font>
  </defs>
  <desc>An example using an embedded font.</desc>
  <text x="100" y="100" font-family="Super Sans, Helvetica, sans-serif"
                font-weight="bold" font-style="normal">Text
    using embedded font</text>
</svg>
```

17.3 The 'font' element

The 'font' element defines an SVG font.

Schema: font

```
  <define name='font'>
    <element name='font'>
      <ref name='font.AT'/>
      <ref name='font.CM'/>
    </element>
```

```
  </define>

  <define name='font.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.FontAdvOrigCommon.attr'/>
    <optional>
      <attribute name='horiz-origin-x' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
  </define>

  <define name='font.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='font-face'/>
        <ref name='missing-glyph'/>
        <ref name='glyph'/>
        <ref name='hkern'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='svg.FontAdvOrigCommon.attr' combine='interleave'>
    <optional>
      <attribute name='horiz-adv-x' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
  </define>
```

Attribute definitions:

`horiz-origin-x = "<number>"`
The X-coordinate in the font coordinate system of the origin of a glyph to be used when drawing horizontally oriented text. (Note that the origin applies to all glyphs in the font.)

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: no.

`horiz-adv-x = "<number>"`
The default horizontal advance after rendering a glyph in horizontal orientation. Glyph widths are required to be non-negative, even if the glyph is typically rendered right-to-left, as in Hebrew and Arabic scripts.

Animatable: no.

Each **'font'** element must have a **'font-face'** child element which describes various characteristics of the font.

17.4 The 'glyph' element

The **'glyph'** element defines the graphics for a given glyph. The coordinate system for the glyph is defined by the various attributes in the **'font'** element.

The graphics that make up the **'glyph'** consist of a single path data specification within the **d** attribute.

Schema: glyph

```
  <define name='glyph'>
    <element name='glyph'>
      <ref name='glyph.AT'/>
      <ref name='glyph.CM'/>
    </element>
  </define>

  <define name='glyph.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.FontAdvOrigCommon.attr'/>
    <ref name='svg.D.attr'/>
    <attribute name='unicode' svg:animatable='false' svg:inheritable='false'><text/></attribute>
    <optional><attribute name='glyph-name' svg:animatable='false' svg:inheritable='false'><text/></attribute>
    </optional>
    <optional><attribute name='arabic-form' svg:animatable='false' svg:inheritable='false'><text/></attribute>
    </optional>
    <optional>
      <attribute name='lang' svg:animatable='false' svg:inheritable='false'>
        <ref name='LanguageCodes.datatype'/>
      </attribute>
    </optional>
  </define>

  <define name='glyph.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
      </choice>
    </zeroOrMore>
  </define>
```

Attribute definitions:

`unicode = "<string>"`
One or more Unicode characters indicating the sequence of Unicode characters which corresponds to this glyph. If a character is provided, then this glyph corresponds to the given Unicode character. If multiple characters are provided, then this glyph corresponds to the given sequence of Unicode characters.

One use of a sequence of characters is ligatures. For example, if **unicode="ffl"**, then the given glyph will be used to render the sequence of characters "f", "f", and "l".

It is often useful to refer to characters using XML character references expressed in hexadecimal notation or decimal notation. For example, **unicode="ffl"** could be expressed as XML character references in hexadecimal notation as **unicode="ffl"** or in decimal notation as **unicode="ffl"**.

The **unicode** attribute contributes to the process for deciding which glyph(s) are used to represent which character(s). See glyph selection rules. If the **unicode** attribute is not provided for a given **'glyph'**, the glyph cannot be accessed in SVG Tiny 1.2.

Animatable: no.

glyph-name = "<name> [, <name>]* "
An optional name for the glyph. It is recommended that glyph names be unique within a font. The glyph names can be used in situations where Unicode character numbers do not provide sufficient information to access the correct glyph, such as when there are multiple glyphs per Unicode character. The glyph names can be referenced in kerning definitions.

Animatable: no.

d = "path data"
The definition of the outline of a glyph, using the same syntax as for the **d** attribute on a **'path'** element. See Path data.

See below for a discussion of this attribute.

Animatable: no.

arabic-form = "initial | medial | terminal | isolated"
For Arabic glyphs, indicates which of the four possible forms this glyph represents.

Animatable: no.

lang = "%LanguageCodes;"
The attribute value is a comma-separated list of language names as defined in [RFC3066]. The glyph can be used if the **xml:lang** attribute exactly matches one of the languages given in the value of this parameter, or if the **xml:lang** attribute exactly equals a prefix of one of the languages given in the value of this parameter such that the first tag character following the prefix is "-". If the attribute is not specified, then the glyph can be used in all languages.

Animatable: no.

horiz-adv-x = "<number>"
The horizontal advance after rendering the glyph in horizontal orientation. If the attribute is not specified, the effect is as if the attribute were set to the value of the font's **horiz-adv-x** attribute.

Glyph widths are required to be non-negative, even if the glyph is typically rendered right-to-left, as in Hebrew and Arabic scripts.

Animatable: no.

The graphics for the **'glyph'** are specified using the **d** attribute. The path data within this attribute is processed as follows:

- Any relative coordinates within the path data specification are converted into equivalent absolute coordinates
- Each of these absolute coordinates is transformed from the font coordinate system into the **'text'** element's current coordinate system such that the origin of the font coordinate system is properly positioned and rotated to align with the current text position and orientation for the glyph, and scaled so that the correct **'font-size'** is achieved.
- The resulting, transformed path specification is rendered as if it were a **'path'** element, using the styling properties that apply to the characters which correspond to the given glyph, and ignoring any styling properties specified on the **'font'** element or the **'glyph'** element.

In general, the **d** attribute renders in the same manner as system fonts. For example, a dashed pattern will usually look the same if applied to a system font or to an SVG font which defines its glyphs using the **d** attribute. Many implementations will be able to render glyphs quickly and will be able to use a font cache for further performance gains.

17.5 The 'missing-glyph' element

The **'missing-glyph'** element defines a graphic to use if there is an attempt to draw a glyph from a given font and the given glyph has not been defined. The attributes on the **'missing-glyph'** element have the same meaning as the corresponding attributes on the **'glyph'** element.

Schema: missing-glyph

```
<define name='missing-glyph'>
  <element name='missing-glyph'>
    <ref name='missing-glyph.AT'/>
    <ref name='missing-glyph.CM'/>
  </element>
</define>

<define name='missing-glyph.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.FontAdvOrigCommon.attr'/>
  <ref name='svg.D.attr'/>
</define>

<define name='missing-glyph.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
    </choice>
  </zeroOrMore>
</define>
```

17.6 Glyph selection rules

When determining the glyph(s) to draw a given character sequence, the **'font'** element is searched from its first **'glyph'** element to its last in logical order to see if the upcoming sequence of Unicode characters to be rendered matches the sequence of Unicode characters specified in the **unicode** attribute for the given **'glyph'** element. The first successful match is used. Thus, the "ffl" ligature needs to be defined in the font before the "f" glyph; otherwise, the "ffl" will never be selected.

17.7 The 'hkern' element

The 'hkern' element defines kerning pairs for horizontally-oriented pairs of glyphs.

Kern pairs identify pairs of glyphs within a single font whose inter-glyph spacing is adjusted when the pair of glyphs are rendered next to each other. In addition to the requirement that the pair of glyphs are from the same font, SVG font kerning happens only when the two glyphs correspond to characters which have the same values for properties 'font-family', 'font-size', 'font-style' and 'font-weight'.

An example of a kerning pair are the letters "Va", where the typographic result might look better if the letters "V" and the "a" were rendered slightly closer together.

Right-to-left and bidirectional text in SVG is laid out in a two-step process, which is described in Relationship with bidirectionality. If SVG fonts are used, before kerning is applied, characters are re-ordered into left-to-right (or top-to-bottom, for vertical text) visual rendering order. Kerning from SVG fonts is then applied on pairs of glyphs which are rendered contiguously. The first glyph in the kerning pair is the left (or top) glyph in visual rendering order. The second glyph in the kerning pair is the right (or bottom) glyph in the pair.

For convenience to font designers and to minimize file sizes, a single 'hkern' can define a single kerning adjustment value between one set of glyphs (e.g., a range of Unicode characters) and another set of glyphs (e.g., another range of Unicode characters).

The 'hkern' element defines kerning pairs and adjustment values in the horizontal advance value when drawing pairs of glyphs which the two glyphs are contiguous and are both rendered horizontally (i.e., side-by-side). The spacing between characters is reduced by the kerning adjustment. (Negative kerning adjustments increase the spacing between characters.)

Schema: hkern

```
<define name='hkern'>
  <element name='hkern'>
    <ref name='hkern.AT'/>
    <ref name='hkern.CM'/>
  </element>
</define>

<define name='hkern.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <optional>
    <attribute name='u1' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='g1' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='u2' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='g2' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <attribute name='k' svg:animatable='false' svg:inheritable='false'>
    <ref name='Number.datatype'/>
  </attribute>
</define>

<define name='hkern.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

u1 = "[<character> | <urange>] [, [<character> | <urange>]]* "
A sequence (comma-separated) of Unicode characters (refer to the description of the unicode attribute to the 'glyph' element for a description of how to express individual Unicode characters) and/or ranges of Unicode characters (see description of ranges of Unicode characters in [CSS2]) which identify a set of possible first glyphs in the kerning pair. If a given Unicode character within the set has multiple corresponding 'glyph' elements (i.e., there are multiple 'glyph' elements with the same unicode attribute value, but different glyph-name values), then all such glyphs are included in the set. Comma is the separator character; thus, to kern a comma, specify the comma as part of a range of Unicode characters or as a glyph name using the g1 attribute. The total set of possible first glyphs in the kerning pair is the union of glyphs specified by the u1 and g1 attributes.

Animatable: no.

g1 = "<name> [, <name>]* "
A sequence (comma-separated) of glyph names (i.e., values that match glyph-name attributes on 'glyph' elements) which identify a set of possible first glyphs in the kerning pair. All glyphs with the given glyph name are included in the set. The total set of possible first glyphs in the kerning pair is the union of glyphs specified by the u1 and g1 attributes.

Animatable: no.

u2 = "[<number> | <urange>] [, [<number> | <urange>]]* "
Same as the u1 attribute, except that u2 specifies possible second glyphs in the kerning pair.

Animatable: no.

g2 = "<name> [, <name>]* "
Same as the g1 attribute, except that g2 specifies possible second glyphs in the kerning pair.

Animatable: no.

k = "<number>"
The amount to decrease the spacing between the two glyphs in the kerning pair. The value is in the font coordinate system. This attribute is required.

Animatable: no.

At least one each of u1 or g1 and at least one of u2 or g2 must be provided.

17.8 Describing a font

17.8.1 Overview of font descriptions

A font description provides the bridge between an author's font specification and the font data, which is the data needed to format text and to render the abstract glyphs to which the characters map - the actual scalable outlines or bitmaps. Fonts are referenced by properties, such as the **'font-family'** property.

Each specified font description is added to the font database and so that it can be used to select the relevant font data. The font description contains descriptors such as the location of the font data on the Web, and characterizations of that font data. The font descriptors are also needed to match the font properties to particular font data. The level of detail of a font description can vary from just the name of the font up to a list of glyph widths.

For more about font descriptions, refer to the font chapter in the CSS2 specification [CSS2 Fonts].

17.8.2 The 'font-face' element

The **'font-face'** element is an XML structure which corresponds directly to the @font-face facility in CSS2. It can be used to describe the characteristics of any font, SVG font or otherwise.

When used to describe the characteristics of an SVG font contained within the same document, it is recommended that the **'font-face'** element be a child of the **'font'** element it is describing so that the **'font'** element can be self-contained and fully-described. In this case, any **'font-face-src'** elements within the **'font-face'** element are ignored as it is assumed that the **'font-face'** element is describing the characteristics of its parent **'font'** element.

Schema: font-face

```
<define name='font-face'>
  <element name='font-face'>
    <ref name='font-face.AT'/>
    <ref name='font-face.CM'/>
  </element>
</define>

<define name='font-face.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.External.attr'/>
  <optional><attribute name='font-family' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='font-style' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='font-weight' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='font-size' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='font-variant' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='font-stretch' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='unicode-range' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='panose-1' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='widths' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional><attribute name='bbox' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
  <optional>
    <attribute name='units-per-em' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='stemv' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='stemh' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='slope' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='cap-height' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='x-height' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='accent-height' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='ascent' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='descent' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
    </attribute>
  </optional>
  <optional>
    <attribute name='ideographic' svg:animatable='false' svg:inheritable='false'>
      <ref name='Number.datatype'/>
```

```
      </attribute>
    </optional>
    <optional>
      <attribute name='alphabetic' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='mathematical' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='hanging' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='underline-position' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='underline-thickness' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='strikethrough-position' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='strikethrough-thickness' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='overline-position' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='overline-thickness' svg:animatable='false' svg:inheritable='false'>
        <ref name='Number.datatype'/>
      </attribute>
    </optional>
  </define>

  <define name='font-face.CM'>
    <zeroOrMore>
      <choice>
        <ref name='svg.Desc.group'/>
        <ref name='font-face-src'/>
      </choice>
    </zeroOrMore>
  </define>
```

Attribute definitions:

`font-family` = "<string>"
Same syntax and semantics as the **'font-family'** descriptor within an @font-face rule.

Animatable: no.

`font-style` = "all | [normal | italic | oblique] [, [normal | italic | oblique]]*"
Same syntax and semantics as the **'font-style'** descriptor within an @font-face rule. The style of a font. Takes on the same values as the 'font-style' property, except that a comma-separated list is permitted.

If the attribute is not specified, the effect is as if a value of "all" were specified.

Animatable: no.

`font-weight` = "all | [normal | bold |100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900] [, [normal | bold |100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900]]*"
Same syntax and semantics as the **'font-weight'** descriptor within an @font-face rule.

The weight of a face relative to others in the same font family. Takes on the same values as the 'font-weight' property with three exceptions:

- relative keywords (bolder, lighter) are not permitted
- a comma-separated list of values is permitted, for fonts that contain multiple weights
- an additional keyword, 'all', is permitted, which means that the font will match for all possible weights; either because it contains multiple weights, or because that face only has a single weight.

If the attribute is not specified, the effect is as if a value of "all" were specified.

Animatable: no.

`font-size` = "<string>"
Same syntax and semantics as the **'font-size'** descriptor within an @font-face rule.

Animatable: no.

`unicode-range` = "<urange> [, <urange>]*"
Same syntax and semantics as the **'unicode-range'** descriptor within an @font-face rule. The range of ISO 10646 characters [UNICODE] possibly covered by the glyphs in the font. Except for any additional information provided in this specification, the normative definition of the attribute is in [CSS2].

If the attribute is not specified, the effect is as if a value of "U+0-10FFFF" were specified.

Animatable: no.

`units-per-em` = " <number> "
Same syntax and semantics as the **'units-per-em'** descriptor within an @font-face rule. The number of coordinate units on the em square, the size of the design grid on which glyphs are laid out.

This value is almost always necessary as nearly every other attribute requires the definition of a design grid.

If the attribute is not specified, the effect is as if a value of "1000" were specified.

Animatable: no.

panose-1 = "[<integer>]{10}"

Same syntax and semantics as the **'panose-1'** descriptor within an @font-face rule. The Panose-1 number, consisting of ten decimal integers, separated by whitespace. Except for any additional information provided in this specification, the normative definition of the attribute is in [CSS2].

If the attribute is not specified, the effect is as if a value of "0 0 0 0 0 0 0 0 0 0" were specified.

Animatable: no.

stemv = " <number> "

Same syntax and semantics as the **'stemv'** descriptor within an @font-face rule.

Animatable: no.

stemh = " <number> "

Same syntax and semantics as the **'stemh'** descriptor within an @font-face rule.

Animatable: no.

slope = " <number> "

Same syntax and semantics as the **'slope'** descriptor within an @font-face rule. The vertical stroke angle of the font. Except for any additional information provided in this specification, the normative definition of the attribute is in [CSS2].

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: no.

cap-height = " <number> "

Same syntax and semantics as the **'cap-height'** descriptor within an @font-face rule. The height of uppercase glyphs in the font within the font coordinate system.

Animatable: no.

x-height = " <number> "

Same syntax and semantics as the **'x-height'** descriptor within an @font-face rule. The height of lowercase glyphs in the font within the font coordinate system.

Animatable: no.

accent-height = " <number> "

The distance from the origin to the top of accent characters, measured by a distance within the font coordinate system.

If the attribute is not specified, the effect is as if the attribute were set to the value of the **ascent** attribute. If this attribute is used, the **units-per-em** attribute must also be specified.

Animatable: no.

ascent = " <number> "

Same syntax and semantics as the **'ascent'** descriptor within an @font-face rule. The maximum unaccented height of the font within the font coordinate system.

If the attribute is not specified, the effect is as if the attribute were set to the difference between the **units-per-em** value and the **vert-origin-y** value for the corresponding font.

Animatable: no.

descent = " <number> "

Same syntax and semantics as the **'descent'** descriptor within an @font-face rule. The maximum unaccented depth of the font within the font coordinate system.

If the attribute is not specified, the effect is as if the attribute were set to the **vert-origin-y** value for the corresponding font.

Animatable: no.

widths = "<string>"

Same syntax and semantics as the **'widths'** descriptor within an @font-face rule.

Animatable: no.

bbox = "<string>"

Same syntax and semantics as the **'bbox'** descriptor within an @font-face rule.

Animatable: no.

ideographic = " <number> "

For horizontally oriented glyph layouts, indicates the alignment coordinate for glyphs to achieve ideographic baseline alignment. The value is an offset in the font coordinate system. If this attribute is provided, the **units-per-em** attribute must also be specified.

Animatable: no.

alphabetic = " <number> "

Same syntax and semantics as the **'baseline'** descriptor within an @font-face rule. For horizontally oriented glyph layouts, indicates the alignment coordinate for glyphs to achieve alphabetic baseline alignment. The value is an offset in the font coordinate system. If this attribute is provided, the **units-per-em** attribute must also be specified.

Animatable: no.

mathematical = " <number> "

Same syntax and semantics as the **'mathline'** descriptor within an @font-face rule. For horizontally oriented glyph layouts, indicates the alignment coordinate for glyphs to achieve mathematical baseline alignment. The value is an offset in the font coordinate system. If this attribute is provided, the **units-per-em** attribute must also be specified.

Animatable: no.

hanging = " <number> "

For horizontally oriented glyph layouts, indicates the alignment coordinate for glyphs to achieve hanging baseline alignment. The value is an offset in the font coordinate system. If this attribute is provided, the **units-per-em** attribute must also be specified.

Animatable: no.

underline-position = " <number> "

The ideal position of an underline within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

underline-thickness = " <number> "
The ideal thickness of an underline, expressed as a length within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

strikethrough-position = " <number> "
The ideal position of a strike-through within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

strikethrough-thickness = " <number> "
The ideal thickness of a strike-through, expressed as a length within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

overline-position = " <number> "
The ideal position of an overline within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

overline-thickness = " <number> "
The ideal thickness of an overline, expressed as a length within the font coordinate system. If this attribute is provided, the units-per-em attribute must also be specified.

Animatable: no.

17.8.3 The 'font-face-src' element

The 'font-face-src' element, together with the 'font-face-uri' and 'font-face-name' elements described further down correspond to the **'src'** descriptor within an @font-face rule. (Refer to the descriptions of the [@font-face rule] and ['src' descriptor] in the CSS2 specification.)

A 'font-face-src' element contains either a 'font-face-uri' or a 'font-face-name' element. The 'font-face-src' element is used for referencing fonts defined elsewhere.

Schema: font-face-src

```
<define name='font-face-src'>
  <element name='font-face-src'>
    <ref name='font-face-src.AT'/>
    <ref name='font-face-src.CM'/>
  </element>
</define>

<define name='font-face-src.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
</define>

<define name='font-face-src.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
      <ref name='font-face-uri'/>
      <ref name='font-face-name'/>
    </choice>
  </zeroOrMore>
</define>
```

17.8.4 The 'font-face-uri' element

The 'font-face-uri' element is used within a 'font-face-src' element to reference a font defined inside or outside of the current SVG document.

When a 'font-face-uri' is referencing an SVG font, then that reference must be to an SVG 'font' element, therefore requiring the use of a fragment identifier (see [IRI]). The referenced 'font' element can be local (i.e., within the same document as the 'font-face-uri' element) or remote (i.e., within a different document).

Schema: font-face-uri

```
<define name='font-face-uri'>
  <element name='font-face-uri'>
    <ref name='font-face-uri.AT'/>
    <ref name='font-face-uri.CM'/>
  </element>
</define>

<define name='font-face-uri.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <ref name='svg.XLinkRequired.attr'/>
</define>

<define name='font-face-uri.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

xlink:href = " <iri> "
An IRI reference

Animatable: no.

17.8.5 The 'font-face-name' element

The 'font-face-name' element is used within a 'font-face-src' element to name the font.

Schema: font-face-name

```
<define name='font-face-name'>
  <element name='font-face-name'>
    <ref name='font-face-name.AT'/>
    <ref name='font-face-name.CM'/>
  </element>
</define>

<define name='font-face-name.AT' combine='interleave'>
  <ref name='svg.Core.attr'/>
  <optional>
    <attribute name='name' svg:animatable='false' svg:inheritable='false'><text/></attribute>
  </optional>
</define>

<define name='font-face-name.CM'>
  <zeroOrMore>
    <choice>
      <ref name='svg.Desc.group'/>
    </choice>
  </zeroOrMore>
</define>
```

Attribute definitions:

name = " <string>"
The name of the font to use.

Animatable: no.

17.9 Font Module

The Font Module contains the following elements:

Font

- font
- glyph
- font-face
- missing-glyph
- hkern
- font-face-src
- font-face-uri
- font-face-name

SVG Tiny 1.2 - 20050413 Top Contents Previous Next Elements Attributes

18 Metadata

Contents

18.1 Introduction

Metadata is structured data about data.

In the computing industry, there are ongoing standardization efforts towards metadata with the goal of promoting industry interoperability and efficiency. Content creators should track these developments and include appropriate metadata in their SVG content which conforms to these various metadata standards as they emerge.

The W3C has a Semantic Web Activity which has been established to serve a leadership role, in both the design of enabling specifications and the open, collaborative development of technologies that support the automation, integration and reuse of data across various applications. The Semantic Web Activity builds upon the earlier W3C Metadata Activity, including the definition of Resource Description Framework (RDF). The specifications for RDF can be found at:

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

Another activity relevant to most applications of metadata is the Dublin Core, which is a set of generally applicable core metadata properties (e.g., Title, Creator/Author, Subject, Description, etc.).

Individual industries or individual content creators are free to define their own metadata schema but are 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 an existing framework such as RDF and to use custom metadata schema in combination with standard metadata schema, rather than totally ignore the standard schema.

18.2 The 'metadata' element

Metadata which is included with SVG content should be specified within 'metadata' elements. The contents of the 'metadata' should be elements from other XML namespaces, with these elements from these namespaces expressed in a manner conforming with the "Namespaces in XML 1.1" Recommendation [XML-NS].

Authors should provide a 'metadata' child element to the outermost 'svg' element within a stand-alone SVG document. The 'metadata' child element to an 'svg' element serves the purposes of identifying document-level metadata.

The DTD definitions of many of SVG's elements (particularly, container and text elements) place no restriction on the placement or number of the 'metadata' sub-elements. This flexibility is only present so that there will be a consistent content model for container elements, because some container elements in SVG allow for mixed content, and because the mixed content rules for XML [XML-MIXED] do not permit the desired restrictions. Representations of future versions of the SVG language might use more expressive representations than DTDs which allow for more restrictive mixed content rules. It is strongly recommended that at most one 'metadata' element appear as a child of any particular element, and that this element appear before any other child elements (except possibly 'desc' or 'title' elements) or character data content. If metadata-processing user agents need to choose among multiple 'metadata' elements for processing it should choose the first one.

Schema: metadata

```
<define name='metadata'>
  <element name='metadata'>
    <ref name='DTM.AT'/>
    <ref name='DTM.CM'/>
  </element>
</define>
```

18.3 An example

Here is an example of how metadata can be included in an SVG document. The example uses the Dublin Core version 1.1 schema. (Other XML-compatible metadata languages, including ones not based on RDF, can be used also.)

Example: 21_01.svg

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in" version="1.2" baseProfile="tiny"
    xmlns = 'http://www.w3.org/2000/svg'>
    <desc xmlns:myfoo="http://example.org/myfoo">
      <myfoo:title>This is a financial report</myfoo:title>
      <myfoo:descr>The global description uses markup from the
        <myfoo:emph>myfoo</myfoo:emph> namespace.</myfoo:descr>
      <myfoo:scene><myfoo:what>widget $growth</myfoo:what>
      <myfoo:contains>$three $graph-bar</myfoo:contains>
        <myfoo:when>1998 $through 2000</myfoo:when> </myfoo:scene>
   </desc>
    <metadata>
      <rdf:RDF
           xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
           xmlns:dc = "http://purl.org/dc/elements/1.1/" >
        <rdf:Description about="http://example.org/myfoo"
             dc:title="MyFoo Financial Report"
             dc:description="$three $bar $thousands $dollars $from 1998 $through 2000"
             dc:publisher="Example Organization"
             dc:date="2000-04-11"
             dc:format="image/svg+xml"
             dc:language="en" >
          <dc:creator>
            <rdf:Bag>
              <rdf:li>Irving Bird</rdf:li>
              <rdf:li>Mary Lambert</rdf:li>
            </rdf:Bag>
          </dc:creator>
        </rdf:Description>
      </rdf:RDF>
    </metadata>
</svg>
```


SVG Tiny 1.2 - 20050413 | Top | Contents | Previous | Next | Elements | Attributes

19 Extensibility

Contents

19.1 Foreign namespaces and private data

SVG allows inclusion of elements from foreign namespaces anywhere with the SVG content. In general, the SVG user agent will include the unknown elements in the DOM but will otherwise ignore unknown elements. (The notable exception is described under Embedding Foreign Object Types.)

Additionally, SVG allows inclusion of attributes from foreign namespaces on any SVG element. The SVG user agent will include unknown attributes in the DOM but will otherwise ignore unknown attributes.

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 content 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 content 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:

Example: 23_01.svg

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in" version="1.2" baseProfile="tiny"
     xmlns = 'http://www.w3.org/2000/svg'>
  <defs>
    <myapp:piechart xmlns:myapp="http://example.org/myapp"
                    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 SVG graphics elements which
       draw the pie chart -->
</svg>
```

19.2 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 at a given point within the SVG content tree. One particular example of this is to provide a frame for XML content styled with CSS or XSL so that dynamically reflowing text (subject to SVG transformations and compositing) could be inserted into the middle of some SVG content. Another example is inserting a MathML expression into an SVG drawing.

The **'foreignObject'** element allows for inclusion of a foreign namespace which has its graphical content drawn by a different user agent. The included foreign graphical content 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 with a proper **xmlns** (see "Namespaces in XML 1.1" [XML-NS]) attribute specification is embedded recursively. One situation where this can occur is when an SVG document fragment is embedded within another non-SVG document fragment, which in turn is embedded within an SVG document fragment (e.g., an SVG document fragment contains an XHTML document fragment which in turn contains yet another SVG document fragment).

Usually, a **'foreignObject'** will be used in conjunction with the **'switch'** element and the **requiredExtensions** attribute to provide proper checking for user agent support and provide an alternate rendering in case user agent support is not available.

19.3 The 'foreignObject' element

Schema: foreignObject

```
  <define name='foreignObject'>
    <element name='foreignObject'>
      <ref name='foreignObject.AT'/>
      <ref name='foreignObject.CM'/>
    </element>
  </define>

  <define name='foreignObject.AT' combine='interleave'>
    <ref name='svg.Core.attr'/>
    <ref name='svg.Conditional.attr'/>
    <ref name='svg.XLinkEmbed.attr'/>
    <ref name='svg.Focus.attr'/>
    <ref name='svg.External.attr'/>
    <ref name='svg.Properties.attr'/>
    <ref name='svg.Transform.attr'/>
    <ref name='svg.XYWH.attr'/>
  </define>

  <define name='foreignObject.CM'>
    <empty/>
  </define>
```

Attribute definitions:

x = " <coordinate> "
The x-axis coordinate of one corner of the rectangular region into which the graphics associated with the contents of the **'foreignObject'** will be rendered.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

y = " <coordinate> "
The y-axis coordinate of one corner of the rectangular region into which the referenced document is placed.

If the attribute is not specified, the effect is as if a value of "0" were specified.

Animatable: yes.

width = " <length> "
The width of the rectangular region into which the referenced document is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

height = " <length> "
The height of the rectangular region into which the referenced document is placed.

A negative value is an error (see Error processing). A value of zero disables rendering of the element.

Animatable: yes.

xlink:href = " <iri> "

An IRI reference.

Animatable: yes.

19.4 An example

Here is an example:

Example: 23_02.svg

```
<?xml version="1.0" standalone="yes"?>
<svg width="4in" height="3in" version="1.1"
 xmlns = 'http://www.w3.org/2000/svg'>
  <desc>This example uses the 'switch' element to provide a
        fallback graphical representation of an paragraph, if
        XMHTML is not supported.</desc>
  <!-- The 'switch' element will process the first child element
       whose testing attributes evaluate to true.-->
  <switch>
    <!-- Process the embedded XHTML if the requiredExtensions attribute
         evaluates to true (i.e., the user agent supports XHTML
         embedded within SVG). -->
    <foreignObject width="100" height="50"
                   requiredExtensions="http://example.com/SVGExtensions/EmbeddedXHTML">
      <!-- XHTML content goes here -->
      <body xmlns="http://www.w3.org/1999/xhtml">
        <p>Here is a paragraph that requires word wrap</p>
      </body>
    </foreignObject>
    <!-- Else, process the following alternate SVG.
         Note that there are no testing attributes on the 'text' element.
         If no testing attributes are provided, it is as if there
         were testing attributes and they evaluated to true.-->
    <text font-size="10" font-family="Verdana">
      <tspan x="10" y="10">Here is a paragraph that</tspan>
      <tspan x="10" y="20">requires word wrap.</tspan>
    </text>
  </switch>
</svg>
```

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 must 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 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 capability is not a requirement.

19.5 Extensibility Module

The Extensibility Module contains the following element:

- foreignObject