W3C Working Draft

Scalable Vector Graphics (SVG) 1.2

W3C Working Draft 27 October 2004

This version:
http://www.w3.org/TR/2004/WD-SVG12-20041027/
Previous version:
http://www.w3.org/TR/2004/WD-SVG12-20040510/
Latest version of SVG 1.2:
http://www.w3.org/TR/SVG12/
Latest SVG Recommendation:
http://www.w3.org/TR/SVG/
Editor:
Dean Jackson, W3C, <dean@w3.org>
Authors:
See Author List

Abstract

SVG is a modularized XML language for describing two-dimensional graphics with animation and interactivity, and a set of APIs upon which to build graphics-based applications. This document specifies version 1.2 of Scalable Vector Graphics (SVG).

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 of the Scalable Vector Graphics (SVG) 1.2 specification. The SVG Working Group plans to submit this specification for consideration as a W3C Candidate Recommendation after examining feedback to this draft. Comments for this specification should have a subject starting with the prefix 'SVG 1.2 Comment:'. Please send them to www-svg@w3.org, the public email list for issues related to vector graphics on the Web. 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 24 November 2004.

This document lists the changes from SVG 1.1. It is not a complete language description.

This document has been produced by the SVG Working Group as part of the Graphics Activity within the W3C Interaction Domain.

This document was produced under the 23 January 2002 CPP as amended by the W3C Patent Policy Transition Procedure. The Working Group maintains a public list of patent disclosures relevant to this document; that page also includes instructions for disclosing a patent. 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.

Table of Contents

- 1 Introduction

- 2 Profiling SVG
- 3 XML Binding Language for SVG
- 4 Flowing text and graphics
- 5 Multiple pages
- 6 Text enhancements
- 7 Streaming
- 8 Progressive rendering
- 9 Vector effects
- 10 Rendering model
- 11 Painting enhancements
- 12 Media
- 13 Animation
- 14 Extended links
- 15 Application development
- 16 Events and Scripting
- 17 Non-graphical enhancements
- Appendix A: DOM enhancements
- Appendix B: API enhancements
- Appendix C: SVG DOM Subset
- Appendix D: Feature strings
- Appendix E: List of events
- Appendix F: RelaxNG schema for SVG 1.2
- Appendix G: Media Type registration for image/svg+xml
- Appendix H: References
- Appendix I: Change log

Full Table of Contents

- 1 Introduction
- 2 Profiling SVG
- 3 XML Binding Language for SVG
    - 3.1 Introduction
    - 3.2 Simple example
    - 3.3 sXBL restrictions with SVG
        - 3.3.1 Standalone SVG documents restriction
        - 3.3.2 Non-SVG element restriction

- 3.4 sXBL rendering and event handling rules with SVG
- 3.5 sXBL bindings for SVG resources
- 3.6 sXBL bindings for visual effects
- 3.7 The traitDef element
- 3.8 Trait mutation events
- 3.9 Document loading
- 3.10 Scripts and handlers
- 3.11 CSS Stylesheets
- 3.12 Focus and navigation
- 3.13 Error handling

- 4 Flowing text and graphics
  - 4.1 The flowRoot element
  - 4.2 The flowRegion element
  - 4.3 The flowRegionExclude element
  - 4.4 The flowDiv element
  - 4.5 The flowPara element
  - 4.6 The flowSpan element
  - 4.7 The flowRegionBreak element
  - 4.8 The flowLine element
  - 4.9 The flowTref element
  - 4.10 The flowImage element
  - 4.11 The flowRef element
  - 4.12 Text Flow
  - 4.13 Determining Strip Location
    - 4.13.1 Calculating Text Regions
  - 4.14 Alignment
    - 4.14.1 The text-align property
    - 4.14.2 The progression-align property
    - 4.14.3 Overflow
  - 4.15 Example
  - 4.16 DOM Interfaces
- 5 Multiple pages
  - 5.1 Displaying Pages
  - 5.2 Timing and animation
  - 5.3 The pageSet element
  - 5.4 The page element
  - 5.5 The page-orientation property
  - 5.6 Navigation between pages

- 6 Text enhancements
  - 6.1 Editable Text Fields
    - 6.1.1 The editable attribute
    - 6.1.2 CSS pseudo class for editable text
  - 6.2 Text Selection
- 7 Streaming
  - 7.1 The timelineBegin attribute
  - 7.2 The streamedContents attribute
- 8 Progressive rendering
- 9 Vector effects
  - 9.1 The vectorEffect element
  - 9.2 Common vector effect primitive attributes
  - 9.3 The veStrokePath element
  - 9.4 The veSetback element
  - 9.5 The veAffine element
  - 9.6 The veReverse element
  - 9.7 The veJoin element
  - 9.8 The veUnion element
  - 9.9 The veIntersect element
  - 9.10 The veExclude element
  - 9.11 The veFill element
  - 9.12 The veStroke element
  - 9.13 The veMarker element
  - 9.14 The veMarkerPath element
  - 9.15 The vePath element
  - 9.16 The vePathRef element
  - 9.17 Examples
  - 9.18 The vector-effect property
- 10 Rendering model
  - 10.1 Enhanced Alpha Compositing
    - 10.1.1 Introduction
    - 10.1.2 Alpha compositing
    - 10.1.3 The clip-to-self property
    - 10.1.4 The enable-background property
    - 10.1.5 The knock-out property
    - 10.1.6 The comp-op property
  - 10.2 Enhanced Transformations
    - 10.2.1 The user space transformation

        - 10.2.2 ViewBox to Viewport transformation
        - 10.2.3 Element Transform Stack
        - 10.2.4 The Current Transform Matrix
        - 10.2.5 The ref() transform value
- 11 Painting enhancements
    - 11.1 Background Fill Property
    - 11.2 Background Fill Opacity Property
    - 11.3 Inheritance into the shadow tree
    - 11.4 The solidColor Element
    - 11.5 Using device colors
        - 11.5.1 The device-color keyword
        - 11.5.2 DOM Interface
    - 11.6 ICC named colors
    - 11.7 Specifying paint
    - 11.8 Controlling the rendering color space
        - 11.8.1 The rendering-color-space property
    - 11.9 Filter Region extensions
    - 11.10 Prefetching resources
        - 11.10.1 The prefetch element
    - 11.11 Referencing external stylesheets
    - 11.12 Increased switch availability
    - 11.13 The min-unit-scale and max-unit-scale attributes
    - 11.14 Testing for formats
    - 11.15 Testing for font availability
    - 11.16 Overlaying graphics
        - 11.16.1 The overlay property
            - 11.16.1.1 The overlay-host property
        - 11.16.2 Example
    - 11.17 Modifications to cursors
        - 11.17.1 The cursor property
        - 11.17.2 Inline cursor content
    - 11.18 Highlighting
    - 11.19 Automatic text length
    - 11.20 More rendering hints
        - 11.20.1 The cache property
            - 11.20.1.1 The static property
        - 11.20.2 The snap property
- 12 Media

- 12.1 The audio element
  - 12.1.1 Supported audio format
- 12.2 The video element
- 12.3 Alternate content based on display resolutions
  - 12.3.1 The multiImage element
  - 12.3.2 The subImageRef element
  - 12.3.3 The subImage element
  - 12.3.4 Selecting the image for rendering
- 12.4 Media Properties
- 12.5 Loading images

- 13 Animation
  - 13.1 The animation element
    - 13.1.1 Transition effects
      - 13.1.1.1 The transition element
      - 13.1.1.2 The transition attributes
  - 13.2 Media elements
  - 13.3 Time containers
  - 13.4 Attributes for run-time synchronization
  - 13.5 Time Manipulation
  - 13.6 Enhanced ElementTimeControl interface
  - 13.7 Triggering animations using key events
- 14 Extended links
- 15 Application development
  - 15.1 Element focus and navigation
    - 15.1.1 The focusable property
    - 15.1.2 Navigation
    - 15.1.3 Obtaining focus
  - 15.2 Tooltips
    - 15.2.1 The hint element
    - 15.2.2 The tooltip property
- 16 Events and Scripting
  - 16.1 The script element
  - 16.2 The handler element
  - 16.3 DOM Interfaces
  - 16.4 Processing order for user interface event
- 17 Non-graphical enhancements
  - 17.1 Externally referenced documents
  - 17.2 Referencing external titles, descriptions and metadata

        - 17.3 Adding Copyright information to an SVG document
        - 17.4 Specifying a snapshot time
- Appendix A: DOM enhancements
    - A.1 DOM Level 3 Support
    - A.2 Media Interfaces
    - A.3 Conversion of Coordinates
    - A.4 Filtering DOM Events
    - A.5 Modification to getScreenCTM
    - A.6 Accessing the client CTM
    - A.7 Wheel event
    - A.8 Obtaining the bounds of rendered content
    - A.9 Notification of shape modification
        - A.9.1 The ShapeChange event
        - A.9.2 The RenderedBBoxChange event
- Appendix B: API enhancements
    - B.1 SVGTimer Interface
    - B.2 Network interfaces
        - B.2.1 URLRequest interface
        - B.2.2 Security Concerns
        - B.2.3 Socket Connections
    - B.3 Monitoring download progress
        - B.3.1 The Progress event
    - B.4 File Upload
        - B.4.1 Interface FileDialog
        - B.4.2 Interface FilesSelected
        - B.4.3 Interface FileList
        - B.4.4 Interface File
        - B.4.5 URLRequest additions
        - B.4.6 Connection additions
        - B.4.7 Security considerations
    - B.5 Persistent Client-side Data storage
    - B.6 Global Interface
- Appendix C: SVG DOM Subset
- Appendix D: Feature strings
- Appendix E: List of events
- Appendix F: RelaxNG schema for SVG 1.2
- Appendix G: Media Type registration for image/svg+xml
    - G.1 Introduction

    - G.2 Registration of Media Type image/svg+xml
- Appendix H: References
- Appendix I: Change log

Authors

The authors of this specification are the participants of the W3C SVG Working Group.

- Ola Andersson, ZOOMON AB
- Henric Axelsson, Ericsson AB
- Phil Armstrong, Corel Corporation
- Selim Balcısoy, Nokia
- Benoît Bézaire, Corel Corporation
- Robin Berjon, Expway
- Gordon Bowman, Corel Corporation
- Craig Brown, Canon Information Systems Research Australia
- Mike Bultrowicz, Savage Software
- Tolga Çapın, Nokia Inc.
- Mathias Larsson Carlander, Ericsson AB
- Jakob Cederquist, ZOOMON AB
- Suresh Chitturi, Nokia
- Charilaos Christopoulos, Ericsson AB
- Lee Cole, Quark
- Don Cone, America Online Inc.
- Alex Danilo, Canon Information Systems Research Australia
- Thomas DeWeese, Eastman Kodak
- Jean-Claude Dufourd, Group des Ecoles des Télécommunications (GET)
- Jon Ferraiolo, Adobe Systems Inc.
- Darryl Fuller, Schema Software
- 藤沢 淳 (FUJISAWA Jun), Canon
- Christophe Gillette, Motorola (formerly BitFlash)
- Rick Graham, BitFlash
- Vincent Hardy, Sun Microsystems Inc.
- 端山 貴也 (HAYAMA Takanari), KDDI Research Labs
- Scott Hayman, Research In Motion Limited
- Stephane Heintz, BitFlash
- Lofton Henderson, OASIS
- Ivan Herman, W3C
- Bin Hu, Motorola

- Michael Ingrassia, Nokia
- 石川 雅康 (ISHIKAWA Masayasu), W3C
- Dean Jackson, W3C *(W3C Team Contact)*
- Christophe Jolif, ILOG S.A.
- Lee Klosterman, Hewlett-Packard
- 小林 亜令 (KOBAYASHI Arei), KDDI Research Labs
- Thierry Kormann, ILOG S.A.
- Yuri Khramov, Schema Software
- Chris Lilley, W3C *(Working Group Chair)*
- Vincent Mahe, France Telecom
- Philip Mansfield, Schema Software
- 水口 充 (MINAKUCHI Mitsuru), Sharp Corporation
- Luc Minnebo, Agfa-Gevaert N.V.
- 小野 修一郎 (ONO Shuichiro), Sharp Corporation
- Lars Piepel, Vodafone
- Antoine Quint, Fuchsia Design (formerly of ILOG)
- Bruno David Simões Rodrigues, Vodafone
- 相良 毅 (SAGARA Takeshi), KDDI Research Labs
- Sebastian Schnitzenbaumer, SAP AG
- Bradley Sipes, ZOOMON AB
- Andrew Sledd, ZOOMON AB
- Пётр Сороткин (Peter Sorotokin), Adobe Systems Inc.
- 上田 宏高 (UEDA Hirotaka), Sharp Corporation
- Rick Yardumian, Canon Development Americas
- Charles Ying, Openwave Systems Inc.

Top | Next

1 Introduction

This is the specification for SVG 1.2, an extension to SVG 1.1 that provides features requested by the implementor and content design communities.

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

Defining an SVG 1.2 document

SVG 1.2 uses the same definitions for conforming SVG Documents and Document Fragments as SVG 1.1. The differences from SVG 1.1 are:

- The value of the version attribute on the root-most **svg** element must be "1.2".
- There is no official DTD for SVG 1.2, and therefore no official way to specify the DOCTYPE for an SVG 1.2 document. Instead, validation is provided by the SVG 1.2 RelaxNG schema.

The following is an example of an SVG 1.2 file:

```
<?xml version="1.0"?>

<svg xmlns="http://www.w3.org/2000/svg"
version="1.2">

  <rect x="10" y="10" width="10" height="10"/>

</svg>
```

Previous | Top | Next

2 Profiling SVG

The creation of SVG Viewers which correspond to profiles other than Tiny, Basic, or Full is discouraged; experience in SVG and other formats shows that proliferation of viewers with subtly differing capabilities is a hindrance to interoperability.

Sometimes, vertically-focused industries can improve interoperability by defining and clearly documenting an industry-specific profile which uses an existing **baseProfile** as a starting point.

On the other hand, creation of particular profiles for different types of content authoring is encouraged, provided the **baseProfile** is set appropriately to the closest standard profile which is a true superset of the authoring profile. Use of such documented profiles can aid interchange of graphical assets between authoring tools.

As an example, a profile aimed at technical illustration might be based on SVG Basic, omit filter effects, and retain animation and scripting to allow for interactive diagrams. A profile for interchange of graphics arts assets might be based on Full, retain filter effects, and omit animation, scripting, and foreignObject - thus ensuring that graphics conforming to that profile can be easily edited in a variety of graphical editors.

Previous | Top | Next

3 XML Binding Language for SVG

3.1 Introduction

SVG 1.2 supports sXBL for custom markup extensibility via pre-packaged components. sXBL allows developers to create re-usable, higher-level components expressed in a custom XML vocabulary (tag set), where the implementation of these components is defined in terms of a shadow tree containing SVG content. For instance, with sXBL, it is possible to define a set of flowcharting components expressed in a custom flowcharting XML grammar. The sXBL-enabled flowcharting components would achieve their effects by attaching appropriate shadow trees of low-level SVG path and text elements to the SVG document. The shadow trees could include any SVG facilities, such as SVG container elements, SVG graphics elements, animation elements, and low-level event handlers for interactivity.

Except for any SVG-specific rules explicitly mentioned in this specification, the normative definition for all XBL features used by SVG is the sXBL specification.

3.2 Simple example

The following simple example is provided to introduce the reader to the most basic aspects of how sXBL works. For detailed processing rules and further examples, refer to the sXBL specification.

The following is a simple SVG example where a custom element, **myNS:HelloWorld**, acquires the alternate presentation behavior defined by the **xbl:definition** element at the top of the file. The **myNS:HelloWorld** element will be rendered by using the contents of the shadow tree which is attached by the binding. The shadow tree consists of a **text** element which has the string "Hello, world, using sXBL" inside:

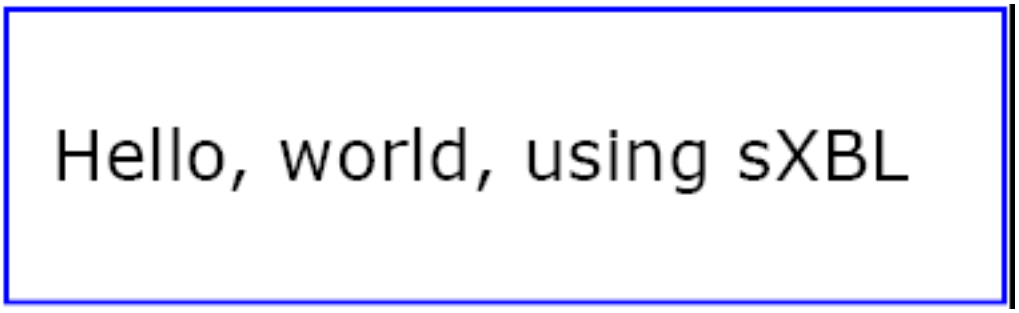
```
<?xml version="1.0"?>
<svg width="10cm" height="3cm" viewBox="0 0 200 60"
     xmlns="http://www.w3.org/2000/svg"
version="1.2"
     xmlns:xbl="http://www.w3.org/2004/xbl"
     xmsns:myNS="http://www.example.com">
  <title>Example xbl01.svg - "hello world" sample
file</title>
  <desc>A simple "Hello, world" sXBL example where
the
         rendering behavior of a custom element
'myNS:HelloWorld'
         consists of an 'svg:text' element which
has the string
         "Hello, world, using sXBL" inside.</desc>
  <defs>
    <xbl:xbl>

      <!-- The following 'xbl:definition' element
defines the
         presentation and interactive behavior that
must be used
         for all 'myNS:HelloWorld' elements in this
document. -->
      <xbl:definition element="myNS:HelloWorld>
        <xbl:template>
          <text>Hello, world, using sXBL</text>
        </xbl:template>
      </xbl:definition>

    </xbl:xbl>
  </defs>

  <rect x="1" y="1" width="198" height="58"
fill="none" stroke="blue"/>
  <g font-size="14" font-family="Verdana"
transform="translate(10,35)">

    <!-- Here is an instance of an 'myNS:
HelloWorld' element.
       The above binding definition attaches a
shadow tree which
       defines alternative rendering and
interactive behavior for this element.
       Instead of the standard SVG behavior where
unknown elements are not rendered,
       the binding definition causes an 'svg:text'
element to be rendered. -->
```

```
    <myNS:HelloWorld/>

  </g>

</svg>
```

The above example results in equivalent rendering to the following SVG. The highlighted sections below (i.e., the **g** and **text** elements) represent the shadow tree which the binding definition (i.e., the **xbl:definition** element defined above) attaches to the custom element. The SVG user agent renders the shadow tree in place of the custom element.

```
<svg width="10cm" height="3cm" viewBox="0 0 200 60"
      xmlns="http://www.w3.org/2000/svg"
version="1.2">
    <title>Example xbl01-equivalent.svg -
          equivalent rendering for "hello world"
sample file</title>

    <rect x="1" y="1" width="198" height="58"
fill="none" stroke="blue"/>
    <g font-size="14" font-family="Verdana"
transform="translate(10,35)">

      <!-- The document is rendered as if the 'myNS:
HelloWorld' element
          were replaced by an SVG 'g' element with
a 'text' element inside. -->
      <g>
        <text>Hello, world, using sXBL</text>
      </g>

    </g>

  </svg>
```

The result is as if the **g** element contained the text directly.



View this image as SVG (SVG 1.2 enabled browsers only)

3.3 sXBL restrictions with SVG

3.3.1 Standalone SVG documents restriction

The sXBL specification defines the terms bound document and the binding source document. The bound document contains the custom element (the "bound element"). The binding source document contains the binding definition (i. e., the **xbl:definition** element) for the custom element.

This specification only defines sXBL processing rules or stand-alone SVG documents where either of the following is true:

- the bound element and its corresponding binding definition are in the same document, and that document is a stand-alone SVG document
- the bound element is in one document and the binding definition is in a separate document, and both documents are stand-alone SVG documents

Future W3C specifications may define sXBL behavior in additional contexts (e. g., when an XBL-enabled SVG document fragment is embedded inline within a parent XHTML document).

3.3.2 Non-SVG element restriction

Within an SVG document fragment, sXBL binding definitions can only be applied to elements which are not in the SVG namespace. Documents which contain binding definitions for elements in the SVG namespace are in error.

The following sample code shows a binding definition for a custom element. This binding definition is conformant to this specification because the custom element belongs to a namespace different than the SVG namespace:

```
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xbl="http://www.w3.org/2004/xbl"
      xmlns:zzz="http://example.com">
  <xbl:xbl>
    <!-- This binding definition is allowed. -->
    <xbl:definition element="zzz:Widget1">
      <!-- ... -->
    </xbl:definition>
  </xbl:xbl>

  <zzz:Widget1/>
</svg>
```

The following example shows three binding definitions that are not allowed:

```
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:svg="http://www.w3.org/2000/svg">
      xmlns:xbl="http://www.w3.org/2004/xbl">
  <xbl:xbl>
    <!-- This binding definition is not allowed
because
         it attempts to define a custom element in
the SVG namespace.
         (Note that in this example the default
namespace
         is the SVG namespace.) -->
    <xbl:definition element="MySVGWidget1">
      <!-- ... -->
    </xbl:definition>
    <!-- Also not allowed. (This time the SVG
prefix is used explicitly.) -->
    <xbl:definition element="svg:MySVGWidget2">
      <!-- ... -->
    </xbl:definition>
    <!-- Redefining SVG elements is also an error.
-->
    <xbl:definition element="svg:path">
      <!-- ... -->
    </xbl:definition>
  </xbl:xbl>

  <MySVGWidget1/>
  <svg:MySVGWidget2/>
</svg>
```

3.4 sXBL rendering and event handling rules with SVG

When a custom element has an sXBL binding definition, the SVG user agent processes binding definitions according to the rules defined in the sXBL specification. SXBL processing of bindings may result in shadow content being attached to custom elements. If shadow content is attached to a custom element (i.e., if the xblShadowTree DOM attribute on the NodeXBL interface is not null), then the custom element's rendering and interactive behavior is defined by the shadow content. More specifically, except for other sXBL-specific details described in this chapter or the sXBL specification, the custom element is renderered and events are handled as follows:

- If an sXBL binding is attached to the custom element and if the xblShadowTree DOM attribute on the NodeXBL interface is not null, then the shadow tree defines the rendering and interactive behavior for the custom element.

  Whereas the standard SVG rule is that non-SVG elements are not rendered (i.e., they behave as if the **display** property is 'none', which is equivalent in rendering behavior to a **defs** element), when an sXBL binding is attached to a custom element and xblShadowTree is not null, the lists of nodes on the xblChildNodes DOM attribute on the NodeXBL interface specify the rendering and interactive behavior of the custom element. Thus, when the custom element is within a rendering context (versus non-rendering contexts, such as described under sXBL bindings for SVG resources and sXBL bindings for visual effects), then the custom element behaves as if it were a **g** element with no attributes and the nodes on xblChildNodes were the children of the **g**.

- The current user coordinate system for the list of nodes on the xblChildNodes DOM attribute before application of any 'transform' attributes or other coordinate system transformations defined on these nodes is the current user coordinate system on the custom element. For example, suppose the bound document contained the following:

```
<svg...>
<g transform="...">
  <circle.../>
```

```
    <MyNS:MyCustomElement/>
  </g>
  <!-- ... -->
</svg>
```

And suppose the complete shadow tree for **MyNS:MyCustomElement** consists of a **rect** and a **rect** element as follows:

```
<rect x=... y=... width=... height=... />
<text transform="...">...</text>
```

Then the current user coordinate system for the **rect** in the shadow content will be the same as the current user coordinate system for the **circle** within the bound document.

The **text** element, however, includes an additional transform attribute. Because of this, the additional transformation specified by the transform attribute is concatenated onto the current user coordinate system inherited from the bound element to define a new current user coordinate system for the **text** element.

- Events flow between the custom element and its shadow content as described in the sXBL specification.

It is important to note that all attributes on custom elements, including SVG-defined attributes, have no direct effect on rendering or interactive behavior. For example, suppose a custom element's schema allowed for **fill** and **stroke** attributes (which are SVG-defined attributes), and further suppose that the following custom element appeared within an SVG file:

```
<svg...>
<xbl:import bindings="..."/>
<MyNS:MyCustomElement fill="red" stroke="blue"/>
</svg>
```

The **fill** and **stroke** attributes are not processed as SVG presentation attributes. As far as the SVG user agent is concerned, unless the binding itself has specific logic to recognize these attributes, the user agent treats these attributes as custom attributes on a custom element and thus the attributes have no effect on rendering or interactive behavior. Note also that attributes in the SVG namespace (e.g., **svg:fill** or **svg:transform**) are treated in the same manner —

the user agent treats these attributes as custom attributes on a custom element.

3.5 sXBL bindings for SVG resources

In the context of SVG, sXBL can be used to define custom gradients, patterns, solid color definitions, markers, clipping paths, masks, and filters. Here is an example of an sXBL binding which defines a custom linear gradient:

```
<!-- Binding source document -->
<?xml version="1.0"?>
<svg
     xmlns="http://www.w3.org/2000/svg"
version="1.2"
     xmlns:xbl="http://www.w3.org/2004/xbl"
     xmsns:myNS="http://www.example.com">
  <defs>
    <xbl:xbl>

      <xbl:definition element="myNS:
BlackWhiteGradient">
        <xbl:template>
          <linearGradient>
            <stop offset="0" stop-color="black"/>
            <stop offset="1" stop-color="white"/>
          </linearGradient>
        </xbl:template>
      </xbl:definition>

    </xbl:xbl>
  </defs>
</svg>

<!-- Bound document -->
<?xml version="1.0"?>
<svg width="10cm" height="3cm" viewBox="0 0 200 60"
     xmlns="http://www.w3.org/2000/svg"
version="1.2"
     xmlns:xbl="http://www.w3.org/2004/xbl"
     xmsns:myNS="http://www.example.com">
  <title>Example xbl02.svg - defining a custom
gradient</title>

  <xbl:import bindings="xbl02-bindings.svg"/>

  <!-- Instantiate the custom gradient -->
  <myNS:BlackWhiteGradient xml:id="grad1"/>
```

```
  <!-- Use the custom gradient to fill a rectangle
-->
  <rect x="1" y="1" width="198" height="58"
        fill="url(#grad1)" stroke="blue"/>
</svg>
```

A reference to a given resource type (e.g., fill="url(#grad1)") is a legal reference if the URI reference resolves to either of the following:

- The URI reference can resolve directly to a resource element of the proper type for the given reference. For example, the 'fill' property can contain a direct URI reference to a **linearGradient** element.
- The URI reference can resolve indirectly to a resource element of the proper type. Instead of directly pointing to a resource of the proper type, the URI reference can resolve to a custom element with an sXBL binding which is defined such that a resource of the proper type is the first and only child element on the xblChildNodes DOM attribute on the NodeXBL interface.. If the URI reference points to a custom element with an sXBL binding, but the first and only child element on xblChildNodes is not the correct type of element for the given reference, then the document is in error.

The following resource types can be defined using sXBL custom elements:

- **linearGradient**
- **radialGradient**
- **pattern**
- **solidColor**
- **marker**
- **clipPath**
- **mask**
- **filter**
- **vectorEffect**

3.6 sXBL bindings for visual effects

In the context of SVG, sXBL can be used to define custom effects such as custom animations, custom filter primitives, custom vector effects, custom gradient stops, and custom geometry for clipping paths. Here is an example of an sXBL binding which defines a custom opacity fade animation:

```
<?xml version="1.0"?>
<svg width="10cm" height="3cm" viewBox="0 0 200 60"
     xmlns="http://www.w3.org/2000/svg"
version="1.2"
     xmlns:xbl="http://www.w3.org/2004/xbl"
     xmsns:myNS="http://www.example.com">
  <title>Example xbl02.svg - defining a custom
animation</title>
  <defs>
    <xbl:xbl>

      <xbl:definition element="myNS:FadeOnClick">
        <xbl:template>
          <animate attributeName="opacity"
begin="click" dur="3s" from="1" to="0"/>
        </xbl:template>
      </xbl:definition>

    </xbl:xbl>
  </defs>

  <rect x="1" y="1" width="198" height="58"
fill="red" stroke="blue">
    <!-- The rectangle will fade to invisible when
the user clicks on it. -->
    <myNS:FadeOnClick/>
  </rect>
</svg>
```

For all of these custom visual effects scenarios, a visual effect can be defined in either of the following ways:

- Direct definition. The given visual effect element (e.g., an animation element, a filter primitive, a vector effect primitive, or a gradient **stop** element) can be included directly in the SVG document, as in the following:

```
<rect ...>
  <animate ...>
</rect>
```

- Indirect definition using an sXBL binding on a custom element. If a custom element has an sXBL binding, after sXBL processing, the child elements

on the xblChildNodes DOM attribute on the NodeXBL interface represent an appropriate sequence of visual effects elements all of the same type (e.g., all are animation elements, all are filter primitives, all are vector effect primitives, all are gradient **stop** elements, all are appropriate shape elements or **use** elements as required for **clipPath** content), then the custom element is processed as if the elements on xblChildNodes replaced the custom element in the SVG document.

If the custom element fulfills some of the requirements for a custom visual effect, but does not fulfill all of the requirements, then the custom element is in error.

Animation elements in SVG change the attributes or properties over time on a target element. If the animation element has an xlink:href attribute, then the URI reference defined by the xlink:href identifies the target element; otherwise, the target element is the parent of the animation element. For XBL-defined custom animations, however, SVG's standard rules for identifying the target element are not workable because XPointer references within the shadow tree are relative to the binding source document, not the bound document, and because the DOM Core parent element in the shadow tree is not the bound element. Because of these issues, the following additional rules apply to identify the target element for any animation elements within shadow trees:

- If the custom element has an explicit **xlink:href** attribute pointing to element E, then all animation elements within the list of xblChildNodes on the NodeXBL interface will use element E as their target element.
- If the custom element does not have an explicit **xlink:href** attribute, and custom element's parent is element P, then all animation elements within the list of xblChildNodes on the NodeXBL interface will use element P as their target element.
- It is an error for animation elements within the list of xblChildNodes to include an **xlink:href** attribute.

3.7 The traitDef element

A trait is a potentially animatable parameter associated with an element. A trait is the value that gets assigned through an XML attribute or CSS style or SMIL animation. In the case of sXBL, it describes an attribute on a custom element, allowing it to be exposed to the animation engine and allowing typed-base DOM access using method calls such as getFloatTrait().

Traits for custom elements are described using the traitDef element.

traitDef Schema

```
  <define name='traitDef'>
    <element name='traitDef'>
      <ref name='attlist.traitDef'/>
    </element>
  </define>

  <define name='attlist.traitDef'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <attribute name='name' svg:
animatable='false' svg:inheritable='false'/>
    <attribute name='namespace' svg:
animatable='false' svg:inheritable='false'>
      <ref name='URI.datatype'/>
    </attribute>
    <attribute name='valueType' svg:
animatable='false' svg:inheritable='false'>
      <choice>
        <value>http://www.w3.org/2001/
XMLSchema#string</value>
        <value>http://www.w3.org/2001/
XMLSchema#float</value>
        <value>http://www.w3.org/2000/
svg#SVGLength</value>
        <value>http://www.w3.org/2000/
svg#SVGMatrix</value>
        <value>http://www.w3.org/2000/
svg#SVGPath</value>
        <value>http://www.w3.org/2000/
svg#SVGRect</value>
        <value>http://www.w3.org/2000/
svg#SVGRGBColor</value>
      </choice>
    </attribute>
  </define>
```

The **name** and **namespace** attributes respectively specify the name and namespace of the attribute.

The **valueType** attribute defines the datatype for the given attribute as a URI in a format compatible with XML Schema Datatypes. For example, a "float" datatype is defined by the URI of "http://www.w3.org/2001/XMLSchema#float". The following values are supported:

- http://www.w3.org/2001/XMLSchema#string
- http://www.w3.org/2001/XMLSchema#float
- http://www.w3.org/2000/svg#SVGLength
- http://www.w3.org/2000/svg#SVGMatrix
- http://www.w3.org/2000/svg#SVGPath
- http://www.w3.org/2000/svg#SVGRect
- http://www.w3.org/2000/svg#SVGRGBColor

Below is an example that defines traits for the **x**, **y**, **width** and **height** attributes for a custom **foo:button** element:

```
<xbl:definition element="foo:button">
  <traitDef name="x" namespace="" valueType="http://
www.w3.org/2000/svg#SVGLength"/>
  <traitDef name="y" namespace="" valueType="http://
www.w3.org/2000/svg#SVGLength"/>
  <traitDef name="width" namespace=""
valueType="http://www.w3.org/2000/svg#SVGLength"/>
  <traitDef name="height" namespace=""
valueType="http://www.w3.org/2000/svg#SVGLength"/>
  <xbl:template>
    ....
  </xbl:template>
</xbl:definition>
```

3.8 Trait mutation events

If a trait value or the animated value of a trait changes for any reason, a TraitMutationEvent is fired.

```
interface TraitMutationEvent : events::Event
{
        // event is dispatched to the element and does not
bubble
        // event names are "TraitValueChanged" and
"TraitAnimValueChanged"
        DOMString traitNamespace;
        DOMString traitLocalName;
}
```

A trait change is different from a regular attribute change. For instance a numeric trait changing its attribute value from "1" to "1.0" does not change the value of the trait. The value of the trait is not attached to the event, because traits can be

of different types. The value must be retrieved from the element.

3.9 Document loading

The sXBL specification defines the term binding source document to represent a document which contains sXBL binding definitions. Binding definitions can be in the original document (what sXBL refers to as the bound document) or in a separate, externally referenced document.

Externally referenced source binding documents are managed in the same manner as SVG manages all externally referenced documents.

3.10 Scripts and handlers

Any **script** and **handler** elements in the SVG namespace contained within sXBL subtrees are treated in the same manner as **script** elements contained directly within SVG content. The defined scripting logic belongs to the document referenced by the ownerDocument DOM attribute on the DOM object corresponding to the given **script** or **handler** element. The scripting context for any scripting logic invoked within shadow trees is the same as any scripting logic defined within the binding source document. Thus, in ECMAScript, if the binding source document has a script element which defines a findAttribute function which is available on a document-global basis, then shadow trees are able to invoke the findAttribute function in the same way as the original content of the binding source document.

3.11 CSS Stylesheets

For binding source documents, any CSS stylesheets embedded within a **style** element in the SVG namespace or referenced by an XML stylesheet processing instruction or a **style** element in the SVG namespace are applied to the document which embeds or contains the reference to the given stylesheet, even if the **style** element is embedded within an sXBL subtree. In particular, any CSS stylesheets embedded within or referenced by a **style** element in the SVG namespace apply to the document referenced by the ownerDocument DOM attribute on the DOM object corresponding to the given **style** element.

Because shadow trees can be regenerated repeatedly during the time a document is presented to the user, there is a potentially difficult specification and implementation issue if **style** elements were allowed to be built dynamically by

sXBL binding definitions and inserted into shadow trees. In order to avoid these complications, this specification includes the restriction that is an error to include **style** elements and xml-stylesheet processing instructions in shadow trees. If either is included within a shadow tree, the document is in error. Note that it is possible to include **style** elements and xml-stylesheet processing instructions within binding source documents and that the corresponding style sheets will apply to the shadow content; thus, this restriction only has to do with the shadow content that is generated by the binding definitions. (Note: Future W3C specifications might remove this restriction partially or completely.)

3.12 Focus and navigation

Shadow content participates in keyboard focus and navigation (e.g., via tabbing between focusable fields) as if the shadow content is replacing the corresponding custom element. For example, suppose there is an sXBL-defined custom element **MyNS:NameWidget** which creates a shadow tree of three SVG editable **text** elements. SVG user agents must allow navigation from editable **text** elements in the original SVG document into the shadow content's three editable **text** elements and back again as if the focusable fields in the shadow content were part of the original document.

The one difference in field navigation behavior due to the presence of sXBL shadow content is that the 'nav-index' property has hierarchical behavior when an sXBL-defined custom element contains focusable elements. The definition for this hierachical behavior for 'nav-index' can be found in the focus section in the sXBL specification.

DOM interfaces for custom elements

All non-SVG elements within an SVG document support the xbl::NodeXBL interface defined in the sXBL specification and also support all of the same interfaces as the **g** element (e.g., all of the methods and properties of SVGElement, SVGTests, SVGLangSpace, SVGExternalResourcesRequired, SVGStyleable, SVGTransformable, events::EventTarget). This allows, for example, the ability to call method getBBox() on a custom element which has renderable content in its shadow tree.

3.13 Error handling

sXBL content within an SVG document is subject to the same error processing rules as the rest of the SVG document. In addition to the standard SVG rules

about correct document formulation, an SVG document containing sXBL elements is technically in error when:

- sXBL elements are included in the content tree in locations not explicitly allowed by the SVG schema. For example, it is an error to include an **xbl:handlerGroup** element as a direct child of an **svg:path** element since this is not allowed by the SVG schema.
- When an sXBL element has child content, an attribute value or a property value which is not permissible according to this specification or the sXBL specification.
- Other situations that are described as being in error in this specification or the sXBL specification.

4 Flowing text and graphics

SVG 1.2 enables a block of text and graphics to be rendered inside a shape while automatically wrapping the objects into lines using the **flowRoot** element. The idea is to mirror, as far as practical, the existing SVG text elements.

4.1 The flowRoot element

The **flowRoot** element specifies a block of graphics and text to be rendered with line wrapping. It contains at least one **flowRegion** element that defines regions into which the children elements of the **flowRoot** should be flowed.

flowRoot Schema

```
<define name='flowRoot'>
  <element name='flowRoot'>
    <ref name='attlist.flowRoot'/>
    <ref name='SVG.flowRoot.content'/>
  </element>
</define>

<define name='attlist.flowRoot' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.Focusable.attrib'/>
  <ref name='SVG.flowalign.attlist'/>
</define>

<define name='SVG.flowRoot.content'>
  <zeroOrMore>
    <ref name='SVG.Description.class'/>
  </zeroOrMore>
  <ref name='SVG.flowRoot.class'/>
</define>

<define name='SVG.flowRoot.class'>
  <ref name='flowRegion'/>
  <zeroOrMore>
    <ref name='SVG.moreFlowRegions.class'/>
  </zeroOrMore>
  <oneOrMore>
    <ref name='flowPara'/>
  </oneOrMore>
</define>

<define name='SVG.flowRoot.class' combine='interleave'>
```

```
    <interleave>
      <ref name='flowRegionExclude'/>
      <zeroOrMore>
        <ref name='flowDiv'/>
      </zeroOrMore>
    </interleave>
  </define>

  <define name='SVG.moreFlowRegions.class' combine='interleave'>
    <ref name='flowRegion'/>
  </define>

  <define name='SVG.flowalign.attlist' combine='interleave'>
    <optional>
      <attribute name='text-align' svg:animatable='true' svg:
inheritable='true'/>
      <attribute name='display-align' svg:animatable='true' svg:
inheritable='true'/>
    </optional>
  </define>
```

4.2 The flowRegion element

The **flowRegion** element contains a set of shapes and exclusion regions in which the text content of a parent **flowRoot** element is drawn into. A **flowRegion** element has basic shapes and **path** elements as children, as well as a **flowRegionExclude** element. The children of a **flowRegion** element are inserted into the rendering tree before the text is drawn and have the same rendering behavior as if they were children of a **g** element.

The child elements create a sequence of shapes in which the text content for the parent **flowRoot** will be drawn. Once the text fills a shape it flows into the next shape. The **flowRegionExclude** child describes a set of regions into which text will not be drawn, such as a cutout from a rectangular block of text.

The child elements of a **flowRegion** can be transformed as usual but the text is always laid out in the coordinate system of the **flowRoot** element. For example, a **rect** child with a 45 degree rotation transformation will appear as a diamond but the text will be aligned along the regular axis.

flowRegion Schema

```
  <define name='flowRegion'>
    <element name='flowRegion'>
      <ref name='attlist.flowRegion'/>
      <ref name='SVG.flowRegion.content'/>
    </element>
  </define>

  <define name='attlist.flowRegion' combine='interleave'>
    <ref name='SVG.Core.attrib'/>
  </define>

  <define name='SVG.flowRegion.content'>
```

```
  <ref name='SVG.flowRegion.class'/>
</define>

<define name='SVG.flowRegion.class'>
  <ref name='rect'/>
</define>

<define name='SVG.flowRegion.class' combine='choice'>
  <choice>
    <ref name='g'/>
    <ref name='use'/>
    <ref name='text'/>
    <ref name='SVG.Shape.class'/>
  </choice>
</define>
```

4.3 The flowRegionExclude element

The **flowRegionExclude** element contains a set of shapes defining regions in which flowed text is not drawn. It can be used to create exclusion regions from within a region of text.

If **flowRegionExclude** is a child of a **flowRegion** then it describes an exclusion region for that particular **flowRegion**. If it is a child of **flowRoot** then it describes exclusion regions for all **flowRegion** children of the **flowRoot**.

flowRegionExclude Schema

```
<define name='flowRegionExclude'>
  <element name='flowRegionExclude'>
    <ref name='attlist.flowRegion'/>
    <ref name='SVG.flowRegion.content'/>
  </element>
</define>
```

4.4 The flowDiv element

The **flowDiv** element specifies a block of text and/or graphics to be inserted into the layout, and marks it as a division of related elements. The children of the **flowDiv** element will be rendered as a block and offset from their parent's siblings both before and after. By separating the logical order of text (in successive **flowDiv** elements) from the physical layout (in regions, which can be presented anywhere on the canvas) the SVG document structure encourages creation of a default, meaningful linear reading order while preserving artistic freedom for layout. This enhances accessibility.

flowDiv Schema

```
<define name='flowDiv'>
  <element name='flowDiv'>
    <ref name='attlist.flowDiv'/>
    <ref name='SVG.flowDiv.content'/>
  </element>
</define>
```

```
<define name='attlist.flowDiv' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Style.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.GraphicalEvents.attrib'/>
</define>

<define name='SVG.flowDiv.content'>
  <zeroOrMore>
    <ref name='SVG.Description.class'/>
    <ref name='flowPara'/>
    <ref name='flowRegionBreak'/>
  </zeroOrMore>
</define>
```

4.5 The flowPara element

The **flowPara** element marks a block of text and graphics as a logical paragraph.

flowPara Schema

```
<define name='flowPara'>
  <element name='flowPara'>
    <ref name='attlist.flowPara'/>
    <ref name='SVG.flowPara.content'/>
  </element>
</define>

<define name='attlist.flowPara' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.Focusable.attrib'/>
  <ref name='SVG.Editable.attrib'/>
  <ref name='SVG.flowalign.attlist'/>
</define>

<define name='SVG.flowPara.content'>
  <zeroOrMore>
    <ref name='SVG.flowPara.class'/>
  </zeroOrMore>
</define>

<define name='SVG.flowPara.class'>
  <choice>
    <ref name='flowSpan'/>
    <text/>
  </choice>
</define>
```

4.6 The flowSpan element

The **flowSpan** element specifies a block of text to be rendered inline, and marks the text as a related span of words. The **flowSpan** element is typically used to allow a subset of the text block, of which it

is a child, to be rendered in a different style or to mark it as being in a different language.

flowSpan Schema

```
<define name='flowSpan'>
  <element name='flowSpan'>
    <ref name='attlist.flowSpan'/>
    <ref name='SVG.flowSpan.content'/>
  </element>
</define>

<define name='attlist.flowSpan' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.Focusable.attrib'/>
</define>

<define name='SVG.flowSpan.content'>
  <zeroOrMore>
    <ref name='SVG.flowSpan.class'/>
  </zeroOrMore>
</define>

<define name='SVG.flowSpan.class'>
  <choice>
    <ref name='flowSpan'/>
    <text/>
  </choice>
</define>
```

4.7 The flowRegionBreak element

When the **flowRegionBreak** element is inserted into the text stream it causes the text to stop flowing into the current region at that point. The text after the **flowRegionBreak** element begins in the next region. If there are no more regions into which text could flow, then the text will stop being rendered at the point of the **flowRegionBreak**.

flowRegionBreak Schema

```
<define name='flowRegionBreak'>
  <element name='flowRegionBreak'>
    <ref name='attlist.flowRegionBreak'/>
    <ref name='SVG.flowRegionBreak.content'/>
  </element>
</define>

<define name='attlist.flowRegionBreak' combine='interleave'>
  <empty/>
</define>

<define name='SVG.flowRegionBreak.content'>
  <empty/>
</define>
```

4.8 The flowLine element

The **flowLine** element is used to force a line break in the text flow. The content following the end of a **flowLine** element will be placed on the next available strip in the **flowRegion** that does not already contain text. This happens even if the **flowLine** element has no children.

If there are no printable characters between adjacent **flowLine** elements then only the first **flowLine** element is rendered.

In all other aspects, the **flowLine** element is functionally equivalent to the **flowSpan** element.

flowLine Schema

```
<define name='flowLine'>
  <element name='flowLine'>
    <ref name='attlist.flowLine'/>
    <ref name='SVG.flowLine.content'/>
  </element>
</define>

<define name='attlist.flowLine' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Style.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.GraphicalEvents.attrib'/>
</define>

<define name='SVG.flowLine.content'>
  <zeroOrMore>
    <ref name='flowSpan'/>
    <ref name='flowImage'/>
    <ref name='flowRegionBreak'/>
  </zeroOrMore>
</define>
```

4.9 The flowTref element

The **flowTref** element is used to insert the child text content of a referenced element. It's effect is analogous to the **tref** element.

flowTref Schema

```
<define name='flowTref'>
  <element name='flowTref'>
    <ref name='attlist.flowTref'/>
    <ref name='SVG.flowTref.content'/>
  </element>
</define>

<define name='attlist.flowTref' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
```

```
    <ref name='SVG.Style.attrib'/>
    <ref name='SVG.Presentation.attrib'/>
    <ref name='SVG.GraphicalEvents.attrib'/>
    <ref name='SVG.XLinkRequired.attrib'/>
  </define>

  <define name='SVG.flowTref.content'>
    <empty/>
  </define>
```

4.10 The flowImage element

The **flowImage** element defines a container for graphics which are to be rendered inline in the text layout. It can be used to insert images or any other graphic object that will flow inline with the text flows.

The **flowImage** element establishes a new viewport for contained graphic elements. If **flowImage** specifies an absolute size, that size is used as the bounding rectangle for the flowImage region. If **flowImage** specifies a percentage as its size, the percentage is represented as a percentage of the current viewport.

In the absence of either **width** or **height** on the **flowImage** element, no new viewport is established. Any contained graphic elements are sized relative to the current viewport. In that case, the bounds of the **flowImage** element are calculated from the bounding box of any contained child graphic elements.

flowImage Schema

```
  <define name='flowImage'>
    <element name='flowImage'>
      <ref name='attlist.flowImage'/>
      <ref name='SVG.flowImage.content'/>
    </element>
  </define>

  <define name='attlist.flowImage' combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <ref name='SVG.Style.attrib'/>
    <ref name='SVG.Presentation.attrib'/>
    <ref name='SVG.GraphicalEvents.attrib'/>
  </define>

  <define name='SVG.flowImage.content'>
    <zeroOrMore>
      <ref name='g'/>
      <ref name='use'/>
      <ref name='text'/>
      <ref name='image'/>
      <ref name='video'/>
      <ref name='flowRoot'/>
      <ref name='flowRef'/>
      <ref name='SVG.Shape.class'/>
      <ref name='SVG.Text.class'/>
    </zeroOrMore>
```

```
  </define>
```

4.11 The flowRef element

The **flowRef** element references a **flowRegion** element. It causes the referenced element's geometry to be drawn in the current user coordinate system along with the text that was flowed into the region.

flowRef Schema

```
<define name='flowRef'>
  <element name='flowRef'>
    <ref name='attlist.flowRef'/>
    <ref name='SVG.flowRef.content'/>
  </element>
</define>

<define name='attlist.flowRef' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Style.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.GraphicalEvents.attrib'/>
  <ref name='SVG.XLinkRequired.attrib'/>
</define>

<define name='SVG.flowRef.content'>
  <zeroOrMore>
    <ref name='SVG.Description.class'/>
  </zeroOrMore>
</define>
```

4.12 Text Flow

Text flow is defined as a post processing step to the standard text layout model of SVG. At a high level the steps for flowing text are as follows:

1. The text is then 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. 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.
4. 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.
5. Each Glyph Group has two extents calculated: it's normal extent, and it's last in text region 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 region. The last in text region extent includes the advance of a trailing soft hyphens but does not include the advance of trailing whitespace or combining marks (ABC width?). The last in text region extent is used when this glyph group is from the last word (in logical order) in this text region.

6. The location of the first strip is determined based on the first word in logical order (see Calculating Text Regions and determining strip location).
7. Words are added to the current Strip in logical order. All the Glyph Groups from a word must be in the same strip and all the glyphs from a Glyph Group must be in the same Text Region.

   When a word is added the line height may increase, it can never decrease from the first word. An increase in the line height can only reduce the space available for text placement in the span.

   The span will have the maximum possible number of words.

8. 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 region and each subsequent glyph at the location of the glyph following the preceding selected glyph (in display order).
9. The next word is selected and the next strip location is determined. Goto Step 7.

4.13 Determining Strip Location

To determine the placement of a strip the Glyph Groups from first word is used. The initial position for the strip is calculated, taking into account the end (in non text progression direction) of the previous strip and the appropriate margin properties.

The line-box is calculated using the initial position as the top/right edge of the line-box, and the line-height of the first word. The 'bottom/right' edge of the line-box must be checked against the margin properties, if it lies within the margin then processing moves to the next flow region.

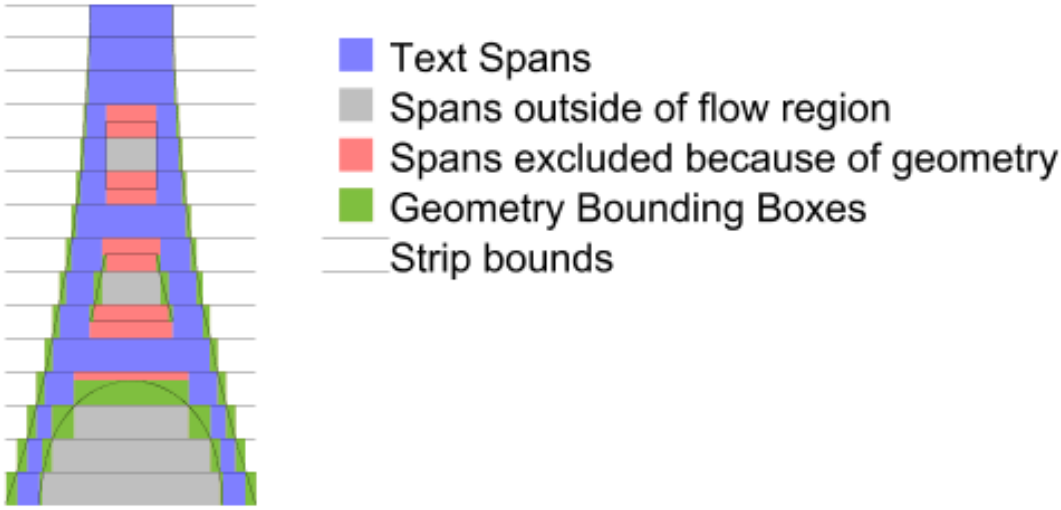
Once the line-box is calculated the Strip and it's associated Text Regions are calculated (see: Calculating Text Regions). If the first word can be placed in the text regions of this Strip then this location is used for the next line of text. If the first word does not fit then the top/right edge is shifted by 'line-advance' and the new line-box is checked. This proceeds until the word fits or end of the flow region is reached at which point processing moves to the next flow region.

4.13.1 Calculating Text Regions

In order to flow text into arbitrary regions it is necessary to calculate what areas of the arbitrary region are available for text placement.

Firstly, the flow region geometry is intersected with the current line-box. The result of this intersection is referred to as the strip. The strip is then split into text regions where ever a piece of geometry from the flow region 'intrudes'. It is important to ignore edges & points that are co-incident with the top or bottom of the line-box.

The diagram below shows the text strips used on a given shape.



The following is a more detailed description of the algorithm:

The current flow region and any applicable exclude regions must be combined into one piece of geometry, simply concatenating the geometry is sufficient as this entire algorithm deals simply with segments of the paths and does not use directionality information until the inclusion tests at the end. The result of the concatenation of the geometry is referred to as the flow geometry.

Next the line-box is calculated, from the top/right edge of the line, the line-height and the bounding box of the flow region. This line-box is intersection with the flow geometry, clipping the flow geometry segments to the line box.

The bounding box is then calculated separately for each of the segments in the intersection.

The left and right (top and bottom respectively for vertical text) edges of the bounding boxes are sorted in increasing coordinate order (x for horizontal text, y for vertical text), for edges at the same location the left/top (or opening) edge is considered less than right/bottom (or closing) edges. The following pseudo code then generates the list of open areas for the current line:

```
Edge [] segs = ...; // The sorted list of edges.

Edge edge = segs[0];
int count = 1;
double start = 0;
for (i=1; i<segs.length; i++) {
    edge = segs[i];
    if (edge.open) {
      // 'open' is true, this is the start of a block out region.
        if (count == 0) {
            // End of an open region so record it.
            rgns.add(new TextRegion(start, edge.loc));
        }
        count++;
    } else {
      // 'open' is false, this edge is the end of a block out region.
        count--;
        if (count == 0) {
            // start of an open area remember it.
```

```
                    start = edge.loc;
                }
            }
        }
```

This gives the regions of the strip that are unobstructed by any flow geometry (from either exclusion or flow regions), however those regions may be outside the flow region (such as in a hole, such as the middle of an 'O'), or inside an exclusion region. Thus the center of each rectangle should be checked first to see if it lies inside any exclusion region if so the rectangle is removed from the list. Second it must be checked for inclusion in the flow region, if it is inside the flow region then the rectangle is available for text placement and becomes a text region for the current strip.

Once all the text regions for a strip are located left and right Margins for horizontal text (top and bottom margins for vertical) as well as indent are applied. Margins are applied to each text region. For the first span in a paragraph (flowPara for flowRegionBreak) the indent is added to the appropriate margin of the first text region. For left to right text this is the left margin of the left most text region, for right to left text this is the right margin of the right most text region, and for vertical text is the top margin of the top most text region.

If the left/right (top/bottom) edges of a text region pass each other due to the application of margins (or indent) the text region is removed from the list. If the text region removed had indent applied the indent is not applied to the next text region in text progression direction it is simply ignored.

Flowing text using system fonts is a difficult operation. Content developers should not expect reproducible results between implementations. The most likely scenario for a reproducible result, although still not completely guaranteed, will be achieved by using SVG Fonts.

4.14 Alignment

4.14.1 The text-align property

Alignment in the inline progression direction in flowing text is provided by the text-align property. It is a modified version of the CSS3 property.

text-align

Value:	start | end | center | justify
Initial:	start
Applies to:	flowText, flowPara, flowDiv elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

For details refer to the CSS3 Text Module. Note that SVG does not allow the value "string" for this property, and that the values "left" and "right" have been removed as they do not make sense in an internationalized context.

The values "start" and "end" are dependent on the writing system being used.

- For left to right horizontal (English, French, etc): start=left and end=right
- For right to left horizontal (Hebrew, Arabic, etc): start=right and end=left
- For top to bottom vertical (vertical Chinese, etc): start=up and end=down

4.14.2 The progression-align property

Alignment in the direction of line progression for flowing text is provided by the **progression-align** property.

progression-align

Value:	before | after | center
Initial:	before
Applies to:	flowRoot, flowPara, flowDiv elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

The **progression-align** property causes blocks of flowed text to align within their containing regions. If the line progression direction is left to right, a setting of "before" will align the block of text at the top of the region, a setting of "after" will align the text at the bottom of the region, and a setting of "center" will vertically center the block of flowed text. The combined line heights of the entire flowed text is used for alignment (as opposed to maximum glyph extents on the flowed text lines).

4.14.3 Overflow

When the last element within a flow cannot be placed within the specified flow regions due to lack of space, then the **flowRegion** element is in an "overflow" state. The opposite state, the "underflow" state, is when the last element within a given flow can be placed within the given flow region.

Whenever a flow region changes from underflow state to overflow state, then the {"http://www.w3.org/2000/svg", "overflow"} event is fired. Whenever a flow region changes from overflow state to underflow state, then the {"http://www.w3.org/2000/svg", "underflow"} event is fired. These events are only fired with state changes. If a flow region is already in the overflow state and new content is appended to the end of the flow, then the state has not changed and therefore the {"http://www.w3.org/2000/svg", "overflow"} must not be fired.

The following is the definition of interface SVGOverflowEvent which is the event interface corresponding to the "overflow" and "underflow" events:

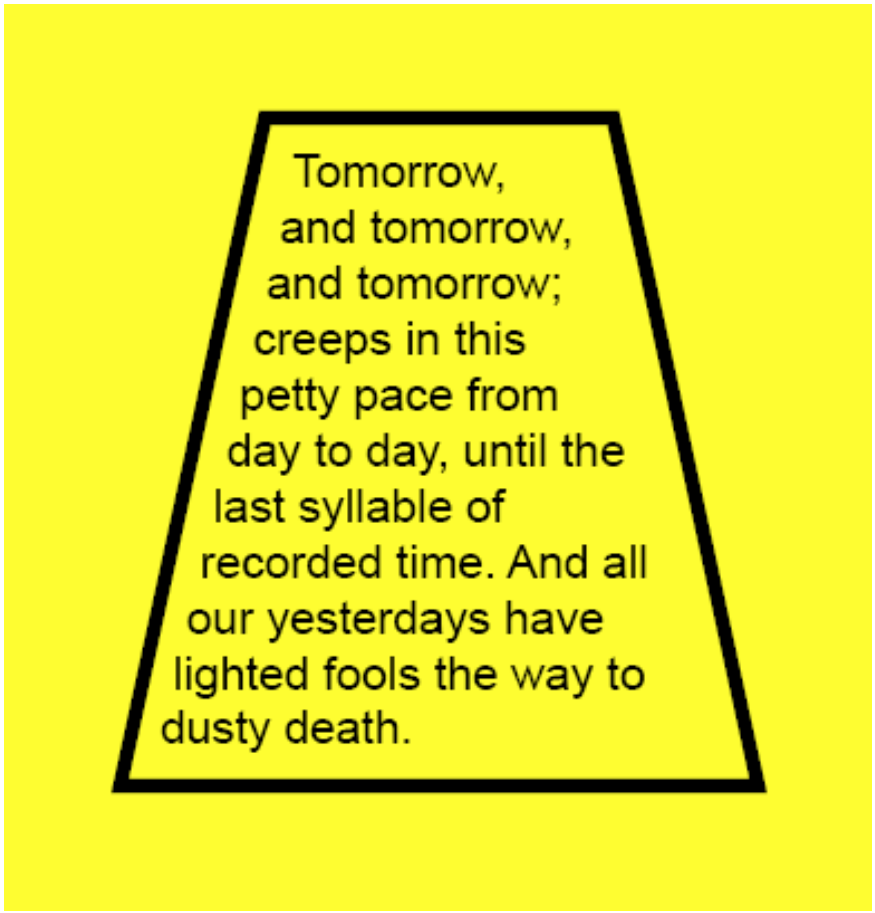
```
interface SVGOverflowEvent : events::Event
{
  readonly attribute SVGFlowParaElement
flowPara;
}
```

The currentTarget for the event is the **flowRegion** element whose overflow state has changed to underflow state, or vice versa.

4.15 Example

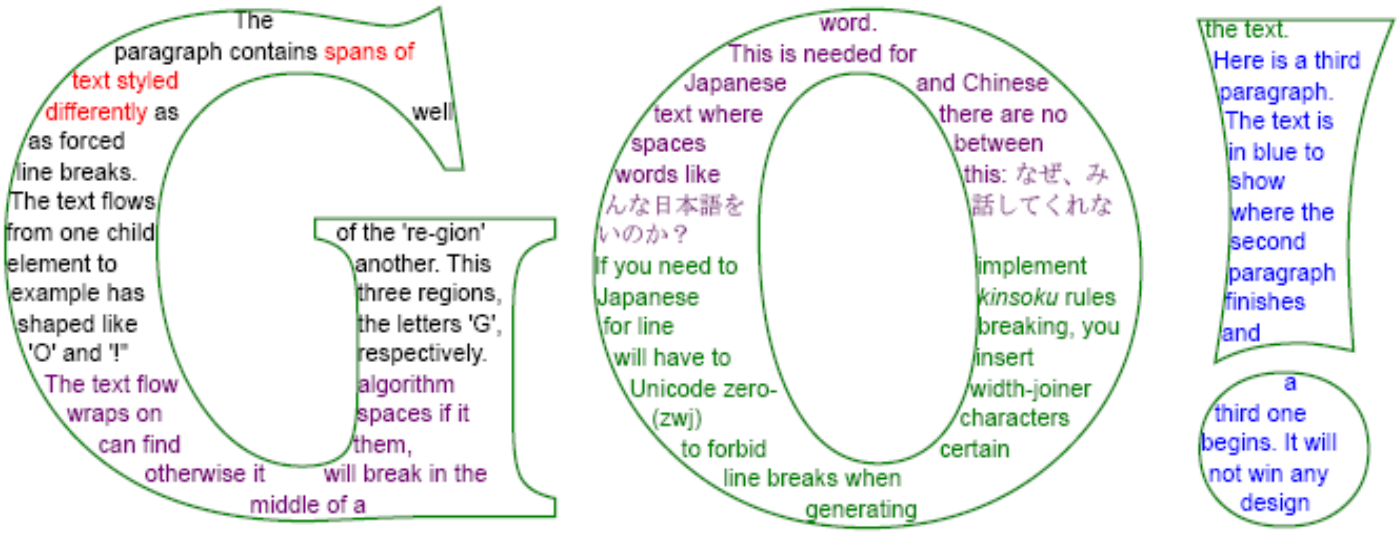
Below is an example of the flowing text capabilities:

```
<svg xmlns:svg="http://www.w3.org/2000/svg" version="1.2"
     xmlns:xlink="http://www.w3.org/1999/xlink"
  width="100%" height="100%" viewBox="0 0 300 310">
  <title>Basic textflow</title>
  <rect x="0" y="0" width="100%" height="100%" fill="yellow"/>
  <flowRoot font-size="16">
    <flowRegion>
      <path d="M100,50L50,300L250,300L300,50z"/>
    </flowRegion>
      <flowPara>Tomorrow, and tomorrow, and tomorrow; creeps in this
       petty pace from day to day, until the last syllable of recorded time.
       And all our yesterdays have lighted fools the way to dusty death.
     </flowPara>
  </flow>
  <path d="M90,40L40,270L260,270L210,40z" fill="none" stroke="black" stroke-
width="5"/>
</svg>
```



View this image as SVG (SVG 1.2 enabled browsers only)

A more complicated example is shown below. It is not included inline. Please see the SVG file for the source.



View this image as SVG (SVG 1.2 enabled browsers only)

4.16 DOM Interfaces

The DOM Interfaces to the flow elements are straightforward. There are no extra attributes or methods, beyond those inherited from other interfaces.

All elements that have textual content derive from the SVGTextContentElement interface. This is the same interface used by the regular, non-flowing, text elements.

```
interface SVGFlowRootElement : SVGElement, SVGTests, SVGLangSpace,
    SVGExternalResourcesRequired, SVGStylable, events:EventTarget {};

interface SVGFlowRegionElement : SVGElement {};

interface SVGFlowRegionsExcludeElement : SVGElement {};

interface SVGFlowDivElement : SVGTextContentElement {};

interface SVGFlowParaElement : SVGTextContentElement {};

interface SVGFlowSpanElement : SVGTextContentElement {};

interface SVGFlowRegionBreakElement : SVGElement {};

interface SVGFlowLineElement : SVGTextContentElement {};

interface SVGFlowTRefElement : SVGTextContentElement,
    SVGURIReferenceElement {};

interface SVGFlowRefElement : SVGElement, SVGURIReferenceElement {};

interface SVGFlowImageElement : SVGElement, SVGExternalResourcesRequired,
             SVGStylable, events:EventTarget {};
```

Previous | Top | Next

5 Multiple pages

An SVG 1.2 document or document fragment can have multiple pages. While the concept of a "page" usually applies to printed media, a page in SVG is defining an ordered list of groups containing graphics, with only one group being displayed at any time. This translates to a printed page for hardcopy output.

The following is an example using multiple pages:

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

  <pageSet>

     <!-- here is the first page -->
     <page>
       <circle cx="100" cy="100" r="20" fill="blue"/
>
     </page>

     <!-- here is the second page -->
     <page>
       <rect x="100" y="100" width="20" height="20"
             fill="red"/>
     </page>

  </pageSet>

</svg>
```

There are two pages in the example above. The first page has a blue circle; The second page has a red rectangle.

All pages are contained within a single **pageSet** element. A **pageSet** may contain any number of **page** elements. Each **page** element defines a single container of

graphical objects.

The **pageSet** and **page** elements introduce different reference scoping rules from the rest of the SVG language. Each **page** element has a local scope. References in a **page** can only be to elements within the same page or in the root SVG document (i.e. before the **pageSet** element). It is an error to refer to content in a different **page** element, regardless of its position in the document. Any resources defined on a **page** are local to that page and go out of scope once the page is no longer being rendered. External references are permitted, but authors are strongly encouraged to use the **externalResourcesRequired** feature in this case.

The scoping mechanism allows an implementation to free any resources required to render a single page if the **streamedContents** attribute is equal to "discard". The behavior is described in the chapter on Streamed Contents. The scoping also provides a simple mechanism to render page ranges by skipping content between page elements until the desired page range is reached.

Below is an example of an SVG document with multiple pages:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.2"
     streamedContents="true">

   <defs>
     <!-- definitions here are always available -->
   </defs>

   <g>
     <!-- graphics here are always visible -->
   </g>

   <pageSet>

      <page>
        <defs>
          <!-- These definitions are local to this page. -->
        </defs>
        <!-- graphics for page 1 go here -->
      </page>

      <page orientation="90">
        <!-- graphics for page 2 go here -->
      </page>

      <page>
        <!-- graphics for page 3 go here -->
```

```
        </page>

    </pageSet>

    <!-- No SVG content is allowed here -->
</svg>
```

No element from the SVG namespace is allowed between the closing **pageSet** tag and the closing **svg** tag. For example:

```
<svg>
    <pageSet>
    </pageSet>
    <!-- No SVG elements are allowed here -->
</svg>
```

This also precludes any content in this location from another namespace that through a binding mechanism such as sXBL would create shadow content using elements from the SVG language. Any document that violates this rule is in error.

The **pageSet** and **page** elements faciliate streamed animation. Many animations involve a set of scenes shown in sequence, such as cartoons. In SVG, each scene in the animation is mapped to a **page** element. Once a scene has ended (the animations running on the **page** complete), then the next scene is shown, or the user agent pauses until the entire next scene/page is downloaded.

5.1 Displaying Pages

In a screen-based user agent, only one page element is displayed at any time on the main canvas. The page displayed is referred to as the "current page". The first **page** in document order is the default "current page". User agents are expected to provide some mechanism for modifying the current page, and therefore allowing the user to navigate between pages. Page navigation may also occur automatically in a streamed document.

User agents are allowed to provide an alternate view, in a separate canvas if possible, to facilitate operations such as print preview, where multiple pages may be displayed at once. If the alternate view of a multiple page document is not animated, then the view should use the begin time of the individual pages.

If printing is invoked on an SVG document, i.e. the root element of the document is an SVG element, all pages should be printed or the user agent will provide a

means for selecting the pages to print. In general if printing is invoked on a document in another namespace, such as XHTML, that contains an SVG document fragment with pageset and pages, the "current page" of the SVG fragment at the begin time for that page should be printed. If an SVG file or document fragment is encapsulated inside a namespace capable of specifying control over the job information (e.g. JDF) then it is expected that this information will be followed.

Content after the opening **svg** tag and before the opening **pageSet** tag is always displayed, regardless of what page is current. Graphical elements here can be considered part of a "master page" that is rendered before the individual **page** contents are rendered.

If an SVG document fragment with multiple pages is embedded in, or referenced by, a document from another namespace, such as XHTML, then typically it is up to that namespace to define the processing of pages. The recommended behavior is that the container document display the "current page" of the SVG fragment, unless it has some mechanism to allow the paged media.

5.2 Timing and animation

The **pageSet** and **page** elements are SMIL time containers. The **pageSet** element is the equivalent of a **seq** element in SMIL. The **page** element is the equivalent of the **par** element in SMIL. For details on their behaviors refer to the SMIL specification references given in section 8.3 and 8.4 respectively. The combination of the **pageSet** and **page** elements and the SMIL timing attributes provide a method to stream long-running declarative animations.

A **page** can have transition effects applied as the user agent moves from one page to the next. The **transin** and **transout** attributes are allowed on the page element.

5.3 The pageSet element

The **pageSet** element is the container for a set of **page** elements.

pageSet Schema

```
<define name="pageSet">
  <element name="pageSet">
    <ref name="attlist.pageSet"/>
```

```
      <ref name="SVG.pageSet.content"/>
    </element>
  </define>

  <define name="attlist.pageSet"
combine="interleave">
    <ref name="SVG.Core.attrib"/>
    <ref name="SVG.Style.attrib"/>
    <ref name="SVG.Presentation.attrib"/>
    <ref name="SVG.GraphicalEvents.attrib"/>
  </define>

  <define name="SVG.pageSet.content">
    <zeroOrMore>
      <ref name="SVG.Description.class"/>
    </zeroOrMore>
    <zeroOrMore>
      <ref name="defs"/>
    </zeroOrMore>
    <zeroOrMore>
      <ref name="page"/>
    </zeroOrMore>
  </define>
```

A pageset element has the time container behavior of the SMIL2 "seq" element. Like **seq**, **pageSet** supports all timing attributes except **endsync**.

5.4 The page element

The **page** element contains graphics that are rendered when the page is the "current page".

page Schema

```
  <define name="page">
    <element name="page">
      <ref name="attlist.page"/>
      <ref name="SVG.page.content"/>
    </element>
  </define>

  <define name="attlist.page"
combine="interleave">
    <ref name="SVG.Core.attrib"/>
    <ref name="SVG.Style.attrib"/>
```

```
    <ref name="SVG.Presentation.attrib"/>
    <ref name="SVG.Focusable.attrib"/>
    <ref name="SVG.Transition.attrib"/>
    <ref name='SVG.AnimationTiming.attrib'/>
    <optional>
      <attribute name="page-orientation" a:
defaultValue="0" svg:animatable="true" svg:
inheritable="false">
        <choice>
          <value>-270</value>
          <value>-180</value>
          <value>-90</value>
          <value>0</value>
          <value>90</value>
          <value>180</value>
          <value>270</value>
        </choice>
      </attribute>
    </optional>
  </define>

  <define name="SVG.page.content">
    <zeroOrMore>
      <ref name="SVG.Description.class"/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name="SVG.Structure.class"/>
        <ref name="SVG.Description.class"/>
        <ref name="SVG.Animation.class"/>
        <ref name="SVG.Handler.class"/>
        <ref name="SVG.Structure.class"/>
        <ref name="SVG.Conditional.class"/>
        <ref name="SVG.Image.class"/>
        <ref name="SVG.MultiImage.class"/>
        <ref name="SVG.Audio.class"/>
        <ref name="SVG.Video.class"/>
        <ref name="SVG.Shape.class"/>
        <ref name="SVG.Text.class"/>
        <ref name="SVG.FlowText.class"/>
        <ref name="SVG.Hyperlink.class"/>
        <ref name="SVG.View.class"/>
        <ref name="SVG.Script.class"/>
        <ref name="SVG.XMLEvents.class"/>
      </choice>
    </zeroOrMore>
  </define>
```

The **page-orientation** attribute is described below.

Conceptually, the **page** element is similar to an **svg** element but without transformation and positioning. All pages are in the coordinate system of their **pageSet**, which is in the coordinate system of the root **svg** element.

The **page** element has the time container behavior of the SMIL "par" element. The **page** element, like **par**, supports all SMIL timing attibutes. Since the **pageSet** element is equivalent to a SMIL **seq**, a **page** element cannot have negative offset values for begin. The dissallowed begin values are listed in the specification of the "seq" element.

5.5 The page-orientation property

When printing an SVG file, the definition of most printing properties, such as paper size, printer tray and double-sided, is controlled by the system (commonly called the job ticket) and is not part of the SVG content. There are two printing properties that can be defined by the SVG content itself, in both cases to assist in on-screen display.

The SVG content typically defines a page dimension or aspect ratio in the topmost **svg** element and its **viewBox**. The **viewBox** transformation applies to printed SVG in the same way as screen display.

The orientation of each **page** element can be controlled by the **page-orientation** property. This enables the content to define whether a portrait or landscape mode is used for display and printing.

page-orientation

Value:	-270 | -180 | -90 | 0 | 90 | 180 | 270
Initial:	0
Applies to:	page elements
Inherited:	no
Percentages:	N/A
Media:	visual print
Animatable:	yes

When displaying a **page** on screen the value of the **page-orientation** property introduces the following transform before the top-level viewport transformation:

0
: No additional transform

-90 or 270
: translate(0 height) rotate(-90)

180 or -180
: translate(width height) rotate(180)

90 or -270
: translate(width 0) rotate(90)

The values for width and height are the dimensions of the top-level **viewBox** in the user coordinate system or the dimensions of the top-level viewport if there is no **viewBox** specified.

This transformation is only applied when displaying on the screen in a manner that expects the orientation of the page to be convenient to the user. For example, a print preview dialog would not apply this transformation.

Below is an example of the **page-orientation** property:

```
<svg width="8.5in" height="11in" viewBox="0 0 612 792">
  <rect x="36" y="36" width="540" height="720"
        fill="yellow" stroke-width="15" stroke="black"/>
  <pageSet>
     <page>
        <text font-size="30" x="72" y="108">This is portrait</
text>
     </page>
     <page page-orientation="90">
        <text font-size="30" x="72" y="108">This is
landscape</text>
     </page>
   </pageSet>
</svg>
```

The above example is equivalent to the following two pages when printed:

```
<svg width="8.5in" height="11in" viewBox="0 0 612 792">
  <rect x="36" y="36" width="540" height="720"
        fill="yellow" stroke-width="15" stroke="black"/>
  <text font-size="30" x="72" y="108">This is portrait</text>
</svg>

<svg width="8.5in" height="11in" viewBox="0 0 612 792">
  <rect x="36" y="36" width="540" height="720"
        fill="yellow" stroke-width="15" stroke="black"/>
```

```
  <text transform="translate(612 0) rotate(90)"
        font-size="30" x="72" y="108">This is portrait</text>
</svg>
```

And equivalent to the following two pages when viewed:

```
<svg width="8.5in" height="11in" viewBox="0 0 612 792">
  <rect x="36" y="36" width="540" height="720"
        fill="yellow" stroke-width="15" stroke="black"/>
  <text font-size="30" x="72" y="108">This is portrait</text>
</svg>

<svg width="11in" height="8.5in" viewBox="0 0 792 612">
    <!-- fake top-level transform -->
  <g transform="translate(0 612) rotate(-90)">
    <rect x="36" y="36" width="540" height="720"
          fill="yellow" stroke-width="15" stroke="black"/>
    <text transform="translate(612 0) rotate(90)"
          font-size="30" x="72" y="108">This is portrait</
text>
  </g>
</svg>
```

In the printing environment, the printing system can override the value of this property. For example, the user may wish to print all pages in portrait mode.

5.6 Navigation between pages

Since **pageSet** and **page** are equivalent to the SMIL **seq** and **par** elements, the "current page" is determined by the current time of the document. The **syncbase** of the first page is that of the **pageSet** element. For subsequent pages the implicit syncbase of a **page** is its previous page sibling. Therefore each **page** will play after the previous page has finished. Hyperlinking to a page will seek the document to the begin time of that page.

It is also suggested that user agents bind input methods, such as the PageUp and PageDown keys if available, to page navigation. These should act the same way as hyperlinking, i.e. the document will seek to the begin time of that page. In a view without animation, such as print preview, the view will use the begin times of each page.

The begin and end of **pageSet** / **page** will raise a "begin" event and "end" event respectively. Repeating of a **page** / **pagSet** using the repeat attribute will raise a "repeat" event.

The SMIL timing attributes **begin**, **end** and **dur** are one simple way to control the duration of each page. Please see the SMIL specification for an explanation of all valid attributes and their use. In this first example each page has a simple duration. Thus the navigation between pages is fixed and will advance to the next page at the completion of the previous page's duration. The first page will have a duration from 0 to 30 seconds, the next page from 30 seconds to 90 seconds, the third page from 90 seconds to 120 seconds, etc.

```
<svg>
   <pageSeq begin="0s">
      <page id="page1" dur="30s">
         ...
      </page>

      <page id="page2" dur="60s">
         ...
      </page>

      <page id="page3" dur="30s">
         ...
      </page>

      <page id="page4" dur="90s">
         ...
      </page>

   </pageSeq>
</svg>
```

The **begin** and **end** attributes, and hence the active duration of pages, can be defined by events, see the SMIL timing attributes and the event value syntax.

All events supported in SVG can be used on SMIL timing attributes that support event values

In this next example, the pages are flipped by clicking on the presentation. The end of the active interval is resolved when the event happens. The begin time is resolved using the implicit syncbase which is the previous page. Each page will begin 0 seconds after the end of the previous page elements duration.

```
<svg>
   <pageSeq begin="0s">
      <page id="page1" begin="0s" end="page1.click">
```

```
            ...
         </page>

         <page id="page2" begin="0s" end="page2.click">
            ...
         </page>

         <page id="page3" begin="0s" end="page3.click">
            ...
         </page>

         <page id="page4" begin="0s" end="page4.click">
            ...
         </page>

      </pageSeq>
   </svg>
```

Once this example has been played and the begin and end times resolved, seeking to a page will not reset these values. The element state will reset when the parent time container repeats or restarts.

6 Text enhancements

6.1 Editable Text Fields

SVG 1.2 introduces editable text fields, moving the burden of text input and editing to the user agent, which has access to system text libraries.

6.1.1 The editable attribute

The **text** and **flowDiv** elements have an **editable** attribute which specifies whether the contents of the elements can be edited in place.

editable = "true" | "false"

> If set to "false" the contents of the text element 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 element 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 provide a way to cut the selected text from the element 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.

In general, user agents should allow for the inline WYSIWYG editing of text with a caret that identifies current position. They should also support system functions such as copy/paste and drag/drop if available.

For WYSIWYG editing the following functionality must be made available:

- movement to the next/previous character (in logical order) with Left/Right arrows for horizontal text and Down/Up keys for vertical text

- movement to the next/previous line with the Down/Up keys for horizontal text and Right/Left for vertical text
- movement to the beginning of the line with the Home key
- movement to the end of the line with the End key
- the system-dependent keyboard binding for copy/cut/paste

For devices without keyboard access, the equivalent system input methods should be used wherever possible to provide the functionality described above.

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 (except for the state associated with an Input Method Editor window).

6.1.2 CSS pseudo class for editable text

To match the two states of editable **text** and **flowText** elements, SVG user agents that suppport CSS style sheets must support the ':edited' pseudo-class. It is used to style the currently editable text field. As such, it is equivalent to the following CSS selectors:

```
svg|text[editable='true']:focus
svg|flowText[editable='true']:focus
```

6.2 Text Selection

The SVGSelection interface and the {"http://www.w3.org/2001/xml-events", "selection"} event allows change notification and access to details on the current text selection. When text selection changes (e.g., a new selection is made, the selection is modified, or an existing selection goes away), whether due to user action or because of DOM calls, the {"http://www.w3.org/2001/xml-events", "selection"} event is dispatched to the top-level svg element.

```
interface SVGDOMRange : range::Range
{
   // ***ElementInstance fields are non-null only when the
corresponding
boundary of the
   // range is on an instance of a <use> element.
   readonly attribute SVGElementInstance
startContainerElementInstance;
   readonly attribute SVGElementInstance
endContainerElementInstance;
};
```

```
interface SVGSelection : SVGDOMRange
{
   readonly attribute boolean       active; // true if
something is selected
};
```

The interface corresponding to the {"http://www.w3.org/2001/xml-events", "selection"} event is as follows:

```
// sent to the top-level svg element when selection changes
in any way
interface SVGSelectionEvent : events::Event
{
};
```

The following attribute is added to the SVGSVGElement interface:

```
interface SVGSVGElement
{
         ....
         readonly attribute SVGSelection selection;
};
```

Previous | Top | Next

7 Streaming

The SVG working group is considering streaming enhancements to the SVG language. Here are two identified uses for streaming:

1. The SVG Print specification has relatively low-end printers as a potential target. These devices may have various limitations such as memory. It might be a requirement of SVGP that elements can be rendered and discarded immediately, thereby removing the need for maintaining an in-memory DOM for the document.
2. For time-based SVG applications viewed on a display device, we would like SVG to allow time-based elements such as animations to start while the rest of the document is still downloading.

To meet the requirements of these use cases, SVG 1.2 adds a method to mark streamed content as available for discard and a method to control the start of an element's local time.

7.1 The timelineBegin attribute

The **timelineBegin** attribute controls the initialization of the timeline on a time container.

timelineBegin = "onLoad" | "onStart"

onLoad:
: This is the default value. The element's timeline starts the moment the SVGLoad event for the element is triggered.

onStart:
: The element's timeline starts at the moment the element's open tag is fully parsed and processed.

For streaming animations, the author would typically set the root **svg** element,

which controls the global timeline, to "onStart", thus allowing the nested timelines to begin as the document arrives. The **page** elements, which normally represent scenes in the animation, typically use the default value of "onLoad", ensuring that the entire scene is loaded before the nested timeline begins.

The **timelineBegin** attribute only affects time containers.

7.2 The streamedContents attribute

The **streamedContents** attribute allows an author to mark the contents of a document as unnecessary. It is allowed on the root **svg** element.

streamedContents = "keep" | "discard"

keep:
: This is the default value. The entire contents of document must be available while the document is being used.

discard:
: The contents of the document may be discarded as they are used. It is not a requirement that the contents are discarded.

In effect, this attribute controls whether or not a user agent is able to discard the contents of the **page** elements that have already been displayed.

Previous | Top | Next

8 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. After the startElement SAX event on a node, that node becomes part of the document tree in the unresolved state. When the node's dependencies are successfully resolved, then the node enters the resolved state. When 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 (like stylesheets or 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**, **vkern**, **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

```
<svg>
  <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

```
<svg>
  <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>
```

According to the proposal above, 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 eRR="true"

```
<svg>
  <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 & eRR=true, rendering is stopped at the **use**)
3. **g**.startElement: no update
4. **g.myGroup**.startElement: no update (use is resolved but eRR=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 eRR="true"

```
    <svg>
      <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 eRR 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

```
<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">
 <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

Previous | Top | Next

9 Vector effects

The default rendering for a shape is to fill the geometry with a specified paint, then stroke the outline with a specified paint and finally the option to draw markers at the vertices.

Vector effects offer the following options:

- Any combination of marking operations (fill, stroke or vertex markers) can be applied in any order to a given geometry.
- Coordinate systems and/or geometry vertex values can me transformed arbitrarily before any marking operation.
- The geometric outline of the stroke can be converted into geometry (via **veStrokePath**) that can then be used for any type of marking operation.
- Logical path geometry operations (join, reverse, union, intersect and exclude) can be applied to a set of shapes.
- Partial definitions of paths can be referenced and then merged together into a shape which subsequently is used for marking operations (e.g., the outline of a country might be assembled by referencing a series of subpaths holding the various pieces of the country's border geometry).

9.1 The vectorEffect element

The **vectorEffect** element defines a transformation of a primitive shape's outline that happens before it is drawn. The transformation is described as a series of vector effect primitive processing nodes, where input(s) and the output of every node can be considered to be SVG path data. Any primitive shape or path element can reference a vector effect through the **vector-effect** property. Alternatively, vector effects can be used as a drawing element; in such case its input is considered to be empty (and the **vePath** element must be used to get input data).

vectorEffect Schema

```
  <define name='vectorEffect'>
    <element name='vectorEffect'>
      <ref name='attlist.vectorEffect'/>
      <ref name='SVG.vectorEffect.content'/>
    </element>
  </define>

  <define name='attlist.vectorEffect'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <optional>
      <attribute name='vectorEffectUnits' a:
defaultValue='userSpaceOnUse'>
        <choice>
          <value>userSpaceOnUse</value>
          <value>objectBoundingBox</value>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='clipout' a:
defaultValue='normal'>
        <choice>
          <value>normal</value>
          <value>clip</value>
        </choice>
      </attribute>
    </optional>
  </define>

  <define name='SVG.vectorEffect.content'>
    <zeroOrMore>
      <ref name='SVG.Description.class'/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name='animate'/>
        <ref name='set'/>
        <ref name='vePath'/>
        <ref name='veSetback'/>
        <ref name='veStrokePath'/>
        <ref name='veAffine'/>
        <ref name='veJoin'/>
        <ref name='veReverse'/>
        <ref name='veUnion'/>
        <ref name='veIntersect'/>
        <ref name='veExclude'/>
        <ref name='veFill'/>
```

```
        <ref name='veStroke'/>
        <ref name='veMarker'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='SVG.VECommon.attrib'
combine='interleave'>
    <optional>
      <attribute name='in'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='result'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
  </define>

  <define name='SVG.VECommon-2.attrib'
combine='interleave'>
    <ref name='SVG.VECommon.attrib'/>
    <optional>
      <attribute name='in2'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
  </define>

  <define name='VEtransform.attrib'
combine='interleave'>
    <optional>
      <attribute name='transform'/>
    </optional>
  </define>

  <define name='VEtransformPath.attrib'
combine='interleave'>
    <optional>
      <attribute name='transformPath'/>
    </optional>
  </define>

  <define name='VEPrimitive.content'>
    <empty/>
  </define>
```

The **vectorEffectUnits** attribute allows "userSpaceOnUse" or "objectBoundingBox" as values. The default is "userSpaceOnUse". The output of the last vector effect node is considered to be the result of the complete vector effect. That path is used when shape's outline is normally used (inside **clipPath**, for text flows and text on a path, and when referenced by a **vePath** element).

The clipout attribute allows "normal" or "clip" as values. The value "normal" means that results of the individual fill operations are composited with each other (using srcOver). The value "clip" means that painting operations should be processed last to first and after every painting operation a clipping should be applied that excludes (clips out) the path just painted from the paintable region. The effect is that areas that were painted already won't be painted over.

This is the default **vectorEffect** that corresponds to the default SVG painting behavior:

```
<vectorEffect>
  <veFill/>
  <veStroke/>
  <veMarker/>
</vectorEffect>
```

The following vector effect results in no rendering (i.e., neither the fill, the stroke, nor the markers are rendered).

```
<vectorEffect/>
```

9.2 Common vector effect primitive attributes

The following attributes are available on all vector effect primitives.

result = name of output

The **result** attribute specifies the location, if necessary, of the output from the current vector effect primitive. It is analogous to the **result** attribute on filter effect primitives. If not specified, the value is empty, and the result is only passed to the following primitive. Animatable: no.

in = "SourcePath" | name of input
in2 = "SourcePath" | name of input

The **in** and **in2** attributes define the input locations for the current vector

effect primitive. The value of "SourcePath" means that the outline of the shape that references this effect is used. Any other values refer to the output of another node within this **vectorEffect** identified by the value of its **result** attribute. If this attribute is omitted, for the first primitive the value "SourcePath" is used, and the output of the previous primitive is used for all other nodes. The path used for "SourcePath"is always pre-transformed to be in the correct coordinate space depending on the value of the **vectorEffectUnits** attribute value. Animatable: no

transform = <transform>
trandformPath = <transform>

The **transform** and **transformPath** attributes supply coordinate space transformations for some vector effect elements. The difference between them is that the **transform** attribute defines the coordinate space where a particular operation occurs so that the input is transformed into that coordinate space, the vector effect operation applied and then the inverse transform applied to the result. In contrast, **transformPath** is simply applied to the input of the particular operation. The default value is empty. Animatable: yes.

9.3 The veStrokePath element

veStrokePath Schema

```
  <define name='veStrokePath'>
    <element name='veStrokePath'>
      <ref name='attlist.veStrokePath'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veStrokePath'
combine='interleave'>
    <ref name='SVG.VECommon.attrib'/>
    <ref name='VEtransform.attrib'/>
    <ref name='SVG.stroke-dasharray.attrib'/>
    <ref name='SVG.stroke-dashoffset.attrib'/>
    <ref name='SVG.stroke-linecap.attrib'/>
    <ref name='SVG.stroke-linejoin.attrib'/>
    <ref name='SVG.stroke-miterlimit.attrib'/>
    <ref name='SVG.stroke-width.attrib'/>
  </define>
```

The **veStrokePath** element produces an outline of the input path's stroke. It is calculated using the given stroke parameters. The default parameter values come from the corresponding computed property values of the source element. Width attribute percentages are interpreted relative to the source element **stroke-width** property.

9.4 The veSetback element

veSetback Schema

```
  <define name='veSetback'>
    <element name='veSetback'>
      <ref name='attlist.veSetback'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veSetback'
combine='interleave'>
    <ref name='SVG.VECommon.attrib'/>
    <ref name='VEtransform.attrib'/>
    <optional>
      <attribute name='setback-offset'/>
    </optional>
  </define>
```

The **veSetback** element performs a path "setback" operation: it breaks the path into individual segments and shortens both ends of every segment by the distance specified by **setback-offset**. If offset is a list of four lengths it is interpreted as follows:

- offset after the path beginnings (produced by M/m commands),
- offset before the "inner" path nodes (when non-M/m command is followed by another non-M/m command)
- offset after the "inner" path nodes
- offset before the path ends (when non-M/m command is followed by M/m command)

9.5 The veAffine element

veAffine Schema

```
  <define name='veAffine'>
    <element name='veAffine'>
      <ref name='attlist.veAffine'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veAffine'
combine='interleave'>
    <ref name='SVG.VECommon.attrib'/>
    <ref name='VEtransformPath.attrib'/>
  </define>
```

The **veAffine** element transforms a path using the specified transformation matrix.

9.6 The veReverse element

veReverse Schema

```
  <define name='veReverse'>
    <element name='veReverse'>
      <ref name='attlist.veReverse'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veReverse'
combine='interleave'>
    <ref name='SVG.VECommon.attrib'/>
  </define>
```

The **veReverse** element reverses the direction of the path. The path is adjusted such that all of the vertices on the path are put in reverse order (i.e., if there are N vertices numbered 0 to N-1, then vertex K is moved to position N-K-1) with adjustments to the path commands and control points to produce the same shape as the original path. In effect, it is as if the path segments have been played backward.

There are a variety of effects that rendering results from a veReverse operation. They include swapping the begin and end markers and a 180 degree orientation change for automatically oriented markers. There are potentially different results when rendering compound paths.

9.7 The veJoin element

veJoin Schema

```
  <define name='veJoin'>
    <element name='veJoin'>
      <ref name='attlist.veJoin'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veJoin'
combine='interleave'>
    <ref name='SVG.VECommon-2.attrib'/>
    <optional>
      <attribute name='connect'>
        <choice>
          <value>none</value>
          <value>line</value>
        </choice>
      </attribute>
    </optional>
  </define>
```

The **veJoin** element joins two paths together by a concatenation operation on the path data commands for the two paths (the path referenced by in2 is concatenated to the end of the path referenced by in). The **connect** attribute can be either "line" or "none". In case of "none", the path segments are merged together. In case of "line" an "M" command that starts the second path is replaced with "L".

9.8 The veUnion element

veUnion Schema

```
  <define name='veUnion'>
    <element name='veUnion'>
      <ref name='attlist.veUnion'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veUnion'
```

```
combine='interleave'>
    <ref name='SVG.VECommon-2.attrib'/>
  </define>
```

The **veUnion** element produces an outline of the union of two shapes. The geometric effect is equivalent to the result of a geometric computation of additive clipping paths (i.e., logical OR'ing of the silhouettes produced by each of the shapes).

9.9 The veIntersect element

veIntersect Schema

```
  <define name='veIntersect'>
    <element name='veIntersect'>
      <ref name='attlist.veIntersect'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veIntersect'
combine='interleave'>
    <ref name='SVG.VECommon-2.attrib'/>
  </define>
```

The **veIntersect** element produces an outline of the intersection of two shapes. The geometric effect is equivalent to the result of a geometric computation of nested clipping paths.

9.10 The veExclude element

veExclude Schema

```
  <define name='veExclude'>
    <element name='veExclude'>
      <ref name='attlist.veExclude'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veExclude'
combine='interleave'>
    <ref name='SVG.VECommon-2.attrib'/>
```

```
  </define>
```

The **veExclude** element produces an outline of the exclusion of the second shape, provided by **in2**, from the first shape, provided by **in**. The geometric effect of veExclude is equivalent to a geometric "clip out" computation where the resulting path geometry is the geometric clip of the first shape against a composite shape consisting of a **veJoin** of: (a) a very large rectangle (i.e., a rectangle whose bounding box is outside the bounds of the first shape) and (b) the second shape after a **veReverse** operation.

9.11 The veFill element

veFill Schema

```
  <define name='veFill'>
    <element name='veFill'>
      <ref name='attlist.veFill'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veFill'
combine='interleave'>
    <optional>
      <attribute name='in'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
    <ref name='SVG.fill.attrib'/>
    <ref name='VEtransform.attrib'/>
  </define>
```

The **veFill** element fills a shape using the paint given by the **fill** property and the opacity value given by the **fill-opacity** property. For the **fill** property, values of "currentStroke" and "currentFill" correspond to the computed value of **stroke** and **fill** on the source element. For the **fill-opacity** property, values of "currentStrokeOpacity" and "currentFillOpacity" correspond to the computed value of **stroke-opacity** and **fill-opacity** on the source element. The output of the **veFill** element is the same as its input.

9.12 The veStroke element

veStroke Schema

```
  <define name='veStroke'>
    <element name='veStroke'>
      <ref name='attlist.veStroke'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veStroke'
combine='interleave'>
    <optional>
      <attribute name='in'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
    <ref name='SVG.fill.attrib'/>
    <ref name='VEtransform.attrib'/>
  </define>
```

The **veStroke** element creates a shape that represents the path of the shape's stroke (taking into account the **stroke** properties on the source element) and then fills the stroke using the paint server given by the **fill** property and the opacity value given by the **fill-opacity** property. For the **fill** property, values of "currentStroke" and "currentFill" mean the computed value of **stroke** and **fill** on the source element. For the **fill-opacity** property, values of "currentStrokeOpacity" and "currentFillOpacity" mean the computed value of **stroke-opacity** and **fill-opacity** on the source element. The output of the **veStroke** element is the same as its input.

Note that the **veStroke** element has **fill** and **fill-opacity** attributes and not **stroke** and **stroke-opacity** attributes. This is because the conceptual model for stroking within vector effects is that you first perform an implicit **veStrokePath** operation to create a new shape, and then you paint the region of (i.e., you "fill") that new shape.

9.13 The veMarker element

veMarker Schema

```
  <define name='veMarker'>
    <element name='veMarker'>
      <ref name='attlist.veMarker'/>
      <ref name='VEPrimitive.content'/>
```

```
    </element>
  </define>

  <define name='attlist.veMarker'
combine='interleave'>
    <optional>
      <attribute name='in'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
    <ref name='SVG.Marker.attrib'/>
  </define>
```

The **veMarker** element draws markers along the input path. Its output is the same as its input.

9.14 The veMarkerPath element

veMarkerPath Schema

```
  <define name='veMarkerPath'>
    <element name='veMarkerPath'>
      <ref name='attlist.veMarkerPath'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.veMarkerPath'
combine='interleave'>
    <optional>
      <attribute name='in'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
    <ref name='SVG.Marker.attrib'/>
  </define>
```

The **veMarkerPath** element behaves like **veMarker** but instead of drawing it produces its output by taking the union of the paths of all the markers.

9.15 The vePath element

vePath Schema

```
  <define name='vePath'>
    <element name='vePath'>
      <ref name='attlist.vePath'/>
      <ref name='SVG.vePath.content'/>
    </element>
  </define>

  <define name='attlist.vePath'
combine='interleave'>
    <optional>
      <attribute name='result'>
        <data type='NMTOKEN'/>
      </attribute>
    </optional>
  </define>

  <define name='SVG.vePath.content'>
    <oneOrMore>
      <ref name='vePathRef'/>
    </oneOrMore>
  </define>
```

The **vePath** element does not take input and its result is the path taken from the shape(s) which it references. The path references come from child **vePathRef** elements. Individual paths are transformed according to the **transformPath** attribtue on **vePathRef** and then joined together in the same manner as **veJoin**.

9.16 The vePathRef element

vePathRef Schema

```
  <define name='vePathRef'>
    <element name='vePathRef'>
      <ref name='attlist.vePathRef'/>
      <ref name='VEPrimitive.content'/>
    </element>
  </define>

  <define name='attlist.vePathRef'
combine='interleave'>
    <ref name='SVG.XLinkRequired.attrib'/>
    <ref name='VEtransform.attrib'/>
```

```
    <optional>
      <attribute name='connect'/>
    </optional>
    <optional>
      <attribute name='reverse'/>
    </optional>
  </define>
```

The **vePathRef** element is a child node for the **vePath** element that references an individual path. It is an error for this element to reference anything but a primitive shape, a path or a text element.

9.17 Examples

A non-scaling line width, with non-scaling markers:

```
<vectorEffect>
  <veFill/>
  <veStroke transform="ref(host)"/>
  <veMarker transform="ref(host)"/>
</vectorEffect>
```

Non-scaling fill/stroke patterns:

```
<vectorEffect>
  <veFill transform="ref(host)"/>
  <veStrokePath in="SourcePath"/>
  <veFill transform="ref(host)" fill="StrokePaint"/>
  <veMarker in="SourcePath" transform="ref(host)"/>
</vectorEffect>
```

An effect that strokes a path twice, once normally and then again with a thinner red stroke:

```
<vectorEffect>
  <veStroke/>
  <veStroke fill="red" stroke-width-adjust="50%"/>
</vectorEffect>
```

Using vector effects to produce shared borders on paths:

```
<defs>
  <path id="border1" d="..."/>
  <path id="border2" d="..."/>
```

```
    <path id="border3" d="..."/>
    <path id="border4" d="..."/>
    <path id="border5" d="..."/>
  </defs>
  <vectorEffect>
    <vePath>
      <vePathRef xlink:href="#border1"/>
      <vePathRef xlink:href="#border2"/>
      <vePathRef xlink:href="#border3"/>
    </vePath>
    <veFill color="red"/>
    <vePath>
      <vePathRef xlink:href="#border4"/>
      <vePathRef reverse="true" xlink:href="#border2"/>
      <vePathRef xlink:href="#border5"/>
    </vePath>
    <veFill color="blue"/>
  </vectorEffect>
```

9.18 The vector-effect property

The **vector-effect** property specifies the vector effect to use when drawing an object. Vector effects are applied before any of the other compositing operations: filters, masks and clips.

vector-effect

Value:	'default' | 'non-scaling-stroke' | 'inherit' | **<uri>**
Initial:	'default'
Applies to:	graphical elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The value 'default' corresponds to the following vector effect:

```
  <vectorEffect>
    <veFill/>
    <veStroke/>
    <veMarker/>
  </vectorEffect>
```

It is the default rendering behaviour from SVG 1.1.

The value 'non-scaling-stroke' is a keyword for a predefined vector effect that causes an object's stroke-width to be unaffected by transformations and zooming. That is:

```
<vectorEffect>
  <veFill/>
  <veStrokePath in="SourcePath"/>
  <veFill transform="ref(host)" fill="currentStroke"/>
  <veMarker in="SourcePath" transform="ref(host)"/>
</vectorEffect>
```

The URI value references a **vectorEffect** element that should be used as the vector effect.

The value "non-scaling-stroke" is designed so that it can be implemented without the entire vector effect engine. For example, profiles of SVG may restrict the values of **vector-effect** to be "default" or "non-scaling-stroke". In effect this requires no processing of vector effects, rather it is always the default rendering order with a different set of transformations.

Previous | Top | Next

10 Rendering model

SVG 1.2 updates the rendering model and rendering operations of SVG 1.1.

10.1 Enhanced Alpha Compositing

10.1.1 Introduction

SVG 1.2 supports the following clipping/masking features:

- alpha compositing, which may be used each time a new element is placed on the canvas. The operation specified determines the combination of the source color and alpha, and the destination color and alpha.
- clipping paths, which use any combination of **path**, **text** and basic shapes to serve as the outline of a (in the absence of anti-aliasing) 1-bit mask, where everything on the "inside" of the outline is allowed to show through but everything on the outside is masked out
- masks, which are container elements which can contain graphics elements or other container elements which define a set of graphics that is to be used as a semi-transparent mask for compositing foreground objects into the current background.

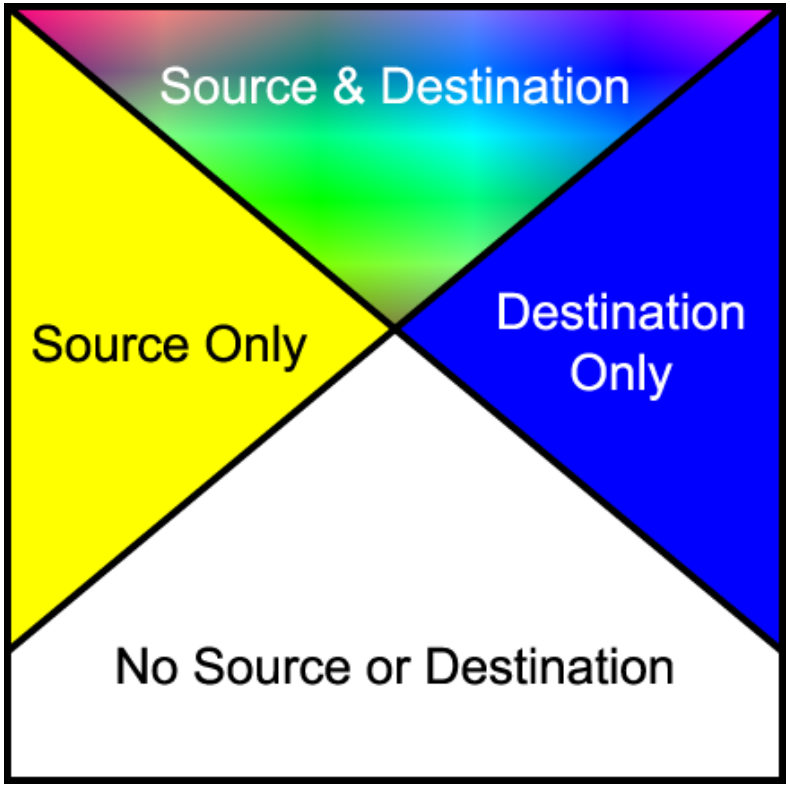
Note that masking with an element containing only color components with full luminance (e.g. r=g=b=1) will produce the equivalent result to compositing using the **src-in** or **dst-in** operators.

10.1.2 Alpha compositing

Graphics elements are composited onto the elements already rendered on the canvas based on an extended Porter-Duff compositing model, in which the resulting color and opacity at any given pixel on the canvas depend on the 'comp-op' specified. The base set of 12 Porter-Duff operations shown below always result in a value between zero and one, and as such, no clamping of output values is required.

In addition to the base set of 12 Porter-Duff operations, a number of blending operations are supported. These blending operations are extensions of the base Porter-Duff set and provide enhanced compositing behavior. The extended operations may result in color and opacity values outside the range zero to one. The opacity value should be clamped between zero and one inclusive, and the pre-multiplied color value should be clamped between zero and the opacity value inclusive.

The following diagram shows the four different regions of a single pixel that are considered when compositing.



Depending on the compositing operation the resultant pixel includes input from one or more of the regions in the above diagram. For the regions where only source or destination are present, a choice of including or not including the input is available. For the region where both are present, various options are available for the combination of input data.

For groups containing compositing operators, the operation used to composite the group onto the canvas is the comp-op property of the container element itself. Other properties on container elements, such as opacity, specify operations that are performed after the children have been combined and before the group is composited onto the background. The **enable-background** and **knock-out** properties specify the state the group buffer is initialized to prior to use, any modification to the compositing of the group's children, and in some cases a post rendering step to be performed after rendering the children and prior to any other post rendering steps.

Implementation note: Various container elements calculate their bounds prior to rendering. For example, rendering a group generally requires an off-screen buffer, and the size of the buffer is determined by calculating the bounds of the objects contained within the group. SVG 1.0 implementations generally calculated the bounds of the group by calculating the union of the bounds of each of the objects within the group. Depending on the compositing operations used to combine objects within a group, the bounds of the group may be reduced, and so, reduce the memory requirements. For example, if a group contains two objects - object A 'in' object B - then the bounds of the group would be the intersection of the bounds of objects A and B as opposed to the union of their bounds.

While container elements are defined as requiring a buffer to be generated, it is often the case that a user agent using various optimizations can choose not to generate this buffer. For example, a group containing a single object could be directly rendered onto the background rather than into a buffer first.

The following variables are used to describe the components of the background, group and extra opacity channel buffers.

```
Sc              Non-premultiplied source color component
Sca             Premultiplied source color component
```

```
Sra Sga Sba     Premultiplied source color component
Sa              Source opacity component
Dc              Non-premultiplied destination color component
Dca             Premultiplied destination color component
Dra Dga Dba     Non-premultiplied destination color component
Da              Destination opacity component
Da(d)           Extra opacity buffer containing the percentage
                of the background channel in the group buffer.
D<n>            Destination buffer <n> where the background is 0,
                groups in the top level svg element 1, nested groups
                2 and so forth
D'              The results of the destination post a compositing step
```

The operation used to place objects onto the background is as follows:

```
Dca' = f(Sc, Dc).Sa.Da  + Y.Sca.(1-Da)  + Z.Dca.(1-Sa)
Da'  =         X.Sa.Da  + Y.Sa.(1-Da)   + Z.Da.(1-Sa)
```

Depending on the compositing operation, the above equation is resolved into an equation in terms of pre-multiplied values prior to rendering. The following are specified for each compositing operation:

```
X, Y, Z, f(Sc, Dc)
```

defined as:

```
f(Sc,Dc)    The intersection of the opacity of the source and destination
            multiplied by some function of the color. (Used for color)

X           The intersection of the opacity of the source and destination.
            (Used for opacity)

Y           The intersection of the source and the inverse of the destination.

Z           The intersection of the inverse of the source and the destination.
```

Depending on the compositing operation, each of the above values may or may not be used in the generation of the destination pixel value.

10.1.3 The clip-to-self property

clip-to-self

Value:	true | false | inherit
Initial:	false
Applies to:	container elements and graphics elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

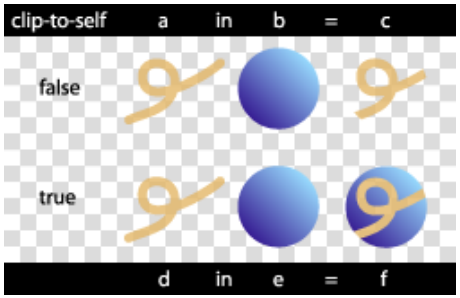
The **clip-to-self** property determines if the object effects pixels not covered by the object. Some compositing operations effect pixels where the source graphic is completely transparent.

For regions not covered by the source graphic, one of two operations can be performed.

1. Setting **clip-to-self** to true means that compositing an object only effects the pixels covered by the object.
2. Setting **clip-to-self** to false means that compositing an object effects all pixels on the canvas by compositing completely transparent source onto the destination for areas not covered by the object.

Note that most compositing operations do not remove the destination and as such for these operations, the **clip-to-self** property has no effect. The compositing operations that remove background are described in the **comp-op** property diagram. They are the operation which remove the right-hand blue region in each diagram. They are clear, src, src-in, dst-in, src-out and dst-atop. For all other operators the **clip-to-self** property has no effect.

The **clip-to-self** property provides compatibility with Java2D.

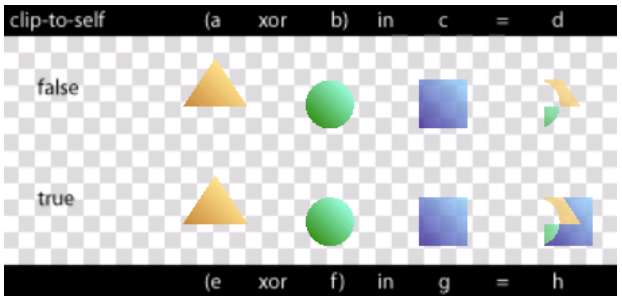


View this image as SVG (SVG 1.2 enabled browsers only)

Container elements where the **clip-to-self** property is set to true only effect the pixels within the extent of the container element. For example, if a container element contains two circles, and the container element has the **clip-to-self** property set to true, then nothing outside the circles is effected. To perform this operation, the renderer needs to keep track of the extent of each of the elements within the container element and ensure that nothing other than the elements is modified. This can be produced by converting each object to a clipping path and unioning the clipping paths together to produce a clipping path that represents the extent of all the elements within the container element. Where a container element contains nested container elements, the operation is performed within the sub-container elements to produce the final path. When the group is composited onto the page, it is composited through the clipping path generated and thus nothing outside the extent of all the elements within the container element is modified.

For filled and stroked shapes and text, the object is directly converted to a clipping path. For images and filters, the bounds of the object are converted to a clipping region.

For some container elements where the **clip-to-self** property is set to false, the container element might effect the background outside bounds of the container element.



View this image as SVG (SVG 1.2 enabled browsers only)

10.1.4 The enable-background property

enable-background

Value:	accumulate | new [**x y width height**] | inherit
Initial:	accumulate
Applies to:	container elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	no

For a container element with **enable-background** set to "new", the container element's buffer is initially cleared to transparent. This buffer is treated as the canvas for the complex group's children. When the complete contents of the container element are rendered onto the buffer, the buffer is composited onto the canvas using the container element's specified compositing operation.

For a container element with **enable-background** set to "accumulate", the corresponding area of the canvas is copied into the container element's buffer. A second buffer which has only an opacity channel is also created. This buffer Da(d) stores the percentage of the background in the group buffer and is initially opaque. The group buffer is treated as the canvas for the children of the group as usual. Additionally, as objects are placed into the group buffer, they are also placed into the Da (d) buffer using one of the operations listed below. Before the group buffer is composited onto the canvas, any remaining background color in the group buffer is removed using the values in the Da(d) buffer. Other post rendering steps such as the opacity are performed after this step, and before compositing the result onto the canvas.

For groups with an enable-background value set to accumulate, the compositing operation used to place the group onto the background is modified. The operation will apply any reduction to the background caused by the objects.

When drawing elements within a container element with **enable-background** set to "accumulate", the standard equations as listed below are used to draw the object into the group buffer. Depending on the compositing operation, one of two operations listed below are used to draw the object into the extra transparency buffer Da (d).

For the operations clear, src, src-in, dst-in, src-out and dst-atop:

```
Da(d)' = 0
```

For all other compositing operations:

```
Da(d)' = Da(d).(1 - Sa)
```

Once the contents of a container element are rendered into the container element's buffer and before operations such as opacity or filters are applied to the buffer, the remaining background is removed from the container element's buffer using the following operations:

```
Dca1'   = Dca1 - Dca0.Da1(d)
Da1'    = Da1 - Da0.Da1(d)
```

At this point Da1(d) should be inverted. The inverted Da1(d) represents the amount of data to be removed from the background when placing the container element onto the background.

```
Da1(d)' = 1 - Da1(d)
```

The next operation to perform is the application of opacity or filters to the container element's buffer. During this step, the operation(s) performed on Da1 should also be performed on Da1(d).

When compositing the group buffer onto the background, rather than the standard compositing operation listed above, the following operations should be used:

```
Dca0' = f(Dc1,Dc0).Da1.Da0 + Y.Dca1.(1-Da0) + Z.Dca0.(1-Da1(d))
Da0'  =          X.Da1.Da0 + Y.Da1.(1-Da0)  + Z.Da0.(1-Da1(d))
```

Note that the last term in the above equations uses the Da(d) buffer rather than Da.

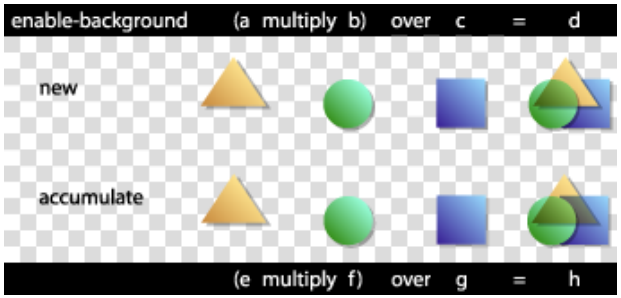
Elements containing a comp-op property value of clear, src, dst, dst-over, src-in, dst-in, src-out, dst-out, src-atop, dst-atop, xor can potentially reduce the opacity of the destination and are only valid where one of the element's ancestorial container element has the enable-background property set to new. For elements without an ancestor with the **enable-background** property set to "new" these operations are technically an error. A user agent should ignore the operation specified and render the element using the src-over compositing operation.

Filters have access to the nearest ancestor group's buffer through the BackgroundImage and BackgroundAlpha images. The buffer created for the ancestor group element of the element referencing the filter, is passed to the filter. Where no ancestors of the element referencing the filter contain an enable-background property value of new, transparent black is passed as input to the filter.

The optional x, y, width, height parameters on the new value indicate the subregion of the container element's user space where input filters have access to the background image. These parameters enable the SVG user agent potentially to allocate smaller temporary image buffers than the effective bounds of the container element. Thus, the values x, y, width, height act as a clipping rectangle on the background image canvas. Negative values for width or height are an error. If not all of the x, y, width and height values or if either width or height are specified as zero then the BackgroundImage and BackgroundAlpha are processed as if enable-background property was set to accumulate.

While container elements are defined as requiring a buffer to be generated, it is often the case that a user agent using various optimizations can choose not to generate this buffer. For example, a group containing a single object could be directly rendered onto the background rather than into a buffer first.

Where a filter references an area of the background image outside the area specified by x, y, width, height, transparent is passed to the filter.



View this image as SVG (SVG 1.2 enabled browsers only)

10.1.5 The knock-out property

knock-out

Value:	true | false | inherit
Initial:	false
Applies to:	container elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	no

For a complex group where the **knock-out** property is set, the buffer is created. The initial contents of the buffer and whether a secondary opacity channel is created depends on the value of the **enable-background** property. For each object within the container element, the object color and opacity replaces that of other objects within the container element, rather than overlaying it. In effect, the destination input to the compositing operations for the complex group's children is the original contents of the buffer, rather than the current buffer for the complex group.

For knock-out = false:

```
Dca1' = f(Sca, Sa, Dca1, Da1)
Da1' = f(Sa, Da1)
```

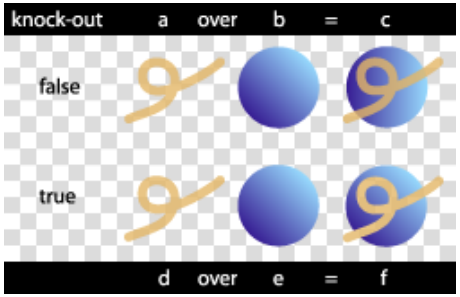
For knock-out = true, enable-background = new:

```
Dca1' = f(Sca, Sa, 0, 0)
Da1' = f(Sa, 0)
```

For 'knock-out' =true ,'enable-background' = accumulate:

```
Dca1' = f(Sca, Sa, Dca0, Da0)
Da1' = f(Sa, Da0)
```

Note that an element in a knockout group that does not have the clip-to-self property set, in effect clears all prior elements in the group.



View this image as SVG (SVG 1.2 enabled browsers only)

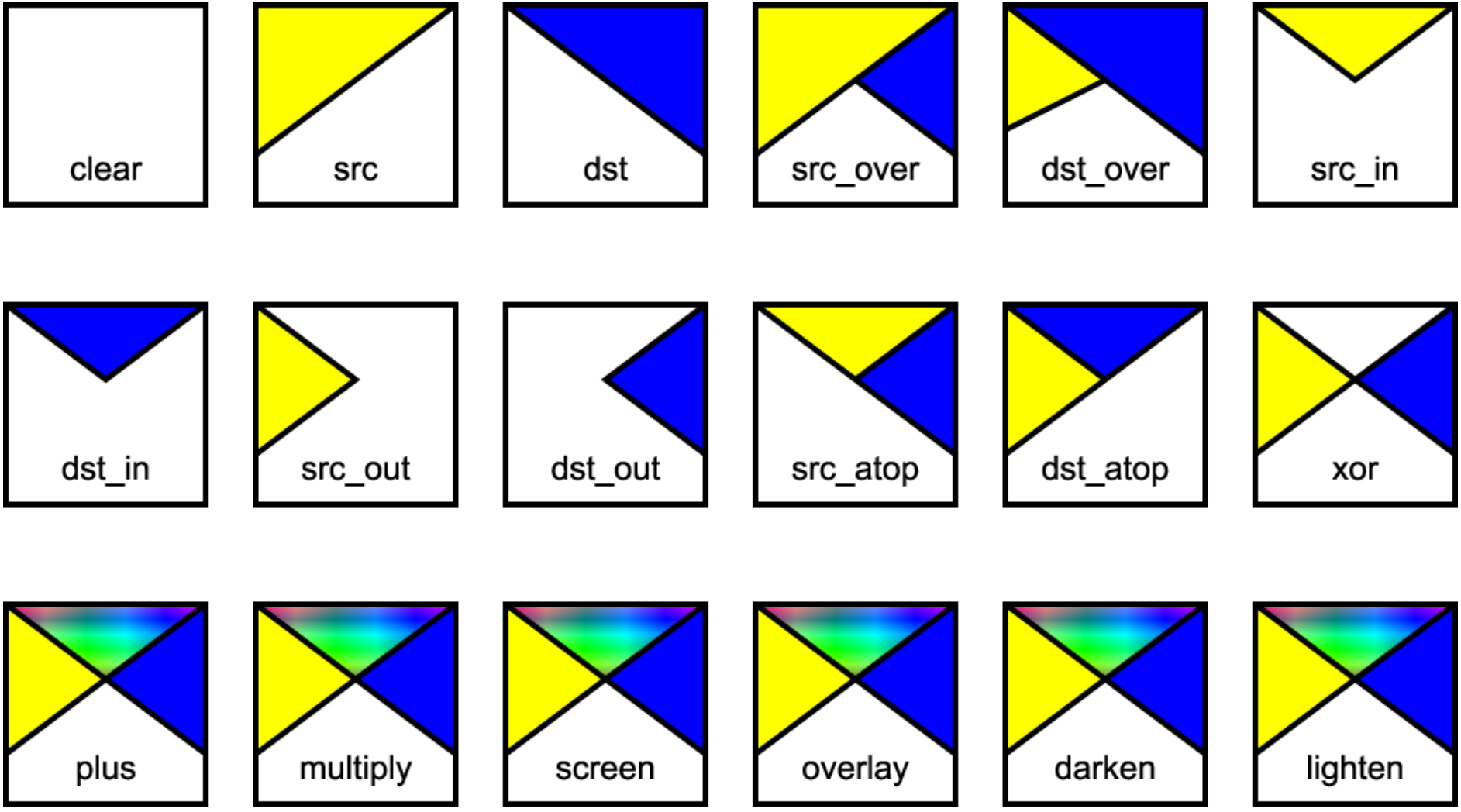
10.1.6 The comp-op property

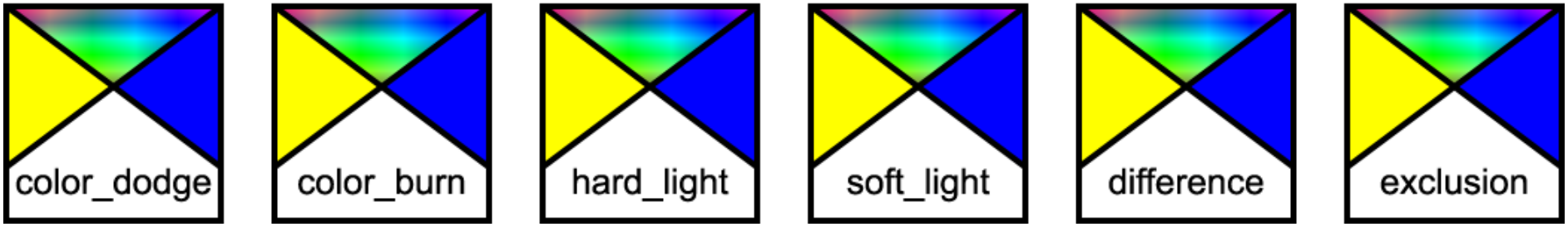
comp-op

Value:	clear | src | dst | src-over | dst-over | src-in | dst-in | src-out | dst-out | src-atop | dst-atop | xor | plus | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | inherit
Initial:	src-over
Applies to:	container elements and graphics elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The **comp-op** property determines the compositing operation used when placing elements onto the canvas. The canvas contains color components and an optional alpha component. When placing new elements onto the canvas, the resulting pixel values on the canvas are calculated using the equations listed in the sections below.

The diagram below shows the sub-pixel regions output by each of the compositing operations.





For many of the operators listed below, the destination is modified in regions of the image where the source is completely transparent. Pixels that the source does not touch are considered transparent, and as such may be modified, depending on the compositing operation. As discussed in the previous section, the bounds of the parent container element can be optimized to save in memory usage and hence, pixel writing requirements. Once the bounds of the parent container element have been determined, each element can only affect the pixels within those bounds.

The following operators change pixels where the source is transparent: clear src src-in dst-in src-out dst-atop

The user agent may be required to create a backing store in which to generate a container element. The size of the backing store for a container element using the default compositing operator src-over is simply the union of the bounds of the sub-elements of the container element. When other compositing operators are used, the bounds of the container element are determined using the compositing operator diagram above. Starting with an empty bounds, the compositing operator specifies that the bounds of each successive object within the container element either replaces the result or is unioned with the result or is intersected with the result. For most compositing operators the bounds are unioned with the result. For the "clear" composite the current result is set to empty. For src, src-out and dst-atop, the bounds are set to the source bounds. For dst, dst-out and src-atop, the bounds are left unchanged. For src-in and dst-in the bounds are intersected with the result.

All color components listed below refer to color component information pre-multiplied by the corresponding alpha value. The following identifiers have the attached meaning in the equations below:

```
Sc  - The source element color value.
Sa  - The source element alpha value.
Dc  - The canvas color value prior to compositing.
Da  - The canvas alpha value prior to compositing.
Dc' - The canvas color value post compositing.
Da' - The canvas alpha value post compositing.
```

clear

Both the color and the alpha of the destination are cleared. Neither the source nor the destination are used as input.

```
f(Sc,Dc) = 0
X        = 0
Y        = 0
Z        = 0

Dca' = 0
Da'  = 0
```

src

The source is copied to the destination. The destination is not used as input.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 1
Z        = 0

Dca' = Sca.Da + Sca.(1 - Da)
     = Sca
Da'  = Sa.Da + Sa.(1 - Da)
     = Sa
```

dst

The destination is left untouched.

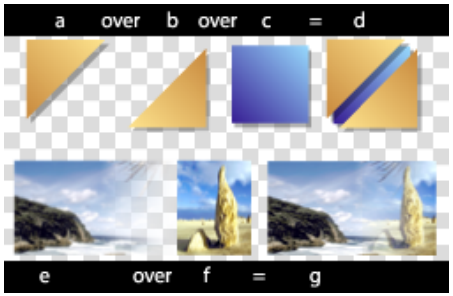
```
f(Sc,Dc) = Dc
X        = 1
Y        = 0
Z        = 1

Dca' = Dca.Sa + Dca.(1 - Sa)
     = Dca
Da'  = Da.Sa + Da.(1 - Sa)
     = Da
```

src-over

The source is composited over the destination.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca.Da + Sca.(1 - Da) + Dca.(1 - Sa)
     = Sca + Dca.(1 - Sa)
Da'  = Sa.Da + Sa.(1 - Da) + Da.(1 - Sa)
     = Sa + Da - Sa.Da
```

The following diagram shows src-over compositing:



View this image as SVG (SVG 1.2 enabled browsers only)

dst-over

The destination is composited over the source and the result replaces the destination.

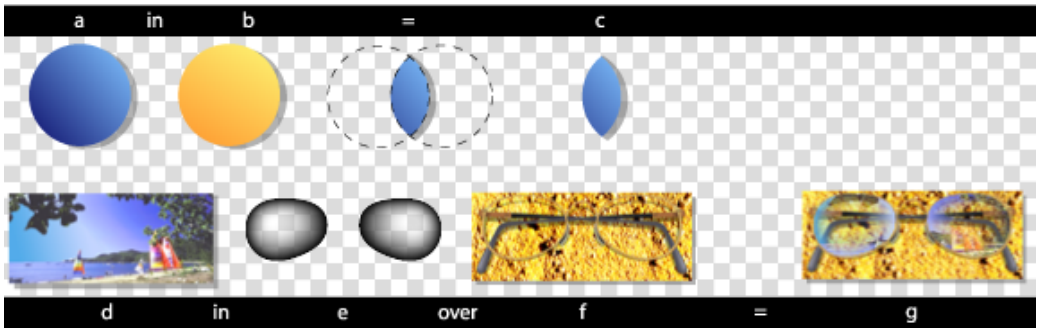
```
f(Sc,Dc) = Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Dca.Sa + Sca.(1 - Da) + Dca.(1 - Sa)
     = Dca + Sca.(1 - Da)
Da'  = Da.Sa + Sa.(1 - Da) + Da.(1 - Sa)
     = Sa + Da - Sa.Da
```

src-in

The part of the source lying inside of the destination replaces the destination.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 0
Z        = 0

Dca' = Sca.Da
Da'  = Sa.Da
```

The following diagram shows src-in compositing:



View this image as SVG (SVG 1.2 enabled browsers only)

dst-in

The part of the destination lying inside of the source replaces the destination.

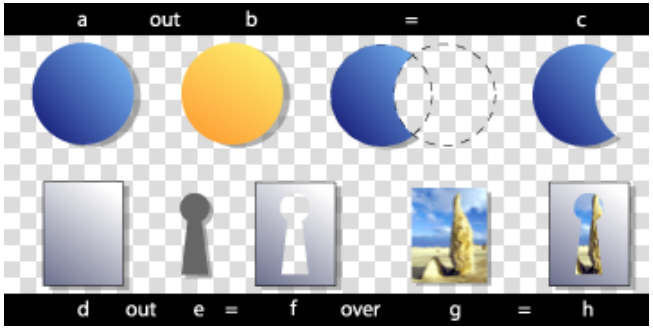
```
f(Sc,Dc) = Dc
X        = 1
Y        = 0
Z        = 0

Dca' = Dca.Sa
Da'  = Sa.Da
```

src-out

The part of the source lying outside of the destination replaces the destination.

```
f(Sc,Dc) = 0
X        = 0
Y        = 1
Z        = 0

Dca' = Sca.(1 - Da)
Da'  = Sa.(1 - Da)
```

The following diagram shows src-out compositing:



View this image as SVG (SVG 1.2 enabled browsers only)

dst-out

The part of the destination lying outside of the source replaces the destination.

```
f(Sc,Dc) = 0
X        = 0
Y        = 0
Z        = 1

Dca' = Dca.(1 - Sa)
Da'  = Da.(1 - Sa)
```

src-atop

The part of the source lying inside of the destination is composited onto the destination.

```
f(Sc,Dc) = Sc
X        = 1
Y        = 0
Z        = 1

Dca' = Sca.Da + Dca.(1 - Sa)
Da'  = Sa.Da + Da.(1 - Sa)
     = Da
```

The following diagram shows src-atop compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

dst-atop

The part of the destination lying inside of the source is composited over the source and replaces the destination.

```
f(Sc,Dc) = Dc
X        = 1
Y        = 1
Z        = 0

Dca' = Dca.Sa + Sca.(1 - Da)
Da'  = Da.Sa + Sa.(1 - Da)
     = Sa
```

xor

The part of the source that lies outside of the destination is combined with the part of the destination that lies outside of the source.

```
f(Sc,Dc) = 0
X        = 0
Y        = 1
Z        = 1

Dca' = Sca.(1 - Da) + Dca.(1 - Sa)
Da'  = Sa.(1 - Da) + Da.(1 - Sa)
     = Sa + Da - 2.Sa.Da
```

The following compositing operators add blending of source and destination colors beyond the base 12 Porter-Duff operations. The behavior of these operators necessitates clamping of the output values after compositing.

plus

The source is added to the destination and replaces the destination. This operator is useful for animating a dissolve between two images.

```
f(Sc,Dc) = Sc + Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca.Da + Dca.Sa + Sca.(1 - Da) + Dca.(1 - Sa)
     = Sca + Dca
Da'  = Sa.Da + Da.Sa + Sa.(1 - Da) + Da.(1 - Sa)
     = Sa + Da
```

multiply

The source is multiplied by the destination and replaces the destination. The resultant color is always at least as dark as either of the two constituent colors. Multiplying any color with black produces black. Multiplying any color with white leaves the original color unchanged.

```
f(Sc,Dc) = Sc.Dc
X        = 1
Y        = 1
Z        = 1

Dca' = Sca.Dca + Sca.(1 - Da) + Dca.(1 - Sa)
Da'  = Sa.Da + Sa.(1 - Da) + Da.(1 - Sa)
     = Sa + Da - Sa.Da
```

The following diagram shows multiply compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

screen

The source and destination are complemented and then multiplied and then replace the destination. The resultant color is always at least as light as either of the two constituent colors. Screening any color with white produces white. Screening any color with black leaves the original color unchanged.

```
f(Sc,Dc) = Sc + Dc - (Sc.Dc)

X        = 1
Y        = 1
Z        = 1

Dca' = (Sca.Da + Dca.Sa - Sca.Dca) + Sca.(1 - Da) + Dca.(1 - Sa)
     = Sca + Dca - Sca.Dca
Da'  = Sa + Da - Sa.Da
```

The following diagram shows screen compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

overlay

Multiplies or screens the colors, dependent on the destination color. Source colors overlay the destination whilst preserving its highlights and shadows. The destination color is not replaced, but is mixed with the source color to reflect the lightness or darkness of the destination.

```
if 2.Dc < Da
  f(Sc,Dc) = 2.Sc.Dc
otherwise
  f(Sc,Dc) = 1 - 2.(1 - Dc).(1 - Sc)
X        = 1
Y        = 1
Z        = 1

if 2.Dca < Da
  Dca' = 2.Sca.Dca + Sca.(1 - Da) + Dca.(1 - Sa)
otherwise
  Dca' = Sa.Da - 2.(Da - Dca).(Sa - Sca) + Sca.(1 - Da) + Dca.(1 - Sa)

Da' = Sa + Da - Sa.Da
```

The following diagram shows overlay compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

darken

Selects the darker of the destination and source colors. The destination is replaced with the source when the source is darker, otherwise it is left unchanged.

```
f(Sc,Dc) = min(Sc,Dc)
X        = 1
Y        = 1
Z        = 1

Dca' = min(Sca.Da, Dca.Sa) + Sca.(1 - Da) + Dca.(1 - Sa)
Da'  = Sa + Da - Sa.Da

or

if Sca.Da < Dca.Sa
  src-over()
otherwise
  dst-over()
```

The following diagram shows darken compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

lighten

Selects the lighter of the destination and source colors. The destination is replaced with the source when the source is lighter, otherwise it is left unchanged.

```
f(Sc,Dc) = max(Sc,Dc)
X        = 1
Y        = 1
Z        = 1

Dca' = max(Sca.Da, Dca.Sa) + Sca.(1 - Da) + Dca.(1 - Sa)
Da'  = Sa + Da - Sa.Da

or

if Sca.Da > Dca.Sa
  src-over()
otherwise
```

```
dst-over()
```

The following diagram shows lighten compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

color-dodge

Brightens the destination color to reflect the source color. Painting with black produces no change.

```
if Sc + Dc >= 1
  f(Sc,Dc) = 1
otherwise
  f(Sc,Dc) = Dc.1/(1-Sc)
X        = 1
Y        = 1
Z        = 1

if Sca.Da + Dca.Sa >= Sa.Da
  Dca' = Sa.Da + Sca.(1 - Da) + Dca.(1 - Sa)
otherwise
  Dca' = Dca.Sa/(1-Sca/Sa) + Sca.(1 - Da) + Dca.(1 - Sa)

Da'  = Sa + Da - Sa.Da
```

The following diagram shows color-dodge compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

color-burn

Darkens the destination color to reflect the source color. Painting with white produces no change.

```
if Sc + Dc <= 1
  f(Sc,Dc) = 0
otherwise
  f(Sc,Dc) = (Sc + Dc - 1)/Sc
X        = 1
Y        = 1
Z        = 1

if Sca.Da + Dca.Sa <= Sa.Da
  Dca' = Sca.(1 - Da) + Dca.(1 - Sa)
otherwise
  Dca' = Sa.(Sca.Da + Dca.Sa - Sa.Da)/Sca + Sca.(1 - Da) + Dca.(1 - Sa)

Da'  = Sa + Da - Sa.Da
```

The following diagram shows color-burn compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

hard-light

Multiplies or screens the colors, dependent on the source color value. If the source color is lighter than 0.5, the destination is lightened as if it were screened. If the source color is darker than 0.5, the destination is darkened, as if it were multiplied. The degree of lightening or darkening is proportional to the difference between the source color and 0.5. If it is equal to 0.5 the destination is unchanged. Painting with pure black or white produces black or white.

```
if 2.Sc < 1
   f(Sc,Dc) = 2.Sc.Dc
otherwise
   f(Sc,Dc) = 1 - 2.(1 - Dc).(1 - Sc)
X        = 1
Y        = 1
Z        = 1
```

```
if 2.Sca < Sa
   Dca' = 2.Sca.Dca + Sca.(1 - Da) + Dca.(1 - Sa)
otherwise
   Dca' = Sa.Da - 2.(Da - Dca).(Sa - Sca) + Sca.(1 - Da) + Dca.(1 - Sa)

Da'  = Sa + Da - Sa.Da
```

The following diagram shows hard-light compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

soft-light

Darkens or lightens the colors, dependent on the source color value. If the source color is lighter than 0.5, the destination is lightened. If the source color is darker than 0.5, the destination is darkened, as if it were burned in. The degree of darkening or lightening is proportional to the difference between the source color and 0.5. If it is equal to 0.5, the destination is unchanged. Painting with pure black or white produces a distinctly darker or lighter area, but does not result in pure black or white.

```
if 2.Sc < 1
   f(Sc,Dc) = Dc.(1 - (1 - Dc).(2.Sc - 1))
otherwise if 8.Dc <= 1
   f(Sc,Dc) = Dc.(1 - (1 - Dc).(2.Sc - 1).(3 - 8.Dc))
otherwise
   f(Sc,Dc) = (Dc + (Dc^(0.5) - Dc).(2.Sc - 1))
X        = 1
Y        = 1
Z        = 1

if 2.Sca < Sa
   Dca' = Dca.(Sa - (1 - Dca/Da).(2.Sca - Sa)) + Sca.(1 - Da) + Dca.(1 - Sa)
otherwise if 8.Dca <= Da
   Dca' = Dca.(Sa - (1 - Dca/Da).(2.Sca - Sa).(3 - 8.Dca/Da)) + Sca.(1 - Da) + Dca.(1 - Sa)
otherwise
   Dca' = (Dca.Sa + ((Dca/Da)^(0.5).Da - Dca).(2.Sca - Sa)) + Sca.(1 - Da) + Dca.(1 - Sa)

Da'  = Sa + Da - Sa.Da
```

The following diagram shows soft-light compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

difference

Subtracts the darker of the two constituent colors from the lighter. Painting with white inverts the destination color. Painting with black produces no change.

```
f(Sc,Dc) = abs(Dc - Sc)
X        = 1
Y        = 1
Z        = 1

Dca' = abs(Dca.Sa - Sca.Da) + Sca.(1 - Da) + Dca.(1 - Sa)
     = Sca + Dca - 2.min(Sca.Da, Dca.Sa)
Da'  = Sa + Da - Sa.Da
```

The following diagram shows difference compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

exclusion

Produces an effect similar to that of 'difference', but appears as lower contrast. Painting with white inverts the destination color. Painting with black produces no change.

```
f(Sc,Dc) = (Sc + Dc - 2.Sc.Dc)
X        = 1
Y        = 1
Z        = 1

Dca' = (Sca.Da + Dca.Sa - 2.Sca.Dca) + Sca.(1 - Da) + Dca.(1 - Sa)
Da'  = Sa + Da - Sa.Da
```

These equations are approximations which are under review. Final equations may differ from those presented here.

The following diagram shows exclusion compositing:

View this image as SVG (SVG 1.2 enabled browsers only)

10.2 Enhanced 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.

10.2.1 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
```

10.2.2 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.

10.2.3 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. Note that for an **svg** element, TS() is the concatenation of transforms found on the ancestors of the **svg** element (**svg** elements themselves do not have a **transform** attribute). Consequently, for an **svg** element, TS(svgId) is the stack of transforms that apply above the VB() transformation (if the **svg** element is not the root element).

Below is an example of the element transform stack:

```
<svg id="root">
  <g id="g" transform="scale(2)">
    <rect id="r" transform="scale(4)"/>
    <svg id="svg0" width="200" height="200" viewBox="0 0 100 100">
      <rect id="r2" transform="scale(0.5)" />
    </svg>
  </g>
</svg>
```

In this example, the transforms are:

```
TS(g) = scale(2)
TS(r) = TS(g) . scale(4) = scale(8)
TS(r2) = scale(0.5)
TS(svg0) = scale(8)
VB(svg0) = scale(2)
```

10.2.4 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 **svg** elements with ids "svg-0" to "svg-n" ("svg-n" being the root).

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

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

```
CTM(elt) = prod{i=0, i=n}(U[i].VB(svg[i]).TS(svg[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, svg[n] refers to the element with the id "svg-n".

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

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

For example, with n=2, we have:

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

This produces the following transformations:

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

          = U.VB(svg[2]).TS(svg[1]).VB(svg[1]).TS(svg[0]).VB(svg[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.

10.2.5 The ref() transform value

In SVG 1.2 the transform attribute has been extended to provide simple constrained transformations using the "ref()" attribute value. A transform attribute can have a value defined in SVG 1.1 or the new "ref" value. The two value types cannot be mixed.

The 'ref(svg, x, y)' transform evaluates to the inverse of the element's parent's CTM multiplied by the closest **svg** element's CTM (from top-most SVG viewport space to the closest **svg** element's user space) but exclusive of the closest **svg** element's user transform, if any. Note that only the outermost **svg** element can have a zoom/pan/rotate user transform. If the closest **svg** element is not the top-most **svg** element there is no user transform to exclude.

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

Where CTM(svg[0].parentElement) evaluates to Identity if 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)
```

In the following example, 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.

In the following example 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 marker will always overlap the middle of the line. Note that the marker will not rotate with the line (e.g., if currentRotate is set) and it will not scale either.

The example below contains nested **svg** elements:

```
<svg id="fullSVGRoot">
  <g id="hostGroup" transform="scale(2)">
    <svg id="tinySVGRoot" baseProfile="tiny" width="400" height="400"
         viewBox="0 0 100 100">
      <rect id="r" transform="ref(svg)" x="5" width="90" y="5"
            height="90" fill="red" />
    </svg>
  </g>
</svg>
```

If the element with id 'fullSVGRoot' has its zoom and pan transform modified, there is an effect on the rectangle's coordinate system. If the "hostGroup" transform is changed, there is an effect on the rectangle.

Using the above definitions, the transform attribute on 'r' evaluates to:

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

So, r's CTM is:

```
CTM(r) = CTM(r.parent).Txf(r)
       = CTM(tinySVGRoot).inv(CTM(tinySVGRoot)).
         CTM(tinySVGRoot.parentElement).VB(tinySVGRoot)
       = CTM(hostGroup).VB(tinySVGRoot)

CTM(hostGroup) = U.VB(fullSVGRoot).TS(hostGroup)

VB(fullSVGRoot) = Identity (no viewBox)
TS(hostGroup) = scale(2)
VB(tinySVGRoot) = scale(4)

CTM(r) = scale(2).scale(4)
       = scale(8)
```

Consequently, the rectangle is always displayed at (20, 20, 360, 360) in the tinySVGRoot viewport space. Since this viewport is subject to the 'fullSVGRoot' user transform, the rectangle will be affected by changes to the root svg's user transform.

11 Painting enhancements

11.1 Background Fill Property

SVG 1.2 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 **background-fill** property defines the paint used to fill the viewport created by a particular element.

background-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 **background-fill**.

```
<svg xmlns="http://www.w3.org/2000/svg"
version="1.2"
     background-fill="red">

  <!-- Everything here has a red background -->

  <!-- This rectangle is not filled, so the red
background
        will show through -->
  <rect x="20" y="20" width="100" height="100"
```

```
        fill="none" stroke="black"/>

  <svg x="40" y="100" width="200" height="200"
       background-fill="green">

    <!-- Everything here has a green background -->

  </svg>

</svg>
```

The filling of the backgound is the first operation in the rendering chain of an element. Therefore:

- The background fill operation happens before filling and stroking.
- The background fill operation occurs before filter effects, and thus is part of the source element's input.
- The background fill operation occurs before compositing, and thus is part of the input to the compositing operations.
- The background fill operation renders into the element's conceptual offscreen buffer, and thus **opacity** applies as usual.
- Background fill is not affected by the **fill** or **fill-opacity** properties.
- As **background-fill** only applies to viewport creating elements, it never undergoes a vector effect.

11.2 Background Fill Opacity Property

The **background-fill-opacity** property specifies the opacity of the **background-fill**.

background-fill-opacity

Value:	**<float>**
Initial:	1.0
Applies to:	viewport-creating elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

```
<svg xmlns="http://www.w3.org/2000/svg"
version="1.2"
     background-fill="red">
```

```
  <!-- Everything here has a red background -->

  <svg x="40" y="100" width="200" height="200"
       background-fill="green" background-fill-
opacity="0.5">

    <!-- Everything here has a half transparent
         green background -->

  </svg>

</svg>
```

11.3 Inheritance into the shadow tree

The **shadowInherit** attribute applies to elements that establish shadow trees and controls property inheritance in those shadow trees. In SVG 1.0 and 1.1, this behavior was fixed depending on the element name. This attribute regularizes control over the inheritance method and gives more flexibility to content creators.

The **shadowInherit** attribute is added to all elements that create shadow trees. It can take one of four values, described below:

onDefine

Properties inherit into the shadow tree at their point of definition, in other words from the element that defines the shadow tree. For example, **marker** elements in SVG 1.1 behave in this way. It is easy to make red markers that are used on multiple paths, and difficult to make markers that are the color of the path on which they are used.

onUse

Properties inherit into the shadow tree at their point of use, in other words from the element that generates the shadow tree. For example, **symbol** elements in SVG 1.1 behave in this way. It is easy to make symbols that are used in multiple places and inherit from the use element that references them.

dynamic

This keyword results in special behavior only in special host-language dependent circumstances where shadow inheritance behavior is variable and dependent on the run-time context. For SVG, the 'dynamic' keyword is equivalent to 'onDefine' except for the **feImage** element. For **feImage**, when referencing a document resource, 'dynamic' is equivalent to 'onUse', but when referencing an element node within a document it is equivalent to 'none'.

none

There is no inheritance into the shadow tree. All properties on the root element of the shadow tree are set to their initial values, as if the shadow tree were in a separate document. This allows SVG content to be created and then re-used without risk of styling changes from the surrounding context.

The default value for **shadowInherit** is 'onUse' except for the following:

- **image**: default is 'none'
- **feImage**: default is 'dynamic' (see description of 'dynamic' above)
- **pattern** and **marker**: default is 'onDefine'

The following example illustrates **shadowInherit**:

```
<svg viewBox="0 0 1000
300"
    xmlns="http://www.w3.org/2000/svg"
version="1.2">

  <desc>Example shadowInherit01 - Illustrate the
effect of
        shadowInherit</
desc>

  <defs
fill="yellow">
    <rect id="rect1" x="0" y="0" width="200"
height="100"/>
  </
defs>

  <use x="100"
y="100"
       xlink:href="#rect1"
shadowInherit="onDefine"
```

```
        fill="red"/
>

  <use x="400"
y="100"
        xlink:href="#rect1"
shadowInherit="onUse"
        fill="green"/
>

  <use x="700"
y="100"
        xlink:href="#rect1"
shadowInherit="none"
        fill="blue"/
>
</
svg>
```

The three rectangles will be colored as follows:

- The first rectangle is yellow because 'onDefine' indicates that the shadow content inherits from the tree where the shadow content is defined.
- The second rectangle is green because 'onUse' indicates that the shadow content inherits from the referencing element (i.e., the element to which the shadow content is attached)
- The third rectangle is black because 'none' indicates that the shadow content does not inherit any property values, neither from where the shadow content was originally defined, nor from the referencing element (i. e., the element to which the shadow content is attached).

11.4 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 (gradients and patterns).

solidColor Schema

```
<define name='solidColor'>
  <element name='solidColor'>
    <ref name='attlist.solidColor'/>
    <ref name='SVG.solidColor.content'/>
  </element>
</define>
```

```
  <define name='attlist.solidColor'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <ref name='SVG.Style.attrib'/>
    <ref name='SVG.Color.attrib'/>
    <ref name='SVG.SolidColor.attrib'/>
    <ref name='SVG.External.attrib'/>
  </define>

  <define name='SVG.SolidColor.attrib'
combine='interleave'>
    <optional>
      <attribute name='solid-color' svg:
animatable='true' svg:inheritable='false'>
        <ref name='SVGColor.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='solid-opacity' svg:
animatable='true' svg:inheritable='false'>
        <ref name='OpacityValue.datatype'/>
      </attribute>
    </optional>
  </define>

  <define name='SVG.solidColor.content'>
    <zeroOrMore>
      <ref name='SVG.Description.class'/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name='stop'/>
        <ref name='animate'/>
        <ref name='set'/>
        <ref name='animateTransform'/>
      </choice>
    </zeroOrMore>
  </define>
```

The **solid-color** property indicates what color to use for this **solidColor** element. The keyword currentColor and ICC colors can be specified in the same manner as within a **<paint>** specification for the **fill** and **stroke** properties.

solid-color

Value: currentColor | **<color>** [icc-color(**<name>** [, **<icccolorvalue>**] *)] | 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

11.5 Using device colors

Certain print applications can improve printing quality by specifying colors by name or in an alternative color format. This is commonly referred to as using 'spot' colors, device colors or inks, and usually means that a particular ink will be used for the color when it is printed. Furthermore, there are applications in the printing press industry where presses can be set up with different inks for different jobs. This means that the content creator will need to create content tailored to the particular press setup in order to obtain the best results.

The **deviceColor** element can be used to indicate an alternative color for a particular paint. This element will be mostly used in closed workflows, since the names of the inks and the parameters (percentages of each ink's color components) rarely have meaning outside the domain of the target device.

deviceColor Schema

```
<define name='deviceColor'>
  <element name='deviceColor'>
    <ref name='attlist.deviceColor'/>
    <ref name='SVG.deviceColor.content'/>
```

```
    </element>
  </define>

  <define name='attlist.deviceColor'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <attribute name='name' svg:
animatable='false' svg:inheritable='false'/>
    <attribute name='uri' svg:animatable='false'
svg:inheritable='false'>
      <ref name='URI.datatype'/>
    </attribute>
  </define>

  <define name='SVG.deviceColor.content'>
    <empty/>
  </define>
```

xlink:href

A URI used to identify the device-specific information included in this element. If the User Agent does not recognize the URI (ie. is not able to recognize the particular device parameters) then the element should be ignored and should not be part of the rendering process.

Animatable: no

name

The name of this device-specific color information. The **name** attribute is used within the **device-color** specification within **<paint>** to reference this **deviceColor** element.

Animatable: no

The **deviceColor** element uses attributes from external namespaces to define the device specific properties that are to be used when the **deviceColor** is referenced from a **<paint>**.

The following example illustrates the use of **deviceColor**. There are two things to note:

1. The **deviceColor** element describes device specific setup information.

2. The **device-color** keyword is used as an alternative **<paint>** specification, achieving the desired **<paint>** when the output is the named device (or when the User Agent is able to understand the device specific information).

```
<svg xmlns="http://www.w3.org/2000/svg"
version="1.2"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:ecpi="http://www.example.com/press/inks">

  <defs>
     <!-- describe a particular output device -->
     <deviceColor name="device-inks"
                  xlink:href="http://www.example.
com/pressInks"
             ecpi:value="Cyan, Magenta, Yellow,
Black, Silver, Gray, Green"/>
  </defs>

  <text x="100" y="150" font-family="Verdana" font-
size="35"
        fill="rgb(22,33,44) device-color(device-
inks, 11,55,66,77,0,0,88)" >

     Hello, out there

  </text>

</svg>
```

In the example above, a supplemental attribute, **value**, from a private namespace has been added. This example **value** attribute provides the definitions of colors or inks to be interpreted in the context of the URI specified. It is in a private namespace so that content and context authors can use any understood format to convey the necessary information. When the particular **deviceColor** element is referenced later by a **device-color** keyword specification, it is generally expected that the number of parameters following the name reference (1st parameter) in the function-like representation for the value of **device-color** alternate in a fill or stroke attribute (for example) will have an understood one-to-one correspondence with the information specified for the value attribute in the the **deviceColor** element. The interpretation of the parameters is implied specifically by the context set by the URI.

11.5.1 The device-color keyword

The **device-color** keyword for specifying device specific **<paint>** colors is to be used only by those agents that understand the full meaning of the context set by the URI in the referred to DeviceColor element. The keyword can only be used with a corresponding color definition, such as a color keyword or RGB definition. The color definition, with ICC Color specification when present, is used if the agent does not understand the rendering context implied by the referenced **device-color** URI. The priority of the use of color specifications are as follows: **device-color** if understood, **color-profile** if present and understood and finally the sRGB specification.

The first parameter in the function like representation **device-color** must be the name for a defined **deviceColor** element, in a manner similar to **color-profile**. The remainder of the parameters are interpreted entirely in light of the information provided in the **deviceColor** element. There is a correspondence between the parameters after the name in the functional representation for **device-color** and the external attributes on the referenced **deviceColor**. In the example above, the interpretation of the values meaning (ink volume to use, percent of total ink volume, or whatever) is strictly in the context of the specification or convention implied by the URI in the referenced **deviceColor** element.

11.5.2 DOM Interface

Interface SVGDeviceColorElement provides access to the **deviceColor** element.

IDL Definition

```
interface SVGDeviceColorElement : SVGURIReference {
  readonly attribute DOMString name; // raises DOMException
on setting
};
```

Attributes

readonly DOMString name

Corresponds to the attribute 'name' on the given element

11.6 ICC named colors

The icc-named-color keyword is used to specify so-called 'spot' colors, also known as named colors. Examples of named colors include Pantone colors.

SVG 1.2 supports device independent ICC named color profiles. These profiles will be of class "Named Color profile" as defined in section 6.1.4 of the ICC profile specification version 4.0.0 . Such profiles may be included in an SVG file via use of the **color-profile** element.

The icc-named-color keyword indicates what named color to use for specifying paint for a given element.

11.7 Specifying paint

The **fill** and **stroke** properties are defined as a value of type **paint**, which is specified as follows:

```
<paint> : none | currentColor | ((color)) [icc-color(<name>[,
<icccolorvalue>]*)][device-color(<name>[,<devicecolorvalue>]
*)][icc-named-color(<name>,<colorname>)] |
<uri> [ none | currentColor | <color> [icc-color(<name>[,
<icccolorvalue>]*)] [device-color(<name>[,<devicecolorvalue>]
*)] [icc-named-color(<name>,<colorname>)]] | inherit <http://
www.w3.org/TR/REC-CSS2/cascade.html#value-def-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> [icc-color(<name> [, <icccolorvalue>])] [device-color(<name> [, <devicecolorvalue>])] [icc-named-color(<name> , <colorname>)]*

The **<color>** is the explicit color (in the 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, with the exception that SVG contains an expanded list of recognized color keywords names. There are three optional syntaxes for specifying icc profile defined and device colors to aid color accuaracy in applications such as printing. These are icc-color, device-color and icc-named-color.

If an optional icc-color specification is provided, then the user agent searches the color profile description database for a color profile description entry whose name descriptor matches **<name>** and uses the last matching entry that is found. (If no match is found, then the ICC color specification is ignored.) The comma-separated list (with optional white space) of **<icccolorvalue>**'s is a set of ICC-profile specific color values, expressed as **<number>**. (In most cases, the **<icccolorvalue>** will be in the range 0-to-1.)

The device-color keyword for specifying device specific colors is to be used only by those agents that understand the full meaning of the context set by the URI in the referred to DeviceColor element. The color definition, with icc-color specification when present, is used if the agent does not understand the rendering context implied by the referenced device-color URI.

The first parameter in the function like representation device-color, **<name>**, must be the name for a defined deviceColor element, resolved in the same manner as color-profile. The remainder of the parameters are interpreted entirely in light of the information provided in the **deviceColor** element. There is a correspondence between the parameters after the name in the functional representation for device-color and the external attributes on the referenced **deviceColor**.

If an optional icc-named-color specification is provided, then the user agent searches the color profile description database in the same manner as for icc-color. The **<colorname>** string is a profile specific color name. (In most cases, the **<colorname>** will match a string in the ICC named color profile.)

If the paint contains a icc-color, device-color or icc-named-color, it must first specify a **<color>** for fallback. If the user agent cannot render a particular optional color definition, they should fallback to the next color definition of lower priority. The priority for the use of color specifications is as follows:

1. icc-named-color.
2. device-color.
3. icc-color.
4. sRGB color.

NOTE:

Note that color interpolation occurs in an RGB color space even if an ICC-based color specification is provided (see color-interpolation and {color-interpolation-filters | http://www.w3.org/TR/SVG11/painting.html#ColorInterpolationFiltersProperty]). Percentages are not allowed on **<icccolorvalue>**'s. For more on ICC-based colors, refer to Color profile descriptions.

<uri> [none | currentColor | <color> [icc-color(<name> [, <icccolorvalue>])] [device-color(<name> [, <devicecolorvalue>])] [icc-named-color(<name>, <colorname>)]*

The **<uri>** is how you identify a paint server such as a gradient, a pattern or a custom paint defined by an extension. The **<uri>** provides the ID of the paint server (e.g., a **gradient**, **pattern** or **solidColor**) to be used to paint the current object. If the URI 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 paint method following this defintion is used if provided; otherwise, the document is in error (see Error processing.

11.8 Controlling the rendering color space

SVG 1.2 adds a new property to give increased control over the color space used for rendering.

11.8.1 The rendering-color-space property

The **rendering-color-space** property defines the color space that an element's rendering operations will take place in. Conceptually this involves the creation of an offscreen buffer with color space is defined by the ICC profile referenced by the property. All fill/stroke/gradient/pattern specifications must be converted to this color space before elements are rendered. Images and the results of filtering must be color converted, when required, to the specified color space before

being composited. After the object/group is rendered the offscreen image must be converted to the color space defined by the **rendering-color-space** property on the object/group's parent before being composited into the parent's offscreen buffer.

The ICC profile referenced must provide forward and reverse conversion, as the implementation will need to convert to and from the specified color space; most ICC profiles provide both conversions. To limit the burden on implementors only three channel ICC profiles are required to be supported. In cases where the specified ICC profile can not be used (such as not being available, or because it has more than three channels of output) the implementation must use the **rendering-color-space** specification from the first ancestor that has a usable profile associated (i.e. it is as if a value was not provided for this property for this element).

Note that standard SVG compositing rules are used. As a consequence highly non-linear color spaces (such as HSV) or non-orthogonal color spaces (such as CMYK) may give unintuitive results when blending colors.

rendering-color-space

Value:	auto | sRGB | linearRGB | **<name>** | **<uri>** | inherit
Initial:	auto
Applies to:	outermost SVG elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Animatable:	yes

auto
: The user agent must defer to the 'color-interpolation' property.

sRGB
: Specifies that the sRGB color space is to be used for rendering.

scRGB
: Specifies that the scRGB color space is to be used for rendering.

linearRGB
: Specifies that the linear sRGB color space is to be used for rendering.

<name>
: A name corresponding to a defined color profile that is in the user agent's color profile description database. The user agent searches the color profile description database for a color profile description entry whose

name descriptor matches **<name>** and uses the last matching entry that is found. If no match is found then it is as if 'sRGB' were specified.

<uri>

A URI reference to the source color profile to use for rendering.

The **color-interpolation** property on gradients and **color-interpolation-filters** property on filter primitives are also extended to support **<name>** and **<uri>** references. This enables gradients and filters where interpolation occurs in an alternate color space. When the gradient is rendered the colors must be converted to the **rendering-color-space** of the object the gradient is applied to (consistent with the current definition of **color-interpolation**).

The **color-interpolation** property on graphics elements is deprecated. The current definition of **color-interpolation** states that when the property is set on an element it controls what color space is used when the child is composited with it's parent.

The problem with this definition is that it implicitly requires converting the content already rendered into the parents buffer to the color space specified by **color-interpolation** for compositing with the child. This made sense when the only supported color spaces were sRGB and linear sRGB, because presumably the compositing code would perform the conversion, composite and convert back all at once, thus preserving the fidelity of the parent buffer. However it would be extremely difficult and computationally expensive for implementations to do this for arbitrary ICC color spaces.

Given this definition, for the use of the **color-interpolation** property to be make sense, the property would generally have to be set on all the children of a grouping element, but not be set on the grouping element. This would allow an implementation to composite all the children in the desired color space, and only convert to the 'parent' color space at the end of the group. However having to set a property on all the children but not on the parent would be extremely fragile as well as error prone for generators of SVG content.

Thus the SVG working group has decided that **rendering-color-space** should replace **color-interpolation** for use on graphics elements, it allows a clearer expression of the authors intent, and makes costly mistakes (both in speed and quality) less likely.

The user agent will go into error if the value of **rendering-color-space** is anything but 'auto' when **color-interpolation** has a value of 'linearRGB'. If **color-interpolation** is removed a future version of the specification the 'auto' value for **rendering-color-space** will be defined to mean 'sRGB'.

11.9 Filter Region extensions

In SVG 1.1, a **filter** defines the area upon which it applies. This makes it difficult to develop a generic filter that can be applied to arbitrary graphics, since the filter must define a large enough area to cover any graphical object to which it is applied. An example of this is a generic "drop shadow" filter, which is commonly specified as a combination of a Gaussian blur **feGuassianBlur**) that is offset **feOffset**) and then composed **feComposite**) with the original source graphic. Since the shadow has to extend beyond the original graphic's boundaries, the filter must be defined to have a larger area than the original graphic. Overestimating this margin has a negative effect on performance, since the complex filter operation has to touch a larger amount of user space (ie. pixels).

In order to solve this problem, SVG 1.2 adds margins to the **filter** and filter primitive elements. These margins are added to the filter region, once any conversion of the filter region from object space into user space has been made.

In particular, the **filterMarginsUnits**, **filterPrimitiveMarginsUnits**, **dx**, **dy**, **dw** and **dh** are added to the **filter** element. The **filterMarginsUnits** specifies the coordinate space of the new margin attributes, which are used to increase or decrease the **filter** element's **x**, **y**, **width** and **height** attributes (once they have been calculated). The **filterPrimitiveMarginsUnits** specifies the units for the new margin attributes on the filter primitives, also named **dx**, **dy**, **dw**, **dh**. These margins attribute override those set on the parent **filter** element.

An example of the new attributes, which defines a generic dropShadow filter:

```
<filter id="dropShadow" x="0" y="0" width="1"
height="1"
          filterMarginsUnits="userSpaceOnUse"
          dx="0" dy="0" dw="5" dh="5" >
      <feGaussianBlur stdDeviation="2"
in="SourceAlpha" />
      <feOffset dx="2" />
      <feMerge>
           <feMergeNode />
           <feMergeNode in="SourceGraphic" />
      </feMerge>
</filter>
```

In the above example, the filter region by default covers the entire bounds of the object (which is not enough to show the shadow). Adding the new margins

extends the width and height by 5 user units each, which is always enough to display the blur (which has a standard deviation of 2 user units) and offset (which is another 2 units).

11.10 Prefetching resources

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.

11.10.1 The prefetch element

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 page element. In this case the prefetch element must appear in a defs block before all defined pagesets and pages 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.

prefetch Schema

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

  <define name='attlist.prefetch'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <ref name='SVG.XLinkRequired.attrib'/>
    <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'/>
    </optional>
    <optional>
      <attribute name='mediaCharacterEncoding'
svg:animatable='false'
         svg:inheritable='false'/>
    </optional>
    <optional>
      <attribute name='mediaContentEncodings'
svg:animatable='false'
        svg:inheritable='false'/>
    </optional>
    <optional>
      <attribute name='bandwidth' svg:
animatable='false'
        svg:inheritable='false'>
        <ref name='NumberOrPercentage.datatype'/>
      </attribute>
    </optional>
  </define>
```

mediaSize = <number> | <percentage>

Defines how much of the media to fetch as a function of the file size of the media. To fetch the entire media without knowing its size, specify 100%. The default is 100%.

When prefetch refers to a resource in the same document (e.g. a page element), the mediaSize attribute indicates the size in bytes of the referred page. 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.

mediaTime = <clock> | <percentage>

Defines how much of the media to fetch as a function of the duration of the media. To fetch the entire media without knowing its duration, specify 100%. The default is 100%. 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 page element), this is the active duration of the referenced page 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 page. 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.

bandwidth = <number> | <percentage>

Defines how much network bandwidth the user agent should use when doing the prefetch. To use all that is available, specify 100%. The default is 100%. Any attribute with a value of "0%" is ignored and treated as if the attribute wasn't specified.

mediaCharacterEncoding = <encodings>

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.

mediaContentEncodings = <encodings>

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.

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 **page** 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:

```
<svg width="400" height="300" version="1.2"
     xmlns="http://www.w3.org/2000/svg"
     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 **page** element in the same document):

```
<svg width="400" height="300" version="1.2"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <desc>
     Example of using prefetch for 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>

  <pageSet>
    <page id="scene1">
      <!-- graphics for scene 1 go here -->
    </page>

    <page id="scene2">
```

```
      <!-- graphics for scene 2 go here -->
    </page>

    <page id="scene3">
      <!-- graphics for scene 3 go here -->
    </page>
  </pageSet>
</svg>
```

11.11 Referencing external stylesheets

SVG 1.2 adds the **xlink:href** attribute to the **style** element, allowing the reference of external stylesheets. The attribute behaves in the same way as the corresponding attribute on the **script** element.

11.12 Increased switch availability

SVG 1.2 allows the **switch** element as a child of any element.

11.13 The min-unit-scale and max-unit-scale attributes

The **min-unit-scale** and **max-unit-scale** attributes refer to the scale factor in the transform between the element's user space and the viewport space. These attributes are allowed to provide one or two values. If two values are provided then the first refers to the scale factor in the horizontal direction and the second value refers to the scale factor in the vertical direction. If one value is provided it is used for both the horizontal and vertical scale factors.

When implementations evaluate these attributes they should pay attention to the possibility that the content is being rendered in a filter using the **filterRes** attribute when calculating the scale factor from the current coordinate system to the viewport coordinate system.

If the scale factor from the current user coordinate system to the viewport coordinate system is greater than **min-unit-scale**, then the test evaluates to true. If the scale factor from the current user coordinate sytsem to the viewport coordinate system is smaller than **max-unit-scale**, then the test evaluates to true. Otherwise, the tests evaluate to false.

The **min-unit-scale** and **max-unit-scale** attributes are often used in conjunction

with the **switch** element. If the **min-unit-scale** and/or **max-unit-scale** attributes are used in other situations, then they represents a simple switch on the given element whether to render the element or not.

11.14 Testing for formats

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.

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.
style(<mime-type>):
Test the MIME-type as a stylesheet language type.
script(<mime-type>):
Test the MIME-type as scripting language type.
foreignObject(namespaceURI):
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)
- style(xslt)

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

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

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

Also the following method is exposed on SVGSVGElement:

boolean isFormatSupported(in dom::DOMString formatDefinition);

11.15 Testing for font availability

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.

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 = list-of-font-names

Below is an example of the **requiredFonts** attribute.

```
<switch>
   <text requiredFonts="FancyPants" id="txt" font-
family="FancyPants">
      <tspan x="..." y="...">....</tspan>
      <tspan x="..." y="...">....</tspan>
      <tspan x="..." y="...">....</tspan>
   </text>
   <flowRoot>
      <flowRegion>
        ...
      </flowRegion>
    <flowPara><flowTRef xlink:href="#txt"></
flowPara>
  </flowRoot>
</switch>
```

In the above example, if the FancyPants font is available on the system, or has been declared as an SVG font elsewhere in the document, then the first child of the switch evaluates to true, and the precise layout of the tspan is used (with the FancyPants font). If the FancyPants font is not available, then the system default font is used inside a flowing region, with the text referenced from the precise layout.

11.16 Overlaying graphics

There are many cases when it is necessary for graphical objects to be drawn above the canvas.

11.16.1 The overlay property

The **overlay** property controls how an element's canvas is composited into the document canvas.

overlay

Value:	'top' | 'none'
Initial:	'none'
Applies to:	**svg** element
Inherited:	no

Percentages: N/A
Media: visual
Animatable: yes

The SVG 1.1 specification says:

Grouping elements such as the **g** have the effect of producing a temporary separate canvas initialized to transparent black onto which child elements are painted. Upon the completion of the group, any filter effects specified for the group are applied to create a modified temporary canvas. The modified temporary canvas is composited into the background, taking into account any group-level masking and opacity settings on the group.

For elements that have the **overlay** property is set to "top" the element's "temporary separate canvas" is not composited to the background as usual. Instead that "temporary separate canvas" is set aside. In other words, the element gets drawn to its canvas, but that canvas is not drawn into to the background yet (and instead drawn later in the whole document's compositing process). Such canvases are commonly referred as overlays, hence the name.

11.16.1.1 The overlay-host property

The **overlay-host** property affects how all "temporary separate canvases" of this element descendants are composited.

overlay-host

Value: 'true' | 'false'
Initial: 'false'
Applies to: **svg** element
Inherited: no
Percentages: N/A
Media: visual
Animatable: yes

If the value of **overlay-host** is true, then after the "modified temporary canvas is composited into the background" for this **svg** element, all "temporary separate canvases" that were set aside in the course of this element's drawing are composited into the background. In other words, overlay host draws all the overlays on top of the other content of the element. Overlays are drawn in the order they were created.

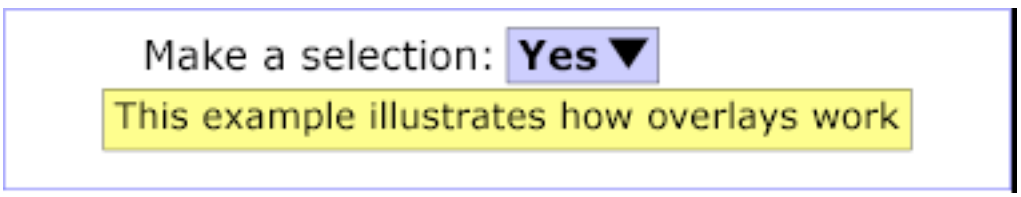
11.16.2 Example

The following example shows how to use the 'overlay' property to create a simple popup menu:

```
<svg xmlns="http://www.w3.org/2000/svg"
version="1.2"
     viewBox="0 0 1000 180">
  <!-- draw the outline of the viewable area --
>
  <rect id="myRect" x="1" y="1" width="998"
height="178"
        fill="none" stroke="blue"/
>

  <g transform="translate
(500,60)">
    <!-- This is the prompt label for the popup --
>
    <text x="-10" y="0" text-anchor="end"
          font-
size="40">
      Make a
selection:
    </
text>

    <!-- This shows the current choice for the popup --
>
    <rect x="0" y="-40" width="150"
height="54"
        fill="#CCF" stroke="black"/
>
    <text x="10" y="0" text-anchor="start"
          font-size="40" font-weight="bold">Yes</
text>

    <!-- The user must click on this triangle to expose the
popup -->
    <path id="triangle" transform="translate(100,-
30)"
          d="M0,0 L 40,0 L 20,30 z"/
>

    <!-- This is the popup menu, which is invisible initially
-->
```

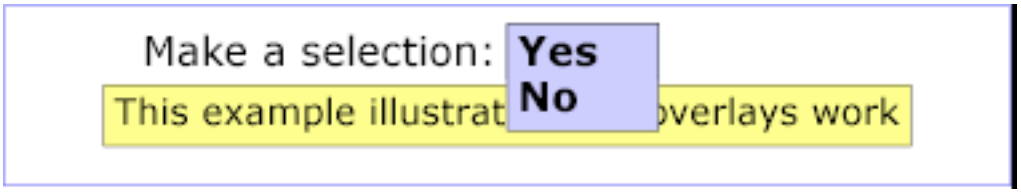
```
    <g id="popup" display="none"
overlay="top">
      <rect x="0" y="-40" width="150"
height="105"
            fill="#CCF" stroke="black"/
>
      <text x="10" y="0" text-anchor="start"
            font-size="40" font-weight="bold">Yes</
text>
      <text x="10" y="45" text-anchor="start"
            font-size="40" font-weight="bold">No</
text>
    </
g>
    <!-- Illustrative declarative animation to expose popup.
-->
    <set xlink:href="#popup"
attributeName="display"
         to="inline" begin="triangle.mousedown" end="mouseup"/
>
  </
g>

  <!-- Additional graphics and text at the bottom --
>
  <rect x="100" y="80" width="800" height="60" fill="#FF8"
stroke="black"/>
  <text id="Note" x="500" y="120" text-anchor="middle" font-
size="36">
    This example illustrates how overlays
work
  </
text>

</svg>
```

In this example, the popup menu is hidden initially. When the user initiates a mousedown event on the triangle, then the popup appears. Here is how the example appears initially:

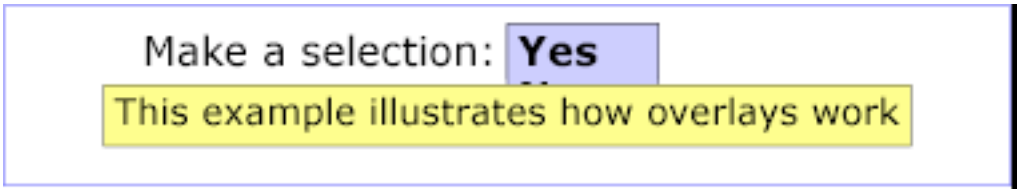


Here is how the example appears once the user has initiated a mousedown

event on the triangle but before any mouseup events occur:



The key thing to observe in this example is the overlay="top" value on the **g** element whose ID is "popup". Because the **g** specifies overlay="top", when the popup is displayed, it will be drawn above the rest of the graphics because its rendering will be on a temporarily canvas when is composited onto the background after the rest of the document is rendered. Without overlay="top", the popup would be obscured by the graphics that appears later in the document. In particular, if overlay="top" were not specified on the popup, the document would have shown the popup as follows:



11.17 Modifications to cursors

11.17.1 The cursor property

In SVG 1.2, the **cursor** property adds a new value "progress", as defined in CSS 2.1. It is often rendered as a spinning beach ball, or an arrow with a watch or hourglass.

11.17.2 Inline cursor content

SVG 1.2 allows cursor appearance to be defined using SVG content. The following is a replacement for the SVG 1.1 section on the cursor element:

The **cursor** element can be used to define a platform-independent custom cursor. A recommended approach for defining a platform-independent custom cursor is through a **cursor** element with either PNG [PNG01] or SVG to define cursor's image and identify the exact position within the image which is the pointer position (i.e., the hot spot). If the **cursor** element has an **xlink:href** attribute, it is assumed that cursor shape is defined through an external image, otherwise cursor element's content is drawn into a separate canvas and the

result is used as cursor's image.

If a bitmap image is used, the PNG format is recommended because it supports the ability to define a transparency mask via an alpha channel. If a different image format is used, this format should support the definition of a transparency mask (two options: provide an explicit alpha channel or use a particular pixel color to indicate transparency). If the transparency mask can be determined, the mask defines the shape of the cursor; otherwise, the cursor is an opaque rectangle. Typically, the other pixel information (e.g., the R, G and B channels) defines the colors for those parts of the cursor which are not masked out. Note that cursors usually contain at least two colors so that the cursor can be visible over most backgrounds.

If **xlink:href** attribute is specified the **width**, **height**, **viewBox** and **preserveAspectRatio** attributes are meaningless and must not be used.

If the cursor image is defined through SVG, the rules are the following:

- the initial state of the cursor image canvas is transparent blank
- the **cursor** element creates its own viewport to draw the cursor content and allows regular viewport attributes (width, height, viewBox and preserveAspectRatio). The **overflow** property does not have any effect because nothing can be drawn outside of the cusor's image dimensions.
- animations and DOM modifications are allowed in the cursor element and should be visible.
- if a **viewBox** is used, the hot spot is defined in terms of the cursor's element children (i.e. after application of the viewBox)

Attribute definitions:

x = " <coordinate>"
: The x-coordinate of the position in the cursor's coordinate system which represents the precise position that is being pointed to. If the attribute is not specified, the effect is as if a value of "0" were specified. Animatable: yes.

y = " <coordinate>"
: The y-coordinate of the position in the cursor's coordinate system which represents the precise position that is being pointed to. If the attribute is not specified, the effect is as if a value of "0" were specified. Animatable: yes.

xlink:href = " <uri>"
: A URI reference to the file or element which provides the image of the

cursor. Animatable: yes.

width = " <length>"
The horizontal dimension of the cursor image if the cursor is defined through SVG. Animatable: yes

height = " <length>"
The vertical dimension of the cursor image if the cursor is defined through SVG. Animatable: yes

preserveAspectRatio
As usual

viewBox
As usual

11.18 Highlighting

SVG 1.1 allows a target object to be denoted, either with a view element or with a viewTarget as part of an SVG view specification in a fragment identifier. The target object(s) are to be highlighted.

While there are some common highlighting strategies, e.g. a thick red outline for black and white schematics, in the fully general case there is no one presentation of highlighting that is guaranteed to be visible in all cases. An element might already have a thick red outline, for example. Approaches using filter effects for color inversion can also be non-obvious in certain cases.

Accordingly, the SVG 1.1 description: "Indicates the target object associated with the view. If provided, then the target element(s) will be highlighted." does not specify how the highlighting is to be achieved or even if the highlighting must be visually distinct from the non-highlighted case.

In SVG 1.2, highlighting in a conforming SVG viewer must be visually detectable. The default rendering is left application dependent, it just has to be visible. Since content creators are in a much better position to decide on an appropriate visual effect for their graphics, a dynamic pseudo-class ':highlight' is provided in SVG 1.2. It matches the target element(s) that are to be highlighted. Appropriate styling can thus be specified by the author.

```
:highlight { stroke-width: 5; stroke-color: red }
#map .country:highlight {filter: url
(#DropShadowBehind)}
```

The first example applies a thick red stroke to elements that are highlighted. The

second applies a filter effect, but only to elements of class "country" which are descendents of the element with id="map".

The SVG 1.2 vector effects have a nice synergy with highlighting, since they allow (for example) multiple strokes to be applied to an element.

```
:highlight { vector-effect: url
(#ThickRedSecondStroke) }
```

11.19 Automatic text length

SVG 1.2 adds a new keyword, "auto", to the allowed values of the **textLength** attribute.

textLength = " <length> | auto"

The author's computation of the total sum of all of the advance values that correspond to character data within this element, including the advance value on the glyph (horizontal or vertical), the effect of properties **kerning**, **letter-spacing** and **word-spacing** and adjustments due to attributes **dx** and **dy** on **tspan** elements. This value is used to calibrate the user agent's own calculations with that of the author.

The purpose of this attribute is to allow the author to achieve exact alignment, in visual rendering order after any bidirectional reordering, for the first and last rendered glyphs that correspond to this element; thus, for the last rendered character (in visual rendering order after any bidirectional reordering), any supplemental inter-character spacing beyond normal glyph advances are ignored (in most cases) when the user agent determines the appropriate amount to expand/compress the text string to fit within a length of textLength.

A negative value is an error.

The "auto" value is as if the author's computation exactly matched the value calculated by the user agent; thus, no advance adjustments are made.

If the attribute is not present, it is as if "auto" was specified.

11.20 More rendering hints

There exist cases where a user agent could achieve better performance if it were able to cache an offscreen buffer for a particular group or element. An example of such a use case are commonly called sprites.

SVG 1.2 adds three new properties that provide rendering hints to the user agent.

11.20.1 The cache property

The **cache** property suggests how much resources should be allocated to dynamic update.

cache

Value:	'true' | 'false' | 'auto'
Initial:	'auto'
Applies to:	graphical elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The value 'true' means that the element is expected to be redrawn often. The user agent may be able to assign extra resources in order to increase the rendering performance.

The value 'false' means that the element is not expected to be redrawn often. The user agent may be able to conserve resources that it normally allocated to redrawing.

The default value 'auto' means that the user agent is expected to use a reasonable compromise between speed of redraw and resource allocation.

11.20.1.1 The static property

The **static** property suggests whether or not an element is expected to be modified often.

static

Value:	'true' | 'false' | 'auto'
Initial:	'auto'
Applies to:	graphical elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The value 'false' means that the element is expected to be modified often.

The value 'true' means that the element is not expected to be modified often. This suggests that user agent may be able to allocate resources, such as an offscreen buffer, that would allow increased performance in redraw. It does not mean that the element will never change. If an element is modified when **static** has the value 'true', then redraw might have reduced performance.

The default value 'auto' means that the user agent is expected to use a reasonable compromise between speed of update and resource allocation.

11.20.2 The snap property

The **snap** property suggests whether or not an element is expected to be displayed in many places and to appear close to identical in each display.

snap

Value:	'true' | 'false' | 'auto'
Initial:	'auto'
Applies to:	graphical elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The value 'false' means that the element is not expected to be displayed in many locations, or that it is not required to look identical in every location.

The value 'true' means that the element is expected to be rendered in many

places and different instances are expected to look as close as possible if all rendering parameters are the same and the user space transformation matrices differ only in the translation component. A typical use of this would be a symbol on a map. A user agent could choose to implement this hint by applying an additional translation to user space before drawing in order to align the user space with device pixels.

The default value 'auto' means that the user agent is expected to use a reasonable compromise between the two values above.

Previous | Top | Next

12 Media

12.1 The audio element

The **audio** element specifies an audio file which is to be rendered to provide synchronized audio. The usual SMIL animation 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.

audio Schema

```
<define name='audio'>
  <element name='audio'>
    <ref name='attlist.audio'/>
    <ref name='SVG.audio.content'/>
  </element>
</define>

<define name='SVG.audio.content'>
  <zeroOrMore>
    <ref name='SVG.Description.class'/>
  </zeroOrMore>
  <zeroOrMore>
    <choice>
      <ref name='SVG.Animation.class'/>
      <ref name='SVG.Handler.class'/>
    </choice>
  </zeroOrMore>
</define>
```

The following example illustrates the use of the audio element. When the button is pushed, the audio file is played three times.

```
<svg width="100%" height="100%" version="1.2"
     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="audio/vorbis"
        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:

The DOM interface for the **audio** element is shown below:

```
interface SVGAudioElement :
                SVGMediaElement,
                SVGURIReference,
                SVGLangSpace {

    readonly attribute SVGAudio audio;
};
```

12.1.1 Supported audio format

SVG user agents are required to support the Ogg Vorbis audio format.

Profiles of SVG may impose restrictions or remove support for Ogg Vorbis. They may also require support for other formats.

12.2 The video element

The **video** element specifies a video file which is to be rendered to provide synchronized video. The usual SMIL animation 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 also includes 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.

video Schema

```
  <define name='video'>
    <element name='video'>
      <ref name='attlist.video'/>
      <ref name='SVG.video.content'/>
    </element>
  </define>

  <define name='SVG.video.content'>
    <zeroOrMore>
      <ref name='SVG.Description.class'/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name='SVG.Animation.class'/>
        <ref name='SVG.Handler.class'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='attlist.video'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <ref name='SVG.Conditional.attrib'/>
    <ref name='SVG.Opacity.attrib'/>
    <ref name='SVG.Graphics.attrib'/>
```

```
    <ref name='SVG.Focusable.attrib'/>
    <ref name='SVG.Clip.attrib'/>
    <ref name='SVG.Mask.attrib'/>
    <ref name='SVG.Compositing.attrib'/>
    <ref name='SVG.Filter.attrib'/>
    <ref name='SVG.Tooltip.attrib'/>
    <ref name='SVG.GraphicalEvents.attrib'/>
    <ref name='SVG.Cursor.attrib'/>
    <ref name='SVG.XLinkEmbed.attrib'/>
    <ref name='SVG.External.attrib'/>
    <ref name='SVG.Media.attrib'/>
    <ref name='SVG.Style.attrib'/>
    <ref name='SVG.BackgroundFill.attrib'/>
    <ref name='SVG.AnimationTimingNoMinMax.
attrib'/>
    <ref name='SVG.AnimationSync.attrib'/>
    <ref name='SVG.Transition.attrib'/>
    <optional>
      <attribute name='x' svg:animatable='true'
svg:inheritable='false'>
        <ref name='Coordinate.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='y' svg:animatable='true'
svg:inheritable='false'>
        <ref name='Coordinate.datatype'/>
      </attribute>
    </optional>
    <attribute name='width' svg:
animatable='true' svg:inheritable='false'>
      <ref name='Length.datatype'/>
    </attribute>
    <attribute name='height' svg:
animatable='true' svg:inheritable='false'>
      <ref name='Length.datatype'/>
    </attribute>
    <optional>
      <attribute name='preserveAspectRatio' a:
defaultValue='xMidYMid meet'
        svg:animatable='yes' svg:
inheritable='false'>
        <ref name='PreserveAspectRatioSpec.
datatype'/>
      </attribute>
    </optional>
    <ref name='SVG.transform.attrib'/>
  </define>
```

The following example illustrates the use of the **video** element. The video content is partially obscured by other graphics elements. Experiments within the SVG working group have shown that adequate performance can be obtained by rendering the video in an offscreen buffer and then transforming and compositing it in the normal way, so that it behaves like any other graphical primitive such as an image or a rectangle. It may be scaled, rotated, skewed, displayed at various sizes, and animated.

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

The DOM interface for the **video** element is shown below:

```
interface SVGVideoElement :
                SVGMediaElement,
                SVGURIReference,
                SVGLangSpace,
                SVGStylable,
                SVGTransformable {
  readonly attribute SVGAnimatedLength x;
  readonly attribute SVGAnimatedLength y;
  readonly attribute SVGAnimatedLength width;
  readonly attribute SVGAnimatedLength height;
  readonly attribute SVGAnimatedPreserveAspectRatio
preserveAspectRatio;

  // a reference to the media stream interface
  readonly attribute SVGVideo video;
};
```

This specification does not mandate support for any particular video format.

12.3 Alternate content based on display resolutions

12.3.1 The multiImage element

The **multiImage** element specifies one or more images which are to be rendered to provide a multiresolution image. Unlike the **image** element, there is no **xlink:href** on the **multiImage** element; instead it has child elements, **subImage** and **subImageRef**, to provide the content. This allows alternate image content at different resolutions to be provided, and selected automatically by the viewer depending on the displayed size of the multiImage and the current zoom level.

The usual SMIL animation features are used to start and stop the entire **multiImage** at the appropriate times.

The DOM interface for the multiImage element is shown below:

```
interface SVGMultiImageElement :
                SVGMediaElement,
                SVGLangSpace,
                SVGStylable,
                SVGTransformable {
  readonly attribute SVGAnimatedLength x;
  readonly attribute SVGAnimatedLength y;
  readonly attribute SVGAnimatedLength width;
  readonly attribute SVGAnimatedLength height;
  readonly attribute SVGAnimatedPreserveAspectRatio
preserveAspectRatio;
};
```

12.3.2 The subImageRef element

The **subImageRef** element is used to provide alternate resources to use depending on rendering conditions. The **subImageRef** element provides an alternate resource reference to be used for a range of rendering scales as defined by **min-pixel-size** and **max-pixel-size**, but is otherwise like an SVG 1.1 **image** element. An **xlink:href** is used to link to the image content. It can point to raster image formats such as PNG and JPEG, to SVG images, or to **symbol** elements in SVG images.

The following example illustrates the use of the **multiImage** element with three **subImageRef** children to link to three JPEG files of the same scene at three different resolutions.

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

```
version="1.2"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="420" height="340" viewBox="0 0 400 280">
  <desc>SVG 1.2 multiImage example with
subImageRef&lt;/desc>
  <multiImage  x="20" y="20" width="240"
height="360">
      <subImageRef xlink:href=".m8.jpg" max-pixel-
size="2"/>
       <subImageRef xlink:href="m4.jpg" min-pixel-
size="2" max-pixel-size="4"/>
       <subImageRef xlink:href="m2.jpg" min-pixel-
size="4" max-pixel-size="8"/>
  </multiImage>
</svg>
```

12.3.3 The subImage element

The **subImage** element is a container element used to provide alternate resources to use depending on rendering conditions. The **subImage** element provides an alternate resource to be used for a range of rendering scales as defined by **min-pixel-size** and **max-pixel-size**, but is otherwise like a **g** element.

The following example illustrates the use of the **multiImage** element with a **subImageRef** for the low resolution version and a **subImage** for the high resolution one, containing a nested **svg** element and four images.

```
<svg version="1.1" width="100%" height="100%"
viewBox="0 0 180 180"
  xmlns="http://www.w3.org/2000/svg" xmlns:
xlink="http://www.w3.org/1999/xlink">
  <multiImage x="0" y="0" width="180" height="180">
    <subImageRef xlink:href="lores.jpg" min-pixel-
size="1"/>
    <subImage max-pixel-size="1">
      <svg width="100%" height="100%" viewBox="0 0
360 360">
        <image x="0" y="0" width="180" height="180"
xlink:href="hr-0.jpg"/>
        <image x="180" y="0" width="180"
height="180" xlink:href="hr-1jpg"/>
        <image x="0" y="180" width="180"
height="180" xlink:href="hr-2.jpg"/>
        <image x="180" y="180" width="180"
height="180" xlink:href="hr-3.jpg"/>
```

```
      </svg>
    </subImage>
  </multiImage>
</svg>
```

12.3.4 Selecting the image for rendering

For **multiImage** elements that have one or more child **subImage** or **subImageRef** elements the viewer has a choice between several possible resources. The choice is made based on the current rendering conditions and the values of the **min-pixel-size** and **max-pixel-size** on the elements.

The **min-pixel-size** and **max-pixel-size** attributes both describe the size of a pixel in the current coordinate system. This may be a single value or a space separated list of two values. If two values are provided then the first refers to the size of a pixel in the horizontal direction and the second value refers to the size of a pixel in the vertical direction in the local coordinate system. If one value is provided it is used for both horizontal and vertical directions. If the attribute is not provided then the resource's range is considered unbounded on that side.

Thus **min-pixel-size** and **max-pixel-size** define a range of resolutions that the resource from the associated **xlink:href** is considered applicable to.

In cases where the current rendering resolution lies outside of any specified range, the viewer should select the resource whose range is closest to the current rendering resolution. Likewise in cases where the current rendering resolution lies within multiple ranges the viewer should the resource whose range is closest to the current rendering. The viewer may use a resource which is not the closest to the current resolution, if the preferred resource is unavailable (not yet dowenloaded, or unreachable, or of an unsupported image format). Thus for example a lower resolution image can be displayed while a higher resolution image is being downloaded.

12.4 Media Properties

The **audio** and **video** elements described above both refer to a set of media attributes. For SVG 1.2 this set includes a single property, **audio-level**.

audio-level

Value:	**<float>**
Initial:	1

Applies to:	**audio**, **video** and container elements
Inherited:	no
Percentages:	N/A
Media:	visual
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 you set 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 a child to override the reduction in audio-level on the parent.

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.

12.5 Loading images

The SVG 1.1 specification does not make it clear 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.

Previous | Top | Next

13 Animation

13.1 The animation element

The **animation** elements specifies an SVG document or an SVG document fragment which provided synchronized animated vector graphics. Like **video**, the **animation** element is a graphical object with size determined by its **x**, **y**, **width** and **height** attributes.

animation Schema

```
  <define name='animation'>
    <element name='animation'>
      <ref name='attlist.animation'/>
      <ref name='SVG.animation.content'/>
    </element>
  </define>

  <define name='SVG.animation.content'>
    <zeroOrMore>
      <ref name='SVG.Description.class'/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name='SVG.Animation.class'/>
        <ref name='SVG.Handler.class'/>
      </choice>
    </zeroOrMore>
  </define>

  <define name='attlist.animation'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <ref name='SVG.Conditional.attrib'/>
    <ref name='SVG.Opacity.attrib'/>
    <ref name='SVG.Graphics.attrib'/>
```

```
    <ref name='SVG.Focusable.attrib'/>
    <ref name='SVG.Clip.attrib'/>
    <ref name='SVG.Mask.attrib'/>
    <ref name='SVG.Filter.attrib'/>
    <ref name='SVG.Compositing.attrib'/>
    <ref name='SVG.GraphicalEvents.attrib'/>
    <ref name='SVG.Tooltip.attrib'/>
    <ref name='SVG.Cursor.attrib'/>
    <ref name='SVG.XLinkEmbed.attrib'/>
    <ref name='SVG.External.attrib'/>
    <ref name='SVG.AnimationTiming.attrib'/>
    <ref name='SVG.Media.attrib'/>
    <ref name='SVG.Filter.attrib'/>
    <ref name='SVG.Compositing.attrib'/>
    <ref name='SVG.GraphicalEvents.attrib'/>
    <ref name='SVG.Tooltip.attrib'/>
    <ref name='SVG.Cursor.attrib'/>
    <ref name='SVG.XLinkEmbed.attrib'/>
    <ref name='SVG.External.attrib'/>
    <ref name='SVG.AnimationTiming.attrib'/>
    <ref name='SVG.Media.attrib'/>
    <ref name='SVG.BackgroundFill.attrib'/>
    <ref name='SVG.AnimationTimingNoMinMax.
attrib'/>
    <ref name='SVG.AnimationSync.attrib'/>
    <optional>
      <attribute name='x' svg:animatable='true'
svg:inheritable='false'>
        <ref name='Coordinate.datatype'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='y' svg:animatable='true'
svg:inheritable='false'>
        <ref name='Coordinate.datatype'/>
      </attribute>
    </optional>
    <attribute name='width' svg:
animatable='true' svg:inheritable='false'>
      <ref name='Length.datatype'/>
    </attribute>
    <attribute name='height' svg:
animatable='true' svg:inheritable='false'>
      <ref name='Length.datatype'/>
    </attribute>
    <optional>
      <attribute name='preserveAspectRatio' a:
defaultValue='xMidYMid meet'
```

```
          svg:animatable='yes' svg:
inheritable='false'>
        <ref name='PreserveAspectRatioSpec.
datatype'/>
      </attribute>
    </optional>
    <ref name='SVG.transform.attrib'/>
  </define>
```

The usual SMIL animation 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.

In the case where the animation element references an external SVG document the same set of rules as used when referencing external SVG images apply.

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** attribute has no meaning. The rules for applying **x**, **y**, **width** and **height** are the same set of rules that apply for the **use** element.

The following example illustrates the use of the **animation** element.

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

  <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" width="100" height="100"
             xlink:href="#movieClip"/>

  <animation begin="2"
             x="100" y="300" width="100" height="100"
             xlink:href="#movieClip"/>

</svg>
```

The DOM interface for the **animation** element is shown below:

```
interface SVGAnimationElement:
  SVGMediaElement,
  SVGURIReference,
  SVGLangSpace,
  SVGStylable,
  SVGTransformable {

readonly attribute SVGAnimatedLength x;
readonly attribute SVGAnimatedLength y;
readonly attribute SVGAnimatedLength width;
readonly attribute SVGAnimatedLength height;
readonly attribute SVGAnimatedPreserveAspectRatio
preserveAspectRatio;

};
```

13.1.1 Transition effects

There exist a number of usage scenarios for transition effects between views of SVG. For example, an image slideshow or multipage SVG documents.

SMIL 2.0 defines syntax for allowing the transition between multimedia elements to include a transition, such as a fadein/fadeout. There is a comprehensive set of transition effects defined by SMPTE and listed in an appendix of SMIL 2.0 Transition Effects

SVG 1.2 adds the transition effects from SMIL 2.0, in particular the **transition** element and the **transIn** and **transOut** attributes.

13.1.1.1 The transition element

The **transition** element defines a single transition class. A **transition** element can be the child of any class, although normally they are defined within a **defs** section.

transition Schema

```
  <define name='transition'>
    <element name='transition'>
      <ref name='attlist.transition'/>
      <ref name='SVG.transition.content'/>
    </element>
  </define>

  <define name='attlist.transition'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <optional>
      <attribute name='dur'/>
    </optional>
    <optional>
      <attribute name='startProgress'>
        <data type='float'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='endProgress'>
        <data type='float'/>
      </attribute>
    </optional>
    <optional>
      <attribute name='direction' a:
defaultValue='forward'>
        <choice>
          <value>forward</value>
          <value>reverse</value>
        </choice>
      </attribute>
    </optional>
    <optional>
      <attribute name='fadeColor' a:
defaultValue='black'/>
    </optional>
```

```
    <!-- refer to the full schema for the
transition type
         attributes. -->
    <ref name='attlist.transition-types'/>
  </define>

  <define name='SVG.transition.content'>
    <zeroOrMore>
      <ref name='SVG.Description.class'/>
    </zeroOrMore>
  </define>

  <define name='SVG.Transition.attrib'
combine='interleave'>
    <ref name='transIn.attrib'/>
    <ref name='transOut.attrib'/>
  </define>

  <define name='transIn.attrib'>
    <optional>
      <attribute name='transIn'/>
    </optional>
  </define>

  <define name='transOut.attrib'>
    <optional>
      <attribute name='transOut'/>
    </optional>
  </define>
```

Attributes:

type
: The type of transition. This attribute is required and must be one of the transition families listed in the SMIL 2.0 Transitions Taxonomy.

subtype
: The subtype of the transition. If specified then the attribute must be a recognized subtype of the transition type. If not specified, then the subtype is the default for the particular transition type.

dur
: The duration of the transition. If specified it must be a clock-value. The default value is "1s".

startProgress
: This is the amount of progress through the transition at which to begin

execution. Legal values are real numbers in the range 0.0-1.0. For instance, we may want to begin a crossfade with the destination image being already 30% faded in. For this case, startProgress would be 0.3. The default value is 0.0.

endProgress

This is the amount of progress through the transition at which to end execution. Legal values are real numbers in the range 0.0-1.0 and the value of this attribute must be greater than or equal to the value of the startProgress attribute. If endProgress is equal to startProgress, then the transition remains at a fixed progress for the duration of the transition. The default value is 1.0.

direction

This specifies the direction the transition will run. The legal values are "forward" and "reverse". The default value is "forward". Note that this does not impact the media being transitioned to, but only affects the geometry of the transition. For instance, if you specified a type of "barWipe" and a subtype of "leftToRight", then the media would be wiped in by a vertical bar moving left to right. However, if you specified direction="reverse", then it would be wiped in by the same vertical bar moving right to left. Another example is the type of "starWipe" and subtype of "fourPoint". For this transition, running the transition forward reveals the destination media on the inside of a four-point star which starts small and gets larger as the transition progresses. Running this transition in reverse would reveal the destination media in the area outside of a large four-point star. The star begins large and gets smaller as the transition progresses. Note that not all transitions will have meaningful reverse interpretations. For instance, a crossfade is not a geometric transition, and therefore has no interpretation of reverse direction. Transitions which do not have a reverse interpretation should ignore the direction attribute and assume the default value of "forward".

fadeColor

If the value of the "type" attribute is "fade" and the value of the "subtype" attribute is "fadeToColor" or "fadeFromColor", then this attribute specifies the starting or ending color of the fade. If the value of the "type" attribute is not "fade", or the value of the "subtype" attribute is not "fadeToColor" or "fadeFromColor", then this attribute is ignored. The allowed values are any color value or a reference to a **solidColor** object. The default value is "black".

None of the attributes on the **transition** element are animatable.

SVG 1.2 does not allow extensions to the transition types specified by SMIL 2.0.

13.1.1.2 The transition attributes

Once a **transition** has been defined in a document, then an transition effect is applied by referencing the **transition** from the **transIn** and **transOut** attributes. The transition attributes are added to the elements that can undergo a transition (listed below).

Transitions specified with a **transIn** attribute will begin at the beginning of the animation's active duration. Transitions specified with a **transOut** attribute will end at the end of the animation's active duration or end at the end of the element's fill state if a non-default fill value is applied.

Note that in SVG, the transition effects only apply to animations modifying the **xlink:href** attribute of graphical elements: **image**, **video**, **use**, **page** and **feImage**.

The value of the attributes is a semicolon-separated list of **transition** element identifiers. The transition used is the first supported transition in the list.

Below is an example of a transition:

```
<svg width="100%" height="100%" version="1.2"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">

  <defs>
    <transition id="fadeToRed2s" type="fade"
subtype="fadeToColor"
                fadeColor="red" dur="2s"/>
    <transition id="fadeFromRed1s" type="fade"
subtype="fadeFromColor"
                fadeColor="red" dur="1s"/>
  </defs>

  <image x="0" y="0" width="100" height="100"
         transIn="fadeFromRed1s"
         transOut="fadeToRed2s">

    <animate attributeType="XML"
             attributeName="xlink:href"
             begin="0s"
             dur="60s"
             values="1.jpg;2.jpg;3.jpg;4.jpg"
             fill="freeze"/>
```

```
  </image>

</svg>
```

13.2 Media elements

SVG supports media elements similar to the SMIL Media Elements. Media elementsdefine 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:

- **video**
- **audio**
- **animation**

13.3 Time containers

SVG supports a subset of SMIL 2.0's feature set for nested time containers. All time containers support all or some of the SVG Timing Attributes. The following elements in the SVG language define a new time container:

- **svg**: is a parallel time container. All descendant elements represent time-based elements with a default timeAction of 'display'.
- **pageSet**: is a sequential time container. Its **page** children elements represent time-based elements with a default timeAction of 'display'.
- **page**: is a parallel time container. All descendant elements represent time-based elements with a default timeAction of 'display'.

Time containers and timeAction are defined by the SMIL2.0 spec.

13.4 Attributes for run-time synchronization

SVG supports the following attributes on time containers and media elements from SMIL 2.0 to control run-time synchronization of different time containers:

- 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 2.0 specification.

13.5 Time Manipulation

SVG 1.2 adds the **speed** attribute from the SMIL 2.0 Timing Manipulations Module.

The **speed** attribute controls the local playback speed of an element, to speed up or slow down the effective rate of play relative to the parent time container. The **speed** attribute is supported on all timed elements. The argument value does not specify an absolute play speed, but rather is relative to the playback speed of the parent time container. The specified value cascades to all time descendants. Thus if a **par** and one of its children both specify a speed of 50%, the child will play at 25% of normal playback speed.

Values less than 0 are allowed, and cause the element to play backwards. An element can only play backwards if there is sufficient information about the simple and active durations. Specifically:

- The element simple duration must be resolved and may not be indefinite.
- The element active duration must be resolved and not indefinite, OR the element active duration must be constrained by a resolved simple duration for its associated time container. There must be a means of defining active time running backwards.

If the cascaded speed value for the element is negative and if either of the above two conditions is not met, the element will begin and immediately end (i.e. it will behave as though it had a specified active duration of 0).

13.6 Enhanced ElementTimeControl interface

In SVG 1.2, a document may contain multiple time containers, such as multiple pages or **video** elements. In order to pause a particular container or timed element, the ElementTimeControl interface has been extended.

```
interface ElementTimeControl {
   void            beginElement();
   void            beginElementAt(in float offset);
   void            endElement();
   void            endElementAt(in float offset);
```

```
    void            pauseElement();
    void            unpauseElement();
    readonly boolean elementPaused;
}
```

The ElementTimeControl interface is added to all media elements and all time containers.

13.7 Triggering animations using key events

SVG 1.2 updates the definition of the accessKey syntax used to control the animation begin or end.

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.

Previous | Top | Next

14 Extended links

SVG 1.0 and 1.1 use XLink simple links; a link anchor can only link to a single link target. It is often desirable to have a link anchor point to multiple targets. XLink extended links are the clear design choice to add an extended link capability. Extended links contain multiple locators. In SVG 1.2, activating an extended link causes a menu to be displayed; the text of each menu item is taken from the title on each locator.

xa Schema

```
<define name='xa'>
  <element name='xa'>
    <ref name='attlist.xa'/>
    <ref name='SVG.xa.content'/>
  </element>
</define>

<define name='attlist.xa' combine='interleave'>
  <ref name='SVG.Core.attrib'/>
  <ref name='SVG.Conditional.attrib'/>
  <ref name='SVG.Style.attrib'/>
  <ref name='SVG.Presentation.attrib'/>
  <ref name='SVG.GraphicalEvents.attrib'/>
  <ref name='SVG.External.attrib'/>
  <ref name='SVG.transform.attrib'/>
  <ref name='SVG.transform-host.attrib'/>
  <optional>
    <attribute name='target'>
      <ref name='LinkTarget.datatype'/>
    </attribute>
  </optional>
</define>

<define name='SVG.xa.content'>
  <zeroOrMore>
    <ref name='SVG.Description.class'/>
```

```
    </zeroOrMore>
    <zeroOrMore>
      <ref name='loc'/>
    </zeroOrMore>
    <zeroOrMore>
      <choice>
        <ref name='SVG.Animation.class'/>
        <ref name='SVG.Structure.class'/>
        <ref name='SVG.Conditional.class'/>
        <ref name='SVG.Image.class'/>
        <ref name='SVG.Style.class'/>
        <ref name='SVG.Shape.class'/>
        <ref name='SVG.Text.class'/>
        <ref name='SVG.Marker.class'/>
        <ref name='SVG.Profile.class'/>
        <ref name='SVG.Gradient.class'/>
        <ref name='SVG.Pattern.class'/>
        <ref name='SVG.Clip.class'/>
        <ref name='SVG.Mask.class'/>
        <ref name='SVG.Filter.class'/>
        <ref name='SVG.Cursor.class'/>
        <ref name='SVG.Hyperlink.class'/>
        <ref name='SVG.View.class'/>
        <ref name='SVG.Script.class'/>
        <ref name='SVG.Font.class'/>
      </choice>
    </zeroOrMore>
  </define>
```

The extended-link equivalent to the **a** element is called **xa** and the locator elements are called **loc**. Here is a simple example using **xlink:title** attributes.

```
<xa>
  <loc xlink:href="http://example.com/A.svg"
    xlink:title="Schematic diagram"/>
  <loc xlink:href="http://example.org/B.svg"
    xlink:title="Parts list"/>
  <!-- content of the link goes here -->
  <path d="M50,50L100,20L30,100z"/>
</xa>
```

The content model of **xa** is one or more **loc** elements, followed by the same content that an **a** element can have. For improved internationalization, **title** children are allowed as children of the **loc** element as well as **xlink:title** attributes on the **loc** element itself. In addition, a **switch** element may be used to give multilingual titles. Here is the same example, but using **title** elements.

```
<xa>
  <loc xlink:href="http://example.com/A.svg">
    <title>Schematic diagram</title>
  </loc>
  <loc xlink:href="http://example.org/B.svg">
    <title>Parts list</title>
  </loc>
  <!-- content of the link goes here -->
  <path d="M50,50L100,20L30,100z"/>
</xa>
```

The behavior of an extended link depends on the number of **loc** children whose display property has a value other than 'none':

0
: No action, link is disabled

1
: The link is traversed to the single location

2 or more
: A menu is constructed for each displayed **loc** element,using the text from the title element s. The size of the menu is such that the longest string is not clipped. Once one of the menu items is chosen,the menu disappears, and the selected locator is traversed.

Here is an example with multilingual menu items - Japanese, Russian, French, and an English fallback

```
<xa>
  <loc xlink:href="http://example.com/A.svg">
    <switch>
      <title systemLanguage="jp">図面</title>
      <title systemLanguage="ru">Диаграмма</title>
      <title systemLanguage="fr">Diagramme schématise</title>
      <title>Schematic diagram</title>
    </switch>
  </loc>
  <loc xlink:href="http://example.org/B.svg">
    <switch>
      <title systemLanguage="jp">部品一覧</title>
      <title systemLanguage="ru">Список частей</title>
      <title systemLanguage="fr">Liste des pièces</title>
      <title>Parts list</title>
    </switch>
  </loc>
```

```
    <!-- content of the link goes here -->
    <path d="M50,50L100,20L30,100z"/>
  </xa>
```

Previous | Top | Next

15 Application development

15.1 Element focus and navigation

15.1.1 The focusable property

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.

focusable

Value:	"true" | "false" | "auto"
Initial:	"auto"
Applies to:	container elements and graphics elements
Inherited:	no
Percentages:	N/A
Media:	visual
Animatable:	yes

The **focusable** property determines 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 value "true" means that the element is keyboard-aware and should be treated as any other UI component that can get focus. The value "false" means that it should not. The value "auto" is equivalent to "false", except that it acts like "true" for the following cases:

- the **a** element
- the **text** and **flowDiv** elements with **editable** set to "true"

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.

Custom (i.e., non-SVG) elements can also be focusable, including custom elements with sXBL bindings. (See XML Binding Language for SVG.)

> **NOTE:**
>
> The ability to specify a focusable property for a custom element probably will require a stylesheet because custom elements do not support SVG's presentation atttributes.

15.1.2 Navigation

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").

In the author wishes to change the specified field navigation order (and, wherever not specified, the default tab order), they must catch the input event related to the navigation and then cancel the event.

Navigation order is determined using the **nav-index**, **nav-left**, **nav-right**, **nav-up**, and **nav-down** properties from the CSS 3 Basic User Interface Module. In most cases, the value 'auto' will cause the user agent to use document order for navigation order. This property only applies to focusable elements.

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, then it is possible to specify focus navigation order such that the tab key takes the user from r1 to c1 to g1.

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 symbol contains a focusable element, and that symbol is referenced by two use elements, then the document will have two separate focusable fields, not just one.)
- If an image 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.
- Focusable elements within shadow trees created by sXBL bindings (see XML Binding Language for SVG) are included in the set of focusable fields that participate in field navigation operations. For more on focusable elements within shadow trees, refer to the Focus and navigation section on sXBL

The one difference with referenced content and shadow trees has to do with the handling of nav-index. For all of the above cases, **nav-index** includes a hierarchical aspect as defined in the sXBL specification (see sXBL 'nav-index' rules).

15.1.3 Obtaining focus

When the user agent gives an element focus it receives a {"http://www.w3.org/2001/xml-events", "DOMFocusIn"} event.

The SVGElement and SVGElementInstance interfaces have a focus() method that, when called, requests that the User Agent give focus to the particular element. Calling focus() on an element that is not focusable causes focus to go to the nearest focusable ancestor element. If there is no focusable ancestor element, then focus goes to the document. If focus() is called on an SVGElementInstance that corresponds to a given use element, then that SVGElementInstance can get focus only if its corresponding element is focusable; otherwise, focus goes to the nearest focusable ancestor element (perhaps spanning from the shadow instance tree into the main document) or, if there are no focusable ancestors, to the document.

The SVGDocument interface have next() and previous() methods which move the focus onto the respectively next or previous focusable element.

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.

15.2 Tooltips

The SVG 1.1 language does not specify a method for the declarative display of tooltips on SVG content. While it suggests that the content of the **title** element could be displayed as a tooltip, it does not provide any control for the content developer. This meant that developers resorted to implementing their own tooltip functionality using scripting.

SVG 1.2 adds declarative tooltip support through the **tooltip** property. This allows for the display of textual tooltips to be declaratively controlled for each element in the SVG document. User agents should use the platform tooltip system if there is one available. On platforms without native support for tooltips, user agents can implement their own tooltip system. This means that the appearance, position and behavior of the tooltip is implementation or platform specific.

However, in the visual environment, the user agent must attempt to display the entire tooltip in a readable manner, avoiding clipping by the edge of the device or canvas. If possible the tooltip should be implemented as a temporary graphic that is superimposed on the top of the SVG canvas at the location of the pointer if one exists. Also, if possible, the content of the tooltips should be made available to accessibility implementations on the device.

15.2.1 The hint element

The **hint** element is another level of metadata for an element. The contents of the **hint** element are intended to be displayed by a tooltip. It differs from the **title** and **desc** elements in that a **hint** may give instructions to the user, as opposed to describe the element that it belongs to.

hint Schema

```
  <define name="hint">
    <element name="hint">
      <ref name="attlist.hint"/>
      <ref name="SVG.hint.content"/>
    </element>
  </define>
  <define name="attlist.hint"
combine="interleave">
    <ref name="SVG.Core.attrib"/>
  </define>
  <define name="SVG.hint.content">
    <choice>
      <ref name="SVG.XLinkEmbed.attrib"/>
      <text/>
    </choice>
  </define>
```

Below is an example of the **hint** element.

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

  <circle cx="50" cy="50" r="25" fill="red"
tooltip="enable">
    <title>The target area</title>
    <hint>Click here to start the animation</hint>
    <animate .... />
  </circle>

</svg>
```

15.2.2 The tooltip property

The tooltip property specifies whether or not tooltips should be displayed for this element.

tooltip

Value:	enable | disable | inherit
Initial:	enable
Applies to:	graphics and container elements
Inherited:	yes
Percentages:	N/A

Media: visual
Animatable: yes

If the **tooltip** property is set to "enable" then the content of the tooltip is the text content of the **hint** child of an element. For elements that do not have a **hint** child, the tooltip content is the content of its parent's tooltip. If the **tooltip** property is set to "disable" then the content of the tooltip is empty. The value of the tooltip for a root **svg** element without a **hint** child is empty. In the case of an empty value, the tooltip will not be displayed.

Tooltips and the CSS **hover** property both track the mouseover and mouseout events. Tooltips may require the pointer to be stationary over the target for a short period of time. Tooltips and **hover** represent additive effects. The **hover** processing should occur first followed by tooltip processing. That is, tooltip processing is placed after **hover** processing and before hyperlink processing. Tooltips respond to pointer events in the same manner as **hover**, thus do not activate on elements with either the **display** or **pointer-events** properties set to "none".

Previous | Top | Next

16 Events and Scripting

16.1 The script element

A **script** element is a place for 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). The SVG specification defines the behavior for two specific script types:

text/ecmascript

This type of executable content must be source code for the ECMAScript programming language (Ed 3 compact profile ES 327). This code is executed in the context of this element's owner document's global scope as explained above.

Also, the script global object can implement the EventListenerInitializer2 interface (described below) 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 **scriptContentType** of "text/ecmascript".

SVG implementations that load external resources through protocols (such as HTTP) that support transfer encoding, must accept external script files with gzip compression ("Transfer-Encoding: gzip" for HTTP).

application/java-archive

This type of executable content must be an external resource that contains a Java Jar archive. 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 attempt to cast it to the EventListenerInitializer2 interface (defined below). If that fails, it should cast it to the deprecated EventListenerInitializer interface (for SVG 1.1 compatibility). Then the initializeEventListeners method should be called (both interfaces has this method) with the script element itself (for EventListenerInitializer2) or its owner document (for EventListenerInitializer) as a parameter. If a class listed in SVG-Handler-Class implements neither EventListenerInitializer nor EventListenerInitializer2, it is an error.

Note that the user agent can 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.

Other language bindings are encouraged to adopt one of the two strategies described above.

16.2 The handler element

The **handler** element is a convenient way to attach and detach event listeners to XML elements. It is based on XML Events.

SVG 1.2 makes the following modifications to XML Events:

- Namespaced events: As SVG 1.2 supports DOM Level 3 Events, which are namespaced, XML Events in SVG 1.2 much also allow namespaced events. SVG 1.2 follows the rules defined in DOM Level 3 (namespaces are specified as "namespace-prefix:event-name").
- URIREFs 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 URIREFs for those attributes, with the following restriction: Only documents that are declaratively referenced as part of the current document, via the use, image and feImage elements, can be referred to. Referring to any other arbitrary external document is an error.

handler Schema

```
  <define name='handler'>
    <element name='handler'>
      <ref name='attlist.handler'/>
      <ref name='SVG.handler.content'/>
    </element>
  </define>

  <define name='attlist.handler'
combine='interleave'>
    <ref name='SVG.Core.attrib'/>
    <attribute name='ev:event' svg:
animatable='false' svg:inheritable='false'>
      <choice>
        <data type='NMTOKEN'/>
        <data type='QName'/>
      </choice>
    </attribute>
    <attribute name='type' svg:
animatable='false' svg:inheritable='false'>
      <ref name='ContentType.datatype'/>
    </attribute>
  </define>

  <define name='SVG.handler.content'>
    <choice>
```

```
      <group>
        <ref name='SVG.XLinkRequired.attrib'/>
      </group>
      <text/>
    </choice>
  </define>
```

scriptContentType

Specifies the scripting language that will be used to create a listener (see the above definition in **script** for possible values of this attribute). This attribute is mutally exclusive with **xlink:href**.

xlink:href

Specifies the script **element** that will be used to create a listener. This attribute is mutually exclusive with **scriptContentType**.

During the document's loading time (or when a **handler** element is inserted in the tree after the document has been loaded), the user agent creates an event listener object for each **handler** element. This is done by locating an appropriate EventListenerInitializer2 object and calling the createEventListener method on it. If the xlink:href attribute is used it must point to a local **script** element. In this case scope object associated with that **script** element is used. If the **scriptContentType** attribute is used then the document's global scope for that script type is used. It is an error to specify script types that inherently don't have global scope objects (e.g. "application/java-archive"). If neither **xlink:href** nor **scriptContentType** are specified, the document's value for **scriptContentType** is used. For the case of ECMAScript, if createEventListener function is not found in global scope, the following pseudo-code is used:

```
function createEventListener( node )
{
 var body = getTextData(node); // concatenated text from all
text, cdata and entity reference children
 return document.global.eval("function(evt){" + body + "}")
}
```

Other scripting lanuage bindings are encouraged to provide similar default behavior.

Once a listener is created it is attached to appropriate event targets according to XMLEvents specification. If the **handler** element is removed, the corresponding

event listener is removed from its targets. If the **handler** element is inserted into the tree again, the same event listener is reused. If the **handler** element's attributes or children are modified, the corresponding listener is removed from its targets, and a new listener is created and added back to the targets.

> **NOTE:**
>
> In the Java environment one would typically place data that is needed to create an event listener either as **handler** element's inline character data, as a custom attribute (say my:name attribute) or as custom markup inside the **handler** element. createEventListener then can read this information since the **handler** element is passed to it as a parameter.

16.3 DOM Interfaces

The EventListenerInitializer2 parameters are expressed purely in Core W3C DOM types. This is done so that this definition is reusable for other mark-up languages.

```
interface EventListenerInitializer
{
  void initializeEventListeners( in SVGDocument doc);
}

interface EventListenerInitializer2
{
  void initializeEventListeners( in dom::Element
scriptElement );
  events::EventListener createEventListener( in dom::Element
handlerElement );
}
```

16.4 Processing order for user interface event

The following is a replacement for section 16.5 in SVG 1.1.

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 (e.g., by invoking the preventDefault() DOM method), then the event is passed on for:

- Processing of any relevant dynamic pseudo-classes (i.e., :hover, :active and :focus), after which 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 **image** element, when the **image** element references an SVG file
- Custom elements with an sXBL binding

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.

Previous | Top | Next

17 Non-graphical enhancements

17.1 Externally referenced documents

For the following 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:image** 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; stylesheets are applied and the CSS cascade is run; timelines are initiated and animations execute; sXBL transformations are applied; etc. For example, if document A.svg includes an **svg:image** 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 URI 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 URLs for resource documentsit 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 documents are 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. Thus, if a primary document references both a **gradient** element and a filter effect from the resource document, the resource document is loaded only once.

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; timelines are initiated and animations execute; sXBL transformations are applied; etc.

Because of HTTP redirects, the same source document might be retrieved from multiple different source URIs. The rule for SVG is that documents are considered to be unique based on string comparisons of the full URI after resolving relative URIs into absolute URIs and after taking into account HTTP redirects (i.e., use the post-redirect URI instead of the original source URI).

17.2 Referencing external titles, descriptions and metadata

SVG 1.2 adds the **xlink:href** attribute to the **title**, **desc** and **metadata** elements, allowing them to reference of external content. In each case the URI must point to data that can be included as content of the referencing element.

17.3 Adding Copyright information to an SVG document

SVG encourages the use of a common metadata format for inclusion of copyright information. Metadata relevant to the data copyright of the entire document should be added to **metadata** element of the topmost **svg** element. This allows the author to unambiguously state the licensing terms for the entire document. The scheme may also be used elsewhere in the document, for pieces that have different licensing. For example, an SVG font may have specific licensing details expressed in its own **metadata** element.

Note that inclusion of this metadata does not provide the author with a method in which to protect or enforce their copyright, it simply bundles the copyright information with the content in a defined manner. Providing methods, technical or non-technical, for data protection is currently beyond the scope of the SVG specification. For more information on protecting XML content see XML Encryption.

The suggested vocabulary for Copyright information is the Creative Commons Metadata Set. This does not exclude the use of other metadata schemes.

Below is an example of providing a copyright license that allows public distribution and reproduction with attribution but disallows commercial use:

```
<svg>
  <metadata>
    <rdf:RDF xmlns:cc="http://web.resource.org/cc/"
             xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#">
      <cc:work rdf:about="">
        <cc:license>
           <cc:permits rdf:resource="http://web.resource.org/cc/
Reproduction"/>
```

```
            <cc:permits rdf:resource="http://web.resource.org/cc/
Distribution"/>
            <cc:requires rdf:resource="http://web.resource.org/cc/
Attribution"/>
            <cc:prohibits rdf:resource="http://web.resource.org/
cc/CommercialUse"/>
          </cc:license>
        </cc:work>
      </rdf:RDF>
    </metadata>

    <!-- graphics go here -->

</svg>
```

Below is an example which prohibits reproduction and distribution.

```
<svg>
  <metadata>
    <rdf:RDF xmlns:cc="http://web.resource.org/cc/"
             xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#">
      <cc:work rdf:about="">
        <cc:license>
          <cc:prohibits rdf:resource="http://web.resource.org/
cc/Reproduction"/>
          <cc:prohibits rdf:resource="http://web.resource.org/
cc/Distribution"/>
        </cc:license>
      </cc:work>
    </rdf:RDF>
  </metadata>

  <!-- graphics go here -->

</svg>
```

Below is an example which refers to an external license:

```
<svg>
  <metadata>
    <rdf:RDF xmlns:cc="http://web.resource.org/cc/"
             xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#">
      <cc:work rdf:about="">
        <cc:license rdf:resource="http://www.example.org/
copyright"/>
      </cc:work>
    </rdf:RDF>
  </metadata>

  <!-- graphics go here -->
```

```
</svg>
```

Like all metadata processing in SVG, there is no requirement for the SVG User Agent to take any action when it finds copyright information. However, it is recommended that in the cases where a User Agent does understand the Copyright information, then it should use that information. For example, it may display the copyright for source code in a message to the user before it displays the source.

17.4 Specifying a snapshot time

SVG 1.2 adds the **snapshotTime** attribute to **svg** elements. The attribute only has an effect on the root **svg** element.

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. Animatable: no

Below is an example of an svg file with a snapShotTime specified. A User Agent displaying a number of svg files in a file directory by rendering a thumbnail image of each file can use the snapShotTime as the time-to-render when generating the image. The look of the thumbnail for a UA honoring the snapShotTime and for a UA not honoring it is showed below the example.

```
<svg version="1.2" snapShotTime="3" baseProfile="tiny"
     xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
     viewBox="0 0 400 300">

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

User agent with snapShotTime support.

User agent without snapShotTime support.

Previous | Top | Next

Appendix A: DOM enhancements

A.1 DOM Level 3 Support

The SVG 1.2 DOM builds upon and is compatible with the Document Object Model (DOM) Level 3 Specification. In particular:

- The SVG 1.2 DOM requires complete support for DOM Level 3 Core.
- The SVG 1.2 DOM requires support for relevant aspects of the DOM Level 3 Events.
- The SVG 1.2 DOM requires complete support for DOM Level 3 XPath.

Other than those areas listed above, the SVG 1.2 DOM follows the same requirements as the SVG 1.1 DOM.

A.2 Media Interfaces

With the addition of **audio** and **video** to SVG 1.2, there is a expanded set of interfaces for media elements. Each of the interfaces for the **audio**, **video**, **image** and **feImage** elements have an attribute for accessing the media object.

There is an oversight in the SVG 1.0 DOM in that the SVGImageElement interface does not allow access to the DOM of the image it refers to (if that image is an SVG document).

To provide this functionality, SVG 1.2 adds a document attribute to the SVGMedia interface, which is the Document interface of the referenced meda, if one is available. It is unlikely that the SVG specification will describe the format for the returned Document except in the case of a referenced SVG image.

Many media streams have embedded metadata. For example, most digital cameras export images in the EXIF format that embeds information on the photograph, such as the presence of a flash, the date the picture was taken and

the model of camera used. SVG 1.2 provides an interface to the referenced media metadata, if the user agent is able to access and process it.

While it would have been consistent to provide an XML-like DOM interface to the referenced media's metadata, this is difficult due to the vast range of metadata schemes across the large number of media formats. Instead, SVG 1.2 provides attribute access to the most common metadata, such as the width and height of visual media, and the duration of timed media. For other properties, there is method for string-based lookup.

The SVG Media interfaces are described below:

```
interface SVGMedia {

    readonly dom::Document document;

    // we can have a simple metadata access this way:
    DOMString getMetadata( in DOMString name );
};


interface SVGVisualMedia : SVGMedia {

    // width and height in pixels (or natural width and
height for vector fomats)
    readonly unsigned float width;
    readonly unsigned float height;

};

interface SVGImage : SVGVisualMedia {

    // Defined for binary (pixel-based) images only
    unsigned SVGColor getPixel( in unsigned long x, in
unsigned long y );

};


interface SVGVideo : SVGVisualMedia {

    readonly unsigned float duration;

};
```

```
interface SVGAudibleMedia : SVGMedia {

    readonly unsigned float duration;

};

interface SVGAudio : SVGAudibleMedia {

    // empty

};
```

Note that the getPixel() method on SVGImage may return invalid data in the cases where the image has not yet been loaded.

The above interfaces are accessed from each of the **image**, **feImage**, **video** and **audio** interfaces as shown below:

```
interface SVGImageElement {
    ....
    readonly SVGImage image;
    ....
};

interface SVGFeImageElement {
    ....
    readonly SVGImage image;
    ....
};

interface SVGVideoElement {
    ....
    readonly SVGVideo video;
    ....
};

interface SVGAudiolement {
    ....
    readonly SVGAudio audio;
    ....
};
```

A.3 Conversion of Coordinates

While it is possible to convert client space coordinates, such as from a pointer

event, to user coordinates using the SVG DOM, SVG 1.2 adds a more convenient way to process the conversion.

Below is the new method to be added to the SVGLocatable interface.

```
interface SVGLocatable {
   // ...

   // converts (clientXArg, clientYArg) coordinates into the
   // current element's user space
   SVGPoint convertClientXY(in long clientXArg, in long
clientYArg)
}
```

A.4 Filtering DOM Events

When an event listener is attached to an element, that listener receives all matching events dispatched by the element. In many cases the listener only requires events that match certain criteria (e.g. a particular attribute name for DOM mutation events or only events in the bubbling phase).

The SVGEventFilter interface is an event listener and event target that filters some events, allowing the script or user agent to only process what is necessary.

```
interface SVGEventFilter : events::EventTarget, events::
EventListener
{
  // value for phase and button that prevents filtering
  const unsigned short DONT_FILTER = 0xFFFF;

  attribute unsigned short phase;
  attribute EventTarget target; // null does not filter out
anything
  attribute unsigned short button; // for mouse events

  // for mouse motion events, false does not filter out
anything
  attribute boolean dragOnly;

  // for mutation events, null does not filter out anything
  attribute DOMString attrLocalName;

  // for mutation events, active only when attrLocalName is
non-null
```

```
    attribute DOMString attrNamespace;
};
```

SVGEventFilters are created using the SVGGlobal interface.

The default state of the created filter does not filter any events. The user must set the properties in order to filter.

SVGEventFilter is an event listener and is also an event target, so that other event listeners can be registered with it. It matches the event it receives against the set of criteria that are expressed as properties and either passes that event object to all of its own event listeners if it finds a match or ignores the event if it does not find a match.

A.5 Modification to getScreenCTM

SVG 1.2 updates the definition of the getScreenCTM() method.

getScreenCTM()

> Returns the transformation matrix from current user units (i.e., after application of the transform attribute, if any) to the screen coordinate system - the coordinate system used by screenX/Y of the MouseEvent interface.

A.6 Accessing the client CTM

SVG 1.2 adds the method getClientCTM() to the SVGLocatable interface.

getClientCTM()

> Returns the transformation matrix from current user units (i.e., after application of the transform attribute, if any) to the client coordinate system - the coordinate system used by clientX/Y of the MouseEvent interface.

A.7 Wheel event

Many devices today have a rotational input method, such as the wheel on a mouse or the "jog dial" of a phone or PDA.

The "wheel" event is triggered when the user rotates the rotational input device. This event may only be registered on the root-most **svg** element.

```
interface WheelEvent : UIEvent {
 readonly attribute int wheelDelta;
}
```

wheelDelta

Indicates the number of "clicks" the wheel has been rotated. A positive value indicates that the wheel has been rotated away from the user (or in a right-hand manner on horizontally aligned devices) and a negative value indicates that the wheel has been rotated towards the user (or in a left-hand manner on horizontally aligned devices).

A "click" is defined to be a unit of rotation. On some devices this is a finite physical step. On devices with smooth rotation, a "click" becomes the smallest measurable amount of rotation.

A.8 Obtaining the bounds of rendered content

In SVG 1.2, the SVGLocatable interface is extended with two new methods:

```
interface SVGLocatable {

  ...

  SVGRect getResultBBox();
  SVGRect getRegionBBox();
  ...

};
```

getResultBBox()

Returns the bounding box of the object, taking into account the geometry and geometry-based drawing operations (stroke, clip, markers, vectoreffects, markers with stroke)

getRegionBBox()

Returns the bounding box of the object, taking into account all geometry

and rendering operations (including filters and masks using regions).

A.9 Notification of shape modification

A.9.1 The ShapeChange event

The svg:ShapeChange event is fired when the raw geometry of an object is changed. Only basic shapes and paths fire this event. This does not take into account markers, masks, clipping or filters but focuses only on core geometry attributes of a given element. However, animations do trigger shape change events. The event target is the element (basic shapes or path) that has had its shape changed.

```
interface svg:ShapeChange : Event { }
```

A.9.2 The RenderedBBoxChange event

The svg:RenderedBBoxChange event is fired when the bounding box of an element's rendered output is modified. This takes into account all SVG features that impact on the rendering of an object: clipping, masking, vector-effects, stroking, filling, filters, markers, the **display** property and animation. The target of the event is the element on which the event listener has been registered.

```
interface svg:RenderedBBoxChange : Event {
 readonly attribute SVGRect boundingBox;
};
```

boundingBox

The new bounding box of rendered content for the target.

Previous | Top | Next

Appendix B: API enhancements

B.1 SVGTimer Interface

The SVGTimer interface provides an API for scheduling execution.

```
interface SVGTimer : events::EventTarget // emits "timer"
event
{
        // initial delay before the first execution is fired
        // 0 means fire as soon as possible
        attribute long delay;

        // period between each execution, if -1, no
additional
        // execution will be fire
        attribute long interval;

        // the current state of the timer
        readonly attribute boolean running;

        // start the timer
        void start()

        // stop the timer
        void stop()

};
```

An SVGTimer object is always either in the running (attribute running is true) or waiting (attribute running is false) state.

delay

This property specifies the time delay in milliseconds. After delay period has passed while the object is in the "running" state, the object will fire a

"timer" event. In the "running" state this attribute is dynamically updated to reflect the remaining delay. It also can be dynamically assigned. If this attribute is 0, it means that event will fire as soon as possible. Assigning negative value is equivalent to calling the stop() method.

interval

This property contains the time interval in milliseconds. If the timer is in the "running" state, then it will fire regularly on the interval.

running

Timer state. Value is true if the timer is running (and will fire in delay milliseconds), false if the timer is waiting. Note that the interval and delay properties can be non-negative if the timer is stopped (but if delay is negative, the timer is stopped).

start()

Change the timer state into "running". If the timer is already in the running state, it has no effect.

stop()

Change the timer state into "waiting". If the timer is already in the waiting state, it has no effect.

Since SVGTimer is an event target, event listeners can be registered on it using regular addEventListener call using "timer" as the event type. Event listeners can access their corresponding SVGTimer object through event object's target property.

Timer events are never fired while some other DOM event is being processed (and are delayed instead). Also, timer events are fired only when the timer is in the "running" state.

SVGTimer instances are created using the SVGGlobal interface.

B.2 Network interfaces

There are two interfaces for sending and retrieving data over the network. The

URLRequest interface is a high level API for communications that can be defined by a URI. The Socket interface is a low level API for socket-level communications.

B.2.1 URLRequest interface

The URLRequest interface enables a client to retrieve data that is addressable by a URI.

```
interface URLRequest : events::EventTarget // emits
"URLResponse" event
with target set to the request
{
  const unsigned short INITIAL   = 0; // after it is created
                                      // or after init is
called
  const unsigned short PENDING   = 1; // after submit is
called
  const unsigned short COMPLETED = 2; // after URLResponse
event comes
  const unsigned short ERROR     = 3; // on lower-level
protocol error
                                      // (e.g. connection
refused)

  // initialize request, not allowed in PENDING state,
  // sets the state to INITIAL
  void init(  in DOMString method, in DOMString uri );

  readonly attribute unsigned short requestState;

  // body of the request, can only
  // be set in INITIAL state
  // Setting causes file content to be removed.
  attribute DOMString requestText;

  void addRequestHeader( in DOMString header, in DOMString
value );
                                    // only in INITIAL state

  // body of the response, only in COMPLETED state
  readonly attribute DOMString responseText;

  // 3-digit status code, only in COMPLETED state
  readonly attribute unsigned long status;
```

```
    // status text, only in COMPLETED state
    readonly attribute DOMString statusText;

    // response header count, only in COMPLETED state
    readonly attribute unsigned long responseHeaderCount;

    // response header iterator, only in COMPLETED state
    NameValuePair getResponseHeader( in unsigned long index )
       raises (DOMException);

    // abort the request, only in PENDING state,
    // sets the state to ERROR
    void abort() raises (DOMException);

    // submit the request, sets the state to PENDING
    void submit() raises (DOMException);

    // add the content from the given file object
    // calling this method sets requestText to become null
    // Only valid in INITIAL state
    void addContentFromFile( in SVGFile file ) raises
  (DOMException);

    // clone this request
    URLRequest cloneRequest();

  };
```

An URLRequest object can be in one of 5 states: INITIAL, PENDING, COMPLETED, ERROR or ABORTED. When an URLRequest object is created its state is set to INITIAL.

The following methods/attributes are present on URLRequest:

requestState

The URLRequest object state.

init()

Initialize this URLRequest object. The "uri" parameter is the URI of the resource to access. The "method" parameter is an access method for the resource. For the HTTP protocol the method names are "GET", "POST", "HEAD" and "PUT" (case insensitive). If this method is called when the object state is PENDING, then an SVG Exception is raised.

requestText

This property is the body of the request. Some URLs and/or access methods don't allow a request body, in which case this property will be ignored. The property is only valid in the INITIAL state.

addRequestHeader()

Adds a header to the request. Some URLs and/or access methods don't allow certain headers; Any header that isn't allowed is ignored. This method is only valid in the INITIAL state.

addContentFromFile()

Add file content to the request body. This method is only valid in the INITIAL state.

submit()

Submit this request for processing. This method is only valid in the INITIAL state, and changes the state to PENDING. When the request is processed the status becomes one of:

- COMPLETED: if system was able to process the request
- ERROR: if the request caused a protocol error
- ABORTED: if the request was aborted before being processed

Once the status is changed from PENDING to either COMPLETED or ERROR, the URLResponse event is fired. The request status never changes while a DOM event is being processed. Only errors that are outside of the domain of URL protocol should cause state to be changed to ERROR; for instance HTTP status 500 should be passed through as COMPLETED (with status field set to 500). Redirection on the protocol level should happen automatically.

responseText

The body of the response to the request. If the response cannot be represented as text, reading this property will raise an DOMException. This property only can be read in the COMPLETED state. Reading the property with any other state raises an DOMException.

status

The protocol-level status code. For HTTP, this is the HTTP-defined three-digit status. This property only can be read in the COMPLETED or ERROR state. Reading the property with any other state raises an DOMException.

statusText

The protocol-level status text. This property only can be read in the COMPLETED state. Reading the property with any other state raises an DOMException.

responseHeaderCount

The number of headers in the reply. This property only can be read in the COMPLETED state. Reading the property with any other state raises an DOMException.

getResponseHeader()

Access the reply headers by index. This method can only be used in the COMPLETED state. Using the method with any other state raises an DOMException. A DOMException is raised if the index is out of bounds.

abort()

Abandon the request and change the state to ABORTED. The URLResponse event is not fired (unless the request is reinitialized and resubmitted). Note that the actual network request might have been already processed by the server; there is no way to reliably cancel network requests once they have been submitted. This method can only be used in the PENDING state. Using the method with any other state raises an DOMException.

cloneRequest()

Create an exact duplicate of this URLRequest object, with the exception of event listeners, which are not cloned. This method can only be used in the INITIAL state. Using the method with any other state raises an DOMException.

The headers used by a URLRequest are defined by the following API:

```
interface URLHeader
{
   readonly attribute DOMString name;
   readonly attribute DOMString value;
};
```

name
The header's name
value
The header's value.

A URLRequest object is created using the "createURLRequest" method on the SVGGlobal interface.

The following code provides an example of how the URLRequest API can be used to emulate the common getURL() and postURL methods.

```
function callback(evt) {
   if (evt.target.requestState == URLRequest.COMPLETED
      && evt.target.status == 200) {
      alert("OK: " + evt.target.responseText);
   } else {
      alert("ERROR!");
   }
}

function getURL(url, callback) {
    var req = createURLRequest();
    req.addEventListener("URLResponse", callback, false);
    req.init("GET", url);
    req.submit();
}

function postURL(url, body, callback) {
    var req = createURLRequest();
    req.addEventListener("URLResponse", callback, false);
    req.init("POST", url);
    req.requestText = body;
    req.submit();
}

function putURL(url, file, callback) {
    var req = createURLRequest();
```

```
    req.addEventListener("URLResponse", callback, false);
    req.init("PUT", url);
    req.addContentFromFile(file);
    req.submit();
}
```

B.2.2 Security Concerns

Note that these interfaces expose possible security concerns. The security model that these interfaces work under is defined by the user agent. However, there are a well-known set of security guidelines used by the browser implementations in this area. For example, most do not allow access to hosts other than the host from which the document was retrieved.

In general the User Agent should prevent access to URLs that refer to host names different from the host name of the actual URL of the corresponding SVGGlobal::document. This applies not only to original URL, but also to any redirected URLs. The access to protocols, different from the protocols of the SVGGlobal::document URL also should be restricted (e.g., a document loaded through HTTP should not have access to HTTPS resources).

B.2.3 Socket Connections

The Connection interface provides an API for socket-level communication.

```
interface ConnectionEvent : events::Event // "ConnectionData"
event
{
   readonly attribute DOMString data;
};

interface Connection : events::EventTarget
{
   readonly attribute boolean connected;

   void setEncoding( DOMString value ); // might be called
before connect

   void connect( in DOMString url ) raises(DOMException)
   void send( in DOMString data );
   void close();
};
```

A Connection object is created using the SVGGlobal interface.

B.3 Monitoring download progress

Many resources, such as raster images, movies and complex SVG content can take a substantial amount of time to download. In some use cases the author would prefer to delay the display of content or the beginning of an animation until the entire contents of a file have been downloaded. In other cases, the author may wish to give the viewer some feedback that a download is in progress (e.g. a loading progress screen).

B.3.1 The Progress event

The ProgressEvent occurs when the user agent makes progress loading a resource (local or external) referenced by an **xlink:href** attribute.

The user agent **must** dispatch a ProgressEvent at the beginning of a load operation (i.e., just before starting to access the resource). This event is of type 'preload'. The value of the 'preload' event's **progress** property is 0.

The user agent **must** dispatch a ProgressEvent at the end of a load operation (i. e. after load is complete and the user agent is ready to render the corresponding resource). This event is of type 'postload' event. The value of the 'postload' event's **progress** property is 1.

The user agent **may** dispatch a loadProgress event between the 'preload' event and the 'postload' events. Such events are of type 'loadprogress'.

All 'loadprogress' events **must** follow to the following constraints:

- the **progress** property on an 'loadprogress' event is strictly greater or equal to zero and strictly smaller than or equal to one.
- for two consecutive 'loadprogress' events, if provided, the progress property of an event **must** be strictly bigger than the value of the **progress** property on the preceding event.
- for two consecutive 'loadprogress' events, the **loaded** property of an event **must** be strictly bigger than the value of the **loaded** property on the preceding event.

In the case where the size of the downloading resource is known, such as from HTTP headers, then the **progress** property reflects the proportion of the current download that has been completed.

In the case where the size of the downloading resource is not known, then the **progress** property will only ever have the value 0 or 1.

The ProgressEvent has three corresponding event attributes on elements: **onpreload**, **onpostload** and **onloadprogress**.

Below is an example of the ProgressEvent:

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

   <script type="text/ecmascript"><![CDATA[
   function showImage(imageHref) {
       var image = document.getElementById('myImage');
       image.setAttributeNS(xlinkNS, "href", imageHref);
   }

   function imageLoadStart(evt) {
       var progressBar = document.getElementById
('progressBar');
       progressBar.setAttributeNS(svgNS, "width", 0);
       var loadAnimation = document.getElementById
('loadAnimation');
       loadAnimation.beginElement();
   }

   function imageLoadProgress(evt) {
       var progressBar = document.getElementById
('progressBar');
       progressBar.setAttributeNS(svgNS, "width", 100*evt.
progress);
   }

   function imageLoadComplete(evt) {
       var progressBar = document.getElementById
('progressBar');
       progressBar.setAttributeNS(svgNS, "width", 100);
        var loadAnimation = document.getElementById
('loadAnimation');
       loadAnimation.endElement();

   }
   ]]></script>

   <image id="myImage" xlink:href="imageA.png" width="300"
height="400"
```

```
            onpreload="imageLoadStart(evt)"
            onloadprogress="imageLoadProgress(evt)"
            onpostload="imageLoadComplete(evt)"/>

   <rect onclick="showImage('imageB.png')" width="120"
         height="30" y="400"/>

   <animate id="loadAnimation" ... />

   <rect id="progressBar" ... />
</svg>
```

IDL Definition

```
interface ProgressEvent : events::Event {
     readonly attribute DOMString       typeArg;
     readonly attribute unsigned long   loaded;
     readonly attribute unsigned long   total;
     readonly attribute float   progress;
     void               initProgressEvent(in DOMString
typeArg,
                                          in unsigned long
loaded,
                                          in unsigned long
total);
};
```

Attributes

readonly DOMString typeArg

Specifies the event type. One of 'preload', 'loadprogress' or 'postload'.

readonly unsigned long total

Specifies the expected total number of bytes expected in a load operation. This value is ignored for a 'preload' and a 'postload' event. For a 'loadprogress' event, it should specify the total number of bytes expected or be -1 which means that it cannot be computed (for example when the size of the downloaded resource is unknown).

readonly unsigned long loaded

Specifies the number of bytes downloaded since the beginning of

the download. This value is ignored for a 'preload' or 'postload' event. For a 'loadprogress' event, this value must be positive.

Methods

initProgressEvent

The initProgressEvent method is used to initialize the value of a ProgressEvent created through the DocumentEvent interface. This method may only be called before the ProgressEvent has been dispatched via the dispatchEvent method, though it may be called multiple times during that phase if necessary. If called multiple times, the final invocation takes precedence.

Parameters

in DOMString typeArg	Specifies the event type.
in unsigned long loaded	Specifies the number of bytes that have been retrieved. This is a positive value.
in unsigned long total	Specifies the expected total number of bytes in this load operation.

No Return Value
No Exceptions

B.4 File Upload

It is desirable for Web applications to have the ability to manipulate as wide as possible a range of user input, including files that a user may wish to upload to a remote server or manipulate inside a rich application. This interface provides application writers with the means to trigger a file selection prompt with which the user can select one or more files. Unlike the file upload forms control available to HTML, this API can be used for more than simply inserting a file into the content of a form being submitted but also allows client-side manipulation of the content, for instance to display an image or parse an XML document.

B.4.1 Interface FileDialog

The FileDialog interface is created with a call to SVGWGlobal::createFileDialog. Its purpose is to control the apparition of a file dialog, and to register the event listeners that will handle the selection of files. The FileDialog object inherits from EventTarget and it is this possible to attach event listeners to it using the addEventListener method.

```
interface FileDialog : events::EventTarget {
```

```
    void open ( );
}
```

The open method takes no parameter and returns nothing. On being called, it prompts the user with a means to select one or more files. It does not need to be a GUI control, but rather whichever input method the user has at his disposal to select files present on the device. On devices that have no file system, it may still open a dialog for data acquisition, eg an interface to a built-in camera. The user agent should make sure that the user cannot forget about the file upload prompt's existence, for instance in the case of a GUI file selection widget by maintaining it on top of the SVG document.

This method operates in an asynchronous manner so that the files that are selected will be communicated to handlers registered on the FileDialog object to listen to the FilesSelected event. An example follows in which when file selection is performed, the handleEvent method of the listener object will be called with a FilesSelectionEvent object.

```
var fd = createFileDialog();
fd.addEventListener("FilesSelected", listener, false);
fd.open();
```

B.4.2 Interface FilesSelected

This is an event dispatched for FilesSelected events. It adds a single field fileList which points to the SVGFileList containing the list of selected files. If the file dialog is cancelled, or if no files are selected, the FileList will be defined but its length will be zero.

```
interface FilesSelected : events::Event {
    readonly attribute FileList fileList;
}
```

B.4.3 Interface FileList

This interface exposes the list of files that has been selected through the file dialog. When none have been selected, its length is zero. It is possible that some platforms will only allow for one file to be selected at a time, however when possible a user agent should provide the option to select multiple files in one pass.

```
interface FileList {
   readonly attribute unsigned long numberOfItems;
```

```
    File getItem    ( in unsigned long index ) raises
( DOMException );
    File removeItem ( in unsigned long index ) raises
( DOMException );
 }
```

The length of the collection can be obtained through the numberOfItems field. Items can be retrieved using getItem, and removed using removeItem. The latter can be used to filter out files that do not match certain criteria (mime type, file extension, etc).

B.4.4 Interface File

This interface describes a single file in a FileList, and allows you to know its name, mime type, and to access its content in various convenient ways.

Please note that in order to be memory-efficient, implementations are not required to load the content of files into memory as soon as they have been selected, but only when their content is required by the program. Even then using the network interfaces an implementation may stream the content of a file to a socket and never need to hold more than a few of its bytes in memory. Note however that in case the implementation only provides the content of the file on demand, since the state of the file system may have changed since the pointer was obtained the content may no longer be available, or might have been modified.

```
interface File {
   readonly attribute DOMString fileName;
   readonly attribute DOMString mimeType;
   readonly attribute DOMString fileSize;
   DOMString getDataAsString    (  ) raises(SVGException);
   DOMString getDataAsBase64    (  ) raises(SVGException);
   DOMString getDataAsHexBinary (  ) raises(SVGException);
}
```

The fileName field provides the name of the file, exclusive of its path.

The mimeType field provides the MIME type of the file, if it is known to the user agent. Where available, it must be provided so as to allow users to filter the content of FileLists based on this criterion as is possible in XForms.

The fileSize field provides the size of the file in bytes. User agents should provide it when it is available, but users should keep in mind that it may not

always be possible for a user agent to know it before the data has been read.

The getDataAsString, getDataAsBase64, and getDataAsHexBinary methods return the content of the file. Since the file may not be a text file, it is possible that getDataAsString might produce unexpected effects in some languages. Note however that languages such as Ecmascript and Java use a simple array of 16-bit bytes to store their string types, and can thus accomodate any binary data. getDataAsBase64 and getDataAsHexBinary are respectively Base64 and hex-binary encoded versions of the content returned by getDataAsString.

If the implementation detects a problem when trying to gain access to the file's content (file is not readable by user, has been removed since the pointer to it was obtained, etc) it must throw an exception. An empty file results in an empty string.

B.4.5 URLRequest additions

In order to avoid costly copying of file content, it is possible to feed files directly into network requests.

```
 interface URLRequest {
    ...
   void addContentFromFile ( in File file ); // adds to the
body, body illegal
+(discarded) for some methods
    ...
 }
```

The addContentFromFile method takes a file and adds it to the body of a request. Note that for some request types (eg GET, HEAD...) it is illegal to have a payload and thus the content will be discarded. If the file cannot be read, an exception will be thrown.

B.4.6 Connection additions

In order to avoid costly copying of file content, it is possible to stream files directly to a socket.

```
interface Connection {
   ...
   void sendFile ( in File file )  raises(SVGException);
   ...
};
```

The sendFile method takes a file and streams it to the socket. If the file cannot be read, an exception will be thrown.

B.4.7 Security considerations

The file upload feature has security implications. However it does not add any security-related issues above those in the common HTML file upload form widget. Rather, it removes the potential risk of having script access to or setting the default value of the form component which has been a cause for concern in a number of HTML user agents.

B.5 Persistent Client-side Data storage

Many applications benefit from the ability to store data between sessions on the client machine. SVG 1.2 adds a simple set of methods to store data specific to the SVG Document in the client. These methods are setPersistentValue and getPersistenValue on the SVGGlobal interface.

The allowed names for persistent values follow the same rules as XML identifiers. The allowed values can be any string, including an XML serialization (which can be consumed by the parseXML() method and generated by the printNode() method).

The user agent keeps a table of persistent values separate for each domain. There is no set limit to how much data can be stored per domain, nor how long. The minimum limits are 10 names, 500 characters and 60 days expiration. If the storage fills beyond the user agent limits, then values may be silently discarded by the user agent.

B.6 Global Interface

The majority of scripted SVG documents in existence make use of the browser specific Window interface, which includes methods such as parseXML and alert. SVG 1.2 specifies an SVGGlobal interface, taking into account the de-facto standard that already exists, as well as adding the new features present in SVG 1.2. SVGGlobal inherits from the Global interface, which is currently defined to be empty. The Global interface is designed to be the parent interface for language specific window objects. In scripting implementations, the methods and attributes defined by the Global object are normally part of the global execution context.

Interface SVGGlobal provides a global object for scripts embedded in a SVG document.

IDL Definition

```
interface Global {}


interface SVGGlobal : Global {

  readonly  attribute dom::Document document;
  readonly  attribute Global        parent;
  readonly  attribute DOMString     location;

  // Move to a new document
  void gotoLocation(in DOMString newURL);

  Node  parseXML(in DOMString source, in Document document);
  DOMString         printNode(in Node node);

  // Timer method.
  SVGTimer createTimer(in long delay, in long interval, in
boolean start);

  // Network methods
  URLRequest createURLRequest();
  Connection createConnection();

  // Mouse capture
  void startMouseCapture(in dom::EventTarget, in boolean
sendAll,
                         in boolean autorelease);
  void stopMouseCapture();

  // File dialog
  FileDialog createFileDialog();

  // Filtering of events
  SVGEventFilter createEventFilter();

  // Persistent client-side data storage
  void setPersistentValue(in DOMString name, in DOMString
value);
```

```
    DOMString getPersistentValue(in DOMString name);

};
```

Attributes

readonly dom::Document document

The Document that this SVGGlobal operates on.

readonly SVGGlobal parent

The SVGGlobal context of this document's parent.

readonly DOMString location

The URI of the current document.

Methods

gotoLocation

Request that the user agent navigates to the given URL. The user agent should handle the resource in the same way as a user navigation and it is not limited to SVG resources.

Parameters

in DOMString newURL	The URL of the new location

No Return Value

No Exceptions

parseXML

Convert the given source string into DocumentFragment that belongs to the given XML document. This document fragment does not get inserted in the tree; this can be done with DOM methods like appendChild or insertBefore. If the second parameter is null, a new XML document is created and the given string is parsed as a standalone XML document.

Parameters

in DOMString source	A string containing a XML document fragment.
in Document document	The Document context for parsing the XML fragment.

Return Value

Node	A Node representing a Document or Document Fragment converted from the original DOMString.

No Exceptions

printNode

Converts a Node into a DOMString. The string is an XML

representation of the Node.

Parameters

in Node node	The Node to be converted.

Return Value

DOMString	A serialized version of the original Node.

No Exceptions

createTimer

Creates an SVGTimer with the given delay and interval.

Parameters

in long delay	Corresponds to the delay attribute on SVGTimer.
in long interval	Corresponds to the interval attribute on SVGTimer.
in boolean start	If true, the returned SVGTimer will be started, otherwise it will be in the waiting state.

Return Value

SVGTimer	An SVGTimer object.

No Exceptions

createURLRequest

Creates an URLRequest object with default parameters.

No Parameters

Return Value

URLRequest	An URLRequest object.

No Exceptions

createConnection

Creates a Connection object with default parameters.

No Parameters

Return Value

Connection	A Connection object.

No Exceptions

startMouseCapture

Limit the dispatch of mouse events to a subtree whose root is defined by the target parameter. Note that this does not affect the platform behaviour of mouse capture, which typically sends all drag-like mousemove events the the widget that received the

mousedown. This method only affects events within the SVG canvas.

Parameters

in dom::EventTarget target	The subtree that will receive all the captured events.
in boolean sendAll	If true, send all mouse events, even those that do not intersect with the specified subtree. For example, scrollbars typically get sent events after they are activated even when the mouse moves outside the scrollbar.
in boolean autorelease	Release the mouse capture once a mouseup event is fired.

No Return Value

No Exceptions

stopMouseCapture

Cancels any mouse capture

No Parameters

No Return Value

No Exceptions

createFileDialog

Creates a FileDialog object.

No Parameters

Return Value

FileDialog	A FileDialog object.

No Exceptions

createEventFilter

Creates an EventFilter object.

No Parameters

Return Value

EventFilter	An EventFilter object.

No Exceptions

startMouseCapture

Limit the dispatch of mouse events to a subtree whose root is defined by the target parameter. Note that this does not affect the platform behaviour of mouse capture, which typically sends all drag-like mousemove events the the widget that received the mousedown. This method only affects events within the SVG

canvas.

Parameters

in dom::EventTarget target	The subtree that will receive all the captured events.
in boolean sendall	If true, send all mouse events, even those that do not intersect with the specified subtree.
in boolean autorelease	Release the mouse capture once a mouseup event is fired.

No Return Value

No Exceptions

setPersistentValue

Store the given (name, value) pair in the user agent's persistent data store, associated with the domain from which the current document was loaded.

Parameters

in DOMString name	The name of the property to be stored.
in DOMString value	The value of the property to be stored.

No Return Value

No Exceptions

getPersistentValue

Retrieve the value for the given name in the user agent's persistent data store, associated with the domain from which the current document was loaded.

Parameters

in DOMString name	The name of the property to be retrieved.

Return Value

DOMString	The value for the given name.

No Exceptions

NOTE:

SVG 1.2 no longer supports the alert, confirm and prompt methods, which were mainly used for debugging purposes, due to their synchronous behaviour and accessibility problems. The SVG Working Group understands that many implementations may still support the methods, but content developers should

know that these methods will produce a non-conformant document.

The SVGDocument interface provides access to the SVGGlobal object.

```
interface SVGDocument {
  ...
  readonly  attribute SVGGlobal     global
  ...
};
```

Previous | Top | Next

Appendix C: SVG DOM Subset

During the later stages of the SVG Mobile 1.1 specifications it became obvious that there was a requirement to subset the SVG and XML DOM in order to reduce the burden on implementations. SVG 1.2 adds new features to the SVG DOM, allowing a subset to be taken that includes as much necessary functionality as possible. SVG 1.2 also proposes a subset, suitable for SVG Tiny implementations.

Furthermore, it should be possible to implement the DOM subset on devices that support SVG Tiny 1.1 although, in this case, the scripting would be external to the SVG document (since SVG Tiny 1.1 does not support inline scripting).

The goal is to provide an API that allows access to initial and animated attribute and property values, to reduce the number of interfaces, to reduce run-time memory footprint using necessary features of the core XML DOM, as well as the most useful features from the Full SVG DOM (such as transformation matrix helper functions).

The IDL definition for the DOM Subset is provided.

A.1 Introduction

This appendix consists of the following parts:

- Overview of the SVG Tiny 1.2 DOM An introduction to the SVG Tiny 1.2 DOM, including a summary of supported features and descriptions by topic of the key features and constraints.
- A definition of all the interfaces in the SVG Tiny 1.2 DOM. (DOM Core APIs, DOM Events APIs, and SVG DOM APIs.)

A.2 Overview of the SVG Tiny 1.2 DOM

The following sections describe the key features and constraints within the SVG Tiny 1.2 DOM.

Note that, like all other W3C DOM definitions, the SVG Tiny 1.2 DOM is programming-language independent. Although this appendix only contain ECMAScript and Java examples, the SVG Tiny 1.2 DOM is compatible with other programming languages.

A.2.1 Document Access

All proposals assume that a Document object is present and is the root for accessing other features. The way the Document object becomes available depends on the usage context. One way to gain access to the Document object is to implement interface EventListenerInitializer within your programming logic. The SVG Tiny user agent will invoke your `initializeEventListeners(dom::Document doc)` method once your programming logic has been loaded and is ready to bind to the document. The Document object typically will be available via various other means, also.

A.2.2 Tree Navigation

SVG Tiny 1.2 DOM only allows navigation of the element nodes in the DOM tree. Two options are available for navigating the hierarchy of elements:

- Individual element nodes with an ID value can be accessed directly via the `getElementById` method on the Document interface.
- The hierarchy of element nodes can be traversed using the facilities on the ElementTraversal interface, along with the `parentNode` attribute on the Node interface

The ElementTraversal interface provides `firstElementChild`, `lastElementChild`, `previousElementSibling` and `nextElementSibling`, which are particularly suitable for constrained devices. These traversal mechanisms skip over intervening nodes between element nodes, such as text nodes which might only contain spaces, tabs and newlines.

A.2.3 Element Creation

SVG Tiny 1.2 DOM allows the creation of new `Elements`:

```
Element myRect = document.createElementNS(svgNS, "rect");
```

The type of elements which can be created through the createElementNS method is restricted to the following elements in the SVG namespace: `<rect>`, `<circle>`, `<ellipse>`, `<line>`, `<path>`, `<text>`, `<image>`, `<use>`, `<a>` and `<g>`.

A.2.4 Element Addition

Node Addition is the ability to add new elements to a document tree.

SVG Tiny 1.2 DOM allows addition and insertion and insertion of a `Node`:

```
String svgNS = "http://www.w3.org/2000/svg";
// Create a new <rect> element
Element myRect
    = document.createElementNS(svgNS, "rect");

// Set the various <rect> properties before appending
...

// Add element to the root of the document
Element svgRoot = document.documentElement();
svgRoot.appendChild(myRect);

// Create a new <ellipse> element
Element myEllipse
    = document.createElementNS(svgNS, "ellipse");

// Set the various <ellipse> properties before insertion
...

// Insert the ellipse before the rectangle
svgRoot.insertBefore(myEllipse, myRect);
```

The types of nodes which can be inserted into the `Document` is limited to the same list specified in the Element Creation section.

A.2.5 Element Removal

Node removal is the ability to remove an element from a document tree. SVG Tiny 1.2 DOM allows the removal

of element Nodes:

```
Element myRect = ...; // See Element creation
Element myGroup = document.getElementById("myGroup");
myGroup.appendChild(myRect);
....
myGroup.removeChild(myRect);
```

Any element nodes in the document can be removed, including both element nodes that were created via the DOM and element nodes that existed in the original document.

A.2.6 Attribute and Property Access

SVG 1.2 adds a new ability to access XML attribute and CSS property values through the SVG DOM through the concept of *traits*. A trait is a potentially animatable parameter associated with an element. Trait is the typed value (e.g., a number, not just a string) that gets assigned through an XML attribute, CSS property or a SMIL animation [SMILANIM]. Traits can be thought of as a unification and generalization of some of the notions of XML attributes and CSS properties.

The trait facilities in the SVG DOM allow for strongly-typed access to certain attribute and property values. For example, there is a `getFloatTrait(...)` method for getting an attribute or property value directly as a float. This contrasts the DOM Core `getAttributeNS(...)` method which always returns a string.

The SVG Tiny 1.2 DOM includes a subset of the trait facilities in the full SVG 1.2 DOM. The trait facilities in the SVG Tiny 1.2 DOM are available on the TraitAccess interface.

Here is an example which uses the trait facilities to get and set the width of a rectangle:

```
float width = myRect.getFloatTrait('width');
width += 10;
myRect.setFloatTrait('width', width);
```

A.2.7 Text Node Access

In the SVG 1.2 Tiny DOM, text node access is available via trait getters and setters. To access or set the text string value for a text element (e.g., a <text> element), you invoke `getTrait()` or `setTrait()` on that text element and pass `#text` as the name of the trait you want to get or set. For example, `MyTextElement.setTrait("#text", "Hello");`

A.2.8 Event Listener Registration and Removal

Event Listener Registration and Removal is the ability to add and remove new event listeners from a `Document`. SVG Tiny 1.2 DOM allows adding and removing `EventListeners`:

```
class MyEventListener implements EventListener {
    public void handleEvent(Event evt) {
        // Do whatever is needed here
    }
}
...
EventListener l = new MyEventListener();

SVGElement myRect = (SVGElement)document.getElementById("myRect");

// Listen to click events, during the bubbling phase
myRect.addEventListener("click", l, false);
....
```

```
// Remove the click listener
myRect.removeEventListener("click", l, false);
```

The SVG 1.2 Tiny DOM only supports the bubble phase. Any attempt to specify event operations on the capture phase will raise a DOMException of type NOT_SUPPORTED_ERR.

Refer to the DOM Events Level 3 specification or the XML Events specification introduction for an explanation of the SVG event flow and the meaning of event targets, event current target, bubble and capture.

A.2.9 Animation

SVG Tiny 1.2 DOM allows code to start or end animation elements.

```
AnimationElement animateColor
    = (AnimationElement) document.getElementById("myAnimation");

// Start the animation 2.5 seconds from now.
animateColor.beginElementAt(2.5);
```

A.2.10 Package naming

The SVG 1.2 Tiny DOM will use the same package names as the SVG 1.2 Full DOM (e.g., org.w3c.dom, org.w3c.dom.events, org.w3c.dom.svg). This allows applications which restrict themselves to the features in the SVG 1.2 Tiny DOM to also run in implementations that support the SVG 1.2 Full DOM.

A.3 Interfaces from DOM Core

DOMException

The `DOMException` class defines a subset of the error codes defined in the DOM Core Level 3 specification.

IDL Definition

```
exception DOMException
{
        unsigned short   code;
};

// ExceptionCode
const unsigned short INDEX_SIZE_ERR = 1;
const unsigned short HIERARCHY_REQUEST_ERR = 3;
const unsigned short WRONG_DOCUMENT_ERR = 4;
const unsigned short NO_MODIFICATION_ALLOWED_ERR = 7;
const unsigned short NOT_FOUND_ERR = 8;
const unsigned short NOT_SUPPORTED_ERR = 9;
const unsigned short INVALID_MODIFICATION_ERR = 13;
const unsigned short INVALID_ACCESS_ERR = 15;
const unsigned short TYPE_MISMATCH_ERR = 17;
```

Defined constants

INDEX_SIZE_ERR	If index or size is negative, or greater than the allowed value.
HIERARCHY_REQUEST_ERR	If any Node is inserted somewhere it doesn't belong.
WRONG_DOCUMENT_ERR	If a node is used in a different document than the one that created it (that doesn't support it)
NO_MODIFICATION_ALLOWED_ERR	If an attempt is made to modify an object where modifications are not allowed.
NOT_FOUND_ERR	If an attempt is made to reference a node in a context where it does not exist .
NOT_SUPPORTED_ERR	If the implementation does not support the requested type of object or operation.
INVALID_MODIFICATION_ERR	If an attempt is made to modify the type of the underlying object.
INVALID_ACCESS_ERR	If a parameter or an operation is not supported by the underlying object.
TYPE_MISMATCH_ERR	If the type of an object is incompatible with the expected type of the parameter associated to the object.

Node

The `Node` interface is the interface for all XML tree model content. This interface is a subset of the `Node` interface defined in the DOM Core Level 3 specification.

IDL Definition

```
interface Node
{
        readonly attribute Node parentNode;
        readonly attribute DOMString namespaceURI;
        readonly attribute DOMString localName;

        Node insertBefore(in Node newChild, in Node refChild) raises(DOMException);
        Node removeChild(in Node oldChild) raises(DOMException);
        Node appendChild(in Node newChild) raises(DOMException);
};
```

No defined constants

Attributes (not yet completed)

Methods

appendChild

Adds `newChild` to the end of the children list for this node.

Parameters

in Node newChild	The new `Node` to add.

Return Value

Node	the newly added node

Exceptions

DOMException	If the operation is not allowed (e.g., if the newChild node type is incompatible with this node) or if addition of the given Node type is not supported by the implementation.

insertBefore

Inserts `newChild` before `refChild`.

Parameters

in Node newChild	The `Node` to insert.
in Node refChild	The `Node` before which `newChild` is inserted

Return Value

Node	the newly inserted node

Exceptions

DOMException	If the operation is not allowed or not supported.

removeChild

Removes a child node.

Parameters

in Node oldChild	The `Node` to remove.

Return Value

Node	the removed node

Exceptions

DOMException	See the DOM Level 3 specification.

Element

The `Element` interface represents an XML element in a `Document`. This interface is a subset of the `Element` interface defined in the DOM Core Level 3 specification.

IDL Definition

```
interface Element : Node
{
};
```

No Defined constants

No methods

Document

The `Document` interface is the interface for an XML `Document` model. This interface is a subset of the `Document` interface defined in the DOM Core Level 3 specification. Note that the `getFirstChild` method returns the root of the document.

IDL Definition

```
interface Document : Node
{
        readonly attribute Element documentElement;
        Element createElementNS(in DOMString namespaceURI, in DOMString qualifiedName)
raises(DOMException);
        Element getElementById(in DOMString elementId);
};
```

No Defined constants

Defined attributes

(Incomplete)

Methods

createElementNS

Create a new element.

Parameters

in DOMString namespaceURI	The namespace uri for the newly created element.
in DOMString qualifiedName	The qualified name for the newly created element.

Return Value

Element	The newly created element

Exceptions

DOMException	See DOM Level 3 specification. In addition, a DOMException (NOT_SUPPORTED_ERR) is thrown if the type of element is not supported by the implementation.

getElementById

Get an element with a given id.

Parameters

in DOMString elementId	The unique `id` of the retrieved element.

Return Value

Element	The matching element or null if none.

No Exceptions

A.4 Interfaces from DOM Events

EventTarget

The interface for DOM nodes which can receive and dispatch `Events` to `EventListeners`. This interface is a subset of the `EventTarget` interface defined in the DOM Level 3 Events specification. Please refer to that specification for details on what the different methods and members mean.

The SVG Tiny DOM only supports the event bubbling phase. If `useCapture` is true, a DOMException of type

NOT_SUPPORTED_ERR is raised.

IDL Definition

```
interface EventTarget
{
        void addEventListener(in DOMString type, in EventListener listener, in boolean
useCapture);
        void removeEventListener(in DOMString type, in EventListener listener, in
boolean useCapture);
};
```

No Defined constants

Methods

addEventListener

Adds a new listener to this target, for the specified event type, during the desired phase.

Parameters

in DOMString type	The type of event to listen to.
in EventListener listener	Will be notified when an event of the desired type happens on this target or one of its descendant.
in boolean useCapture	If true, the listener will be called during the event flow capture phase. Otherwise, the listener will be called during the bubble phase. If the event's target is this target, then the listener will be called during the 'at target' phase of event flow.

No Return Value

No Exceptions

removeEventListener

Removes a listener previously added with an `addEventListener` call.

Parameters

in DOMString type	The type of event that was listened to.
in EventListener listener	The listener that was previously registered.
in boolean useCapture	If true, the listener was listening to events in the capture phase of event flow.

No Return Value

No Exceptions

EventListener

Interface used to receive `Events` from an `EventTarget` This interface is a subset of the `EventListener` interface defined in the DOM Level 3 Events specification. Please refer to that specification for details on what the different methods and members mean.

IDL Definition

```
iinterface EventListener
```

```
{
        void handleEvent(in Event evt);
};
```

No Defined constants

Methods

handleEvent

Handle event.

Parameters

in Event	evt	Contains contextual information about the event.

No Return Value

No Exceptions

Event

Provides information about an event and its propagation. This interface is a subset of the `Event` interface defined in the DOM Level 3 Events specification and defines additional constraints.

IDL Definition

```
interface Event
{
        readonly attribute DOMString type;
        readonly attribute EventTarget currentTarget;
};
```

No defined constants

Defined attributes

(Incomplete)

MouseEvent

The MouseEvent interface provides specific contextual information associated with Mouse events.

IDL Definition

```
interface MouseEvent : Event
{
        readonly attribute long screenX;
        readonly attribute long screenY;
        readonly attribute long clientX;
        readonly attribute long clientY;
        readonly attribute unsigned short button;
};
```

No defined constants

Defined attributes

(Incomplete)

TextEvent

The TextEvent interface provides information associated with Text events.

IDL Definition

```
interface TextEvent : Event
{
        readonly attribute DOMString data;
};
```

No defined constants

Attributes

readonly DOMString data

The text data.

KeyboardEvent

The KeyboardEvent interface provides information associated to a key event.

IDL Definition

```
interface KeyboardEvent : Event
{
        readonly attribute DOMString keyIdentifier;
};
```

No defined constants

Attributes

readonly DOMString keyIdentifier

String indicating which key that was pressed.

ConnectionEvent

The ConnectionEvent interface provides information associated with a socket connection.

IDL Definition

```
interface ConnectionEvent : Event
{
        readonly attribute DOMString data;
};
```

No defined constants
Attributes

readonly DOMString data

The connection data.

Supported events

The following events are supported:

- DOMActivate
- DOMFocusIn
- DOMFocusOut
- activate
- click
- focusin
- keydown
- keyup
- load
- resize
- scroll
- textInput
- zoom
- connectionData

A.5 Interfaces from SMIL DOM

ElementTimeControl

IDL Definition

```
interface ElementTimeControl
{
        boolean beginElementAt( in float offset );
        boolean endElementAt( in float offset );
};
```

A.6 Global Interfaces

Global

IDL Definition

```
interface Global
{
};
```

Connection

IDL Definition

```
interface Connection
{
        void connect( in DOMString uri ) raises(DOMException);
        void send( in DOMString data );
        void close();
        readonly attribute boolean connected;
};
```

A.7 Interfaces for SVG DOM

SVGException

IDL Definition

```
exception SVGException
{
        unsigned short   code;
};

// SVGExceptionCode
const unsigned short SVG_INVALID_VALUE_ERR        = 1;
const unsigned short SVG_MATRIX_NOT_INVERTABLE    = 2;
```

Interface SVGRect

IDL Definition

```
interface SVGRect
{
        attribute float x;
        attribute float y;
        attribute float width;
        attribute float height;
};
```

SVGPoint

IDL Definition

```
interface SVGPoint
{
        attribute float x;
        attribute float y;
};
```

SVGMatrix

IDL Definition

```
interface SVGMatrix
{
        float getComponent(in unsigned long index) raises(dom::DOMException);

        SVGMatrix multiply( in SVGMatrix secondMatrix );
        SVGMatrix inverse() raises( SVGException );
        SVGMatrix translate( in float x, in float y );
        SVGMatrix scale( in float scaleFactor );
        SVGMatrix rotate( in float angle );
};
```

SVGPath

SVG`Path` provides an API to access and manipulate the `'d'` attribute on the <path> element

IDL Definition

```
```

```
interface SVGPath
{
        const unsigned short MOVE_TO = 0x4d; // 'M'
        const unsigned short LINE_TO = 0x4C; // 'L'
        const unsigned short CURVE_TO = 0x43;// 'C'
        const unsigned short QUAD_TO = 0x51; // 'Q'
        const unsigned short CLOSE = 0x5a;   // 'Z'

        readonly attribute unsigned long numberOfSegments;

        unsigned short getSegment( in unsigned long index ) raises(dom::DOMException);
        float getSegmentParam( in unsigned long cmdIndex, in unsigned long
paramIndex ) raises(dom::DOMException);

        void moveTo( float x, float y );
        void lineTo( float x, float y );
        void quadTo( float x1, float y1, float x2, float y2 );
        void curveTo( float x1, float y1, float x2, float y2, float x3, float y3 );
        void close();
};
```

SVGRGBColor

IDL Definition

```
interface SVGRGBColor
{
        readonly attribute unsigned long red;
        readonly attribute unsigned long green;
        readonly attribute unsigned long blue;
};
```

SVGLocatable

IDL Definition

```
interface SVGLocatable
{
        SVGRect   getBBox();
        SVGMatrix getScreenCTM();
        SVGRect   getScreenBBox();
};
```

TraitAccess

`TraitAccess` is an interface to access *trait values* (see Attribute Access). A trait is a potentially animatable parameter associated with an element. Trait is the typed value (e.g., a number, not just a string) that gets

assigned through an XML attribute, CSS property or a SMIL animation [SMILANIM]. Traits can be thought of as a unification and generalization of some of the notions of XML attributes and CSS properties. The trait facilities allow for strongly-typed access to certain attribute and property values. For example, there is a `getFloatTrait(...)` method for getting an attribute or property value directly as a float. This contrasts the DOM Core `getAttributeNS(...)` method which always returns a string.

Each trait corresponds to an attribute or property which is parsed and understood by the element and in most cases animatable. For any given profile there is a well-defined set of traits that all implementations must support. Each increasing profile may support a larger set of traits. If an implementation does not support a trait it must throw an exception. Unlike attributes traits are typed and their values are normalized; for instance SVG path specification is parsed and all path commands are converted to their absolute variants, it is not possible to say through the value of the trait if a path command was absolute or relative. When getting and setting trait values, accessor of the correct type must be used or exception will be thrown.

For XML attributes, the setter methods (e.g., `setTrait(...)`, `setFloatTrait(...)`) add a new attribute for the given element. If an attribute with that name is already present on the element, its value is changed to be that of the `value` parameter. For CSS properties, the setter methods set the specified value for the given property, with equivalent specificity rules to using `setAttributeNS(...)` on a presentation attribute (see presentation attributes). The value which is modified is always a base value (in terms of SMIL animation). For inheritable traits the trait value can always be set to "inherit" (but quering the value will always return the actual inherited value as explained above).

The XML attributes, the getter methods (e.g., `getTrait(...)`, `getFloatTrait(...)`) return the attribute value for the named attribute. If the given attribute has a value, then return that value; else if the attribute does not have a value, and if the attribute is a CSS property then return the computed value; else if the attribute is inheritable and an inheritable value is available, then return the inherited value; else if there is a default value, then return the default value; otherwise, an INVALID_ACCESS_ERR DOMException is raised. In either case (i.e., XML attributes or CSS properties), the returned value corresponds to the *base value* before animation is applied and not the *presentation value* (aka, animated value), where *base value* and *presentation value* corresponds to the SMIL Animation definitions of these terms (see [SMILANIM]). For the attribute "xlink:href" then the returned value follows the processing rules of xml:base.

For both the getter and setter methods, if the trait name does not correspond to a defined attribute or property, an exception is raised.

IDL Definition

```
interface TraitAccess
{
        DOMString getTrait( in DOMString name ) raises (DOMException);
        DOMString getTraitNS( in DOMString namespaceURI, in DOMString name ) raises
(DOMException);
        float getFloatTrait( in DOMString name ) raises (DOMException);
        SVGMatrix getMatrixTrait( in DOMString name ) raises (DOMException);
        SVGRect getRectTrait( in DOMString name ) raises (DOMException);
        SVGPath getPathTrait( in DOMString name ) raises (DOMException);
        SVGRGBColor getRGBColorTrait( in DOMString name ) raises (DOMException);

        void setTrait( in DOMString name, in DOMString value ) raises (DOMException);
        void setTraitNS( in DOMString namespaceURI, in DOMString name, in DOMString
value ) raises (DOMException);
        void setFloatTrait( in DOMString name, in float value ) raises (DOMException);
        void setMatrixTrait( in DOMString name, in SVGMatrix matrix ) raises
(DOMException);
        void setRectTrait( in DOMString name, in SVGRect rect ) raises (DOMException);
        void setPathTrait( in DOMString name, in SVGPath path ) raises (DOMException);
```

```
        void setRGBColorTrait( in DOMString name, in SVGRGBColor color ) raises
(DOMException);
};
```

Attributes

DOMString id

The value of the id attribute on the given element.

Exceptions on setting
DOMException NO_MODIFICATION_ALLOWED_ERR: Raised on an attempt to change the value of a readonly attribute.

Methods

getTrait
Returns the computed value for the trait with the requested localName.
Parameters
• in DOMString localName. The requested trait's local name
Return value
• DOMString the trait's computed value.
Exceptions
• DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to a DOMString (SVG Tiny only). • DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or the localName is NULL.

setTrait
Sets the specified value for the trait with the given localName.
Parameters
• in DOMString localName. The trait's local name • in DOMString value. The trait's new specified value.
No return value
Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as a DOMString.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if the localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getTraitNS

Returns the computed value for the trait with the requested localName and namespaceURI.

Parameters

- in DOMString namespaceURI. The requested trait's namespaceURI.
- in DOMString localName. The requested trait's local name

Return value

- DOMString the trait's computed value.

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to a DOMString (SVG Tiny only).
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or if localName is NULL.

setTraitNS

Sets the specified value for the trait with the given localName and namespaceURI

Parameters

- in DOMString namespaceURI. The trait's namespace URI
- in DOMString localName. The trait's local name
- in DOMString value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as a DOMString.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if the localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getFloatTrait

Returns the computed value for the trait with the requested localName.

Parameters

- in DOMString localName. The requested trait's local name

Return value

- float the trait's computed value.

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to a float (SVG Tiny only).
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or if localName is NULL.

setFloatTrait

Sets the specified value for the trait with the given localName.

Parameters

- in DOMString localName. The trait's local name
- in float value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as a float.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getMatrixTrait

Returns the computed value for the trait with the requested localName.

Parameters

- in DOMString localName. The requested trait's local name

Return value

- SVGMatrix the trait's computed value.

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to an SVGMatrix (SVG Tiny only).
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or if localName is NULL.

setMatrixTrait

Sets the specified value for the trait with the given localName.

Parameters

- in DOMString localName. The trait's local name
- in SVGMatrix value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as an SVGMatrix.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getRectTrait

Returns the computed value for the trait with the requested localName.

Parameters

- in DOMString localName. The requested trait's local name

Return value

- SVGRect the trait's computed value.

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to a SVGRect (SVG Tiny only).
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or if localName is NULL.

setRectTrait

Sets the specified value for the trait with the given localName.

Parameters

- in DOMString localName. The trait's local name
- in SVGRect value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as an SVGRect.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getPathTrait

Returns the computed value for the trait with the requested localName.

Parameters

- in DOMString localName. The requested trait's local name

Return value

- SVGPath the trait's computed value.

Exceptions

- DOMException with error code NOT_SUPPORTED_ERR if the requested trait's computed value cannot be converted to an SVGPath or if the requested trait is not supported on this element or if localName is NULL.

setPathTrait

Sets the specified value for the trait with the given localName.

Parameters

- in DOMString localName. The trait's local name
- in SVGPath value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as an SVGPath.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

getRGBColorTrait

Returns the computed value for the trait with the requested localName.

Parameters

- in DOMString localName. The requested trait's local name

Return value

- SVGRGBColor the trait's computed value.

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if requested trait's computed value cannot be converted to an SVGRGBColor (SVG Tiny only).
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element, or if localName is NULL.

setRGBColorTrait

Sets the specified value for the trait with the given localName.

Parameters

- in DOMString localName. The trait's local name
- in SVGRGBColor value. The trait's new specified value.

No return value

Exceptions

- DOMException with error code TYPE_MISMATCH_ERR if the requested trait's value cannot be specified as an SVGRGBColor.
- DOMException with error code NOT_SUPPORTED_ERROR if the requested trait is not supported on this element.
- DOMException with error code INVALID_ACCESS_ERR if the input value is an invalid value for the given trait, or if localName is NULL.
- DOMException with error code NO_MODIFICATION_ALLOWED_ERR if attempt is made to change readonly trait.

ElementTraversal

This interface provides a way to traverse elements in the DOM tree. It is needed mainly because SVG Tiny DOM does not expose character data Nodes. Each element in SVG Tiny document tree implements this interface.

This applies to elements in the foreign namespaces as well.

IDL Definition

```
interface ElementTraversal
{
        readonly attribute Element firstElementChild;
        readonly attribute Element lastElementChild;
        readonly attribute Element nextElementSibling;
        readonly attribute Element previousElementSibling;
};
```

Attributes

readonly Element firstElementChild
First child element node of this element.

readonly Element lastElementChild
Last child element node of this element.

readonly Element nextElementSibling
Next sibling element node of this element.

readonly Element previousElementSibling
Previous sibling element node of this element.

SVGDocument

`SVGDocument` is the interface for the SVG document.

IDL Definition

```
interface SVGDocument : dom::Document
{
        readonly attribute SVGGlobal global;
};
```

SVGElement

`SVGElement` is the base interface used by all elements in the SVG namespace.

An element's id can be set only if it does not already have an id. DOMException with error code NO_MODIFICATION_ALLOWED_ERR is raised if an attempt is made to change an existing id. Elements with non-null id can be inserted, but cannot be removed from the DOM tree (see removeChild).

IDL Definition

```
interface SVGElement : dom::Element, ElementTraversal, events::EventTarget, TraitAccess
```

```
{
        attribute DOMString id; // raises (DOMException) on setting
};
```

SVGLocatableElement

`SVGLocatableElement` is the base interface used by all graphics and container elements in the SVG namespace.

IDL Definition

```
interface SVGLocatableElement : SVGElement, SVGLocatable
{
};
```

SVGAnimationElement

`SVGAnimationElement` is the base interface used by all animation elements in the SVG namespace.

IDL Definition

```
interface SVGAnimationElement : SVGElement, smil::ElementTimeControl
{
};
```

SVGGlobal

`SVGGlobal` is the interface that provides (among other things) access to a global object for scripts embedded in an SVG document.

IDL Definition

```
interface SVGGlobal : global::Global
{
        global::Connection createConnection();
        void gotoLocation( in DOMString newURI);
        readonly attribute document document;
        readonly attribute global::Global parent;
};
```

SVGSVGElement

SVGSVGElement provides an API to access APIs corresponding to the <svg> element

The DOM attributes currentScale, currentRotate and currentTranslate are equivalent to the 2x3 matrix [a b c d e f] = [currentScale 0 0 currentScale currentTranslate.x currentTranslate.y].[cos(currentRotate) sin(currentRotate) -sin(currentRotate cos(currentRotate) 0 0]. If "magnification" is enabled (i.e., zoomAndPan="magnify"), then the effect is as if an extra transformation were placed at the outermost level on the SVG document fragment (i.e., outside the outermost 'svg' element).

IDL Definition

```
interface SVGSVGElement : SVGLocatableElement
{
        attribute float currentScale; // raises (DOMException) on setting
        attribute float currentRotate;
        readonly attribute SVGPoint currentTranslate;

        readonly attribute SVGRect viewport;

        void            pauseAnimations();
        void            unpauseAnimations();
        boolean         animationsPaused();
        float           getCurrentTime();
        void            setCurrentTime( in float seconds );

        SVGMatrix       createSVGMatrixComponents( float a, float b, float c, float d,
float e, float f );
        SVGRect         createSVGRect();

        SVGPath     createSVGPath();
        SVGRGBColor     createSVGRGBColor( in unsigned long red, in unsigned long
green, in unsigned long blue )
                                            raises(SVGException);
};
```

No Defined constants

Attributes

readonly SVGRect viewport

The position and size of the viewport (implicit or explicit) that corresponds to this 'svg' element. When the user agent is actually rendering the content, then the position and size values represent the actual values when rendering. The position and size values are unitless values in the coordinate system of the parent element. If no parent element exists (i.e., 'svg' element represents the root of the document tree), if this SVG document is embedded as part of another document (e.g., via the HTML 'object' element), then the position and size are unitless values in the coordinate system of the parent document. (If the parent uses CSS or XSL layout, then unitless values represent pixel units for the current CSS or XSL viewport, as described in the CSS2 specification.) If the parent element does not have a coordinate system, then the user agent should provide reasonable default values for this attribute.

The object itself and its contents are both readonly.

float currentScale

This attribute indicates the current scale factor relative to the initial view to take into account user magnification and panning operations, as described under Magnification and panning.

Exceptions on setting

DOMException	INVALID_ACCESS_ERR: Raised on an attempt to change the value of currentScale to zero.

float currentRotate

This attribute indicates the current rotation in the user transform, relative to the coordinate system's origin. The value is in degrees.

readonly SVGPoint currentTranslate

The corresponding translation factor that takes into account user magnification and panning.

Methods

pauseAnimations

Suspends (i.e., pauses) all currently running animations that are defined within the SVG document fragment corresponding to this 'svg' element, causing the animation clock corresponding to this document fragment to stand still until it is unpaused.

No Parameters

No Return Value

No Exceptions

unpauseAnimations

Unsuspends (i.e., unpauses) currently running animations that are defined within the SVG document fragment, causing the animation clock to continue from the time at which it was suspended.

No Parameters

No Return Value

No Exceptions

animationsPaused

Returns true if this SVG document fragment is in a paused state.

No Parameters

Return value

boolean	Boolean indicating whether this SVG document fragment is in a paused state.

No Exceptions

getCurrentTime

Returns the current time in seconds relative to the start time for the current SVG document fragment.

No Parameters

Return value

float	The current time in seconds.

No Exceptions

setCurrentTime

Adjusts the clock for this SVG document fragment, establishing a new current time.

Parameters

in float seconds	The new current time in seconds relative to the start time for the current SVG document fragment.

No Return Value

No Exceptions

createSVGRect

Creates an SVGRect object.

No Parameters

Return value

SVGRect	An SVGRect object with x, y, width and height all initialized to zero.

No Exceptions

EventListenerInitializer

Interface used to set up event initializers, typically at document load time.

The EventListenerInitializer interface needs to be implemented by scripts written in Java (or other compiled languages). It is called when code is loaded and can be bound to a document.

IDL Definition

```
interface EventListenerInitializer
{
        void initializeEventListeners(dom::Document doc);
};
```

The use of this interface in the Java environment, with jar files and manifests, is not yet defined.

A.8 Traits supported in the SVG Tiny 1.2 DOM

Proposed list of supported traits

The table below shows the list of attributes and properties that Tiny DOM implementations must support. Each light gray section lists one or multiple elements for which the subsequent attributes or properties apply. Each attribute row lists the allowed getter and setter (s). This table is not normative.

Property	Trait Getter	Trait Setter
<svg>, <rect>, <circle>, <ellipse>, <line>, <path>, <g>, <image>, <text>, and <a>		
color	getRGBColorTrait	setTrait [inherit] setRGBColorTrait [SVGRGBColor]
display	getTrait	setTrait [inline | none | inherit]
fill	getRGBColorTrait	setRGBColorTrait [null | SVGRGBColor] setTrait(none | currentColor | inherit)
fill-rule	getTrait	setTrait(nonzero | evenodd | inherit)
stroke	getRGBColorTrait [null, SVGRGBColor]	setRGBColorTrait [null | SVGRGBColor] setTrait(none | currentColor | inherit)
stroke-dashoffset	getFloatTrait	setTrait [inherit] setFloatTrait

stroke-linecap	getTrait [butt | round | square]	setTrait [butt | round | square | inherit]
stroke-linejoin	getTrait [miter | round | bevel]	setTrait [miter | round | bevel | inherit]
stroke-miterlimit	getFloatTrait [value >= 1]	setTrait [inherit] setFloatTrait [value >= 1]
stroke-width	getFloatTrait [value >= 0]	setTrait [inherit] setFloatTrait [value >= 0]
visibility	getTrait [visible | hidden | collapse]	setTrait [visible | hidden | collapse | inherit]
<svg>, <text>, <g>, <a>		
font-familly	getTrait [single, computed font-family value]	setTrait [same syntax as font-family attribute]
font-size	getFloatTrait [value >= 0]	setFloatTrait [value >= 0] setTrait [inherit]
font-style	getTrait [normal | italic | oblique]	setTrait [normal | italic | oblique | inherit]
font-weight	getTrait [100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900]	setTrait [normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit]
text-anchor	getTrait [start | middle | end]	setTrait [start | middle | end | inherit]
Attribute	**Trait Getter**	**Trait Setter**
<rect>, <circle>, <ellipse>, <line>, <path>, <g>, <image>, <text>, and <a>		
transform	getMatrixTrait	setMatrixTrait
<circle>		
cx	getFloatTrait	setFloatTrait
cy	getFloatTrait	setFloatTrait
r	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
<ellipse>		
cx	getFloatTrait	setFloatTrait
cy	getFloatTrait	setFloatTrait
rx	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
ry	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
<path> (path-length is not supported)		
d	getPathTrait [non null value]	setPathTrait [non null value]

<rect>		
height	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
width	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
x	getFloatTrait	setFloatTrait
y	getFloatTrait	setFloatTrait
rx	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
ry	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
<image>		
x	getFloatTrait	setFloatTrait
y	getFloatTrait	setFloatTrait
width	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
height	getFloatTrait [value >= 0]	setFloatTrait [value >= 0]
xlink:href	getTrait NS[absolute URI, factoring in xml:base]	setTraitNS [non local-URI value]
<a>		
target	getTrait	setTrait
xlink:href	getTraitNS[absolute URI, factoring in xml:base]	setTraitNS
<text> (Notes: For 'x' and 'y', it is only possible to provide floating point scalar values; an array of x or y values is not supported. 'rotate' attribute is not supported.)		
x	getFloatTrait	setFloatTrait
y	getFloatTrait	setFloatTrait
#text	getTrait [not null]	setTrait [not null]
<svg>		
version	Readonly, throws DOMException of type NO_MODIFICATION_ALLOWED_ERR	getTrait
baseProfile	Readonly, throws DOMException of type NO_MODIFICATION_ALLOWED_ERR	getTrait
viewBox	setRectTrait	getRectTrait
zoomAndPan	setTrait [disable | magnify]	getTrait [disable | magnify]
<line>		
x1	setFloatTrait	getFloatTrait
x2	setFloatTrait	getFloatTrait
y1	setFloatTrait	getFloatTrait

y2	setFloatTrait	getFloatTrait

A.9 IDL

IDL Definition

```
pragmas
{
  java.jni.api.name="org.w3c.dom";

  core.package.vendor="W3C";
  core.package.name="SVG Tiny";
  core.package.id="svgt";
};

[
  comment="subsetted Core DOM";
  java.jni.api.name="org.w3c.dom";
]
module dom
{
    typedef string DOMString;

    interface Node;
    interface Element;
    interface Document;

    exception DOMException
    {
        unsigned short code;
    };

    const unsigned short WRONG_DOCUMENT_ERR = 4;
    const unsigned short INDEX_SIZE_ERR = 1;
    const unsigned short HIERARCHY_REQUEST_ERR = 3;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 7;
    const unsigned short NOT_FOUND_ERR = 8;
    const unsigned short NOT_SUPPORTED_ERR = 9;
    const unsigned short INVALID_STATE_ERR = 11;
    const unsigned short INVALID_MODIFICATION_ERR = 13;
    const unsigned short INVALID_ACCESS_ERR = 15;
    const unsigned short TYPE_MISMATCH_ERR = 17;

    interface Node
    {
        readonly attribute DOMString namespaceURI;
        readonly attribute DOMString localName;
        readonly attribute Node parentNode;

        Node appendChild(in Node newChild) raises(DOMException);
        Node insertBefore(in Node newChild, in Node refChild) raises(DOMException);
        Node removeChild(in Node oldChild) raises(DOMException);
    };

    interface Element : Node
    {
    };
```

```
    interface Document : Node
    {
        Element createElementNS(in DOMString namespaceURI, in DOMString qualifiedName)
raises(DOMException);
        readonly attribute Element documentElement;
        Element getElementById(in DOMString id);
    };

};

module events
{
    typedef dom::DOMString DOMString;
    typedef dom::DOMException DOMException;
    typedef dom::Document Document;
    typedef dom::Element Element;

    interface EventTarget;
    interface EventListener;
    interface Event;

    interface EventTarget
    {
        void addEventListener(in DOMString type, in EventListener listener, in boolean
useCapture);
        void removeEventListener(in DOMString type, in EventListener listener, in
boolean useCapture);
    };

    interface EventListener
    {
        void handleEvent(in Event evt);
    };

    interface Event
    {
        readonly attribute EventTarget currentTarget;
        readonly attribute DOMString type;
    };

    interface MouseEvent : Event
    {
        readonly attribute long screenX;
        readonly attribute long screenY;
        readonly attribute long clientX;
        readonly attribute long clientY;
        readonly attribute unsigned short button;
    };

    interface TextEvent : Event {
        readonly attribute DOMString data;
    };

    interface KeyboardEvent : Event {
        readonly attribute DOMString keyIdentifier;
    };

    interface ConnectionEvent : Event {
            readonly attribute DOMString data;
    };
};

module smil
{
```

```
    interface ElementTimeControl
    {
        void beginElementAt(in float offset);
        void beginElement();
        void endElementAt(in float offset);
        void endElement();
        void pauseElement();
        void unpauseElement();
        readonly attribute boolean elementPaused;
    };
};

module global
{
    interface Connection;

    interface Global {};

    interface Connection : events::EventTarget
    {
        typedef dom::DOMString DOMString;
        typedef dom::DOMException DOMException;
        void connect(in DOMString uri) raises(DOMException);
        void send(in DOMString data);
        void close();
        readonly attribute boolean connected;
    };
};

module svg
{
    typedef dom::DOMString DOMString;
    typedef dom::DOMException DOMException;
    typedef dom::Document Document;
    typedef dom::Element Element;

    interface SVGSVGElement;
    interface SVGRGBColor;
    interface SVGRect;
    interface SVGPoint;
    interface SVGPath;
    interface SVGMatrix;
    interface SVGLocatableElement;
    interface SVGElement;
    interface SVGAnimationElement;
    interface SVGDocument;
    interface SVGGlobal;

    exception SVGException
    {
        unsigned short code;
    };

    const unsigned short SVG_INVALID_VALUE_ERR = 1;
    const unsigned short SVG_MATRIX_NOT_INVERTABLE = 2;

    interface SVGDocument : Document
    {
        readonly attribute SVGGlobal global;
    };

    interface SVGSVGElement : SVGLocatableElement
    {
        attribute float currentScale;
```

```
        attribute float currentRotate;
        readonly attribute SVGPoint currentTranslate;

        readonly attribute SVGRect viewport;

        void        pauseAnimations();
        void        unpauseAnimations();
        boolean     animationsPaused();
        attribute float currentTime;

        SVGMatrix createSVGMatrixComponents(in float a, in float b, in float c, in
float d, in float e, in float f);
        SVGRect createSVGRect();
        SVGPath createSVGPath();
        SVGRGBColor createSVGRGBColor(in long red, in long green, in long blue) raises
(SVGException);
    };

    interface SVGRGBColor
    {
        readonly attribute unsigned long red;
        readonly attribute unsigned long green;
        readonly attribute unsigned long blue;
    };

    interface SVGRect
    {
        attribute float x;
        attribute float y;
        attribute float width;
        attribute float height;
    };

    interface SVGPoint
    {
        attribute float x;
        attribute float y;
    };

    interface SVGPath
    {
        const unsigned short MOVE_TO = 77;
        const unsigned short LINE_TO = 76;
        const unsigned short CURVE_TO = 67;
        const unsigned short QUAD_TO = 81;
        const unsigned short CLOSE = 90;

        readonly attribute unsigned long numberOfSegments;

        unsigned short getSegment(in unsigned long cmdIndex) raises(DOMException);
        float getSegmentParam(in unsigned long cmdIndex, in unsigned long paramIndex)
raises(DOMException);

        void moveTo(in float x, in float y);
        void lineTo(in float x, in float y);
        void quadTo(in float x1, in float y1, in float x2, in float y2);
        void curveTo(in float x1, in float y1, in float x2, in float y2, in float x3,
in float y3);
        void close();
    };

    interface SVGMatrix
    {
        float getComponent(in unsigned long index) raises(DOMException);
```

```
        SVGMatrix mMultiply(in SVGMatrix secondMatrix);
        SVGMatrix mInverse() raises(SVGException);
        SVGMatrix mTranslate(in float x, in float y);
        SVGMatrix mScale(in float scaleFactor);
        SVGMatrix mRotate(in float angle);
    };

    interface SVGLocatable
    {
        SVGRect   getBBox();
        SVGMatrix getScreenCTM();
        SVGRect   getScreenBBox();
    };

    interface SVGLocatableElement : SVGElement, SVGLocatable
    {
    };

    interface TraitAccess
    {
        DOMString getTrait(in DOMString name) raises(DOMException);
        DOMString getTraitNS(in DOMString namespaceURI, in DOMString name) raises
(DOMException);
        float getFloatTrait(in DOMString name) raises(DOMException);
        SVGMatrix getMatrixTrait(in DOMString name) raises(DOMException);
        SVGRect getRectTrait(in DOMString name) raises(DOMException);
        SVGPath getPathTrait(in DOMString name) raises(DOMException);
        SVGRGBColor getRGBColorTrait(in DOMString name) raises(DOMException);

        void setTrait(in DOMString name, in DOMString value) raises(DOMException);
        void setTraitNS(in DOMString namespaceURI, in DOMString name, in DOMString
value) raises(DOMException);
        void setFloatTrait(in DOMString name, in float value) raises(DOMException);
        void setMatrixTrait(in DOMString name, in SVGMatrix matrix) raises
(DOMException);
        void setRectTrait(in DOMString name, in SVGRect rect) raises(DOMException);
        void setPathTrait(in DOMString name, in SVGPath path) raises(DOMException);
        void setRGBColorTrait(in DOMString name, in SVGRGBColor color) raises
(DOMException);
    };

    interface ElementTraversal
    {
        readonly attribute Element firstElementChild;
        readonly attribute Element lastElementChild;
        readonly attribute Element nextElementSibling;
        readonly attribute Element previousElementSibling;
    };

    interface SVGElement : dom::Element, events::EventTarget, TraitAccess,
ElementTraversal
    {
        attribute DOMString id;
    };

    interface SVGAnimationElement : SVGElement, smil::ElementTimeControl
    {
    };

    interface EventListenerInitializer
    {
        void initializeEventListeners( in SVGDocument doc);
    };
```

```
    interface EventListenerInitializer2
    {
        void initializeEventListeners( in dom::Element scriptElement );
        events::EventListener createEventListener( in dom::Element handlerElement );
    };

    interface SVGGlobal : global::Global
    {
       global::Connection createConnection();
       void gotoLocation(in DOMString newURI);
       readonly attribute Document document;
       readonly attribute global::Global parent;
    };


};
```

Previous | Top | Next

Appendix D: Feature strings

This appendix is normative.

The following are the feature strings for the requiredFeatures attribute. These same feature strings apply to the hasFeature method call that is part of the SVG DOM's support for the DOMImplementation interface defined in [DOM3-CORE] (see Feature strings for the **hasFeature** method call). In some cases the feature strings map directly to SVG modules, in others they represent some functionality of the User Agent (that it is a dynamic viewer for example).

Feature String:
http://www.w3.org/TR/SVG12/feature#SVG
User Agent Supports:
At least one of the following (all of which are described subsequently): "http://www.w3.org/TR/SVG12/feature#SVG-static", "http://www.w3.org/TR/SVG12/feature#SVG-animation", "http://www.w3.org/TR/SVG12/feature#SVG-dynamic" or "http://www.w3.org/TR/SVG12/feature#SVGDOM". (Because the feature string "http://www.w3.org/TR/SVG12/feature#SVG" can be ambiguous in some circumstances, it is recommended that more specific feature strings be used.)

Feature String:
http://www.w3.org/TR/SVG12/feature#SVGDOM
User Agent Supports:
At least one of the following (all of which are described subsequently): "http://www.w3.org/TR/SVG12/feature#SVGDOM-static", "http://www.w3.org/TR/SVG12/feature#SVGDOM-animation" or "http://www.w3.org/TR/SVG12/feature#SVGDOM-dynamic". (Because the feature string "http://www.w3.org/TR/SVG12/feature#SVGDOM" can be ambiguous in some circumstances, it is recommended that more specific feature strings be used.)

Feature String:

http://www.w3.org/TR/SVG12/feature#SVG-static

User Agent Supports:

The following features (described below)

- http://www.w3.org/TR/SVG12/feature#CoreAttribute
- http://www.w3.org/TR/SVG12/feature#Structure
- http://www.w3.org/TR/SVG12/feature#ContainerAttribute
- http://www.w3.org/TR/SVG12/feature#ConditionalProcessing
- http://www.w3.org/TR/SVG12/feature#Image
- http://www.w3.org/TR/SVG12/feature#Style
- http://www.w3.org/TR/SVG12/feature#ViewportAttribute
- http://www.w3.org/TR/SVG12/feature#Shape
- http://www.w3.org/TR/SVG12/feature#Text
- http://www.w3.org/TR/SVG12/feature#PaintAttribute
- http://www.w3.org/TR/SVG12/feature#OpacityAttribute
- http://www.w3.org/TR/SVG12/feature#GraphicsAttribute
- http://www.w3.org/TR/SVG12/feature#Marker
- http://www.w3.org/TR/SVG12/feature#ColorProfile
- http://www.w3.org/TR/SVG12/feature#Gradient
- http://www.w3.org/TR/SVG12/feature#Pattern
- http://www.w3.org/TR/SVG12/feature#Clip
- http://www.w3.org/TR/SVG12/feature#Mask
- http://www.w3.org/TR/SVG12/feature#Filter
- http://www.w3.org/TR/SVG12/feature#XlinkAttribute
- http://www.w3.org/TR/SVG12/feature#Font
- http://www.w3.org/TR/SVG12/feature#Extensibility

For SVG viewers, "http://www.w3.org/TR/SVG12/feature#SVG-static" indicates that the viewer can process and render successfully all of the language features in the modules corresponding to the features listed above.

Feature String:

http://www.w3.org/TR/SVG12/feature#SVGDOM-static

User Agent Supports:

All of the DOM interfaces and methods that correspond to the language features for "http://www.w3.org/TR/SVG12/feature#SVG-static".

Feature String:

http://www.w3.org/TR/SVG12/feature#SVG-animation

User Agent Supports:

All of the language features from "http://www.w3.org/TR/SVG12/feature#SVG-static" plus the feature "http://www.w3.org/TR/SVG12/feature#Animation". For SVG viewers running on media capable of rendering time-based material, such as displays, "http://www.w3.org/TR/

SVG12/feature#SVG-animation" indicates that the viewer can process and render successfully all of the corresponding language features.

Feature String:

http://www.w3.org/TR/SVG12/feature#SVGDOM-animation

User Agent Supports:

All of the DOM interfaces and methods that correspond to the language features for "http://www.w3.org/TR/SVG12/feature#SVG-animation".

Feature String:

http://www.w3.org/TR/SVG12/feature#SVG-dynamic

User Agent Supports:

All of the language features from "http://www.w3.org/TR/SVG12/feature#SVG-animation" plus the following features:

- http://www.w3.org/TR/SVG12/feature#Hyperlinking
- http://www.w3.org/TR/SVG12/feature#Scripting
- http://www.w3.org/TR/SVG12/feature#View
- http://www.w3.org/TR/SVG12/feature#Cursor
- http://www.w3.org/TR/SVG12/feature#GraphicalEventsAttribute
- http://www.w3.org/TR/SVG12/feature#DocumentEventsAttribute
- http://www.w3.org/TR/SVG12/feature#AnimationEventsAttribute

For SVG viewers running on media capable of rendering time-based material, such as displays, "http://www.w3.org/TR/SVG12/feature#SVG-dynamic" indicates that the viewer can process and render successfully all of the corresponding language features.

Feature String:

http://www.w3.org/TR/SVG12/feature#SVGDOM-dynamic

User Agent Supports:

All of the DOM interfaces and methods that correspond to the language features for "http://www.w3.org/TR/SVG12/feature#SVG-dynamic".

Feature String:

http://www.w3.org/TR/SVG12/feature#CoreAttribute

User Agent Supports:

Core Attribute Module

Feature String:

http://www.w3.org/TR/SVG12/feature#Structure

User Agent Supports:

Structure Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicStructure
User Agent Supports:
Basic Structure Module

Feature String:
http://www.w3.org/TR/SVG12/feature#ContainerAttribute
User Agent Supports:
Container Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#ConditionalProcessing
User Agent Supports:
Conditional Processing Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Image
User Agent Supports:
Image Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Style
User Agent Supports:
Style Module

Feature String:
http://www.w3.org/TR/SVG12/feature#ViewportAttribute
User Agent Supports:
Viewport Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Shape
User Agent Supports:
Shape Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Text
User Agent Supports:
Text Module

Feature String:

http://www.w3.org/TR/SVG12/feature#BasicText
User Agent Supports:
Basic Text Module

Feature String:
http://www.w3.org/TR/SVG12/feature#PaintAttribute
User Agent Supports:
Paint Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicPaintAttribute
User Agent Supports:
Basic Paint Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#OpacityAttribute
User Agent Supports:
Opacity Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#GraphicsAttribute
User Agent Supports:
Graphics Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicGraphicsAttribute
User Agent Supports:
Basic Graphics Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Marker
User Agent Supports:
Marker Module

Feature String:
http://www.w3.org/TR/SVG12/feature#ColorProfile
User Agent Supports:
Color Profile Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Gradient

User Agent Supports:
Gradient Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Pattern
User Agent Supports:
Pattern Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Clip
User Agent Supports:
Clip Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicClip
User Agent Supports:
Basic Clip Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Mask
User Agent Supports:
Mask Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Filter
User Agent Supports:
Filter Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicFilter
User Agent Supports:
Basic Filter Module

Feature String:
http://www.w3.org/TR/SVG12/feature#DocumentEventsAttribute
User Agent Supports:
Document Events Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#GraphicalEventsAttribute
User Agent Supports:

Graphical Events Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#AnimationEventsAttribute
User Agent Supports:
Animation Events Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Cursor
User Agent Supports:
Cursor Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Hyperlinking
User Agent Supports:
Hyperlinking Module

Feature String:
http://www.w3.org/TR/SVG12/feature#XlinkAttribute
User Agent Supports:
Xlink Attribute Module

Feature String:
http://www.w3.org/TR/SVG12/feature#ExternalResourcesRequired
User Agent Supports:
ExternalResourcesRequired Module

Feature String:
http://www.w3.org/TR/SVG12/feature#View
User Agent Supports:
View Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Script
User Agent Supports:
Script Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Animation
User Agent Supports:
Animation Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Font
User Agent Supports:
Font Module

Feature String:
http://www.w3.org/TR/SVG12/feature#BasicFont
User Agent Supports:
Basic Font Module

Feature String:
http://www.w3.org/TR/SVG12/feature#Extensibility
User Agent Supports:
Extensibility Module

Feature String:
http://www.w3.org/TR/SVG12/feature#TinyBase
User Agent Supports:
All the static and declarative animation language capabilities defined for SVG Tiny in Section 2.

Feature String:
http://www.w3.org/TR/SVG12/feature#TinyInteractivity
User Agent Supports:
All the static and declarative animation language capabilities defined for SVG Tiny in Section 2 as well as support for interactivity.

Feature String:
http://www.w3.org/TR/SVG12/feature#TransformedVideo
User Agent Supports:
The ability to perform any transformation (including scaling) on video content.

Previous | Top | Next

Appendix E: List of events

Event Identifier {event-namespace, event-localname}	Description	DOM3 event category	Event attribute name	
{"http://www.w3.org/2001/xml-events", "DOMFocusIn"} SVG 1.2 alias: {"http://www.w3.org/2001/xml-events", "focusin"} (see Notes below).	Occurs when an element receives keyboard focus.	UIEvent	onfocusin	
{"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 keyboard focus.	UIEvent	onfocusout	
{"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	onactivate	

{"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	onclick
{"http://www.w3.org/2001/xml-events", "mousedown"}	Occurs when the pointing device button is pressed over an element.	MouseEvent	onmousedown
{"http://www.w3.org/2001/xml-events", "mouseup"}	Occurs when the pointing device button is released over an element.	MouseEvent	onmouseup
{"http://www.w3.org/2001/xml-events", "mouseover"}	Occurs when the pointing device is moved onto an element.	MouseEvent	onmouseover
{"http://www.w3.org/2001/xml-events", "mousemove"}	Occurs when the pointing device is moved while it is over an element.	MouseEvent	onmousemove
{"http://www.w3.org/2001/xml-events", "mouseout"}	Occurs when the pointing device is moved away from an element.	MouseEvent	onmouseout
{"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", "DOMSubtreeModified"}	This is a general event for notification of all changes to the document. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	
{"http://www.w3.org/2001/xml-events", "DOMNodeInserted"}	Fired when a node has been added as a child of another node. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	

{"http://www.w3.org/2001/xml-events", "DOMNodeRemoved"}	Fired when a node is being removed from another node. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	
{"http://www.w3.org/2001/xml-events", "DOMNodeRemovedFromDocument"}	Fired when a node is being removed from a document, either through direct removal of the Node or removal of a subtree in which it is contained. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	
{"http://www.w3.org/2001/xml-events", "DOMNodeInsertedIntoDocument"}	Fired when a node is being inserted into a document, either through direct insertion of the Node or insertion of a subtree in which it is contained. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	

{"http://www.w3.org/2001/xml-events", "DOMAttrModified"}	Fired after an attribute has been modified on a node. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	
{"http://www.w3.org/2001/xml-events", "DOMCharacterDataModified"}	Fired after CharacterData within a node has been modified but the node itself has not been inserted or deleted. (The normative definition of this event is the description in the DOM3 Events specification.)	MutationEvent	none	
{"http://www.w3.org/2001/xml-events", "DOMElementNameChanged"}	Occurs after the namespaceURI and/or the nodeName of an Element node have been modified (e.g., the element was renamed using Document. renameNode()). (The normative definition of this event is the description in the DOM3 Events specification.)	MutationNameEvent	none	

{"http://www.w3.org/2001/xml-events", "DOMAttributeNameChanged"}	Occurs after the namespaceURI and/or the nodeName of a Attr node have been modified (e.g., the attribute was renamed using Document.renameNode). (The normative definition of this event is the description in the DOM3 Events specification.)	MutationNameEvent	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	onload	

{"http://www.w3.org/2001/xml-events", "unload"} Deprecated backwards-compatibility alias: {"http://www.w3.org/2001/xml-events", "SVGUnload"} (see Notes below).	Only applicable to outermost 'svg' elements. The unload event occurs when the DOM implementation removes a document from a window or frame.	HTMLEvent	onunload	
{"http://www.w3.org/2001/xml-events", "abort"} Deprecated backwards-compatibility alias: {"http://www.w3.org/2001/xml-events", "SVGAbort"} (see Notes below).	The abort event occurs when page loading is stopped before an element has been allowed to load completely.	HTMLEvent	onabort	
{"http://www.w3.org/2001/xml-events", "error"} Deprecated backwards-compatibility alias: {"http://www.w3.org/2001/xml-events", "SVGError"} (see Notes below).	The error event occurs when an element does not load properly or when an error occurs during script execution.	HTMLEvent	onerror	
{"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	onresize	

{"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	onscroll	
{"http://www.w3.org/2000/svg", "selection"}	Current text selection changes.	none	none	
{"http://www.w3.org/2001/xml-events", "change"}	An element loses the input focus and its value has been modified since gaining focus.	HTMLEvent	none	
{"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	DOM3's SVG Events	onzoom	

	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.			
{"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	onbegin	
{"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	onend	
{"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	onrepeat	

{"http://www.w3.org/2000/svg", "overflow"}	Occurs when a region which can contain a content flow goes from a non-overflow state into an overflow state (i. e., previously, content did not overflow the container, but now content does overflow the container).	none	none	
{"http://www.w3.org/2000/svg", "underflow"}	Occurs when a region which can contain a content flow goes from a overflow state into a non-overflow state (i. e., previously, content overflowed the container, but now content does not overflow the container).	none	none	
{"http://www.w3.org/2001/xml-events", "wheel"}	Occurs when a rotational input device has been activated.	UIEvent	none	
{"http://www.w3.org/2000/svg", "shapechange"}	Occurs when a path or basic shape has its geometry modified.	none	none	
{"http://www.w3.org/2000/svg", "renderedbboxchange"}	Occurs when the bounding box of a shape's rendered output is modified.	none	none	
{"http://www.w3.org/2000/svg", "traitvaluechanged"}	Occurs when the base value of a trait is modified.	none	none	

{"http://www.w3.org/2000/svg", "traitanimvaluechanged"}	Occurs when the base value of a trait is modified.	none	none	
{"http://www.w3.org/2000/svg", "URLResponse"}	Occurs when a response arrives from a web server after sending a URLRequest.	none	none	
{"http://www.w3.org/2000/svg", "timer"}	A timer event.	none	none	
{"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	
{"http://www.w3.org/2000/svg", "filesSelected"}	Raised when a user completes the operation of selecting a list of files.	none	none	
{"http://www.w3.org/2004/xbl", "prebind"}	A new XBL binding is in the process of being attached to the event target. For the normative definition of this event, refer to the sXBL specification.	none	none	
{"http://www.w3.org/2004/xbl", "bound"}	A new XBL binding has been attached to the event target. For the normative definition of this event, refer to the sXBL specification.	none	none	

{"http://www.w3.org/2004/xbl", "unbinding"}	A new XBL binding is in the process of being attached to the event target. For the normative definition of this event, refer to the sXBL specification.	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"}
  - "SVGUnload" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "unload"}
  - "SVGAbort" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "abort"}
  - "SVGError" event is deprecated in favor of {"http://www.w3.org/2001/xml-events", "error"}
  - "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
  - "unload" and "SVGUnload" are aliases for each other
  - "abort" and "SVGAbort" are aliases for each other
  - "error" and "SVGError" 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

Appendix F: RelaxNG schema for SVG 1.2

A modularized RelaxNG schema for SVG 1.2 is available.

Previous | Top | Next

Appendix G: Media Type registration for image/svg+xml

This appendix is normative.

G.1 Introduction

This appendix registers a new MIME media type, "image/svg+xml" in conformance with RegMedia and W3CRegMedia.

G.2 Registration of Media Type image/svg+xml

MIME media type name:

image

MIME subtype name:

svg+xml

Required parameters:

None.

Optional parameters:

None

The encoding of an SVG document is determined by the XML encoding declaration. This has identical semantics to the application/xml media type in the case where the charset parameter is omitted, as specified in

RFC3023 sections 8.9, 8.10 and 8.11.

Encoding considerations:

Same as for application/xml. See RFC3023 , section 3.2.

Restrictions on usage:

None

Security considerations:

As with other XML types and as noted in RFC3023 section 10, repeated expansion of maliciously constructed XML entities can be used to consume large amounts of memory, which may cause XML processors in constrained environments to fail.

SVG documents may be transmitted in compressed form using gzip compression. For systems which employ MIME-like mechanisms, such as HTTP, this is indicated by the Content-Transfer-Encoding header; for systems which do not, such as direct filesystem access, this is indicated by the filename extension and by the Macintosh File Type Codes. In addition, gzip compressed content is readily recognised by the initial byte sequence as described in RFC1952 section 2.3.1.

Several SVG elements may cause arbitrary URIs to be referenced. In this case, the security issues of RFC2396, section 7, should be considered.

In common with HTML, SVG documents may reference external media such as images, audio, video, style sheets, and scripting languages. Scripting languages are executable content. In this case, the security considerations in the Media Type registrations for those formats apply.

In addition, because of the extensibility features for SVG and of XML in general, it is possible that "image/svg+xml" may describe content that has security implications beyond those described here. However, if the processor follows only the normative semantics of this specification, this content will be outside the SVG namespace and will be ignored. Only in the case where the processor recognizes and processes the additional content, or where further processing of that content is dispatched to other processors, would security issues potentially arise. And in that case, they

would fall outside the domain of this registration document.

Interoperability considerations:

This specification describes processing semantics that dictate behavior that must be followed when dealing with, among other things, unrecognized elements and attributes, both in the SVG namespace and in other namespaces.

Because SVG is extensible, conformant "image/svg+xml" processors must expect that content received is well-formed XML, but it cannot be guaranteed that the content is valid to a particular DTD or Schema or that the processor will recognize all of the elements and attributes in the document.

SVG has a published Test Suite and associated implementation report showing which implementations passed which tests at the time of the report. This information is periodically updated as new tests are added or as implementations improve.

Published specification:

This media type registration is extracted from Appendix G of the SVG 1.2 specification.

Additional information:

Person & email address to contact for further information:

Dean Jackson, (dean@w3.org).

Intended usage:

COMMON

Author/Change controller:

The SVG specification is a work product of the World Wide Web Consortium's SVG Working Group. The W3C has change control over these specifications.

Previous | Top | Next

Appendix H: References

DOM

Document Object Model (DOM): Level 2 Core, Arnaud Le Hors et al editors, W3C, See http://www.w3.org/TR/DOM-Level-2-Core/

CSS21

Cascading Style Sheets, level 2 revision 1, Bert Bos, Tantek Celik, Ian Hickson, Hakon Wium Lie editors, W3C, See http://www.w3.org/TR/CSS21

CSS3Color

CSS3 Color Module, Tantek Celik, Chris Lilley editors, W3C, See http://www.w3.org/TR/css3-color

HTML

HTML 4.01 Specification, Dave Raggett, Arnaud Le Hors, Ian Jacobs editors, W3C, See http://www.w3.org/TR/html401/

RFC1952

RFC 1952, GZIP file format specification version 4.3. P. Deutsch, See http://www.ietf.org/rfc/rfc1952.txt

RFC2396

RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee, R. Fielding, L. Masinter see http://www.ietf.org/rfc/rfc2396.txt

RFC3023

RFC 3023, XML Media Types. M. Murata, S. St.Laurent, D. Kohn, see http://www.ietf.org/rfc/rfc3023.txt

SMIL20

Synchronized Multimedia Integration Language (SMIL 2.0), Jeff Ayars et al editors, W3C, See http://www.w3.org/TR/smil20/

SVG11

Scalable Vector Graphics (SVG) 1.1 Specification, Jon Ferraiolo, Jun Fujisawa, Dean Jackson editors, W3C, See http://www.w3.org/TR/SVG11/

SVG2Reqs

SVG 1.1/1.2/2.0 Requirements, Dean Jackson editor, W3C, See http://

www.w3.org/TR/SVG2Reqs/

SVGPrint

SVG Print, Alex Danilo editor, W3C, See http://www.w3.org/TR/SVGPrint/

SVGPrintReqs

SVG Printing Requirements, Jun Fujisawa, Lee Klosterman Craig Brown, Alex Danilo editors, W3C, See http://www.w3.org/TR/SVGPrintReqs/

AWWW

Architecture of the World Wide Web, First Edition Ian Jacobs, Norman Walsh see http://www.w3.org/TR/webarch/

XForms

XForms 1.0, Micah Dubinko, Leigh Klotz, Roland Merrick, T.V. Raman editors, W3C, See http://www.w3.org/TR/xforms/

XHTMLMod

Modularization of XHTML in XML Schema, Daniel Austin, Shane McCarron, Masayasu Ishikawa editors, W3C, See http://www.w3.org/TR/xhtml-m12n-schema/

XMLEvents

XML Events, Shane McCarron, Steven Pemberton, T.V. Raman editors, W3C, See http://www.w3.org/TR/xml-events/

XPath

XML Path Language (XPath) Version 1.0, James Clark, Steve DeRose editors, W3C, See http://www.w3.org/TR/xpath

Previous | Top

Appendix I: Change log

The following is a simplified list of changes since the last public draft:

- Separation into multiple files (for each chapter)
- Slight rewording in abstract and status introduction.
- Replaced RCC chapter with integration of sXBL.
- New **requiredFormats** test attribute in "Testing for document formats" section.
- New **requiredFonts** test attribute in "Testing for required fonts" section.
- Added new authors Jean-Claude Dufourd (ENST), Suresh Chitturi (Nokia) and Vincent Mahe (France Telecom).
- DOM Enhancements renamed as many of the APIs were not specific to documents, but to applications. They also can be used outside of SVG.
- Various updates to SVGGlobal (formerly SVGWindow), including removal of documentStyleSheet and evt attributes, addition of screen and location attributes, addition of navigation method, addition of mouse capture and merging of existing new meethods into the interface.
- Changed return type of SVGImage::getPixel to SVGColor.
- Traits now throw NO *SUPPORTED*ERROR when name is null.
- SVGSVGElement::createSVGRect now described.
- SVGSVGElement::currentScale can throw an exception when setting the scale to 0.
- SVGSVGElement::currentRotate can no longer throw an exception.
- Much of the Progressive Rendering section was missing in the previous draft. We found it and put it back in.
- Slight clarification of the wording in background-fill. Described relationship of background-fill to other compositing operations.
- Removed rgba() syntax from color specifications.
- Added rules for processing of external document references.
- Prefetch element now compatible with SMIL version.
- Multiple pages **page** and **pageSet**) have a more complete definition, with timing values and expected behaviours.
- Added synchronisation attributes from SMIL 2.
- Added **background-fill-opacity** property.

- Expansion of Vector Effects specifications. Changed "compositing" attribute to "clipout".
- Selection event now part of DOM3 events. New Selection interfaces.
- Better specification of focus and navigation.
- Remove any mention of feature to adjust drawing order (eg. z-index).
- Advanced compositing has equations for background removal.
- More comprehensive specification of transformations, and modification to constrained transformations.
- Removed the note that says **rendering-color-space** was in danger of being dropped.
- Font hinting removed from specification.
- Add "auto" keyword to **textLength**.
- Declarative animation of clock time removes **animateClock**).
- Enhanced ElementTimeControl interface.
- New element, **animation**, for displaying animated vector content.
- Expanded documentation of URLRequest. URLHeader renamed.
- SVGTimer interface simplified, and documentation expanded. SVGTimer is now an Event Target.
- Separation of DOM enhancements from new APIs.
- Change specification of getScreenCTM to return values in the coordinate system used by screenX and screenY. Added a getClientCTM method.
- Removed bogus "left", "right", "before" and "after" values for flowing text alignment (the **text-align** property).
- Removed reference to CSS **vertical-align** property. Added **progression-align** property for block alignment of flowing text.
- Added "overflow" and "underflow" events for flow regions.
- Changed **shadowInherit** from a property into an attribute. Removed "initial" value and added "dynamic" value, with better description of defaults. Also added an example.
- Added overlay examples.
- Modified definition of accessKey animation syntax to coordinate with DOM Level 3 Events.
- SVG user agents are required to support the Ogg Vorbis audio format. SVG user agents are not required to support any particular video format.
- New section on processing user interface events
- Definition of ICC named colors.
- Expansion of specifying paint values to include device colors and icc named colors.
- Added interfaces for flow elements.
- Added list of events.
- Added list of feature strings.
- Added event notification for shape changes, and event notification for rendering bounding box modifications.

- Added wheel event.
- Updated images with multiple resolutions.
- Added Media Type registration for image/svg+xml.
- Two new methods on SVGLocatable for obtaining rendered bounds.
- Updated DOM Subset.

```
pragmas
{
  java.jni.api.name="org.w3c.dom";

  core.package.vendor="W3C";
  core.package.name="SVG Tiny";
  core.package.id="svgt";
};

[
  comment="subsetted Core DOM";
  java.jni.api.name="org.w3c.dom";
]
module dom
{
    typedef string DOMString;

    interface Node;
    interface Element;
    interface Document;

    exception DOMException
    {
        unsigned short code;
    };

    const unsigned short WRONG_DOCUMENT_ERR = 4;
    const unsigned short INDEX_SIZE_ERR = 1;
    const unsigned short HIERARCHY_REQUEST_ERR = 3;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 7;
    const unsigned short NOT_FOUND_ERR = 8;
    const unsigned short NOT_SUPPORTED_ERR = 9;
    const unsigned short INVALID_STATE_ERR = 11;
    const unsigned short INVALID_MODIFICATION_ERR = 13;
    const unsigned short INVALID_ACCESS_ERR = 15;
    const unsigned short TYPE_MISMATCH_ERR = 17;

    interface Node
    {
        readonly attribute DOMString namespaceURI;
        readonly attribute DOMString localName;
        readonly attribute Node parentNode;
```

```
    Node appendChild(in Node newChild) raises(DOMException);
    Node insertBefore(in Node newChild, in Node refChild) raises(DOMException);
    Node removeChild(in Node oldChild) raises(DOMException);
  };

  interface Element : Node
  {
  };

  interface Document : Node
  {
    Element createElementNS(in DOMString namespaceURI, in DOMString qualifiedName) raises
(DOMException);
    readonly attribute Element documentElement;
    Element getElementById(in DOMString id);
  };

};

module events
{
  typedef dom::DOMString DOMString;
  typedef dom::DOMException DOMException;
  typedef dom::Document Document;
  typedef dom::Element Element;

  interface EventTarget;
  interface EventListener;
  interface Event;

  interface EventTarget
  {
    void addEventListener(in DOMString type, in EventListener listener, in boolean useCapture);
    void removeEventListener(in DOMString type, in EventListener listener, in boolean useCapture);
  };

  interface EventListener
  {
    void handleEvent(in Event evt);
  };

  interface Event
```

```
  {
    readonly attribute EventTarget currentTarget;
    readonly attribute DOMString type;
  };

  interface MouseEvent : Event
  {
    readonly attribute long screenX;
    readonly attribute long screenY;
    readonly attribute long clientX;
    readonly attribute long clientY;
    readonly attribute unsigned short button;
  };

  interface TextEvent : Event {
    readonly attribute DOMString data;
  };

  interface KeyboardEvent : Event {
    readonly attribute DOMString keyIdentifier;
  };

  interface ConnectionEvent : Event {
        readonly attribute DOMString data;
  };
};

module smil
{
  interface ElementTimeControl
  {
    void beginElementAt(in float offset);
    void beginElement();
    void endElementAt(in float offset);
    void endElement();
    void pauseElement();
    void unpauseElement();
    readonly attribute boolean elementPaused;
  };
};

module global
{
```

```
  interface Connection;

  interface Global {};

  interface Connection : events::EventTarget
  {
    typedef dom::DOMString DOMString;
    typedef dom::DOMException DOMException;
    void connect(in DOMString uri) raises(DOMException);
    void send(in DOMString data);
    void close();
    readonly attribute boolean connected;
  };
};

module svg
{
  typedef dom::DOMString DOMString;
  typedef dom::DOMException DOMException;
  typedef dom::Document Document;
  typedef dom::Element Element;

  interface SVGSVGElement;
  interface SVGRGBColor;
  interface SVGRect;
  interface SVGPoint;
  interface SVGPath;
  interface SVGMatrix;
  interface SVGLocatableElement;
  interface SVGElement;
  interface SVGAnimationElement;
  interface SVGDocument;
  interface SVGGlobal;

  exception SVGException
  {
    unsigned short code;
  };

  const unsigned short SVG_INVALID_VALUE_ERR = 1;
  const unsigned short SVG_MATRIX_NOT_INVERTABLE = 2;

  interface SVGDocument : Document
```

```
  {
    readonly attribute SVGGlobal global;
  };

  interface SVGSVGElement : SVGLocatableElement
  {
    attribute float currentScale;
    attribute float currentRotate;
    readonly attribute SVGPoint currentTranslate;

    readonly attribute SVGRect viewport;

    void       pauseAnimations();
    void       unpauseAnimations();
    boolean    animationsPaused();
    attribute float currentTime;

    SVGMatrix createSVGMatrixComponents(in float a, in float b, in float c, in float d, in float e, in
float f);
    SVGRect createSVGRect();
    SVGPath createSVGPath();
    SVGRGBColor createSVGRGBColor(in long red, in long green, in long blue) raises
(SVGException);
  };

  interface SVGRGBColor
  {
    readonly attribute unsigned long red;
    readonly attribute unsigned long green;
    readonly attribute unsigned long blue;
  };

  interface SVGRect
  {
    attribute float x;
    attribute float y;
    attribute float width;
    attribute float height;
  };

  interface SVGPoint
  {
    attribute float x;
```

```
    attribute float y;
  };

  interface SVGPath
  {
    const unsigned short MOVE_TO = 77;
    const unsigned short LINE_TO = 76;
    const unsigned short CURVE_TO = 67;
    const unsigned short QUAD_TO = 81;
    const unsigned short CLOSE = 90;

    readonly attribute unsigned long numberOfSegments;

    unsigned short getSegment(in unsigned long cmdIndex) raises(DOMException);
    float getSegmentParam(in unsigned long cmdIndex, in unsigned long paramIndex) raises
(DOMException);

    void moveTo(in float x, in float y);
    void lineTo(in float x, in float y);
    void quadTo(in float x1, in float y1, in float x2, in float y2);
    void curveTo(in float x1, in float y1, in float x2, in float y2, in float x3, in float y3);
    void close();
  };

  interface SVGMatrix
  {
    float getComponent(in unsigned long index) raises(DOMException);

    SVGMatrix mMultiply(in SVGMatrix secondMatrix);
    SVGMatrix mInverse() raises(SVGException);
    SVGMatrix mTranslate(in float x, in float y);
    SVGMatrix mScale(in float scaleFactor);
    SVGMatrix mRotate(in float angle);
  };

  interface SVGLocatable
  {
    SVGRect   getBBox();
    SVGMatrix getScreenCTM();
    SVGRect   getScreenBBox();
  };

  interface SVGLocatableElement : SVGElement, SVGLocatable
```

```
  {
  };

  interface TraitAccess
  {
    DOMString getTrait(in DOMString name) raises(DOMException);
    DOMString getTraitNS(in DOMString namespaceURI, in DOMString name) raises
(DOMException);
    float getFloatTrait(in DOMString name) raises(DOMException);
    SVGMatrix getMatrixTrait(in DOMString name) raises(DOMException);
    SVGRect getRectTrait(in DOMString name) raises(DOMException);
    SVGPath getPathTrait(in DOMString name) raises(DOMException);
    SVGRGBColor getRGBColorTrait(in DOMString name) raises(DOMException);

    void setTrait(in DOMString name, in DOMString value) raises(DOMException);
    void setTraitNS(in DOMString namespaceURI, in DOMString name, in DOMString value) raises
(DOMException);
    void setFloatTrait(in DOMString name, in float value) raises(DOMException);
    void setMatrixTrait(in DOMString name, in SVGMatrix matrix) raises(DOMException);
    void setRectTrait(in DOMString name, in SVGRect rect) raises(DOMException);
    void setPathTrait(in DOMString name, in SVGPath path) raises(DOMException);
    void setRGBColorTrait(in DOMString name, in SVGRGBColor color) raises(DOMException);
  };

  interface ElementTraversal
  {
    readonly attribute Element firstElementChild;
    readonly attribute Element lastElementChild;
    readonly attribute Element nextElementSibling;
    readonly attribute Element previousElementSibling;
  };

  interface SVGElement : dom::Element, events::EventTarget, TraitAccess, ElementTraversal
  {
    attribute DOMString id;
  };

  interface SVGAnimationElement : SVGElement, smil::ElementTimeControl
  {
  };

  interface EventListenerInitializer
  {
```

```
    void initializeEventListeners( in SVGDocument doc);
  };

  interface EventListenerInitializer2
  {
    void initializeEventListeners( in dom::Element scriptElement );
    events::EventListener createEventListener( in dom::Element handlerElement );
  };

  interface SVGGlobal : global::Global
  {
    global::Connection createConnection();
    void gotoLocation(in DOMString newURI);
    readonly attribute Document document;
    readonly attribute global::Global parent;
  };


};
```