# Complementing layout information with render information in SBML files

Ralph Gauges, Sven Sahle and Katja Wegner
University of Heidelberg
Im Neuenheimer Feld 267
D-69120 Heidelberg
Germany

October 29, 2009

# 1 Introduction

In 2003 we proposed an extension to the SBML file format that allowed programs to include layout and render information in SBML files to store one or more graphical representations of the SBML model. During the discussions on the SBML mailing list, it soon became evident that a consensus for both layout and render information would not be reached easily, therefore we separated the layout from the render part of the specification and concentrated on the inclusion of layout information into SBML files. Now three years later, we consider the layout extension to be ready for general usage and as a matter of fact, it has been accepted as an official extension to the upcoming SBML Level 3. There are several implementations for it and some programs already use it to exchange layout information on reaction networks. With the growing interest in graphical representations of reaction networks we feel that it is now time to complement the layout extension with a render extension that builds on it and allows the user to define not only the size and location of the objects, but also how they are to be rendered.

# 2 Design decisions

The first and as we think natural decision was to base the render extension on the existing layout extension. Secondly, we tried to make the render extension as flexible as possible in order to not impose any artificial limits on how programs can display their reaction networks.

We wanted to keep the render extension independent of the SBML model as well as of the layout extension, therefore the render information will be stored as one or more separate blocks. There can be one block of render information that applies to all layouts and an additional block for each layout. In the beginning this render information will be stored in the annotation of the `listOfLayouts` element or the annotation of a `layout` element respectively.

The render information consists of a set of styles that are associated with objects from the layout either by a list of ids of layout objects or by roles of layout objects or ids of their corresponding model elements. For example you can define a style that can be applied to all SpeciesReference objects or to all objects that have the role `product`.

Global render information included in the annotation of the `listOfLayouts` element will only be able to define styles that associate render information with roles of elements, it can not associate styles with individual objects from a layout.

Many of the elements used in the current render specification are based on corresponding elements from the SVG specification. This allows us to easily convert a combination of layout information and render information into a SVG drawing. At the same time we profit from the work that has already been done while creating the SVG specification.

# 3 Render information

The render extension provides two locations where styles can be defined. First each layout can have its own set of render information located in the annotation of the `layout` element (local render information). Second, a set of global render information objects located in the annotation of the `listOfLayouts` element can be defined.

It is important to note that each layout can have more than one set of local render information and that it is also possible to define more then one global style. Each style can also reference another style that complements it, this way the user can create styles that are based on other styles. In contrast to local styles, the global styles can not reference individual layout elements by an id, they can only define role based or type based styles.

## 3.1 Local render information

The top level element for the local render information is called `listOf-RenderInformation` which can contain a list of one or more `renderInformation` elements of type **LocalRenderInformation**. In addition to the list of local render information objects, the **ListOfLocalRenderInformation** has two attributes to specify the version of the render information.

**versionMajor** specifies the major version of the render information. Major version do not have to be backwards compatible to any lower major version of the render specification. **versionMinor** specifies the minor version of the render information. All minor versions within a major version have to be compatible.

The **LocalRenderInformation** data type is based on the **RenderInformationBase** data type. The **RenderInformationBase** class is derived from SBMLs **SBase** type and has five attributes. The **id** attribute is of type **SId** like the ids in SBML. It is used to give the **renderInformation** element a unique id through which it can be referenced from other **LocalRenderInformation** objects. The optional attribute **name** gives a **LocalRenderInformation** object a more user friendly name that can be displayed in programs.

The attributes **programName** and **programVersion** are optional and can be used to store information about the program that created the render information. Another optional attribute called **referenceRenderInformation** can be used to specify the id of another local or global render information object that complements the current render information object. So if a program can find no fitting render information in the current render information object, it can go on to the one referenced and see if it can find fitting information there. In order to avoid loops, only render information objects that have already been defined before may be referenced. So local render information objects may reference any global render information object as well as any local render information object that has already been defined and belongs to the same layout.

In addition to those five attributes, the **RenderInformationBase** object has an attribute called **backgroundColor** which defines the background color for rendering the layout. In addition to those attributes, there are three elements. The first element is called `listOfColorDefinitions` and is used to predefine a set of colors to be referenced in styles. The second element `listOfGradientDefinitions` contains linear and radial gradients to be referenced in styles. How colors and gradients can be defined is explained in the section called "Colors and gradients".

The third element is called `listOfLineEndings` and it is used to define a set of line endings that can be applied to path objects. This is explained in more detail in the section called "Line endings".

The **LocalRenderInformation** class extends the **RenderInformationBase** class by one element. The element is called `listOfStyles` and it can hold one or more local style objects. Each local style object is located in an element called `style` and is of type **LocalStyle**.

A **LocalStyle** object has an attribute called **id** that uniquely identifies it. It also has an optional **roleList** attribute which lists all the roles the style applies to and it can have a **typeList** attribute which lists all the element types the style applies to. The valid types for the **typeList** attribute are a combination of one or more of the following values separated by whitespaces:

- COMPARTMENTGLYPH,
- SPECIESGLYPH,
- REACTIONGLYPH,
- SPECIESREFERENCEGLYPH
- TEXTGLYPH,

- GRAPHICALOBJECT and
- ANY

The `ANY` keyword specifies that this styles applies to any type of glyph and would be equivalent to listing all the other keywords. Concerning the valid keywords for the **roleList** attribute we had thought about taking those from some kind of controlled vocabulary. Preferably, this would be some kind of ontology like SBO. The specifics of this will have to be discussed with other interested parties.

For the time being, all layout objects derived from **GraphicalObject** will get an additional attribute called **objectRole**. This attribute can be used to specify a string that specifies the role of the given object. If the same string appear s in the **roleList** of some render information object, the render information applies to the object, but only if there is no render information object that is more specific (see "Style resolution" and "Role resolution" below).

**LocalStyle** objects can have one more optional attribute which is called `idList`. This is simply a list of ids of layout objects the style applies to.

The only subelement of a style is a `g` element which specifies how the element(s) covered by the **idList**, **roleList** and **typeList** are to be rendered. The details of this element are described in the section about grouping.

| **ListOfLocalRenderInformation** inherits from **SBase** | | |
|---|---|---|
| versionMajor | : | unsigned int |
| versionMinor | : | unsigned int |
| renderInformation | : | LocalRenderInformation[1..*] |

| **RenderInformationBase** inherits from **SBase** | | |
|---|---|---|
| id | : | SId |
| name | : | string {use="optional"} |
| programName | : | string {use="optional"} |
| programVersion | : | string {use="optional"} |
| referenceRenderInformation | : | string {use="optional"} |
| backgroundColor | : | string {use="optional" default="#FFFFFFFF" } |
| listOfColorDefinitions | : | ListOfColorDefinitions {use="optional"} |
| listOfGradientDefinitions | : | ListOfGradientDefinitions {use="optional"} |
| listOfLineEndings | : | ListOfLineEndings {use="optional"} |

| **LocalRenderInformation** inherits from **RenderInformationBase** | | |
|---|---|---|
| listOfStyles | : | ListOfLocalStyles {use="optional"} |

| **ListOfLocalStyles** inherits from **SBase** | | |
|---|---|---|
| style | : | LocalStyle[1..*] |

| **LocalStyle** inherits from **Style** | | |
|---|---|---|
| idList | : | string[1..*] {use="optional"} |

| **Style** inherits from **SBase** | | |
|---|---|---|
| id | : | SId |
| roleList | : | string[1..*] {use="optional"} |
| typeList | : | string[1..*] {use="optional"} |
| g | : | Group |

## example:

```
<listOfLayouts xmlns="http://projects.eml.org/bcb/sbml/level2"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <layout id="Layout_1">
    <annotation>
      <listOfRenderInformation
           xmlns="http://projects.eml.org/bcb/sbml/render/version1_0_0">
        <renderInformation id="FancyRenderer_Default"
                            name="default style"
                           programName="FancyRenderer"
                           programVersion="0.1.1">
          <listOfColorDefinitions>
            <colorDefinition ... />
                  ...
          </listOfColorDefinitions>
          <listOfGradientDefinitions>
            <linearGradient ... >
                 ...
            </linearGradient>
            <radialGradient ... >
                 ...
            </radialGradient>
                  ...
```

```
          </listOfGradientDefinitions>
          <listOfLineEndings>
               ...
          </listOfLineEndings>
          <listOfStyles>
            <style id="CompartmentGlyphStyle" typeList="COMPARTMENTGLYPH">
              <g ...>
                ...
              </g>
            </style>
             ...
          </listOfStyles>
        </renderInformation>
      </listOfRenderInformation>
    </annotation>
       ...
 </layout>
</listOfLayouts>
```

## 3.2 Global render information

Global render information is specified very similar to local render information there are only some slight differences that one has to be aware of. Global render information is stored in an element called `listOfGlobalRenderInformation` which contains one ore more `renderInformation` elements of type **GlobalRenderInformation**.

The **ListOfGlobalRenderInformation** object has the same version attributes as the **ListOfLocalRenderInformation** object.

The attributes and elements of **GlobalRenderInformation** objects and **LocalRenderInformation** objects are the same. The only difference here is the fact that **GlobalRenderInformation** objects may only reference ids of other **GlobalRenderInformation** objects in their **referenceRenderInformation** attribute.

The `listOfStyles` element of the **GlobalRenderInformation** object contains one or more `style` elements but this time these are of type **GlobalStyle**. The **GlobalStyle** data type is also very similar to the **LocalStyle** data type but the **GlobalStyle** does not have an **idList** attribute since referencing individual ids from a layout does not make sense for a global render information object. Otherwise global and local render information is specified in the same way.

### example:

```
<listOfLayouts xmlns="http://projects.eml.org/bcb/sbml/level2"
```

| **ListOfGlobalRenderInformation** inherits from **SBase** | | |
|---|---|---|
| versionMajor | : | unsigned int |
| versionMinor | : | unsigned int |
| renderInformation | : | GlobalRenderInformation[1..∗] |

| **GlobalRenderInformation** inherits from **RenderInformationBase** | | |
|---|---|---|
| listOfStyles | : | ListOfGlobalStyles {use="optional"} |

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <annotation>
    <listOfGlobalRenderInformation
          xmlns="http://projects.eml.org/bcb/sbml/render/version1_0_0">
      <renderInformation id="FancyRenderer_GlobalDefault"
                         name="default global style"
                         programName="FancyRenderer"
                         programVersion="0.1.1">
        <listOfColorDefinitions>
             ...
        </listOfColorDefinitions>
        <listOfGradientDefinitions>
             ...
        </listOfGradientDefinitions>
        <listOfLineEndings>
             ...
        </listOfLineEndings>
        <listOfStyles>
             ...
        </listOfStyles>
      </renderInformation>
    </listOfGlobalRenderInformation>
  </annotation>
</listOfLayouts>
```

# 4 Styles

## 4.1 Positions and sizes

Positions and sizes for render elements can be specified as a combination of absolute values where the default unit is pt (1/72 inch) and relative values in % where the % symbol has to be added to the value. Each coordinate can have zero or one relative component and zero or one absolute component. For example to specify a coordinate that is 5 points left of the right edge of the current viewport the user could specify $-5 + 100\%$.

In order to make parsing of coordinate information easier, the absolute component has to be specified before the relative component. If the absolute component is 0.0, only the relative part has to be specified. All values are relative to the bounding box of the corresponding element in the layout. This bounding box basically specifies a canvas for the render elements to be drawn on.

When applying transformations to elements with relative values, the relative values have to be converted to absolute values first.

## 4.2 Colors and gradients

Although, it is possible to specify the color for a graphical primitive directly, colors and especially gradients can be specified in a so called `listOfColorDefinitions` and `listOfGradientDefinitions` element which are subelements of the **RenderInformation** data type. The `listOfColorDefinitions` element holds one or more elements called `colorDefinition` of type **ColorDefinition**. The **ColorDefinition** data type is derived from **SBase** and has two additional attributes. One **id** attribute which uniquely identifies the **ColorDefinition** object within a **RenderInformation** object and an attribute called **value** which holds a color value.

Color values are specified as a 6 to 8 digit hex string which defines the RGBA value of the color. If only the first six digits for the RGB value are given, the alpha value is assumed to be 0xFF which means that the color is totally opaque. Instead of specifying a color value, the value '`none`' can be given which is equal to no drawing at all. To specify '`none`' for the **stop-color** attribute of a gradient is not allowed.

| **ColorDefinition** inherits from **SBase** | | |
|---|---|---|
| id | : | SId |
| value | : | string |

### example:

```
<listOfColorDefinitions>
  <colorDefinition id="darkred" value="#200000" />
       ...
</listOfColorDefinitions>
```

All graphical primitives in the render extension have a **stroke** attribute that is used to specify the color of the stroke that is used to draw the curve or the outline of ellipses, rectangles or polygons. This **stroke** attribute can either hold a color value or it can hold the id of a predefined **ColorDefinition** object.

The `listOfGradientDefinitions` element holds one or more `linearGradient` or `radialGradient` subelements of type **LinearGradient** or **RadialGradient** respectively.

The base class for both gradient types is called **GradientBase** and it has the two attributes **id** and **spreadMethod**. As well as a list of so called "gradient stops". The **id** attribute is used to identify and reference a gradient within a render information.

| **GradientBase** inherits from **SBase** | | |
|---|---|---|
| id | : | SId |
| spreadMethod | : | string {use="optional" default="pad"} |
| stop | : | GradientStop[1..*] |

The **spreadMethod** attribute is optional and specifies the method that is used to continue the gradient pattern if the vector points do not span the whole bounding box of the object the gradient is applied to (see example below). The attribute can have three values called `pad`, `reflect` or `repeat`:

- `pad`: the gradient color at the endpoint of the vector defines how the gradient is continued beyond that point (default value).
- `reflect`: the gradient continues from end to start and then from start to end again and again.
- `repeat`: the gradient pattern is repeated from start to end over and over again.

To specify "gradient stops" a gradient element can hold one or more subelements called `stop` which are of type **GradientStop**. The **GradientStop** data type has two attributes. The first attribute, called **offset**, represents the relative distance from the starting point of the gradient. Depending on the type of gradient, this is either the point defined by the **x1**,**y1** and **z1** attributes (linear gradient) or the **fx**, **fy** and **fz** attributes (radial gradient). The value is given as a positive percentage value (usually somewhere between 0% and 100%). The other attribute is called **stop-stroke** and defines the color for the given gradient stop. The attributes value can either be given as a hexadecimal color value or as the id of a ColorDefinition object from the `listOfColorDefinitions` (see above). To specify the id of another gradient as the value of a **stop-color** attribute is considered an

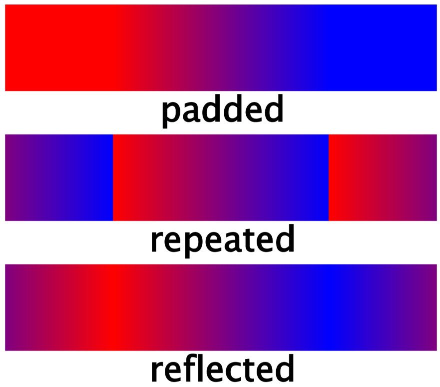


Figure 1: example of different SVG spreadMethod values

error. In case the two points that define the gradient vector are identical, the area is to be painted with a single color taken from the last gradient stop element.

There are a few rules that need to be considered when working with gradient stops. Basically those rules are the same as defined by the SVG specification.

1. the offset value of a gradient stop should be between 0% and 100%. If the offset lies outside of this value, the value is adjusted to be either 0% isf the given value is smaller than 0% or to 100% if the value is greater than 100%.

2. The absolute part in any offset value is ignored, meaning it is considered to be 0.0 even if specified otherwise in a gradient stop.

3. The offset of any gradient stop has to be greater or equal to the offset of the preceding fgradient stop. If a gradient stop has an offset that is smaller than the offset of the preceeding stop, the offset is considered to have the same value as the offset of the preceeding stop.

4. If two gradient stops have the same offset value, the last gradient stop with this offset value determines the color at this point in the gradient.

A `linearGradient` element has six attributes. The attributes **x1**, **y1**, **z1**, **x2**, **y2** and **z2** are all optional and define a vector on which the gradient stops are mapped. If not specified, **x1**, **y1** and **z1** default to 0% and **x2**,**y2** and **z2** default to 100%.

### example:

| **LinearGradient** inherits from **GradientBase** | | |
|---|---|---|
| x1 | : | string {use="optional" default="0%"} |
| y1 | : | string {use="optional" default="0%"} |
| z1 | : | string {use="optional" default="0%"} |
| x2 | : | string {use="optional" default="100%"} |
| y2 | : | string {use="optional" default="100%"} |
| z2 | : | string {use="optional" default="100%"} |

| **GradientStop** inherits from **SBase** | | |
|---|---|---|
| offset | : | string |
| stop-color | : | string |

```
<listOfGradientDefinitions>
  <linearGradient x1="30%" y1="50%" x2="70%" y2="50%">
    <stop offset="0%" stop-color="#0000A0" />
    <stop offset="100%" stop-color="darkred" />
  </linearGradient>
        ...
</listOfGradientDefinitions>
```

The **RadialGradient** data type has seven additinal attributes. The attributes **cx**, **cy** and **cz** define the center of the radial gradient. The attributes are optional and can either be given in absolute or relative coordinates. The default value for all three attributes is 50%. The **r** attribute defines the radius of the gradient and it can also be specified in either absolute or relative coordinates. Specifying negative values for **r** is considered an error. The attributes **fx**, **fy** and **fz** specify the focal point of the gradient. The gradient will be drawn such that the 0% stop is mapped to (**fx**,**fy**,**fz**). The attributes **fx**, **fy** and **fz** are optional. If one is omitted it is considered to equal to the value of **cx**, **cy** and **cz** respectively.

If the focal point, which is determined by the values **fx**, **fy** and **fz** lies outside the circle, the focal point is considered to be located on the intersection of the the line from the center point to the focal point and the sphere determined by the center point and the radius.

If the radius is given in relative values, the relation is to the width as well as the height. This means that if the width of the bounding box and the height of the bounding box are not equal, **cx**,**cy**,**cy** and **r** dont't actually specify a circle, but an ellipse.

### example:

```
<listOfGradientDefinitions>
```

| **RadialGradient** inherits from **GradientBase** | | |
|---|---|---|
| cx | : | string {use=”optional” default=”50%”} |
| cy | : | string {use=”optional” default=”50%”} |
| cz | : | string {use=”optional” default=”50%”} |
| r | : | string {use=”optional” default=”50%”} |
| fx | : | string {use=”optional”} |
| fy | : | string {use=”optional”} |
| fz | : | string {use=”optional”} |

```
  <radialGradient cx="50%" cy="50%" r="20" spreadMethod="repeat">
    <stop offset="10%" stop-color="#000040" />
    <stop offset="90%" stop-color="#0000C0" />
  </radialGradient>
      ...
</listOfGradientDefinitions>
```

## 4.3 Graphical primitives

The graphical primitives polygons, rectangles and ellipses are based on the corresponding elements from SVG.For lines, arcs and general path primitives, we introduce the `curve` element which differs slightly from the layout extension with the same name. Whereas **Point** objects in the layout extension could only contain absolute values for their coordinates, **RenderPoint** objects in the render extension can contain relative coordinate values.

Since polygons are very similar to general path primitives, we use a similar structure to define curves and polygons in the render extension.

All graphical primitives have attributes in common that specify some drawing properties. As mentioned in the ”Colors and gradients” section. Each graphical primitive has a **stroke** attribute that defines the color used for curves and outlines of geometric shapes. In addition to that, the **stroke-width** attribute specifies the width of the stroke and the **stroke-dasharray** is a list of positive integer numbers that specifies the lengths of dashes and gaps that are used to draw the line. The individual numbers in the list are separated by commas.

E.g. ”5,10” would mean to draw 5 points, make a 10 point gap, draw 5 points etc. If the pattern is to start with a gap, the first number has to be 0.

If a style defines a stroke dasharray and this style is applied to a curve from the layout specification, one has to watch out for the fact that the layout curves may contain breaks (if the end point of segment n is not identical to the starting point of segment n+1). In this case each of the unbroken line stretches is considered a seperate curve object and the line stippling is applied to each curve. That means the line stippling is not continously applied through the gap, but it starts again after the gap.

In addition to those attributes, ellipses, polygons and rectangles have an attribute called **fill** that specifies the fill style of those elements. The fill style can either be a hexadecimal color value or the id of a **ColorDefinition** object or the id of a **GradientDefinition** object. Instead of a color or gradient id, '`none`' can be specified which means that the object is unfilled.

Additionally, an attribute called **fill-rule** can be used to specify how the shape should be filled. Allowed values for **fill-rule** are:

- `nonzero` (default) or
- `evenodd`.

For a detailed description on how those attributes work in detail, we would like to refer you to the corresponding documentation in the SVG specification. As time permits we will add our own documentation.

Currently the `fill-rule` attribute is only usefull for polygons. All other shapes can not have alternating areas.

As a common base class for all elements that can be drawn, we introduce the **Transformation** class which contains one attribute called **transform** that specifies an affine transformation matrix in 3D consisting of exactly twelve double values. Since the layout and render extension are only 2D so far, this class is only used as a base class for **Transformation2D** and we leave the complete specification of this class for a future version of this document.

| **Transformation** inherits from **SBase** | | |
|---|---|---|
| transform | : | double[12] {use="optional"} |

Since the current render information specification only defines 2D objects, we derive a second class called **Transformation2D** from **Transformation**. This new class restricts the transformation matrix to specify the six values of a 2D affine transformation. The class **Transformation2D** serves as the base class for all drawable 1D and 2D objects.

| **Transformation2D** inherits from **Transformation** | | |
|---|---|---|
| transform | : | double[6] {use="optional"} |

## 4.4 Transformations

In order to be able to display text that is not aligned horizontally or vertically or to effectively compose groups of objects from primitives, transformations like rotation, translation and scaling are needed. SVG, among other options, allows the user to specify a 3x3 matrix transformation matrix:

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

Since the last row of the matrix is always 0 0 1, the matrix is specified as a six value vector. Therefore, in the render extension each group or graphical primitive is derived from the class **Transformation2D** and can have a `transform` attribute just as in SVG. The allowed value for the attribute has the form: `a, b, c, d, e, f`.

The values for `a`,`b`,`c`,`d`,`e` and `f` depend on the transformation operation components and the order in which those transformation components are executed.

There are five basic transformation operations that can be combined in a affine transformation matrix.

### 4.4.1 Translation

Translating something means moving it some distance along one or more of the axes. The corresponding 2D tranformation matrix is

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

where tx and ty are the distance along the x and y axes by which the object shall be moved.

### 4.4.2 Scaling

Scaling means to multiply all coordintate components of an object by a certain value. The corresponding 2D transformation matrix is

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

where sx and sy are the scaling factors along the x and y axis respectively.

### 4.4.3 Rotation

With a rotation, an object can be rotated around the origin of the coordinate system. The corresponding 2D transformation matrix is

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

where $\alpha$ is the angle of rotation around the origin.

### 4.4.4 Skewing

Skewing is the least used operation and we have to distinguish between skewing along the x or the y axis. The corresponding 2D transformation matrices are

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

$$\begin{bmatrix} 1 & 0 & 0 \\ tab(\beta) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\alpha$ is the skewing angle of skewing along the x axis and $\beta$ is the angle for skewing along the y axis.

Combining several of the operations above means multiplying the transformation matrices that belong to the individual operations. Depending on the matrices that are multiplied, the order of the operations matter, e.g. it makes a difference if an object is tranlated before it is rotatet or if it is rotatet first.

If an object specifies a transformation, this transformation is to be applied to the object prior to any other coordinate properties of the object. E.g. if a rectangle specifies a position of $x = 10$ and $y = 20$ and it also specifies a rotation by 45 degrees, the rotation is applied before the object is placed at $P(10, 20)$. The transformation for an object is always in relation to the objects viewport. For most render objects, this would be the bounding box of the corresponding layout object. For layout curves, e.g. in reaction glyphs or species reference glyphs, the viewport is the complete diagram. For objects defined in line endings, the viewport is the bounding box of the line ending before it is applied to the line.

### example:

```
<g ...>
  <text x="50%" y="50%" text-anchor="middle" stroke="#FF0000"
       font-family="serif" font-size="20.0"
       transform="1.0, 3.0, 2.5, 1.4, 4.0, 5.0">This is a Text</text>
     ...
</g>
```

All objects that are derived from **Transformation2D** can have a transformation, this includes group elements. In contrast to other attributes on groups and children of groups, the transformation is not overwritten if it is specified in a child, but rather all transformations that are defined in an object hierarchy accumulate. E.g. when a group specifies a transformation and a child of the group also sets a transformation, the transformation for the child has to be applied to the child only and the transformation that is set on the group has to be applied to the whole group, i.e. to all children of the group.

| **GraphicalPrimitive1D** inherits from **Transformation2D** | | |
|---|---|---|
| stroke | : | string {use="optional"} |
| stroke-width | : | string {use="optional"} |
| stroke-dasharray | : | unsigned integer[1..*] {use="optional"} |

| **GraphicalPrimitive2D** inherits from **GraphicalPrimitive1D** | | |
|---|---|---|
| fill | : | string {use="optional"} |
| fill-rule | : | string {use="optional"} |

### 4.4.5 Curves

Simple lines and complex curves are represented by a `curve` element. A curve has a `listOfElements` element that can hold an arbitrary number of points and cubic bezier elements in any order . The only restriction is that the first element has to be a point. If the first element is a bezier element, it is to be interpreted as a point.

As mentioned earlier, **RenderPoint** objects used to specify the individual curve segments can contain relative values for their coordinates as well as absolute values. The coordinate values are always with respect to the bounding box of the layout object the render information applies to.

To assign line endings to the start and end of a path object, two new attributes were introduced. They are called **startHead** and **endHead** and specify the id of the line ending that shall be applied to the start and the end of the curve respectively. Both attributes are optional.

How line endings are defined is described in the section called "Line endings".

| **Curve** inherits from **GraphicalPrimitive1D** | | |
|---|---|---|
| startHead | : | SId {use="optional"} |
| endHead | : | SId {use="optional"} |
| listOfElements | : | ListOfElements |

| **ListOfElements** inherits from **SBase** | | |
|---|---|---|
| element | : | RenderPoint[1..*] |

| **RenderPoint** inherits from **SBase** | | |
|---|---|---|
| id | : | SId {use="optional"} |
| x | : | string |
| y | : | string |
| z | : | string {use="optional" default="0.0"} |

| **RenderCubicBezier** inherits from **RenderPoint** | | |
|---|---|---|
| basePoint1_x | : | string |
| basePoint1_y | : | string |
| basePoint1_z | : | string {use="optional" default="0.0"} |
| basePoint2_x | : | string |
| basePoint2_y | : | string |
| basePoint2_z | : | string {use="optional" default="0.0"} |

## example:

```
 <g ...>
  <curve stroke-width="2.0" stroke="#000000" >
   <listOfElements>
     <element xsi:type="RenderPoint" x="0%" y="50%" />
     <element xsi:type="RenderPoint" x="100%" y="50%" />
     <element xsi:type="RenderCubicBezier" x="0%" y="50%" basepoint1_x="50%" basepoint1_y="90%
              basepoint2_x="50%" basepoint2_y="90%" />
   </listOfElements>
  </curve>
    ...
</g>
```

### 4.4.6 Polygons

A **Polygon** object is made up of a `polygon` element which contains a `listOfElements` that defines the edge of the polygon.

The major difference to the **Curve** object is that the individual curve segments can only be straight lines and the last point of the curve is connected to the first, so the polygon is always closed. Therfore, the polygon can have a fill style that determines how the inside of the polygon is to be rendered.

| **Polygon** inherits from **GraphicalPrimitive2D** | | |
|---|---|---|
| listOfElements | : | ListOfElements |

## example:

```
<g ...>
  <polygon stroke="#000000" stroke-width="3" fill="#FF0000">
    <listOfElements>
      <element xsi:type="RenderPoint" x="100%" y="33%"/>
      <element xsi:type="RenderPoint" x="20%" y="100%"/>
      <element xsi:type="RenderPoint" x="50%" y="0"/>
      <element xsi:type="RenderPoint" x="80%" y="100%"/>
      <element xsi:type="RenderPoint" x="0" y="33%"/>
    </listOfElements>
  </polygon>
     ...
</g>
```

### 4.4.7 Rectangles

The **Rectangle** object was taken from the SVG specification and allows the definition of rectangles with or without rounded edges.

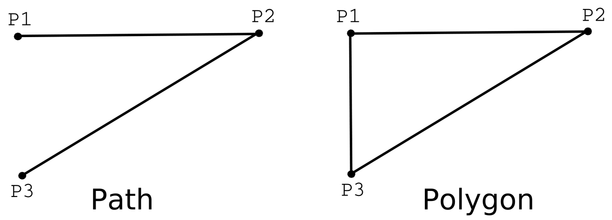


Figure 2: Rendering of a Path vs. rendering of a Polygon with the same base points

The rectangle has the attributes **x**, **y** and **z** to specify its position within the bounding box of the enclosing layout object and a **width** and **height** attribute that specifies the width and height of the rectangle, either in absolute values or as a percentage of the width and height of the enclosing bounding box. The default value for the optional **z** attribute is 0.0.

Additionally the rectangle has two optional attributes **rx** and **ry** that specify the radius of the corner curvature. If only **rx** or **ry** is specified, the other is presumed to have the same value as the one given. The default value for **rx** and **ry** is 0.0 which means that the edges are not rounded. The relative values in rx and ry are in relation to the width and the height of the rectangle respectively. So a value of 10% for rx means the radius of the corner is 10% of the width of the rectangle.

| **Rectangle** inherits from **GraphicalPrimitive2D** | | |
|---|---|---|
| x | : | string |
| y | : | string |
| z | : | string {use="optional" default="0.0"} |
| width | : | string |
| height | : | string |
| rx | : | string {use="optional" default="0.0"} |
| ry | : | string {use="optional" default="0.0"} |

## example:

```
<g ...>
 <rectangle x="0%" y="0%" width="100%" height="100%" rx="5%"
            fill="darkred" stroke="#000000" />
      ...
</g>
```

### 4.4.8 Ellipses

The definition of an ellipse was also taken directly from SVG. The `ellipse` element has the attributes **cx**, **cy** and **cz** to specify the center of the ellipse and **rx** and **ry** to specify the radius of the ellipse along the x-axis and the y-axis respectively. If only **rx** or **ry** is specified, the other is presumed to have the same value. Circles are a special case of an ellipse where **rx** and **ry** are equal. Again **cz** is optional and its default value is 0.0.

| **Ellipse** inherits from **GraphicalPrimitive2D** | | |
|---|---|---|
| cx | : | string |
| cy | : | string |
| cz | : | string {use=”optional” default=”0.0”} |
| rx | : | string |
| ry | : | string {use=”optional” default=rx} |

## example:

```
<g ...>
 <ellipse cx="50%" cy="50%" rx="30%" fill="#00FF00" stroke="#000000" />
     ...
</g>
```

### 4.4.9 Text elements

In order to draw text, we use the `text` element from SVG with slight modifications. Like the `text` element in SVG, our text element has the optional attributes **font-family** to specify which font to use and **font-size** to specify the size of the font. If specified, **font-size** must be a positive value. It can be either an absolute value or a relative value. In the case of a relative value it specifies a percentage of the height of the corresponding object. Combinations of absolute and relative values as for the point objects in other objects are not allowed.

For reasons of simplicity, we limit the display of text to normal text, outlined or filled-outlined text are not supported. Also in order to simplify the text display we think it would be best practice if programs would limit the choice of the **font-family** attribute to the generic families `serif`, `sans-serif` and `monospace`. But since those only apply to western languages, it can make sense to use other values for **font-familie** in certain cases.

The horizontal alignment of a text element can be specified by the **text-anchor** attribute. Allowed values are `start`, `middle` and `end`. SVG does not seem to provide any means for the vertical alignment of text. Since we feel that this is an important feature, we have added a corresponding attribute called **vtext-anchor** which determines the vertical justification of the text element. The values that are allowed for **vtext-anchor** are `top`, `middle` and `bottom`.

The alignment attributes do not have default values because this would disable inheritance. Only the top level group in a style does have default values for the alignment attributes.

Since we have a right handed coordiante system, the positive y axis normally faces downward on the screen if the positive z-axis goes into the screen. This means that text actually has to be renderer with the top towards lower y-values.


Figure 3: example text with marked baseline, ascent and descent

If the **vtext-anchor** is given as `top`, the top of the text has to be aligned with the bottom end (lower y value) of the bounding box (see Figure 4.4.9). If **vtext-anchor** is `bottom`, the bottom of the text has to be aligned with the top of the bounding box (higher y value) (see Figure 4.4.9). If **vtext-anchor** is `middle`, the vertical center of the text box has to be aligned with the vertical center of the bounding box (see Figure 4.4.9).

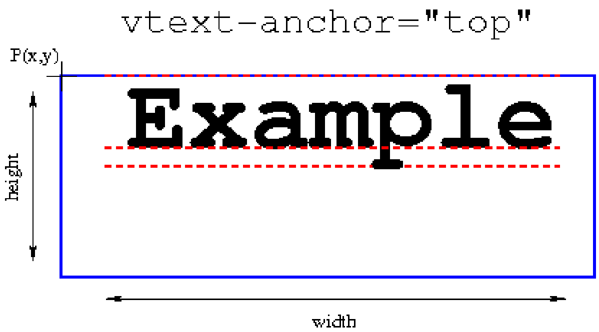


Figure 4: vertical text alignment `top`

The text element can also have offset values for the `x`,`y` and `z` value. Those offsets are applied to the text after it has been positioned according to the anchor attributes. The default value for these three attributes is 0.0.

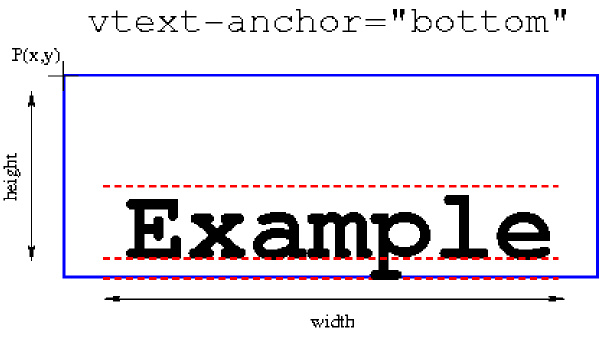


Figure 5: vertical text alignment `bottom`

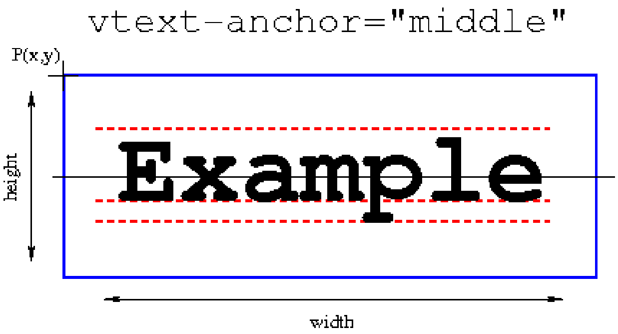


Figure 6: vertical text alignment `middle`

The `text` element has two more attributes. One is called **font-weight** and specifies whether a font is to be drawn bold. The only values allowed for **font-weight** are `bold` and `normal`. Likewise the **font-style** attribute determines whether a font is to be drawn italic or normal and consequently the only allowed values are `italic` and `normal`. Both attributes are optional.

**example:**

```
 <g ...>
  <text x="50%" y="50%" text-anchor="middle" stroke="#FF0000"
        font-family="serif" font-size="20.0" >This is a Text</text>
     ...
</g>
```

### 4.4.10 Bitmaps

To include bitmaps into a graphical representation we use the `image` element from SVG. The `image` element in SVG can also be used to include complete

| **Text** inherits from **GraphicalPrimitive1D** | | |
|---|---|---|
| x | : | string |
| y | : | string |
| z | : | string {use=”optional” default=”0.0”} |
| font-family | : | string {use=”optional”} |
| font-size | : | string {use=”optional”} |
| font-weight | : | string {use=”optional”} |
| font-style | : | string {use=”optional”} |
| text-anchor | : | string {use=”optional”} |
| vtext-anchor | : | string {use=”optional”} |

SVG vector images which we explicitly exclude in this version of the proposal since we think it would be too complex. If the need for the inclusion of SVG drawings arises, it is only a matter of rephrasing this specification.

The `image` element has six attributes. The **x**, **y** and **z** attributes specify the position of the image within the bounding box and the **width** and **height** attributes specify its width and height. The **z** attribute is optional and its default value is 0.0. The actual image data is not embedded in the render information, but the `image` element has an attribute called **href** that references an external JPEG or PNG file. To simplify things, the reference has to be an absolute or relative path to a local file. Non-local image ressources (e.g. from the net) are currently not supported. If the referenced image is larger then the given width and height, it has to be scaled to the given dimensions. If the reference ressource can not be found, it is up to the application if nothing is drawn or some placeholder is displayed. Preferably the user would get some kind of notification about the missing ressource.

| **Image** inherits from **Transformation2D** | | |
|---|---|---|
| x | : | string |
| y | : | string |
| z | : | string {use=”optional” default=”0.0”} |
| width | : | string |
| height | : | string |
| href | : | string |

### example:

```
<g ...>
 <image x="10%" y="10%" width="80" height="100" href="Glucose.png" />
    ...
```

```
</g>
```

## 4.5 Grouping

Like in SVG, several graphical primitives can be grouped inside a `g` element to generate more complex render information.

| **Group** inherits from **GraphicalPrimitve2D** | | |
|---|---|---|
| font-family | : | string {use="optional"} |
| font-size | : | string {use="optional"} |
| font-weight | : | string {use="optional"} |
| font-style | : | string {use="optional"} |
| text-anchor | : | string {use="optional"} |
| vtext-anchor | : | string {use="optional"} |
| startHead | : | SId {use="optional"} |
| endHead | : | SId {use="optional"} |

**stroke**, **stroke-width**, **stroke-dasharrays**, **transform**, **fill**,**fill-rule**, **font-family**, **font-size**, **font-weight**, **font-style** and **text-anchor** attributes can be applied to groups. If any of those attributes is specified for a **Group** object, it specifies the corresponding attribute for all graphical primitives and groups defined within this group. If a graphical primitive or a group redefines one or more of those attributes, the newly defined values take effect. The outermost group in a style always has default values for the attributes, all other embeded elements don't have default values for their attributes. This way it is easy to distinguish between an attribute that has really been set and one that has not been set. The default values for the outermost `group` element are listed in table 1.

It might seem a little unusual that the default values for **stroke-width** and **font-size** are set to 0. The reason for this is that a style that only contains an empty `group` is meant to define that the element the style applies to is not to be rendered. Since the render information for curves in **SpeciesReferenceGlyph** and **ReactionGlyph** objects as well as the render information for **TextGlyph** objects is defined via attributes from the outermost `group` element of a style (see below), the `group` element would explicitly have to define the **stroke-width** or the **font-size** to be 0 which would be inconsistent with the implied meaning of an empty group. The outermost `group` element can also contain information about arrow heads to

| attribute | default value |
|---|---|
| stroke | none |
| stroke-width | 0.0 |
| stroke-dasharrays | *empty list* |
| transform | 1.0, 0.0, 0.0, 0.0, 1.0, 0.0 |
| fill | **none** |
| fill-rule | string {use=”optional” default=”nonzero”} |
| font-family | sans-serif |
| font-size | 0 |
| font-weight | normal |
| font-style | normal |
| text-anchor | start |
| vtext-anchor | top |
| startHead | **none** |
| endHead | **none** |

Table 1: Attribute default values.

be used on curves specified in the layout. This information is given via the **startHead** and **endHead** attributes just like for `curve` elements. These attributes only apply to **Curve** objects from the layout, not to **RenderCurve** objects within the group. Since those two attributes only make sense on the outermost group of a style, they are to be ignored on all other groups. The default value for those attributes is `none` which means that no line ending is to be drawn.

Each `group` element also has an **id** attribute through which it can be identified. In addition to those attributes a **Group** object can contain 0 or more child elements that form the render information. These child elements have to be elements derived from **Transformation2D**, so right now this would be **Images** or everything derived from **GraphicalPrimitive1D**, e.g. `rectangles`, `ellipses`, `curves`, `polygons`, `text` elements or `groups`.

**example:**

```
 <g stroke="#000000" font-family="serif" >
   <rectangle x="0%" y="0%" width="100%" height="100%"
              fill="blueLinearGradient"  />
   <text x="50%" y="50%" font-size="80%" text-anchor="middle"
         stroke="#FF0000" />
</g>
```

# 5 Line endings

In many graphs the relations between nodes are depicted by lines and often the type of relation is encoded in the line ending. For this reason, the render extension provides ways to specify a set of arbitrary line endings and means to apply those to path objects. The individual line endings are defined in an element called `listOfLineEndings` which comes right before the `listOfStyles`.

The individual line endings are defined as **Group** objects just like styles. Therefore, arbitrarily complex line endings can be defined. Each line ending is encapsulated in an element called `lineEnding` and contains two subelements.

The first element is called `boundingBox` and it specifies the viewport that is used to draw the line ending. Just like the bounding boxes of the layout extension, this bounding box contains a `position` and a `dimensions` subelement. The `dimensions` element specifies the size of the viewport for the line ending along each of the axes. The `position` element specifies the offset from the end of the curve that the line ending is applied to. A position of $(0.0, 0.0, 0.0)$ means that the origin of the line endings bounding box is mapped directly to the end of the curve. For a description on how the mapping is calculated in all other cases see the section called "Mapping line endings to curves".

The second subelement is a `group` element that holds the render information for the line ending.

The two attributes of the `lineEnding` element are the **id** attribute which is used to specify a unique id for the line ending by which it can be referenced and an attribute called **enableRotationalMapping**. The **enableRotationalMapping** attribute specifies whether a line ending will be rotated depending on the slope of the line it is applied to or if it is drawn just the way it was specified. The default value for the attribute is `true` which means that the line ending is rotated depending on the slope of the line. A more detailed description of this mapping is given in figure 5.

In order to declare that a certain line ending is to be used on a path object, the `curve` element has two attributes called **startHead** and **endHead** which hold the id of a line ending definition for the start and for the end of the path respectively.

The top level group in a line ending differs from top level groups in normal graphical elements in one respect. The top level group of a line ending inherits all attributes from the line it is applied to save for the attributes for the line endings themselves. This way a stylesheet can define one line ending which can be applied to lines of different colors and it inherits the color from the line. If the group also inherited the attributes for the line endings and it contained a `curve` element itself, we would have generated an endless loop.

| **LineEnding** inherits from **GraphicalPrimitive2D** | | |
|---|---|---|
| enableRotationalMapping | : | boolean default=true |
| boundingBox | : | BoundingBox |
| g | : | Group |

### example:

```
<lineEnding id="SimpleArrowHead">
 <boundingBox>
   <position x="-10.0" y="-4.0" />
   <dimensions width="12.0" height="8.0"/>
 </boundingBox>
 <g>
   <polygon>
     <curve>
       <listOfCurveSegments>
         <curveSegment xsi:type="LineSegment">
           <start x="100%" y="50%" />
           <end x="0%" y="100%" />
         </curveSegment>
         <curveSegment xsi:type="LineSegment">
           <start x="0%" y="100%" />
           <end x="0%" y="0%" />
         </curveSegment>
       </listOfCurveSegments>
     </curve>
   </polygon>
 </g>
</lineEnding>
```

## 5.1 Mapping line endings to curves

In order to apply a line ending which is defined using only 2D coordinates onto a line which has been defined using 3D coordinates, we need to define a kind of mapping. The first definition we make is that the origin of the line ending viewport is mapped to the end of the line to which the line ending is applied. If the **enableRotationalMapping** attribute is set to `false`, the line endings coordinate system is the same as the global coordinate system used to draw the layout, only the origin is moved to that end of the line the line ending is applied to. If the **enableRotationalMapping** attribute is set to `true`, which is the default, we define that the x,y-plane of the line endings

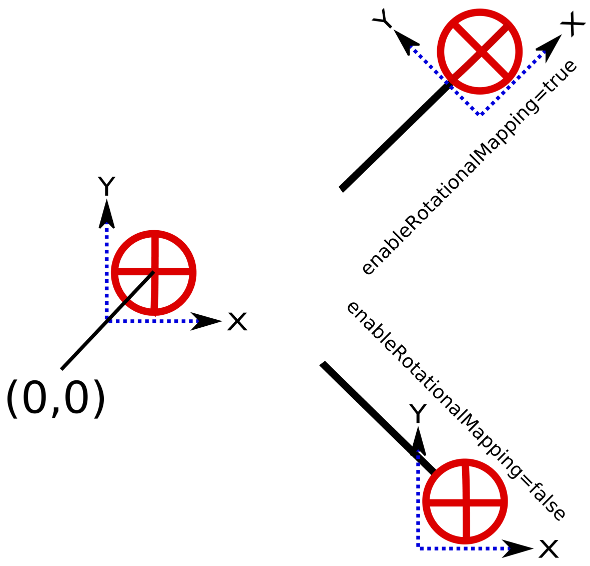


Figure 7: example of a line ending with and without rotation mapping enabled

viewport is mapped to the plane that results from taking the unit vector of the slope of the line and the unit vector that results from orthonormalizing the slope vector and a second vector that has no component along the z axis. If the slope of the line has a positive component along the x axis, the orthonormalized vector also has to have a positive component along the y axis. In order to retain the right handed coordinate system, the z axis of the line endings coordinate system is perpendicular to the plane created by the other two vectors and has a positive component along the global coordinate systems z-axis. Likewise if the slope has a negative component along the global coordinate systems x axis, the y component of the orthonormalized second vector has a negative component along the y axis of the global coordinate system and to retain the right handed coordinate system, the third vector which is perpendicular to the plane made by the slope and its orthonormalized vector, has a positive component along the global coordinate systems z axis.

If the slope of the line points directly along the positive z axis of the global coordinate system, the line endings coordinate system is mapped to the line ending by a -90 rotation around the y axis of the line endings coordinate system and a translation of the origin of the line endings coordinate system to the end of the line. If the slope points directly down the negative z axis, the line endings coordinate system has to be rotated by +90 around its y axis before translation to the position of the curves end.

This may all sound very complicated, but in the end, the calculations to be done are not difficult and straight forward.

The mathematical description of the mapping and an example are given in Appendix A.

# 6 Style resolution

To resolve which style applies to a certain object, one should follow the rule that more specific style definitions take precedence over less specific ones and that if there are several styles with the same specificity, the first one encountered in the file is to be used. In essence, this means that a program first has to search the local render information for a style that references the id of the object. If none is found, it searches for a style that mentions the role of the object. If it has one, see next section. If it does not find one, it searches for a style for the type of the object.

If a render information references another render information object via its **referenceRenderInformation** attribute, the program has to go through that one as well to see if a more specific render information is present there. If the chain of referenced RenderInformation objects has been searched and no style has been found that fits, it is up to the program how the object is rendered.

If several type based styles are found that would fit, a style that applies to only one type takes precedence over a style that applies to several types.

If a program explicitly wants to define render information that states that some objects are not to be rendered at all, it has to define a style that does nothing, i.e. has no render information but applies to the objects that should not be rendered.

# 7 Role resolution

This render extension explicitly provides means to write render information that renders layout objects based on certain roles those render objects or their corresponding model objects have. So far SBML models or layouts do not contain such role information or only for a limited number of objects if one would consider the role attribute of SpeciesReferenceGlyph objects to fall into this category. Although there is currently no means to specify these roles, there are already initiatives underway that try to complement

SBML files with more biological information based on ontologies. One of these initiatives, the sboTerms, is about to be included into SBML Level 2 Version 2. This ontology or a similar one could provide this role information for layout objects in the future.

For the time being, we define an additional attribute called **objectRole** for all layout objects derived from **GraphicalObject** including **GraphicalObject** itself. The attribute specifies a user defined role string. render information including the same role string in its **roleList** attribute applies to the object. This is only true if no more specific render information takes precedence (see "Style resolution").

A specific style can reference one or more roles to which it applies. When a program tries to determine which style applies to a specific object it might have to determine the role of the object layout first. If the layout object itself has a role, this will be taken, otherwise if the layout object is associated with an object in the model, the program should get the role from the associated object. If none of them has a role, no role based style can be applied to the object.

# 8 Style information for reaction glyphs and species reference glyphs

When defining a style for a **ReactionGlyph** or **SpeciesReferenceGlyph** object, one has to distinguish between layout objects that only specify a bounding box for the object and those that specify a curve. In the case of a bounding box, you want to define complete render information, whereas in the case of a curve, you only want to set certain attributes that determine certain aspects of how the curve should be drawn, e.g. its color. To resolve this conflict, the style for such an object has to define render information for both cases. The render information for the case of a bounding box is specified just like render information for any other object within a group. Render information for the case of a curve is defined by the appropriate attributes that are in effect in the outermost **Group** object itself. Those attributes include **stroke**, **stroke-width** and **stroke-dasharray**. Additionally **startHead** and **endHead** can be specified to define line endings for layout curve objects. If the group does not define one or more of these attributes, the default value is used (see section "Grouping").

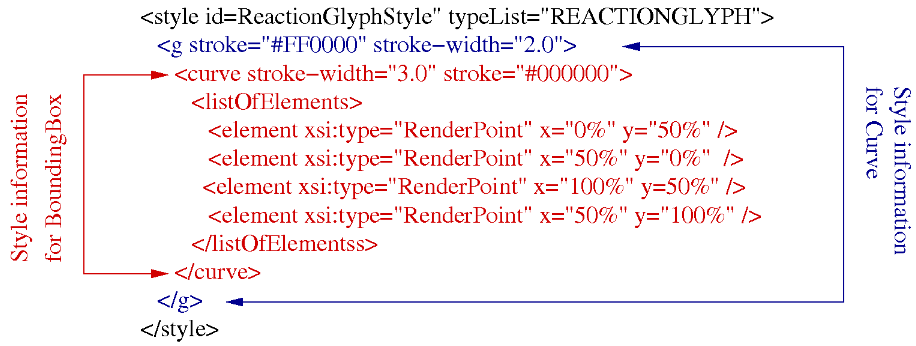


Figure 8: style with render information for objects with curve or bounding box

# 9 Style information for text glyphs

Just as in the case of curves in **ReactionGlyphs** and **SpeciesReferenceGlyphs**, **TextGlyphs** can be considered render information which is located in the layout. A **TextGlyph** specifies the text to be rendered, it therefore does not need additional render information in the form of a `text` element. On the other hand, it needs render information in the form of font properties. Just as for the **Curve** object for **ReactionGlyphs** and **SpeciesReferenceGlyphs**, this render information is taken from the font related attributes of the outermost group element of the style that is used to render a **TextGlyph**. Any additional information within the group is ignored. If the group does not specify any of the **font-family**, **font-size**, **font-weight**, **font-style**, **text-anchor** or **vtext-anchor** attributes, the default values are to be used.

# 10 Uniqueness of ids

Since local and global render information objects can reference other render information objects, programs creating render information need to make sure that all the ids are unique within the reference history. In other words, a render information object that references another render information object must make sure that none of its ids is equal to an id in any of the directly or indirectly referenced render information objects.

An exception to this rule is to create e.g. a color definition with the same id as the color definition in a referenced style in this case interpreting programs can assume that this color definition is supposed to override the color definition with the same name in the referenced render information object. Likewise it is also possible to override a color definition with a gradient and vice versa, line ending definitions on the other hand can only be replaced by other line ending definitions.

# 11 Appendix A

The mapping of arrow heads to line endings involves some transformations which we would like to illustrate with two examples. The first example as depicted in Figure 9 defines a straight line and a line ending which is to be applied to the end of the line. The line ending specifies a bounding box with a size of 4x4 and a position of $P(-2, -2)$. The origin of the line ending is at $o(0.0, 0.0, 0.0)$ and the bounding box extends along the positive x- and y-axes. The position of the bounding box is the offset by which the origin of the bounding box has to be translated from the endpoint of the curve.

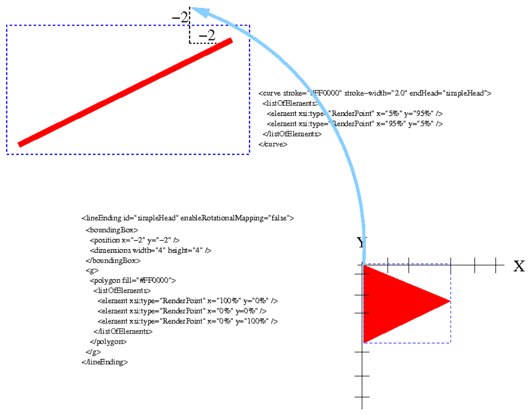


Figure 9: Curve with arrow head and no rotational mapping

Since the arrow head in the first example explicitly disables rotation mapping by specifying **enableRotationalMapping=false** in the definition of the line ending, the process of mapping the arrow head to the line is simply a matter of moving the origin of the line endings coordinate system to the end point of the line $E(ex, ey)$ plus the offset that is specified as the position $P(px, py, pz)$ of the line endings bounding box $F = E + P = (ex + px, ey + py, ez + pz)$. In our example the origin of the line endings coordinate system has to be moved 2 units up and two to the left of the and of the curve that the line ending is applied to.

The result of this operation is depicted in Figure 10.

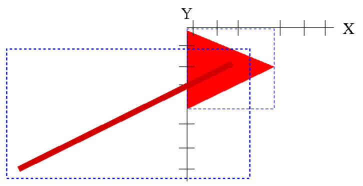


Figure 10: Curve with mapped arrow head and no rotational mapping

The second example is very similar to the first example, only now, the rotational mapping for the arrow head is enabled. This means that we now have to execute two steps in order to map the arrow head to the line ending.

First we need to rotate the arrow head so that the x-axis of the arrow heads coordiante system is aligned with the slope $s = \frac{dy}{dx}$ of the curve.

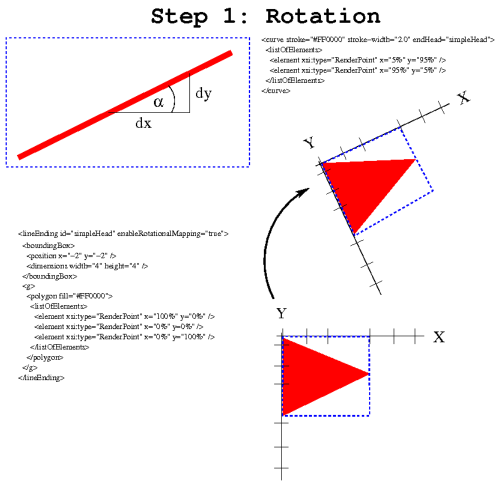


Figure 11: Step 1: Rotation

The rotation of the arrow head involves the following steps:

1. calculate the normalized direction vector of the slope:
   We first need to find the two points that determine the slope at the end of the line. One point is always the endpoint of the line ($E(ex, ey, ez)$). The second point depends on whether the last element of the line is a straight line or if it is a bezier element. If it is a bezier element, the second point is the second basepoint of the bezier element, if it is a straight line, it is either the preceeding point or the endpoint of the preceeding bezier element. We call this second point $S(sy, dy, sz)$.

The direction vector can be calculated as $v(vx, vy, vz) = (ex - sy.ey - sy, ez - sz)$. To normalize the vector we have to calculate the length $l = \sqrt{vx^2 + vy^2 + vz^2}$ of the direction vector and divide all elements of $v$ by this length. $v_n(v_n x, v_n y, v_n z) = (vx/l, vy/l, vz/l)$

2. calculate the normalized vector that is

   (a) orthogonal to the direction vector of the line

   (b) located in the plane x- and y-axis

   If the direction vector is parallel to the y-axis ($vx = 0.0$), the orthogonal vector $w$ is parallel to the x-axis ($w(vy, 0, 0)$). For all other cases $w$ is $w(wx, wy, wz) = (-v_n y * v_n x, 1 - v_n y^2, -v_n y * v_n z)$.
   Again we have to normalize this vector by dividing through its length $n = \sqrt{wx^2 + wy^2 + wz^2}$, which results in the normlized vector $w_n(w_n x, w_n y, w_n z) = (wx/n, wy/n, wz/n)$.

3. create the transformation matrix that converts the original coordinate system into the coordinate system that is made up of the two calculated vectors:
   The transformation matrix that results from the two normalized vector that we calculated in the steps above is $m = \begin{pmatrix} v_n x & w_n x & 0.0 & 0.0 \\ v_n y & w_n y & 0.0 & 0.0 \\ v_n z & w_n z & 0.0 & 1.0 \end{pmatrix}$

The second step moves the origin of the arrow heads coordinate system to the endpoint of the line, which is exactly the same as we did in the first example.

Mapping of an arrow head to the beginning of a curve is exactly the same as for the end of a curve, only the direction of the line has to be reversed and in case of a cubic bezier, one has to use the first basepoint rather than the second basepoint.

# 12 Changes

## 12.1 Draft ??/??/2009

- Add the **backgroundColor** attribute to the render information objects

- **fill-rule** in **GraphicalPrimitive2D** no longer has a default value. If there was a default value, the inheritance of attributes would not work

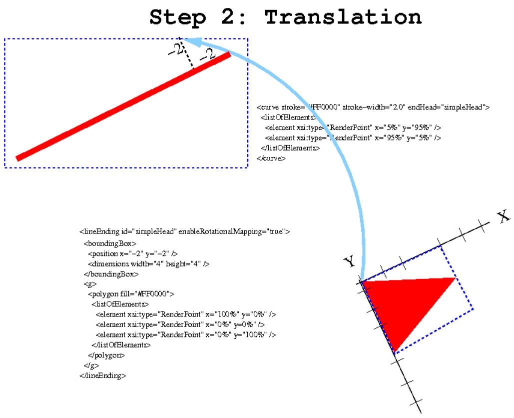


Figure 12: Step 2: Translation

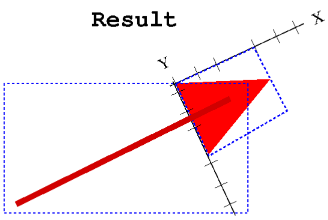


Figure 13: Curve with mapped arrow head and rotational mapping

because one can not distuinguish between a default value and a value set by the user.

- Rephrase the paragraph about default values for group elements. It now states that only the outermost group in a style has default values for its arguments, all elements within that group don't have default values for any of their attributes.

- Clearify what **rx** and **ry** in the rectangle relate to. So far it was not specified if they relate to the size of the bounding box or the size of the rectangle. Now it is made clear that they are relative to the size of the rectangle.

- Add some explanations about handling gradient stops.

- Add documentation on handling certain cases for center and focal points values on radial gradients.

- Remove curves from the list of elements that have a **fill** attribute, because they don't since they are derived from **GraphicalPrimitive1D**.
- Add some words about accumulating transformations.
- Removed the `inherit` type for **fill-rule** because fill rules are inherited from the group anyway if they are not specified.
- Added an attribute, `vtext-anchor`, to texts to specify a vertical alignment for a text. The same attribute has also been added to the `group` element.
- Rewrote how to interpret the offsets on a `text` element.
- Line endings now inherit all attributes from the line they are applied to save for the line ending attributes themselves since this would lead to an endless loop.
- References in images can only be to local ressources. Image files from the net or other places are not supported.
- Specify that general transformations as specified by the **transform** attribute are to be applied to objects before any other transformation, e.g. offsets.
- Added some more sentences on how transformations are to be applied to objects, and what coordinate system they apply to.
- Specified what happens with line stippling if a layout curve has gaps.
- Changed the curve and polygon specification to simplify the design.
- Added some examples for vertical text placement.
- Rewrote and simplified the section about the placement of line endings.

## 12.2 Draft 01/30/2008

- The **LocalRenderInformation** and the **GlobalRenderInformation** type now have a common base class called **RenderInformationBase**.
- All classes for rendered 2D objects are now derived from the new classes **Transformation** and **Transformation2D**. The **Transformation** now holds the **transform** attribute which has been part of **GraphicalPrimitive1D**. The consequence of this is that **Images** which are now also derived from **Transformation2D** can be transformed.

- The section on transformations has been extended to explain what the six elements of the **transform** attribute represent.
- The **fill-rule** attribute has been missing from the **Group** class and has now been added. Some more small changes in the section about grouping.

Thanks to Frank Bergmann for the valuable feedback, for providing me with examples and his help in testing the implementation.