SVG Graphics

SVG stands for Scalable Vector Graphics. It is an XML-based format for vector graphics suitable for Web. An SVG picture can be an integral part of HTML document or a standalone picture. Advantages: As a vector image it can be scaled. It can be created and edited in a text editor, and works well with CSS, Java and other web technology.

1. Basic structure

An SVG picture is enclosed in a paired `<svg>` tag. This then can be normally manipulated using html tools, e.g.

```
<p style="text-align:center">
  <svg>
    specification
  </svg>
</p>
```

If it is a standalone picture, then it should also have the appropriate XML header, and a specification in the `svg` tag:

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg">
  specification
</svg>
```

The `svg` tag can contain a number of attributes. Some are global, some can be global but one can also use them with individual elements. Typical basic example:

```
<svg viewBox="0 0 180 50" width="180" height="50" stroke="black" border="1 mm">
```

Canvas settings:

The canvas is set up automatically by the `svg` element. The x-axis points right, the y-axis points down. The default unit is pixels (`px`), and the default place for the origin $(0, 0)$ is in the upper-left corner of the canvas. Further specifications are done globally by attributes of the `<svg>` element.

• `viewBox="`*a b c d*`"` sets up the actual canvas and coordinate system we plan to use, where (a,b) are the coordinates of the upper-left corner and c, d define the width and height of the canvas, so in fact its lower-right corner has coordinates ($c-a$,$d-b$). For instance, after `viewBox="0 0 13 12.7"` we would address the upper-left corner as $(0, 0)$, which is the default; on the other hand, after `viewBox="-200 -200 400 400"` we get a canvas where the origin is located in the middle. Graphical elements outside of the declared viewBox are not rendered.

• `width=`*dim* and `height=`*dim* set up the size for the picture when shown, unless overridden by outside parameters like CSS. It can be in pixels (`width="200"`), but other units are possible, e.g. `width="2 cm"`.

• `preserveAspectRatio` determines how the picture is treated when it is scaled to conform to `width` and `height` specifications. When not present, it is interpreted as `preserveAspectRatio="none"` and the picture is stretched in both directions as needed.

`preserveAspectRatio="xMaxYMax meet"` will place the lower-right corner of the picture to the lower-right picture of the viewing area and stretch the picture while preserving the aspect ratio as much as possible while it fits inside. `"xMinYMax meet"` aligns the lower-left corners. Other combinations are possible. `xMidYMax` will align the lower edge (masimal values of y for the viewport and the image's viewBox) and aligns the x-center of the picture with the x-center of the viewport before scaling.

`preserveAspectRatio="xMaxYMax slice"` and other combinations will align the picture according to the specification, but scale it so much that the entire viewport is filled, and the excess parts of the picture are not shown.

Style attributes:

Global style specifications can be placed directly in the `<svg>` tag, either as individual specifications (see `stroke=` in the example above) or as one style specification (in later versions of SVG):

```
<svg style="stroke:#000000;stroke-width:5;">
```

When the picture is a part of a web-page, it is recommended to do global settings using the CSS approach, in this case we would place

```
svg{
    border: 1px solid;
    stroke: #000000;
  }
```

in the style section of the HTML header.

For local settings we place these attributes directly in graphical elements. There we can place them directly, or as a `style` attribute with colons. The latter appeared only in later versions of SVG.

• `border="`*dim*`"` defines border around the whole picture. When not set, no border is drawn.
• `stroke="`*color*`"` sets up the color for the lines. We can use names (`green` etc.), RGB numbers `#0000FF` and `rgb(0,0,255)`, or inherit the text color from the parent elements with `stroke="currentColor"`. Note that a graphical element with undefined stroke will be constructed but not drawn.
Some popular colors: `white`, `black`, `green`, `blue`, `red`, `yellow`, `lime`, `purple`, `violet`.
We use `stroke:`*color*`;` when defined as style.
• `stroke-width="`*dim*`"` sets up the thickness of lines. `stroke-width="5"` renders lines 5 px thick.
• `stroke-linecap="round"` sets up how lines should end. Other values are `butt` and `square` where a square is attached at the end of the line, so it exceeds the starting and ending coordinate by half its width. The last possible value is `inherit`, where the rule is taken from a parent element.
• `stroke-linejoin="mitter"` sets up how lines should join. This makes sharp corners, the other options is `round` and `bevel` with a squarish joint. The length of the spike when two lines join can be restricted by `stroke-miterlimit="`*dim*`"`.
• `stroke-dasharray="`*dim dim ...* `"` defines the pattern for a dashed line. The numbers (separated by spaces or commas) determine the length of black and white (more precisely, stroke and background colored) segments, for instance `stroke-dasharray="4 1 2 3"` would draw a black segment of length 4px, then white of length 1 px etc. When an odd number of length specifications is given, the pattern is doubled, so for instance `stroke-dasharray="2 1 2"` is interpreted as `"2 1 2 2 1 2"` and produces long black, short white, long black, long white, short black, long white. In particular, `stroke-dasharray="13"` gets doubled to the pattern `"13 13"` and produces evenly dashed line.
• `stroke-dashoffset="`*dim*`"` defines the offset of the pattern defined by `stroke-dasharray`.
• `stroke-opacity="`*number*`"` sets opacity of lines. The values are 0 through 1.
• `fill="`*color*`"` specifiers whether inside of curves should be filled (when a color is specified) or not (value `none`). Another interesting value is `transparent`. Note that the default is filled, with the pen color.
• `fill-rule=` specifies how the regions to be filled are determined. The default is `nonzero`, other values are `evenodd` and `inherit`, where the rule is taken from parent elements.
• `fill-opacity="`*number*`"` sets opacity of fills. The values are 0 through 1.
• `opacity="`*number*`"` sets opacity of the whole element. The values are 0 through 1.

Transformations:
The attribute `transform` expects a list of transforms separated by spaces.
• `translate(`*x y*`)` can also have the coordinates separated by a comma. The *y* specification can be omitted, then it is assumed to be 0. There are also specialized versions `translateX` and `translateY`.
• `rotate(`*angle*`)` causes a clockwise rotation in degrees around the origin. If we want a different center of roation, we use `rotate(`*angle cx,cy*`)`. There are also versions `rotateX` and `rotateY`. When specifying the angle, we can use the special unit `turn`, as in `rotate(0.5turn)`.
• `scale(`*c*`)` scales the image *c* times. We apply different scaling in the basic directions with `scale(`*cx cy*`)`. Specific attributes are `scaleX` and `scaleY`.
• `skewX(`*angle*`)` and `skewY(`*angle*`)` skew the image in the given direction. We can also use `skew(`*angle*`,`*angle*`)`
• `matrix(`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`)` uses the transform $x = ax + by + c$, $y = dx + ey + f$.
Or(?), the following interpretation: *a* is `scaleX`, *b* is `skewY`, *c* is `skewX`, *d* is `scaleY`, *e* is `translate x` and *f* is `translate Y`.

2. Graphics elements

These are typically non-paired, and as such they use the XML convention of ending with `/>`. The first SVG version featured paired elements, and both should work well. The attribute `stroke` must be set for the shape to be drawn.
When shapes from elements overlap, the later specification is drawn over the previous ones.

2a. Lines

The element `<line x1="`*a*`" y1="`*b*`" x2="`*c*`" y2="`*d*`" />` draws a line from the point (*a*,*b*) to the point (*c*,*d*). As usual, missing units are interpreted as pixels. It allows attributes influencing the line.
```
<line x1="10" y1="23" x2="310" y2="13" stroke="red" stroke-width="2mm" />
```
We can also use the CSS form
```
<line x1="10" y1="23" x2="310" y2="13" style="stroke:red;stroke-width:2mm" />
```

The element `<polyline points="`*points*`" />` draws a broken line connecting the specified points. The basic form of specification is a list of coordinates separated by spaces *x1 y1 x2 y2 ...* , but we can insert commas to add readability, or split it across lines.
```
<polyline points="0 0, 30 45, 78 10" />
```
Some people prefer `<polyline points="0,0 30,45 78,10" />`

2b. Polygons

The element `<rect x="`a`" y="`b`" width="`c`" height="`d`" />` creates a rectangle with upper-left corner given by optional arguments (a,b) and dimensions as specified. We can also supply `rx` and `ry` as radii for rounded corners.

```
<rect x="10" y="13" width="150" height="100" rx="5" ry="10" fill="#BBC42A" />
```

The element `<polygon points="`*points*`" />` draws a broken line connecting the specified points that also cycles back to the first point. The point specification is just like for `polyline`.

`<polygon points="0 0 30 0 15 20" />` draws a triangle.

2c. Circles

The element `<circle cx="`a`" cy="`b`" r="`R`" />` draws a circle centered at (a,b) with radius R.

```
<circle cx="25" cy="74" r="13" fill="black" stroke="green" stroke-width="4" />
```

If no center is specified, then the default $(0, 0)$ is used.

The element `<ellipse cx="`a`" cy="`b`" rx="`A`" ry="`B`" />` draws an ellipse centered at (a,b) with horizontal half-axis A and vertical half-axis B.

```
<ellipse cx="50" cy="50" rx="30" ry="16" style="stroke:blue;fill:green;opacity:0.5" />
```

2d. Path

The element `<path d="`*specification*`" />` is the most powerful element for drawing. The specification consists of pairs (separated by spaces or commas where necessary) of the form *action data*. Each action is called by a specific letter. If this letter is upper-case, then the data is understood to be absolute. If this letter is lower-case, the data is understood to be relative to the latest position. Note: The "latest position" means the latest position of the previous action, that is, the current point when the action started.

Actions:

• `M` is Move-to and expect two numbers. The currect position is moved to the new one (relative when `m` is used) without creating any graphics.

• `L` is Lineto and expects two numbers. A straight segment is drawn ending at the specified position (absolute or relative when `l` is used).

• `H` is Horizontal Lineto and expects one number. A horizontal straight segment is drawn to location whose x coordinate is specified and y coordinate is taken from the current point.

• `V` is Vertical Lineto and expects one number. A vertical straight segment is drawn.

• `Z` is ClosePath and does not accept arguments. The latest point is connected with the first one using a straigh segment, creating a closed path. Line-joining rules apply when closing the path.

The following codes all create the same triangle.

```
<polygon points="0 0, 30 0, 15 20" />
<path d="M 0 0 L 30 0 L 15 20 Z" />
<path d="M 0 0 l 30 0 l -15 20 Z" />
<path d="M 0 0 h 30 l -15 20 Z" />
<path d="M0,0h30l-15,20Z" />
```

Note that we set a new "first point" every time we use the Move-to operation.

• `A` is Arc and expect seven numbers: *rx ry ang l s a b*.

The point (a,b) determines the end of the arc, relative if the action `a` is used. The arc is taken from an ellipse with half-axes *rx* and *ry* that is rotated by *ang* degrees. Finally, *l* is the "large-arc" flag with values 0 (small arc is used) or 1 (large arc); and *s* is the 'sweep" flag with values 1 (clockwise) and 0 (counter-clockwise).

For instance, we get a circle using

```
<path d="M 30 20 a 10 10 0 0 0 -10 -10 a 10 10 0 0 0 -10 10
    a 10 10 0 0 0 10 10 a 10 10 0 0 0 10 -10 z" />
```

We put the `z` at the end to specify that we want the curve closed.

• `Q` is Quadratic Bezier Curve. A quadratic Bezier curve is determined by its endpoints and one control point that determines directions of the curve at both endpoints. The current point is one end, so two pairs of points are expected:

`<path d="Q`*c d a b*`" />` draws a curve from the current point to the endpoint (a,b) with control point (c,d).

• `T` is used for a continuation of an existing quadratic Bezier curve segment, with the control point automatically chosen so that the ingoing and outgoing angle at the current point are equal and the resulting curve looks good. Just one point is expected, the endpoint of this segment. In this way one can create complex shapes by repeating the `T` action with anchor points.

When a `T` action is not preceded by another `T` or `Q` action, the `L` action is used instead.

• `C` is Cubic Bezier Curve. A cubic Bezier curve is determined by its endpoints and each also has a control point determining the direction of the curve at this endpoint and the intensity of this influence. The current point is one end, so three pairs of points are expected:

`path d="C`*c1 d1 c2 d2 a b*`"` draws a curve from the current point with control point (*c1*,*d1*) to the endpoint (*a*,*b*) with its control point (*c2*,*d2*).

• `S` is used for a continuation of an existing cubic Bezier curve segment. The first control point is determined as a mirror of the second control point of the previous endpoint (now the current point), so that the curve connects smoothly. Thus only the endpoint and its control point need to be specified.

```
<path d="M0,0 C5,5 10,5 15,0 S25,-5 30,0" />
```

or

```
<path d="M0,0 C5,5 10,5 15,0 s10,-5 15,0" />
```

Heart:

```
<path d="M10,30 A20,20 0,0,1 50,30 A20,20 0,0,1 90,30 Q90,60 50,90 Q10,60 10,30 z" />
```

2e. Text

The paired element `<text x="`*a*`" y="`*b*`">` places text in the picture. It is aligned so that the first character starts horizontally at location *a* and its baseline is at level *b*.

```
<text x="3" y="13">This is a text</text>
```

We can also use `dx=` and `dy=` for relative coordinates with respect to the previous text element.

The actual position can be changed using the `text-anchor` option. The value `start` is the default, the left edge of the first character is placed at the specified location. The value `end` positions the text so that the right edge of its last character is at the specified location. The third possibility is `middle`.

The SVG engine does not recognize named entities, so instead of `&alpha;` we have to use `&#945;` etc.

The color of the text is (surprisingly) defined by the `fill` attribute. The usual `font-size="`*dim*`"` and `font-family=` attributes apply. One can use a generic font name or a font family name. More fonts can be listed and priority is from the first, so font families should come first and a generic font name last as a fallback mechanism.

`serif` is a generic name for fonts with serifs, popular families are `"Times"`, `"Palatino"`, `"Lucida Bright"`.

`sans-serif` is a generic name for font families like `"Arial"`, `"Helvetica"`, `"Verdana"`, `"Trebuchet MS"`, `"Open Sans"`, `"Lucida Sans"`.

`monospace` is a generic name for computer-like font families, for example `"Courier"`, `"Courier New"`, `"Lucida Console"`.

`cursive` is a generic name for italic font families like `"Comic Sans MS"`, `"Apple Chancery"`, `"Brush Script MT"`.

`"fantasy"` is a generic name for decorative styles.

One can also try things like `"Leckerli One"` for some fun.

Most families can be further modified using `font-style=` with values `normal`, `oblique` and `italic`, another modifier is `font-weight=` with values `normal` and `bold`, but also numerical value, with 500 being normal and 700 bold. Another attribute is `font-variant=` with many possible values, including `normal`, `none`, or `small-caps`.

Finally, we have the `font-size=` attribute that accepts a dimension, or a percentage, or predefined sizes `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx-large`, and also relative sizes `smaller`, `larger`. We can also use `inherit` or `initial`.

The attribute `text-decoration=` accepts values `underline`, `overline` and `line-through`.

The attribute `textLength="`*dim*`"` specifies to what length the text should be scaled, and `lengthAdjust` determines how the scaling is achieved, with values `spacing` and `spacingAndGlyphs` that also widens characters if necessary.

We can influence spacing of characters with `kerning=`. The default value is `auto`, we can also use `inherit` or supply a numerical value that spreads characters. This can be further modified with the `letter-spacing=` attribute, with values `normal`, `inherit` or a numerical value.

There are more attributes that can manipulate the text. We can use the standard transformations that apply to the whole text.

```
<text x="3" y="13" fill="red" transform="rotate(90,20,13)">This is a text</text>
```

There is also a special attribute `rotate="`*numbers*`"`. The first number specifies rotation for the first character, the second number for the second character etc., the last number applies to the rest of the text.

Finally, we can enclose some parts of the argument of a `text` element using `<tspan>` and `</tspan>`. This allows a part fo the text to be treated differently, changing colors, locations and orientations. The following sets the text in three lines.

```
<text x="10" y="10" fill="black">One <tspan x="10" y="30">two </tspan>
    <tspan x="10" y="50">three</tspan></text>
```

We could also use

```
<text x="10" y="10" fill="black">One <tspan x="10" dy="20">two </tspan>
    <tspan x="10" dy="20">three</tspan></text>
```

as the relative dimensions are taken with respect to the previous part of text. Note that this also sets up a new position for the text that comes afterwards. Thus

```
<text x="10" y="10" fill="black">One <tspan dy="20" font-size="larger">two </tspan>
    <tspan dy="-50">three</tspan></text>
```

will print the three words one after another, with "two" printed 20 pixels below the baseline (y-axis points down) and larger, while "three" is printed $20 - 50 = -30$ pixels below the baseline, that is, above the baseline.

Special rule: The `dy` attribute also accepts a list of values, then the first applies to the first character, the second to the second character etc., the last applies to the remaining characters of the argument of this `tspan`.

The paired element `<textPath>` should be placed within the `<text>` environment, and expects a reference to a curve that we have set up previously (see `defs` below). The argument is then printed along this curve.

```
<defs>
  <polyline id="peak" points="0,0 30,30 60,0" />
</defs>
<text fill="black">
    <textPath href="#peak">This text goes up and down.</textPath>
</text>
```

If the path to be used is an actual `path`, we can specify it directly in the `<textPath>` specification as the attribute `path=`. We can adjust the length of the text as with `<text>`. We can also specify the side of the path on which the text should be positioned using `side=` with default value `left` and the alternative `right`. When characters are placed, they are rotated and moved. Using the attribute `method` we can ask for preservation of shapes (value `align`) ot deformation allowing for better flow of text (value `stretch`).

We can also specify how far along the curve the text should start using `startOffset=` that expects either a dimension or a percentage.

3. Structural elements

`<g>` marks a **group**, a container for grouping related graphics together.

`<g id=`*name*`>`
elements
`</g>`

This example shows one use of grouping: This shape is rendered, but also stored under the *name* and can be used again.

The second important use of groups is when we want to apply common transformations to more objects.

```
<g name="rotated">
<rect x="3" y="3" width="14" height="14" stroke="red" />
<circ cx="10" cy="10" r="5" />
</g>
```

`<defs>` also groups elements into a group that can be named, but they are not rendered, just named.

```
<defs name="heart">
<rect x="3" y="3" width="14" height="14" />
<circ cx="10" cy="10" r="5" />
</defs>
```

`<symbol>` groups and draws the elements just like `<g>`, the difference being that it accepts more attributes. A very useful one is the `viewBox` attribute that sets up the dimensions of this particular group. This allows it to be scaled when named and placed.

`<use>` is the element that places named groups in a picture. Attributes can be used, but only the `x`, `y`, `width`, and `height` attributes take precedence, the others (colors etc.) will be overridden by the attributes of the groups we are referring to if present.

```
<use href="#heart" stroke="blue" />
```

Here the blue color will be used.

```
<use href="#rotated" stroke=blue />
```

Here the rectangle will be drawn in red.

We need not use groups to create named objects:

```
<circle id="dot" cx="5" cy="0" r="3" fill="blue" />
<use href="#circle" x="10" y="0" />
<use href="#circle" x="15" y="0" />
```

The `x` and `y` attributes are used to shift the coordinates in the referred group. The `width` and `height` attributes have effect only when referring to a `symbol` element.

4. Clipping

Sometimes we only want a part of the shape that we draw to actually appear. First we define a mask that determines what will be visible. To this end we use the `clipPath` environment within which we define the shape of the mask using `path` or similar drawing commands. Since we will be referring to it later, we have to assign it a name.

```
<clipPath id="MyClip">
  <rect x="-100" y="0" width="200" height="100" />
</clipPath>
```

Then we can draw the shape to be clipped as usual, but we add a special option referring to our mask:

```
<circle cx="0" cy="0" r="100" clip-path="url(#MyClip)" />
```

Only the part of the circle inside our mask is drawn, in this case it is the lower half of the circle.

For more complicated curves we may need to specify how the inside is determined. This is done using the option`clip-rule` with the same values that we saw earlier for fill. For instance,

```
<circle cx="0" cy="0" r="100" clip-path="url(#MyClip)" clip-rule="evenodd" />
```

5. Note on dimensions

Sometimes we want to specify specific dimensions in real-world units, for instance milimeters. Some attributes allow for it, for instance `stroke-width`, or the `width` attribute of the `rect` element, but others do not, for instance the specification of the `path` element. This duality causes trouble when we want to combine several elements with specific dimensions, but some are capable of direct unit specification and some are not.

One approach is to draw such elements with pixel specifications and scale them accordingly. One milimeter is typically 3.7795275591 pixels, so we can draw, say,

```
<rect x="20 cm" y="40 cm" width="10 cm" height="13 cm" />
<path d="M 20,40 A5,5 0,1,1 30,40" transform="scale(3.7795)" />
```

and the two shapes will fit, forming a Roman window with rounded top.

This has some disadvantages. One is related to the fact that various visualisation devices may have slightly different pixel to mm conversion, throwing our pictures off. Another disadvantage is that svg has in fact two coordinate systems, and mixing them may easily lead to clashes that are hard to work out without a deeper understanding of svg mechanics.

Thus it is better to keep real dimensions out of element specifications and stick with the default, that is, the pixels. Then we can scale the resulting picture using the `width` and `heigh` parameters of `<svg>`.

For instance, assume that we set up the basic canvas with dimensions 400×500 using `viewBox`. If we add `width="400 mm" height="500 mm"` to `<svg>`, then the resulting pictures will be rescaled, in fact it will be enlarged by the coefficient 3.78 or so. In effect, all numbers in element specifications will be interpreted in milimeters. If we add `width="400 cm" height="500 cm"` to `<svg>`, then all numbers in element specifications will be interpreted in centimeters.

Setting up `viewBox="0 0 2100 2970" width="210 mm" height="297 mm"` we obtain a picture with dimensions of the standard A4 paper size where coordinates are interpreted with respect to the unit 0.1 mm. This has two advantages. First, we can obtain a more precise placing without the need for decimal numbers. The second advantage is related to text. Note that in the setup that we have here the actual shapes will get shrinked when fitting the prescribed dimensions. This should cause no troubles.

On the other hand, consider the setup `viewBox="0 0 210 297" width="210 mm" height="297 mm"`. Now the numbers are interpreted as milimeters, and the picture that we draw is enlarged to fit the size. When we place some text, then the thickness of letters is first taken according to our specification, but then it is adjusted so that it is visible/printable in the original (pixel) size. Since we eventually intend to enlarge it almost four times, then it is very likely that the letters are very small in the original picture, and therefore they will be drawn significantly thicker than we want them to be without us being able to do anything about it. After the resulting picture is enlarged to the desired size, our text will look like boldface (or worse).