

<Angle-brackets/> on the Branch Line

John Lumley, jωL Research,

Abstract

As a retirement 'hobby', somewhat removed from the computing *milieu*, the author has started building a model railway in his garden. Surveying the extant tools for designing such layouts and finding them “not quite right”, he started building a design tool himself, using the familiar technologies of XSLT₃ and SVG executing in a browser, employing Saxon-JS as the processing platform. By adding animations, the tool expanded beyond simple design to in effect become an active model train system. The results of this were demonstrated, with some success, at Markup UK in 2018. This paper describes the design of this tool in some detail, as well as possible developments since that demonstration.

1. Introduction

The author decided to take up a retirement “hobby” as a change from wrestling with programmatic complexities. Having chosen to build a garden railway, having been trained as an engineer and having read some of the sage advice from those already “in the scene”, it was clear that the layout would need some careful design. Issues such as maximum gradients, minimum turn radii and *loading gauge* clearances required a clear and calculated design. Naturally there are CAD tools specifically targetted at model railways, but equally well, I found none of them to be *just right*.

So, having spent many years developing software, and in recent times being deeply immersed in XML technologies, particularly XSLT_{3.0}[XSLT] and SVG[SVG], I decided to build a specific design tool with these technologies. Given Saxon-JS[Saxon-JS] as the XSLT execution engine, the tool was run through a browser connecting to a *localhost* web server.

The main design used an XML definition of garden “background” and the possible layouts, and at first calculated all the resulting geometry, producing both a tabular summary and a set of SVG graphic elements that could be displayed on a grid. This permitted for example interferences between tracks and garden elements (e.g. bushes) to be examined. Simple XHTML controls were added to allow various aspects of the display to be altered dynamically, using Saxon-JS's interactive modes (e.g. `ixsl: on-change`) to alter style or class properties of parts of the XHTML/SVG DOM tree. Textual styling (fonts, colours. etc.) were defined in a set of CSS stylesheets.

Once a simple system was operational, the “picture” was enhanced, both by supporting an isometric view of the garden/layout, but also more “realistic” graphics for the track and other aspects.

A little experimentation showed that the animation facilities present in SVG should allow objects to move around the paths of the track. A simple facility was added to enable “block” objects to be run, moving from section to section under controllable and alterable speeds. Simple click interaction allowed the points to be changed, so the path of these blocks could be altered whilst they were still running.

The model for these “locomotives” was improved to support a three-dimensional definition consisting of a number of orthogonal rectangular blocks and cylinders, from which an isometric SVG view of a simple locomotive could be displayed. This would then be animated to follow the path of the current track section, with tangential rotation to “point forward” and with suitable rotation animations on the wheels. Simple sound effects (running sound, whistles etc.) were added to the design.

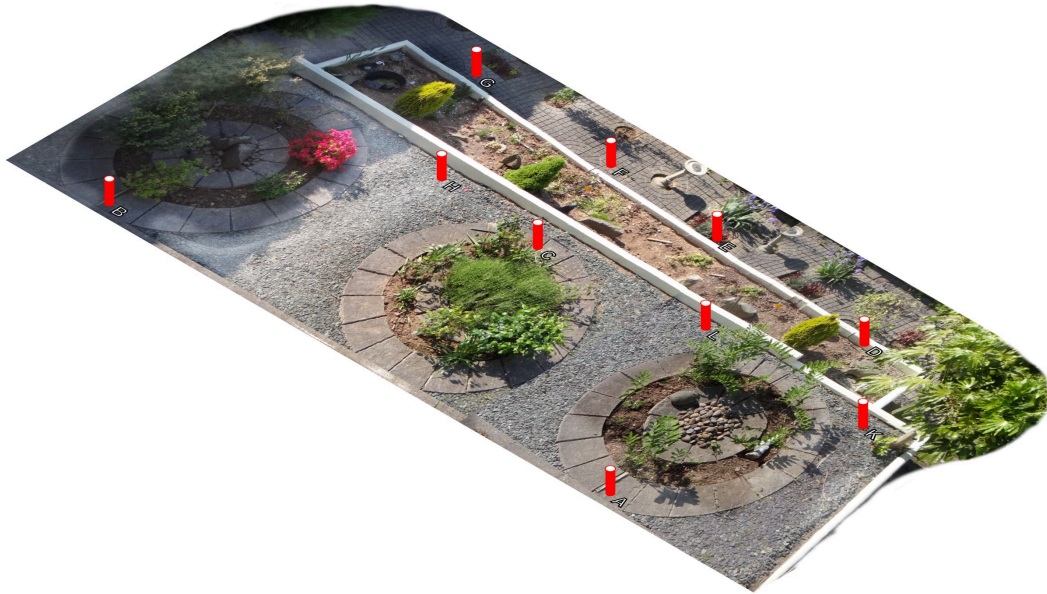
Finally, this system was demonstrated at MarkupUK 2018 in the DemoJam session, with some success.

In this paper I describe the deign and implementation details of the system that was demonstrated, and outline some additional possible developments. In conclusion I discuss how suitable the combination of XSLT₃, SVG and Saxon-JS has been to tackling this design task.

2. Overall Design

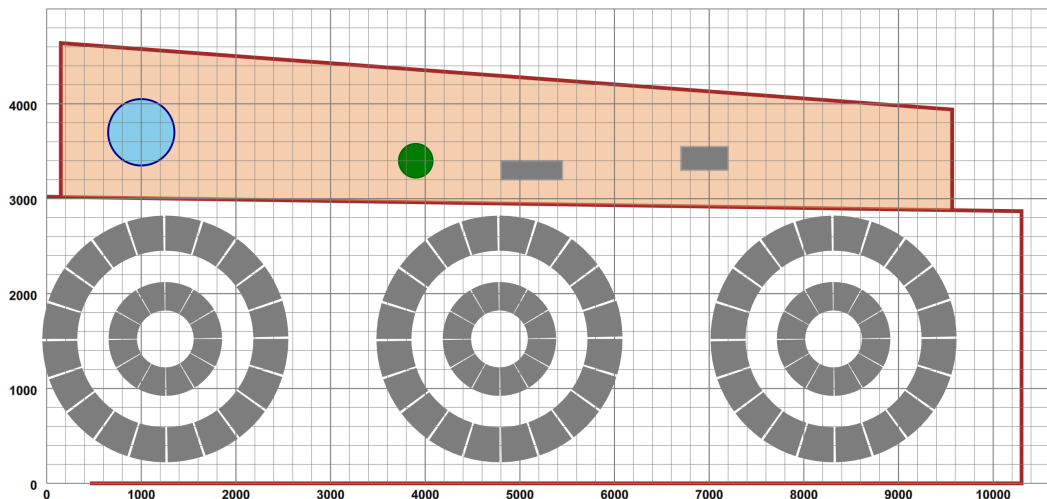
The system designed is of course influenced by the external factors of the garden itself and the track components from which the railway will be built. The garden area chosen is approximately 10m x 4m on two levels:

Figure 1. The Garden for the Railway



Both levels are substantially flat¹ with a step of about 250mm between, so it should be possible to climb a connecting embankment. (Generally gradients should be less than 1 in 40 and certainly no more than 1 in 25). The area was surveyed (marked by the red survey points shown above) and a simplified plan of the garden drawn up:

Figure 2. The garden plan



I decided that the railway would be built at SM₃₂ gauge/scale, also known as 16mm. The track has a gauge of 32mm, and is taken to represent a 2ft narrow-gauge line² so the scale is 16mm to the foot or 1:19. As such, models of narrow-gauge locomotives are large enough to be totally steam-powered. The tracks themselves would not carry electrical power — all engines would be self-powered, and remotely controlled. The commercially available track had a small set of points of different tightness and flexible track sections of some 900mm length. This meant that apart from the fixed-design points, the rest of the track could be “freeform”, subject of course to a recommended minimum turn radius, which whilst being dependent upon locomotive wheelbase, would be about 900-1000 mm.

The original design consists of five major sections:

¹Only when laying out the track bed did it become apparent that several elevation changes O(50mm) existed on the upper level.

²Many of the UK’s “little trains”, such as the Ffestiniog and the Talylyn, run on 1’ 11½” gauge track

- A declarative description, as an XML structure, of the design environment, consisting of background components (e.g. pictures of the garden and schematics of fixed sections such as walls, paving and plants) and a series of layouts. A layout is described as a sequence of (mainline) track sections of straights, curves and points, each represented by an XML element describing length, radius and/or turn angle. Branch lines are children sequences of a **point** element. Where necessary track connections between leaves of the tree are joined to make a complete graph through named link declarations.
- A geometry computational engine, written in XSLT₃, which calculates the position and orientation of each track section, and produces a map of the layout, keyed by section 'name', each entry describing both the track segments of the section and the two-way connectivity between section ends.
- A graphical display of the design as an SVG tree. Background elements are generated as SVG groups from the environment description. Track components are generated from unit descriptions and positioned with *use* instructions. Within this, some components which can differ in display dependent upon state, such as points, are represented by several views, each classed separately. The overall display can be subject to transform, most notably an isometric one. Textural styling and initial visibility is defined in a series of CSS stylesheets.
- An XSLT₃ stylesheet, using Saxon-JS extensions, and invoked from an outer XHTML document, which populates the XHTML with a series of interactive controls, and generates the detailed layout internal structures and SVG graphics to be embedded in the web page. Templates respond to interaction, such as button state changes, or clicking on points levers, altering the local CSS state of other components and controls.
- Adding “railway engines” as SVG objects, which are presented in both plan and isometric views from a simple “block-and-cylinder” model. An event-based system animates these to run along tangential paths of the track sections, using SVG animation facilities. Speed and direction of travel can be controlled interactively for multiple engines. Events are generated at the conclusion of animations, and are caught by templates that consult the layout map to determine the next sector to enter, then calculate the necessary animation duration, given length and speed, and start up the path-following animation. Speed change involves stopping a current animation, recalculating duration for the remaining section path and restarting a new animation partway through. Issues on collision detection (“train crashes”) will be discussed.

As far as the software mix is concerned, the top-level XHTML document contains some constant background components and **div** containers which will be populated, a **script** element containing a very small set of global JavaScript functions, for primary control of animations and mapping from screen to SVG co-ordinates, and an invocation of Saxon-JS with a precompiled program from an XSLT source of some 20 files and perhaps some 3000 source lines. This program takes as input a file containing definitions of the garden, possible layouts and locomotives. Textural styling is supported by a set of associated (static) CSS files.

There are a number of (Javascript) libraries for supporting SVG effects and animation, and pretty much all the written guides to “advanced” SVG use a combination of some of these, but I wanted to explore how much could be done almost entirely in XSLT_{3.0}. All the programming is limited to XDM data types, XHTML, SVG, CSS and XSLT_{3.0} with Saxon-JS interaction extensions, with a minimum of (perhaps a dozen) globally defined small JavaScript functions, mostly to invoke, query and stop SVG animations.

3. Layout Topology and Geometry

The original motivation was as a tool to design my planned garden railway, in terms of a connected set of track components that satisfied the requirements of i) being constructed from obtainable parts and ii) lay within the limits of bend curvature and track gradient that were recommended for such railways. For the present, given the flat nature of the garden, apart from the step between sections, vertical gradients have been ignored — how they could be added is discussed later.

I considered attempting a “drag and drop” style of interaction, but decided against this, especially as all straights and curves could be “freeform” so a small set of track parts wasn’t really appropriate. The starting point was a choosing an XML representation that focussed on continuous sequences of track components, describing the “main line”, implemented as a sequence of elements, such as:

Example 1. A simple layout

```
<layout name="simple">
  <start x="400" y="400" orient="30"/>
  <straight name="section1" length="1000"/>
```

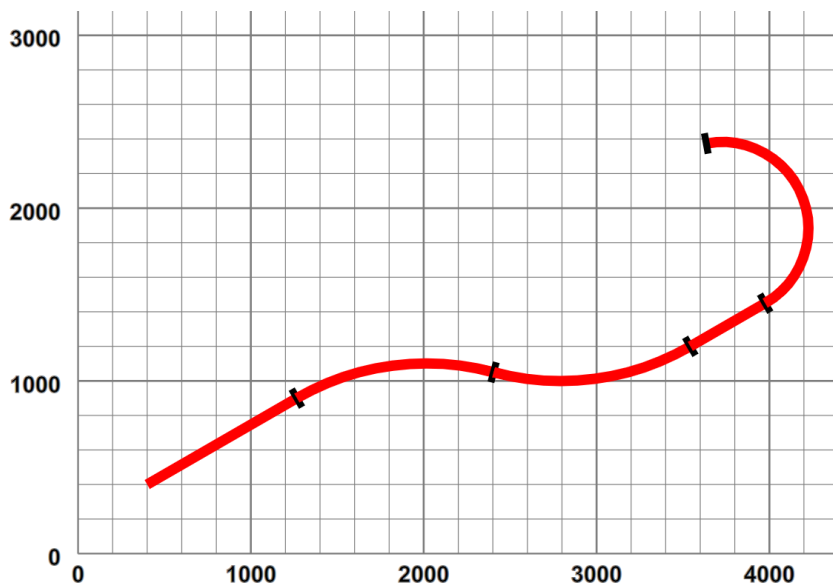
```

    <curve r="1500" angle="-45"/>
    <curve r="1500" angle="45"/>
    <straight length="500"/>
    <curve r="500" length="1400"/>
</layout>

```

which defines a layout *simple* that contains one section *section1*. This starts at the point (400,400) with an orientation of +30° from the positive X axis. The first section is a 1000 long³ straight, preserving orientation, followed by a circular arc curve, of radius 1500, turning left though a positive angle of 45°, followed by a similar right turn, a short straight and a tighter left-hand bend defined by radius and curve length, rather than angle. When plotted out this section looks like:

Figure 3. Simple layout - pictorially



Circular arcs were chosen as the only curve representation as i) they support a design method of “turn this tightly for x degrees”, ii) they are supported directly in SVG and iii) their geometry is simple to calculate. Polynomial splines could have been used, but they are difficult to define in terms of curve length. In real railway engineering, curves are defined by *Cornu spirals* - where the curvature ($1/\text{radius}$) is a piecewise linear function of arc length — lateral (centripetal) acceleration increases at a uniform rate as a train moves along such a curve at constant speed. SVG alas does not support such curves.

Layouts that have such a simple topology (a single contiguous section) tend to be somewhat boring. Alternative routes involve switching between different sections joined by *points*⁴. In our layout definition a point is represented as an element, *whose child is the “branch line”*:

Example 2. A simple branch line

```

<layout name="simplePoint" start="section1">
  <start x="400" y="400" orient="30"/>
  <straight name="section1" length="1000"/>
  <point id="P1" radius="small" turn="left">
    <spur>
      <straight name="branch1" length="580"/>
      <curve r="2000" angle="-40"/>
    </spur>
  </point>
  <curve name="section2" r="1500" angle="-45"/>
  <curve r="1500" angle="45"/>
  <straight length="500"/>
  <curve r="500" length="1400"/>
</layout>

```

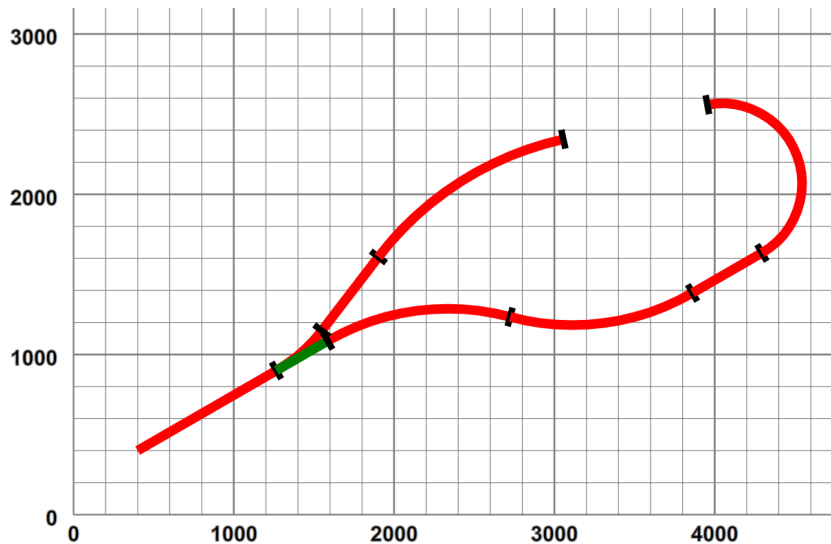
³Any consistent distance units could be used of course, but for this case it's simplest to use millimetres.

⁴In American terminology *turnouts*.

</layout>

The branch line itself is defined by a **spur** element, whose children define a set of sections. The point defines its type, in this case a small radius point and its handedness — here the branch turns off to the left. This layout looks like:

Figure 4. Simple branch line - pictorially



The point obviously has two possible paths, one straight on, the *not-set* track, shown in green, and the turning branch, the *set* track. The layout now consists of three sections, *section1* leading up to the point P_1 , followed by *section2* as the mainline and *branch1* on the branch.

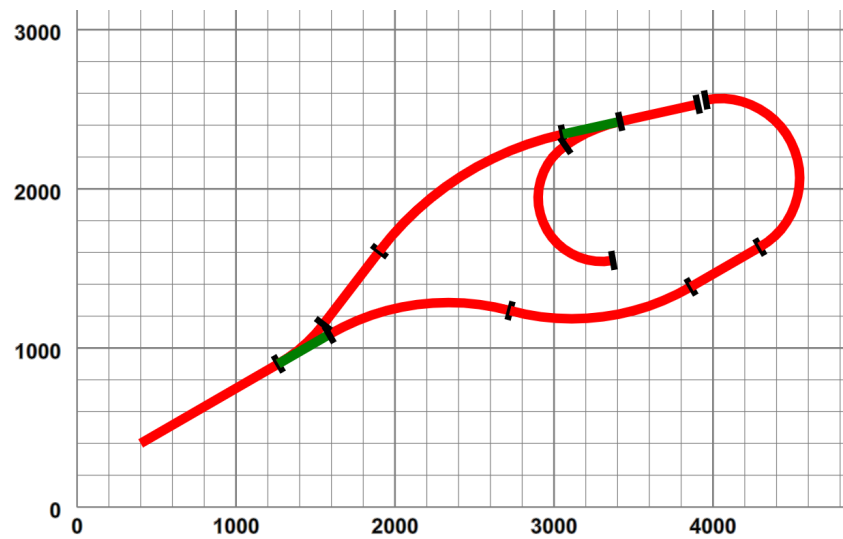
This “tree” representation can obviously be extended, such as adding a point on the branch line, with a sub-branch line such as:

Example 3. A layout with two points

```
<layout name="twoPoints" start="section1">
  <start x="400" y="400" orient="30"/>
  <straight name="section1" length="1000"/>
  <point id="P1" radius="small" turn="left">
    <spur>
      <straight name="branch1" length="580"/>
      <curve r="2000" angle="-40"/>
      <point id="P2" radius="small" dir="trailing" turn="left">
        <spur>
          <curve r="400" angle="155"/>
        </spur>
      </point>
      <straight length="500"/>
    </spur>
  </point>
  <curve name="section2" r="1500" angle="-45"/>
  <curve r="1500" angle="45"/>
  <straight length="500"/>
  <curve r="500" length="1400"/>
</layout>
```

which looks like:

Figure 5. Two points pictorially



Observant readers will note that the new point has been added in technically a *trailing* condition, i.e. proceeding from the start it is only possible to enter the siding in reverse⁵. This leads us on to considerations of representing the layout *topology*.

3.1. Representing the topology

If we want to use a layout for any purpose other than design (such as interactive animation), we don't just need the geometry of the layout: we also need to represent the topology — which sections are joined when points are in a given state? If a train leaves one section, which is the one it will enter, if any? To do this we represent contiguous sections of track and points as components with two or three *ports*:

- face The port which faces against an oncoming vehicle in normal travel, i.e. trains usually start from the face port. For *points* this is the entry from which the exit track (*trail* or *spur*) depends upon the state of the point.
- trail The port from which a vehicle emerges in normal travel, i.e. trains usually end a section leaving the trail port. For points entered in the normal switched direction this is the exit when the point is *not set*.
- spur Only defined for points, the exit port when the point has been *set*⁶.

Using these definitions we can describe the topological relations between component sections in a simple map:

Figure 6. Topology of a two-point layout

id	type	down	parts	length	face	trail	spur
branch1	section	false	2	1,976	P1.spur	P2.trail	
branch2	section	true	1	1,082	P2.spur		
branch3	section	false	1	500	P2.face		
section1	section	false	1	1,000		P1.face	
section2	section	false	4	4,256	P1.trail		
P1	points	false	1	369	section1.trail	section2.face	branch1.face
P2	points	false	1	369	branch3.face	branch1.trail	branch2.face

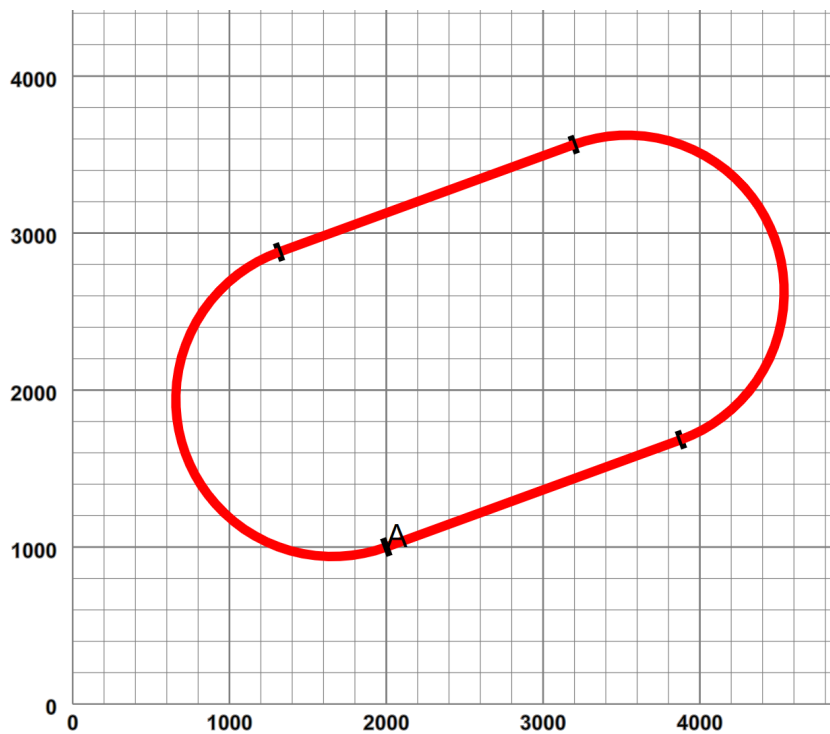
This map has an entry for each component describing its type and its port connections in terms of a component/port pair to which that port attaches. Note however that *branch2* (the “backward” spur from point *P2*) is labelled *down=false*.

⁵Early railway practice only used trailing points on higher-speed main lines, to reduce risk of derailment from partially opened points.
⁶In theory an engine entering a set of points from the *trail* or *spur* direction, when the points are set *against* that direction, i.e. when *set* from *trail* or *not set* from *spur*, may be able to “force” an automatic points switch, but this is not recommended practice.

This means that the “main” direction (i.e. proceeding from P_2 along *branch2*) of that section of track is in a reversed sense to the rest of the layout — the importance of this will become apparent later.

Thus far we have a layout that has no loops or paths of multiple connection, and whilst totally representable by a tree is not completely useful, especially if one wants to leave a train running around the layout indefinitely. Suppose we have a simple oval loop:

Figure 7. An oval becomes a loop



which starts at 2000,1000, and loops back through two straights and two curves to an end point co-incident in position *and orientation* with the start. To “close the loop”, we have to convert our tree to a graph, in this case with “self-pointers” by adding a specific link directive

Example 4. Describing a graph linkage

```
<layout name="oval" start="A">
  <start x="2000" y="1000" orient="20"/>
  <straight name="A" length="2000"/>
  <curve r="1000" angle="180"/>
  <straight length="2000"/>
  <curve r="1000" angle="180"/>
  <link>A.trail A.face</link>
</layout>
```

id	type	down	parts	length	face	trail	spur
A	section	false	4	10,283	A.trail	A.face	

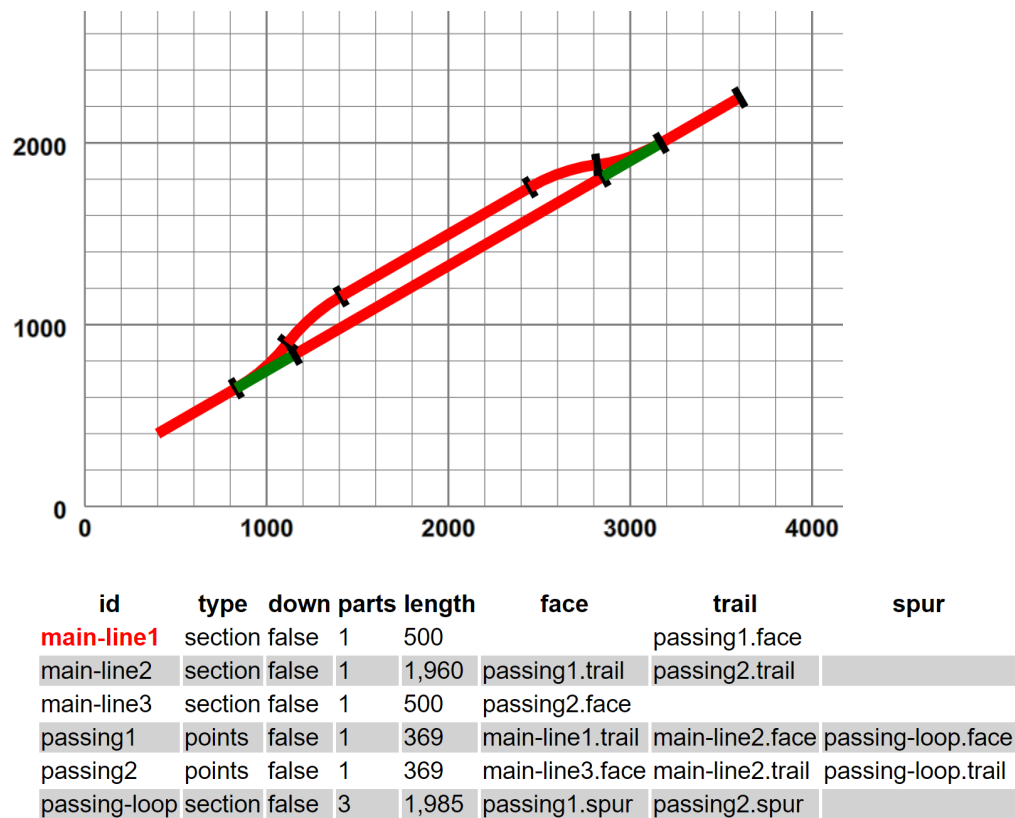
Now a vehicle finishing at *A.trail* can proceed happily into *A* again through *A.face* and similarly in a reverse direction. Of course in this case we could infer from the geometrical co-incidence that such a link may be required, but sometimes the geometry isn't quite accurate enough. Here is a passing loop:

Example 5. A passing loop

```
<layout name="passingLoop" start="main-line1">
  <start x="400" y="400" orient="30"/>
  <straight name="main-line1" length="500"/>
  <point id="passing1" radius="small" turn="left">
    <spur>
      <curve name="passing-loop" r="1000" angle="-22.5"/>
      <straight length="1200"/>
      <curve r="1000" angle="-22.5"/>
    </spur>
  </point>
  <straight name="main-line2" length="1960"/>
  <point id="passing2" radius="small" dir="trailing" turn="right"/>
  <straight name="main-line3" length="500"/>
  <link>passing-loop.trail passing2.spur</link>
</layout>
```

Where now we have specifically linked the passing loop component onto the trailing point spur:

Figure 8. Passing loop graphically and topologically



But linking isn't quite as straightforward. Suppose in our earlier example we consider the "small gap" between *section2* and *branch3* is joinable, and we specifically add a link declaration:

Example 6. Linking arbitrary branches

```
<layout name="twoPointsLinked" start="section1">
  <start x="400" y="400" orient="30"/>
  ...
```

```

<point id="P1" radius="small" turn="left">
  <spur>
    ...
    <point id="P2" radius="small" dir="trailing" turn="left">
      <spur>
        <curve name="branch2" r="400" angle="155"/>
      </spur>
    </point>
    <straight name="branch3" length="500"/>
  </spur>
</point>
<curve name="section2" r="1500" angle="-45"/>
...
<curve r="500" length="1400"/>
<link>section2.trail branch3.trail</link>
</layout>

```

This link introduces a requirement for a “polarity shift” — a locomotive proceeding *forwards* from *section2* would find itself running in the *reverse* direction in *branch2*. To permit smooth continuous operations, our “cyber-locomotives” have a “running in the wrong-direction” property (which is *xored* with *reverse*), and when similar ports are connected with similar “down-line” properties, a dummy *swap* component is inserted in the link, which will invert this property as a vehicle transits⁷:

Figure 9. Swapping direction across links.

id	type	down	parts	length	face	trail	spur
branch1	section	false	2	1,976	P1.spur	P2.trail	
swap1	swap		0	0	section2.trail	branch3.trail	
branch2	section	true	1	1,082	P2.spur		
swap2	swap		0	0	branch3.trail	section2.trail	
branch3	section	false	1	500	P2.face	swap2.face	
section1	section	false	1	1,000		P1.face	
section2	section	false	4	4,256	P1.trail	swap1.face	
P1	points	false	1	369	section1.trail	section2.face	branch1.face
P2	points	false	1	369	branch3.face	branch1.trail	branch2.face

(*swap1* and *swap2* could in theory be the same, but the implementation is easier to use one for each direction, and the additional cost minimal.)

3.2. Computing the geometry

The original intention of the design tool was to automate the calculation of track geometry. This proved to be relatively easy, using a simple vector arithmetic package with a triple vector datatype of *x, y, orientation*⁸, and the `xsl:iterate` instruction processing the track component sequences through template application as the track is “constructed”. For example here is the code to process a **straight** element:

```

<xsl:template match="straight" as="map(*)" mode="makeTrack">
  <xsl:param name="start" as="map(*)"/>
  <xsl:param name="options" as="map(*)" select="map{}" tunnel="true"/>
  <xsl:variable name="length" select="@length" as="xs:double"/>
  <xsl:variable name="straight"
    select="v:new($length, 0) => v:rotateDeg($start?orient)"/>
  <xsl:variable name="end" select="v:add($start, $straight)"/>
  <xsl:variable name="path" select="p:line($start, $end)"/>

```

⁷Such an issue is faced by two-rail electric power systems on railways with such “re-entrancy”

⁸Adding a *z* (height) component would be simple, being altered by `length * gradient`. It is safe to assume that gradients will never be steep enough to make significant effects on planar (*x, y*) positions.

```

<xsl:variable name="pieces" as="element(*)">
  <g class="straight">
    <g class="schematic">
      <path d="{ $path }"/>
      <xsl:sequence select="r:join($end)"/>
    </g>
    <g class="way"
      transform="translate({ $start?x }, { $start?y })
        rotate({ $start?orient })">
      <xsl:if test="$options?layTrack">
        <xsl:sequence select="r:straight($length)"/>
      </xsl:if>
    </g>
  </g>
</xsl:variable>
<xsl:sequence select="map{
  'type':string(name()),
  'orient.start' : $start?orient,
  'orient.end' : $start?orient,
  'pieces': $pieces,
  'length': $length,
  'path': $path,
  'start' : $start,
  'end': $end,
  'name': string((@name,
    'S-'||string(accumulator-before('trackNo')))[1])
}"
/>
</xsl:template>

```

`$start` is an input parameter which is a map whose principal members are `x`, `y` and `orient`⁹. The new end point, including its orientation, is calculated effectively by

```
v:add($start, v:new($length,0) => v:rotateDeg($start?orient))
```

where `v:rotateDeg($in,$rot)` rotates a vector (and its end orientation) by `$rot` degrees. During this operation the (SVG) graphic pieces for the schematic and the track pictures are constructed (see below) and added to the resulting map as well as other needed information, such as track section length. Each piece is named, using an `xsl:accumulator` to generate something suitable in the absence of a specific `@name` value.

This template is executed from an `xsl:iterate` instruction processing the children of a `layout` or a `spur`:

```

<xsl:template match="rail|spur|layout" as="map(*)" mode="makeTrack">
  <xsl:param name="start" as="map(*)">
    <xsl:apply-templates select="start" mode="#current"/>
  </xsl:param>
  <xsl:iterate select="* except (start | link)">
    <xsl:param name="start" select="$start" as="map(*)"/>
    <xsl:choose>
      <xsl:when test="not(self::break)">
        <xsl:variable name="part" as="map(*)">
          <xsl:apply-templates select="." mode="#current">
            <xsl:with-param name="start" select="$start"/>
          </xsl:apply-templates>
        </xsl:variable>
        <xsl:sequence select="$part"/>
        <xsl:next-iteration>
          <xsl:with-param name="start" select="$part?end"/>
        </xsl:next-iteration>
      </xsl:when>
    </xsl:choose>
  </xsl:iterate>

```

⁹Orientation is held in degrees and converted to radians as required. SVG describes its rotations in degrees and I know fairly closely what 30°, 45° and 225° look like, but not 1.5 radians.

```

        <xsl:otherwise>
            <xsl:break/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:iterate>
</xsl:template>

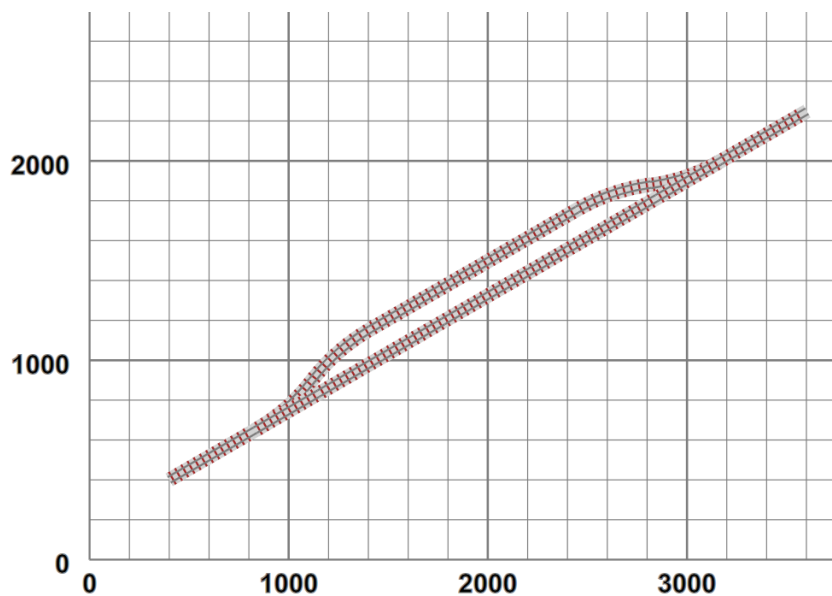
```

For each subsequent iteration the `$start` parameter becomes the `end` property of the `$part` just generated. Needless to say processing a `curve` is similar to that for `straight`, though the calculation of the chord, end point and the appropriate SVG elliptical arc are more complex. For the `point` we need to construct *two* sections: the *not set* (straight on) track section and its end point, and the *set* section with its attached branch line, which is constructed by a recursive call on the iteration above, with the branch `spur` element as context and the spur position and orientation as the `$start` parameter.

4. Drawing pictures

Thus far we have drawn schematic representations of the track as SVG line-based components. With a little work SVG is entirely capable of generating much more detailed views, with a lot of possibility of caching intermediate and reused sections. For example:

Figure 10. More detailed track



In this case the track is generated from a sequence of “rail-and-sleeper” subsection definitions, displayed via SVG’s `use` directive:

```

<g xmlns="http://www.w3.org/2000/svg" class="way"
  transform="translate(3769.549241302635,2599) rotate(30)">
  <use href="#track10" x="0" y="0"/>
  <rect class="ballast" x="360" y="-36" width="140" height="72"/>
  <use href="#sleeper" x="378" y="0"/>
  <use href="#sleeper" x="414" y="0"/>
  <use href="#sleeper" x="450" y="0"/>
  <line class="rail SM32" x1="360" y1="-16" x2="500" y2="-16"/>
  <line class="rail SM32" x1="360" y1="16" x2="500" y2="16"/>
</g>

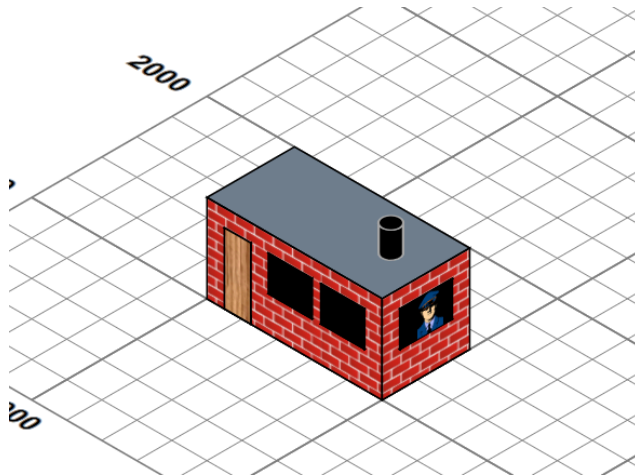
```

In this case we have a “pre-built” 10-sleeper section of straight track (`#track10`), followed by “ballast”, three sleepers and two rails to display the remainder of the required length. All these are sized to the actual dimensions of the track being used. This is translated and rotated into the required start position.

4.1. Isometric Views

Planar views are useful, but they don't give a picture of what one might see, where the third dimension has some importance. Luckily an *isometric* transformation can give a view "from above and aside". This involves applying a transform of `translate(3000,0) rotate(30) skewX(-30) scale(1,0.8660254037844387)` to the graphics and altering some pieces to support a pseudo-3D view. For example, let us add a simple building:

Figure 11. A simple building

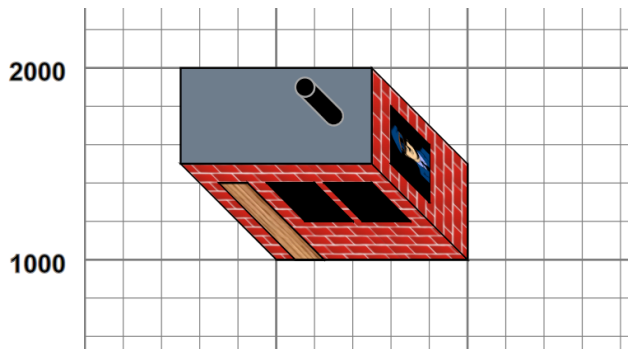


I could perhaps have looked at using a full 3D modelling package which was capable of generating SVG outputs, but my needs were modest and could perhaps be handled by a simple declarative model, processed completely with XSLT to generate suitable SVG. The building is defined by a simple XML structure of boxes and a cylinder:

```
<buildings>
  <resources> ... </resources>
  <group x="1000" y="1000"
    fill="url(#brickWall)" stroke-width="10" stroke="black">
    <box width="1000" height="500" rotateZ="0" depth="500" z="0">
      <top fill="slategrey"/>
    </box>
    <box height="1" width="150" depth="400"
      z="0" x="100" y="0" fill="url(#wood)"/>
    <box height="1" width="250" depth="200" fill="black"
      z="200" x="350" y="0" />
    <box height="1" width="250" depth="200" fill="black"
      z="200" x="650" y="0"/>
    <box width="1" height="300" depth="200" fill="black"
      z="200" x="1000" y="100">
      <east >
        <svg:image xlink:href="images/officer-in-uniform.png"
          x="100" y="0" height="200"/>
      </east>
    </box>
    <cylinder radius="50" length="150" axis="z" fill="black"
      z="500" x="800" y="250" stroke="darkgrey"/>
  </group>
</buildings>
```

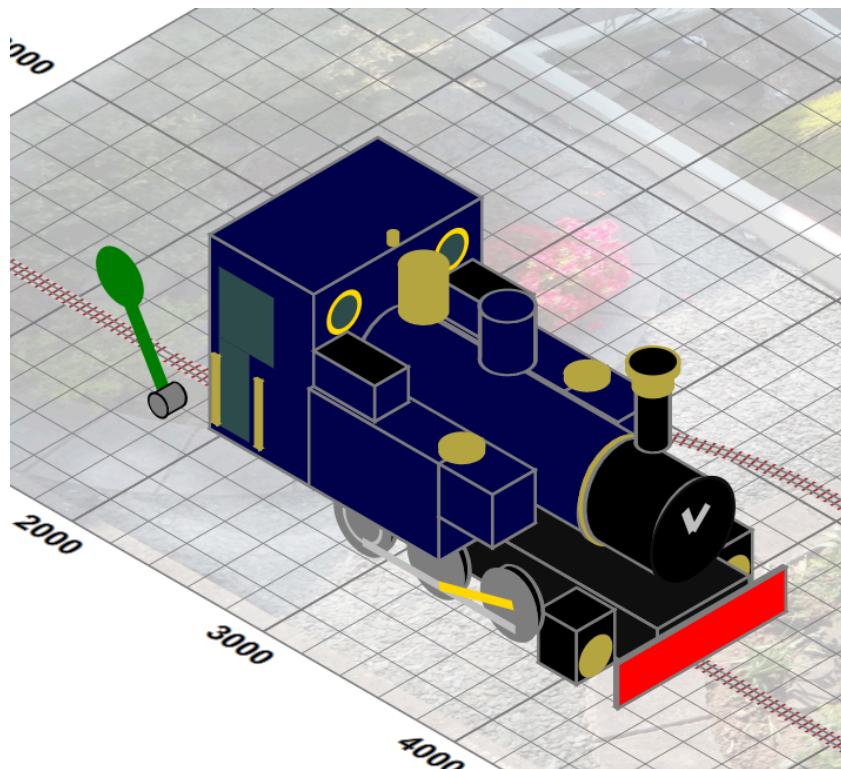
which is then used to generate an SVG group that look like:

Figure 12. An iso-orthogonal building



such that when the entire SVG group, within which lie all picture pieces (grid, plan, track etc..), is subject to isometric projection, the building appears to have depth and height. (We also produce a true orthogonal view, so we can look at the scene from “directly above”.) Currently the repertoire is orthogonally-oriented rectangular blocks and cylinders, with named “faces” to which styling and content can be attached (*top*, *south* and *east* for blocks, with *bottom*, *north* and *west* normally hidden, and *surface*, *top* and *bottom* for cylinders.). Components are currently positioned absolutely and can be grouped. Using this we can build models of the complexity of:

Figure 13. The Lady Anne



which is defined by some 50 components, some of which are repeats of common substructures, implemented by bindings and interpolations of XSLT variables. This ability to style and add content to the named *faces* of the component parts is important. For example, adding the “smokebox handle” to the boiler front of *Lady Anne* merely requires:

```
<cylinder class="boilerFront" x="151" z="80" axis="x"
  radius="27" length="45">
  <end class="boilerEnd">
    <svg:g class="silver" stroke="silver" stroke-width="5">
      <line x1="0" y1="0" x2="10" y2="-10"/>
      <line x1="0" y1="0" x2="-5" y2="-14"/>
    </svg:g>
  </end>
</cylinder>
```

```

    </svg:g>
  </end>
</cylinder>

```

and the graphic components will be placed and transformed correctly to sit *in the boiler front*. As we will see later, it is critical that the SVG views of these model engines must be such that they produce the expected picture when subjected to an isometric transformation, as shown for the building, as the trajectory paths trains must follow (which are effectively *on the flat*) are themselves subjected to the same projection.

5. Interaction

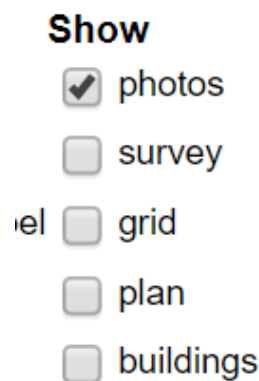
The tool has two main types of interaction: animations, discussed in the next section, and view selection. Most of the view selection is based on switching the **display** style of graphical or user interface element on and off, through controls that are generated from declarative descriptions. For example:

Figure 14. Controls for display options

```

<div name="show">
  <title>Show</title>
  <option default="">photos</option>
  <option>survey</option>
  <option>grid</option>
  <option>plan</option>
  <option>buildings</option>
  ...
</div>

```



declares a group of controls, from which a group of labels and checkboxes are generated, some of which are preset and whose rendering is shown above. Control of display is performed by a generic XSLT template, which fields change events on the generated **input** checkboxes, all of which are class-labelled as *show*:

```

<xsl:template match="input[@class eq 'show']" mode="ixsl:onChange">
  <ixsl:set-style name="display" object="id(@value)"
    select="if(ixsl:get(.,'checked')) then 'inline' else 'none'"/>
</xsl:template>

```

The **@value** of the **input** is taken to be the *id* of an element (either XHTML or SVG) that contains all items of the given type and *display* style modified accordingly. Generic hide/reveal controls for object with a given *class* token are supported by a similar template.

Switching between orthogonal and isometric views of the garden/plan/layout involves modifying a top-level transform attribute on the SVG and setting a class token to indicate the given view. As all (3D) components have both orthogonal and isometric views, each class-labelled, simple CSS compound rules such as `.viewISO .partORTHO, .viewORTHO .partISO {display: none;}` and `.viewISO .partISO, .viewORTHO .partORTHO {display: inline;}` ensure that only the correct class components are displayed for the current view.

Points obviously have state and this needs to be changed to direct trains to suitable parts of the layout. We construct an XHTML “signal box” where all the point controls are checkboxes and through which specific points can be set into *switched* or *unSwitched* classes. CSS styling ensures that the appropriate components for the given state are displayed. Sometimes determining which control effects which point can be problematic. A solution to this is to support clicking on the (SVG) points themselves, or an adjacent lever. This is achieved by the templates:

```

<xsl:template match="*[contains-token(@class, 'pointLever')]"

```



```

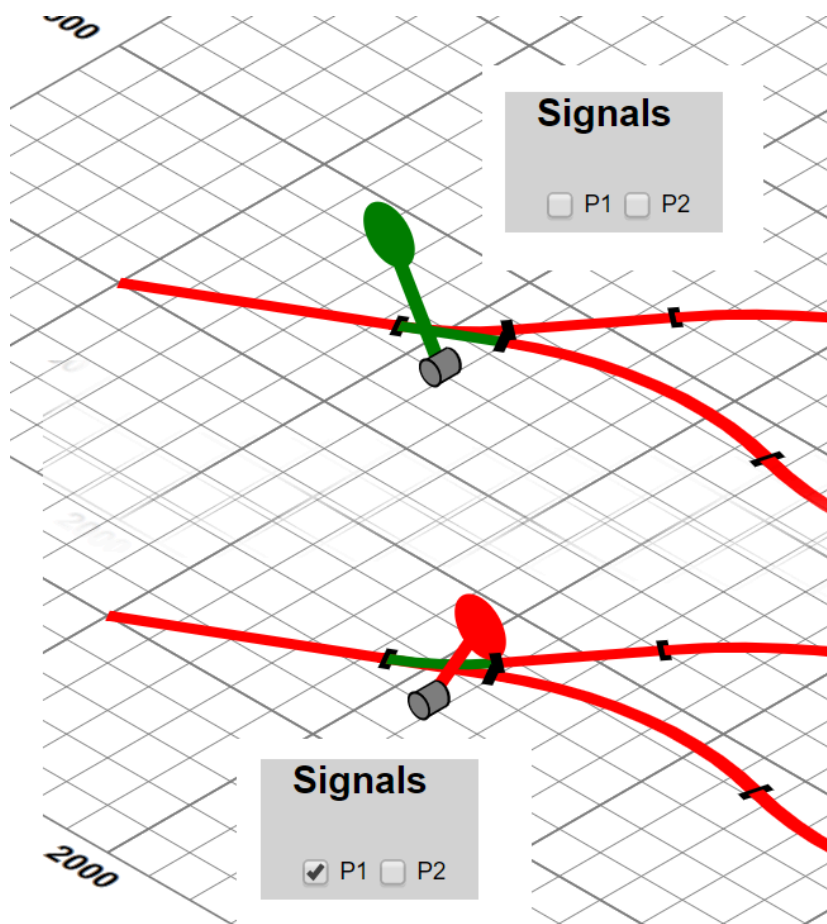
    mode="ixsl:onclick">
    <xsl:variable name="point"
      select="ancestor::*:g[contains-token(@class, 'point')][1]"/>
    <xsl:variable name="point.state"
      select="id($point/@id||'-state')"/>
    <xsl:sequence select="ixsl:call($point.state,'click',[])"/>
  </xsl:template>

  <xsl:template match="input[contains-token(@class, 'pointState')]"
    mode="ixsl:onchange">
    <xsl:variable name="checked" select="ixsl:get(., 'checked')"/>
    <xsl:sequence select="js:playAudio(id('pointChange'))"/>
    <xsl:for-each select="id('point-' || @value, .)/*:g[1]">
      <ixsl:set-attribute name="class"
        select="if($checked) then 'switched' else 'unSwitched'"/>
    </xsl:for-each>
  </xsl:template>

```

where clicking on the (SVG) point lever dispatches another *click* event to the appropriate state control in the signal box. Controls in the signal box respond to changes by playing the *pointChange* sound effect and changing the (un)switched class of the actual signal, which changes which of the graphic groups is displayed:

Figure 15. Changing points with a signal box



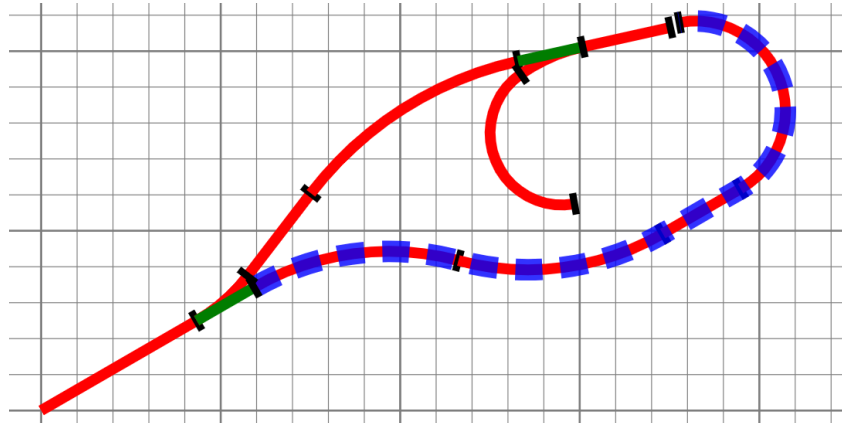
6. Animations

SVG supports animations based on SMIL event-driven models. Of particular interest in this case is the use of *path-based* animation where a given SVG group can be successively translated along a given path. As trains move along tracks, and

in our design tracks are defined by sections from which SVG **path** definitions can be constructed easily, we should be able to simulate the movement of trains around our tracks. And so it proved.

The basic animation we used is effectively “move this group *g* along this path *p* in a duration of *dur* seconds.” For each section of the layout (i.e. a contiguous run of straight and curves, or the *set* and *not set* short sections of points), we calculate both a path description (the *d* property of **svg:path**) and the total length. For example the dashed blue line is the defined (single) path for the *section2* track section, for which a total length of 4,256 has been calculated :

Figure 16. A track section path



Assuming we wish our “train” to run at 100mm/s (a scale speed of ~7km/hr, i.e. a brisk walking pace), then the animation should take 42.5 seconds. This is achieved by forming up an **svg:animateMotion** definition element:

```
<animateMotion xmlns="http://www.w3.org/2000/svg"
  id="train.animation" xlink:href="#train"
  begin="indefinite" fill="freeze" repeatCount="1"
  calcMode="linear" keyTimes="0;1" keyPoints="0;1"
  rotate="auto"
  dur="42.5" onend="eventEnded('train;section2.trail') >
  <mpath xlink:href="#section2.path" />
</animateMotion>
```

The graphics group that will be subject to the animation

Conditions for the start of the animation — in this case the animation waits until it is triggered explicitly. When the animation has finished freeze the graphics state, i.e leave the graphics translated to the end of the path and do not repeat.

keyTimes and **keyPoints** define a piecewise-linear mapping between proportions of the duration and proportions of the total length — this is used to support moving in reverse and altering “speed”.

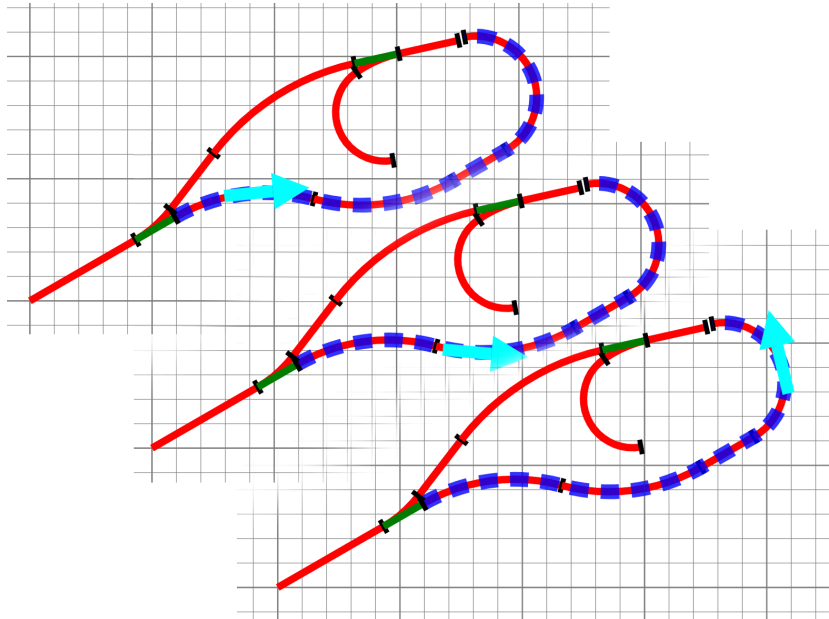
auto adds a rotation transform to the animated graphics corresponding to the current path tangent direction, so the graphics object “turns” along the path.

When the animation completes the global function **eventEnded()** will be executed with an argument containing information about which train has completed a move and where — in this case arriving at the *trail* port of *section2*.

A reference to the path to be followed.

The animation is started by invoking the **beginElement()** function method of the animation element through a minimal global JavaScript function. Thus our “train”(in this case a cyan arrow) progresses along *section2* as below:

Figure 17. Movement along a track section.



When the animation finishes, the `onend` statement is invoked, which is fielded by the global JavaScript function `eventEnded()`.

```
var ignoreEvent = false;
function eventEnded(e) {
    if(!ignoreEvent) {
        var event = new Event("change",{"bubbles":true});
        var store = this.document.getElementById("event");
        store.value = e;
        store.dispatchEvent(event);
    }
    ignoreEvent = false;
}
```

There are cases (described below) when we need to ignore an end event temporarily.

A (hidden) checkbox element in the DOM tree that is used to hold the event information as its `value` property. Propagating an event that the value of the event information store has changed.

After this function has executed, the checkbox `id('event')` receives a *change* event which is caught by an XSLT template:

```
<xsl:template match="*:input[@id eq 'event']" mode="ixsl:onChange">
    <xsl:variable name="layout" as="map(*)"
        select="$layouts(f:radioValue('layouts', .))"/>
    <xsl:variable name="parts" select="tokenize(@value, ';')"/>
    <xsl:choose>
        <xsl:when test="exists($parts[3])">
            <!-- There is a new section to enter -->
            <xsl:call-template name="runTrain">
                <xsl:with-param name="engine" select="$parts[1]"/>
                <xsl:with-param name="trackComponentID" select="$parts[3]"/>
                <xsl:with-param name="tracks" select="$layout?tracks"/>
            </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
            <!-- There is a no new section to enter - end of the line -->
            <xsl:for-each select="id($parts[1])">
                <ixsl:set-attribute name="position" select="$parts[2]"/>
            </xsl:for-each>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

```

</xsl:for-each>
<xsl:variable name="engine" select="$parts[1]"/>
<xsl:call-template name="stopEngine">
  <xsl:with-param name="engine" select="$engine"/>
</xsl:call-template>
<xsl:call-template name="reverseEngine">
  <xsl:with-param name="engine" select="$engine"/>
</xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

```

There are a number of possible layouts, held as a named map global variable. Which is the active one is determined by the value of the *layouts* radio button set.

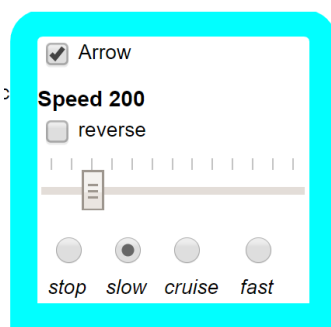
This template expects the value of the *event* checkbox to be a string of the form *train;current port[next port]*.

If there is a next port, then the train is run on that new section from that port, on the current layout.

If not then the train is assumed to have reached the end of the line. It is stopped and the direction reversed, so that, as a convenience to the driver, “opening the throttle again” again will cause the train to move back along the section.

The trains are controlled by a simple interactive XHTML control group (obviously of class *cab*):

Figure 18. The Engine Cab



```

<div id="Arrow.cab" class="cab arrow">
  <div class="toggler">
    <input class="run" type="checkbox"
      value="Arrow" />
    <label class="text">Arrow</label>
  </div>
  <label class="title">Speed
    <span class="value">0</span></label>
  <div name="direction" class="direction">
    <div class="toggler">
      <input class="direction" type="checkbox"
        value="reverse"/>
      <label class="text">reverse</label>
    </div>
  </div>
  <input type="range" min="0" max="1200"
    value="0" list="tickmarks" />
  <div class="radio speed">
    ...
    <div class="toggler">
      <input class="speed" type="radio"
        value="200" />
      <label class="text">slow</label>
    </div>
    ...
  </div>
</div>

```

Apart from selecting a locomotive to run, the only current action is to *change its speed or direction of travel*. A number of XSLT templates detect changes in the cab input controls such as:

```

<xsl:template match="input[contains-token(@class, 'speed')]"
  mode="ixsl:onChange">
  <xsl:variable name="cab"
    select="ancestor::div[contains-token(@class, 'cab')]" />
  <xsl:variable name="run" select="$cab//input[@class eq 'run']" />
  <xsl:variable name="value" select="@value" />
  <ixsl:set-property object="$cab//input[@type eq 'range']"
    name="value" select="number($value)" />
  <xsl:for-each select="$cab//span[contains-token(@class, 'value')]">

```

```

        <xsl:result-document href="?. " method="ixsl:replace-content">
            <xsl:sequence select="string($value)"/>
        </xsl:result-document>
    </xsl:for-each>
    <xsl:if test="ixsl:get($run, 'checked')">
        <xsl:variable name="engine" select="$run/@value"/>
        <xsl:for-each select="id($engine)">
            <ixsl:set-attribute name="speed" select="$value"/>
        </xsl:for-each>
        <xsl:call-template name="changeVelocity">
            <xsl:with-param name="engine" select="$engine"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>

```

which detects a change in the *stop*, *slow*, *cruise*, *fast* radio button set. The selected speed is the `@value` of the set, which is written into a `span` element within the cab `div` and used to set the slider to a suitable point. If the engine is running (the top left checkbox checked), then the demanded speed is written as an attribute onto the selected engine object and then the *changeVelocity* template is invoked.

The key idea here is to determine *how far the current animation has progressed*, from which the remaining distance to travel can be determined. This is computed by a global JavaScript function with the animation object *a* as argument:

```

function animProgress(a) {
    if(a.getAttribute("dur")==0 ||
        a.getAttribute("dur")=="indefinite") {
        return 0;
    }
    var startTime;
    try{
        startTime = a.getStartTime();
    } catch(e) {
        return 0;
    }
    var t_ratio=(a.getCurrentTime() - startTime)/a.getSimpleDuration();
    return t_ratio;
}

```

which calculates the ratio of elapsed to total animation duration. In cases where the animation is not active (for which I can't find a simple test), the exception on finding start time is caught. Given the remaining distance and desired speed, a new duration can be determined and the animation restarted using the `keyPoints` property to start somewhere down the animation path, e.g. `keyPoints="0.5;1"` would be used for a speed change halfway along the track section¹⁰.

The animation is restarted by invoking the `beginElement()` method — the `ignoreEvent` flag is used to prevent the implicit `endElement()` event, triggered before the restart, that would normally be used to signal completion of traversal of a section, propagating to the XSLT templates. In the case that the locomotive is running in reverse, the key points are reversed, e.g. `keyPoints="0.66;0"` would be used for a speed change one-third of the way backwards through a section.

In the absence of such speed changes a running locomotive involves animation movement along the current section until the end event is executed, fielded by the XSLT template shown earlier, which then starts animation along the next specified section. In the case of entering points, the state of the point is examined (from the status of the point control in the signal box!) and the correct path and next section determined for the animation¹¹. When a locomotive enters a *swap* section, described above, its internal *running in the wrong direction* flag is inverted and it passes on to the following section.

A small number of other animation effects have been added. Firstly locomotives have wheels, which can be animated to rotate at a rate and direction suitable for their diameter and the locomotive's speed, using the animation element:

```

<animateTransform type="rotate" begin="indefinite"
    attributeName="transform" from="0" to="360"
    dur="..." attributeType="XML" repeatCount="indefinite"/>

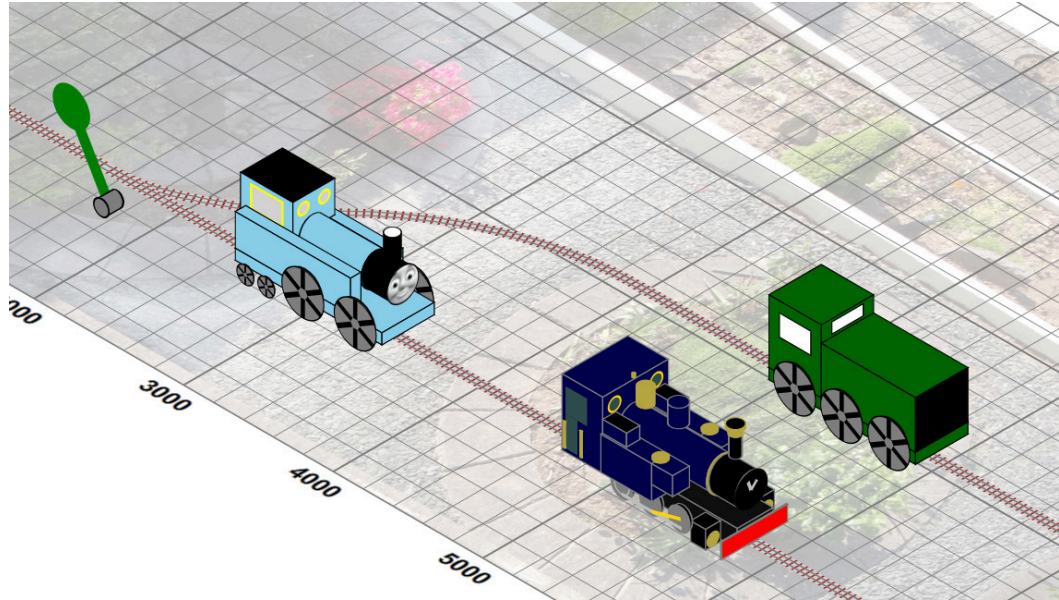
```

¹⁰The current animation may itself already involve a "partial" path, as a consequence of a previous change in speed — this is determined from the existing `@keyPoints` value on the `animateMotion` element to determine the "distance to go".

¹¹Changing a point while a locomotive is moving through it will not effect the locomotive's path.

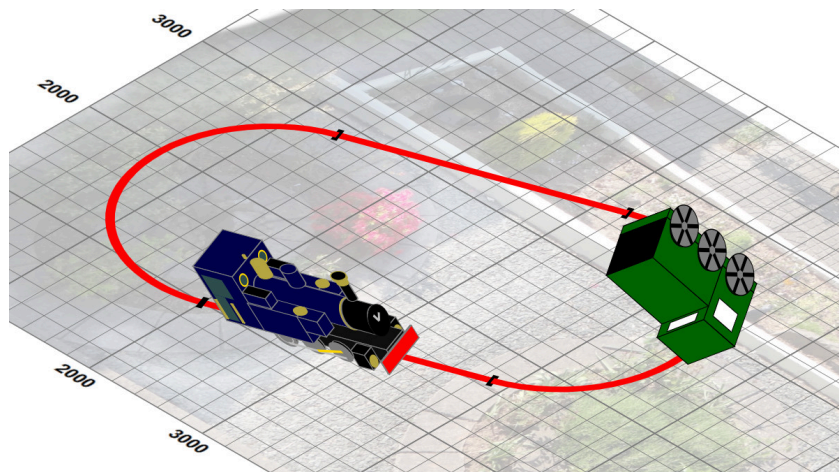
Secondly, locomotives can be given running sound effects by invoking `play()` method on an `audio` element when they start movement, and can “whistle” when they enter a (zero length) *whistle* pseudo-track section. The end point of this development was a case where multiple engines could be run on a layout, stopping, starting, reversing and changing their speed independently and altering points to move them to different sections of the layout:

Figure 19. Three engines running simultaneously



But there is a problem with the isometric view “trick” and automatic path tangent rotation:

Figure 20. On the ceiling



The animation rotation transformation is applied before the isometric projection and our 3D trick no longer works with significant rotations. How this may be overcome is discussed in the next section.

7. Developments

There are a small number of developments I have been working on, but at the time of writing they are incomplete. This section describes these ideas.

7.1. True 3D models and view rotation

The 3D model used so far is a collection of orthogonally arranged rectangular blocks and cylinders, declared in an order that reflects isometric view shadowing. For example an *engine frame* block is defined before the *boiler* cylinder, to appear

underneath it. From this model suitable SVG components can be generated to simulate a 3D view when subjected to a uniform isometric transformation. But to support a non-orthogonal rotation of such a model about the z-axis, to overcome the “on the ceiling” effect, the situation becomes somewhat more complex. There are three points to consider:

- What is a suitable graphic for a block or cylinder when rotated by θ degrees about the z-axis? A key requirement is that the “faces” model of additional styling and content must still be supported.
- As a group of 3D parts is rotated, their obscuration relationships alter and any views must accommodate this. How should a set of component parts be “depth-ordered” in the direction of the isometric view, when the ensemble is rotated significantly?
- How is the appropriate rotated view displayed as a locomotive turns?

Constructing the isometric-prepared components of a rotated block is a little tricky. The top surface is always visible and can just be rotated as required. Ignoring any visibility of the base, only two of the four vertical sides will be visible dependent upon rotation change ranges of 45° and 135° . Each visible face is subjected to additional scaling and skew dependent on the rotation angle, so that it is correctly sized, positioned and any additional content “stays in place”. The situation for horizontally aligned cylinders is very much more complex, and at the time of writing is work in progress.

To “view-order” an ensemble of rotated components it would be helpful if a (possibly multiple) value can be computed that can be used as sort keys to arrange the parts into appropriate order using `xsl:perform-sort/xsl:sort+`. This can be so for some very simple cases, but in general parts must be pairwise-compared, which requires some sorting function that uses a *compare* function, rather than a key-generator. Sadly, XPath sort functions all use a “key” model, so a generic XSLT higher-order pairwise sorting function may have to be constructed.

Calculating the rotation views *on the fly* would be catastrophically expensive, so the solution chosen is to generate a series of groups, each corresponding to a defined angle of rotation and labelled suitably (e.g. `class="rotate-45"` for a view rotated by -45°). It would also be possible to generate the set of views offline and include in the runtime. However they can be sizeable — an interval of 5° , which certainly doesn't appear “smooth” would require 72 separate versions.

Assuming there is such a series of views of an engine, we need to arrange for the display property of the (approximately) correct rotation view to be switched from *none* to *inline*. But we do know for a given locomotive which section it is in and can map from the proportion of the animation completed to the tangential orientation at that point. (As we use only straights and circular arcs, the tangent angle is a piecewise linear function of the “section proportion”, running from 0 to 1. This profile is added to the map entry for the section.) Given that the speed of the engine is known, we can thus predict how long it will be until the current rotation view should be superseded by the next one. This is enabled through a template `rotateTrain` which both makes visible the suitable view and schedules a further `rotateTrain` call after a suitable wait.

7.2. Collision detection, *a.k.a.* train crashes

As designed, my locomotives are ætheral beings, able to glide seamlessly and smoothly through each other. To prevent this, we need to detect collision or interference. SVG does have some primitive collision detection based on bounding box overlap, but given the isometric 3D nature of our engines, this is unlikely to be accurate, and certainly over-enthusiastic. Moreover, normal movement of our engines is both highly restrained, i.e. to track sections, and predictable, as they travel at known rates.

A simple approach, ignoring engine “size” and treating them as point entities, is to consider only cases where two (or more) engines are in the same section¹², travelling in either the same or opposite directions. Such a case can be checked when either a locomotive enters a new section, or the speed of an engine is changed. In such circumstances, we know both where, in distance, each locomotive is, and how fast they are approaching each other. Hence in the case of a predicted collision we can schedule an action (using `ixsl:scheduleAction`) to trigger a “crash notification” after the required interval. However there is also the difficulty of a subsequent speed change altering this — this requires the ability to delete some of the currently active scheduled actions, which has proved highly problematic.

7.3. Difficulties

Apart from the headache-inducing issue of calculating the geometry of the edge of the visible curved surface of a rotated cylinder, most of the difficulty has been managing the animations and events. In particular it appears that an active animation cannot be stopped and deleted or restarted without invoking any associated `onend` event. The temporary solution, of dubious robustness, uses a global flag to suppress subsequent event propagation.

¹²Much of nineteenth-century railway signalling development was of course to stop such a situation happening in the first place.

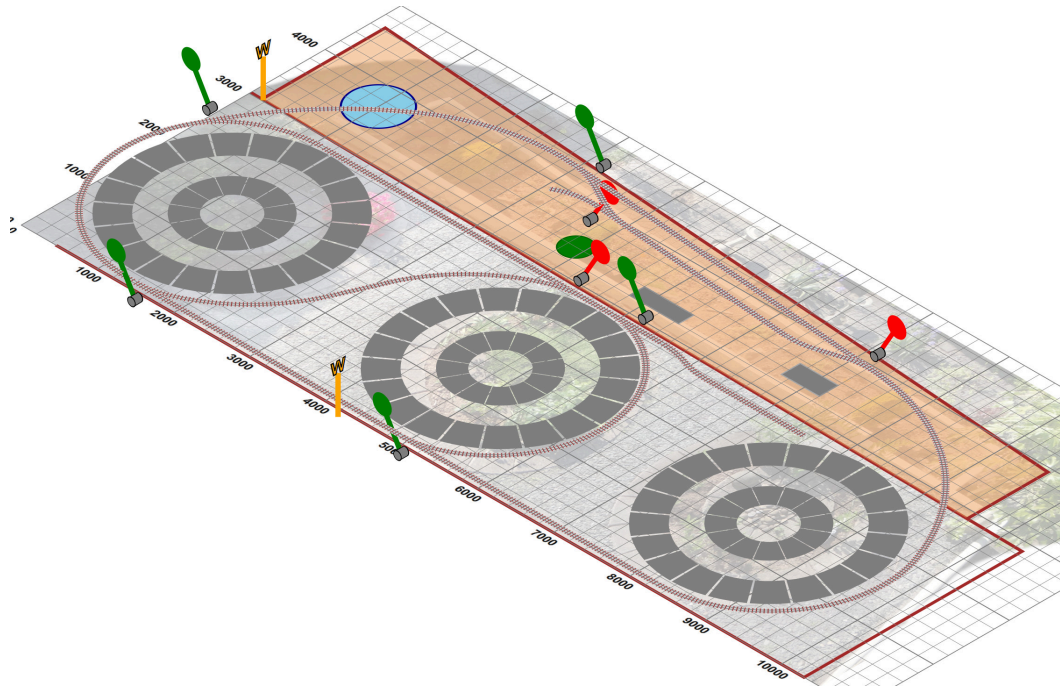
8. Conclusion

I originally built a small tool, using XSLT, that “did the geometry calculations” for a layout I was designing. A graphical view is always helpful, and generating SVG to do so was straightforward. Developing the isometric view led towards a more pictorial aspect to the output. Adding very simple animation opened the possibility of building something more akin to a “train set”, and showed some of the ways controls and active state could be mixed in an XSLT/Saxon-JS/SVG/browser environment. And this led to the idea of a demonstration at MarkupUK 2018...

The implementation needed a very small number of global JavaScript functions, that were invoked in XSLT/XPath expressions through the Saxon-JS function mapping namespace <http://saxonica.com/ns/globalJS>. All the rest of the code is XSLT3.0, with Saxon-JS extensions, generating all necessary XHTML and SVG structures, with templates fielding and processing events both from interaction and animation. Once up and running, the system is of course stateful — the speed, direction and current track section of engines, the switched *set* state of points etc. This state information is stored as attributes on the DOM tree.

Did it help with the original purpose — designing a garden railway? Well this was the layout design demonstrated at Markup2018:

Figure 21. The layout as proposed



and this is what currently exists:

Figure 22. Lady Anne on the Garden Line



Without Saxon-JS this project wouldn't have even been attempted and thanks are due to my colleagues Mike Kay and Debbie Lockett for the excellence of that product. The author is of course extremely grateful for the many votes cast in his direction at last year's MarkupUK DemoJam — without them he wouldn't have had to write this paper.

References

- [1] Debbie Lockett and Michael Kay. *Saxon-JS: XSLT 3.0 in the Browser..* Balisage: The Markup Conference . 2016. <https://doi.org/10.4242/BalisageVol17.Lockett01>.
- [2] *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 2011. World Wide Web Consortium (W3C). <https://www.w3.org/TR/SVG11/>.
- [3] *XSL Transformations (XSLT) Version 3.0*. 2017. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xslt-30/>.