

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



SVG-based Knowledge Visualization

DIPLOMA THESIS

Miloš Kaláb

Brno, spring 2012

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Tomáš Gregar Ph.D.

Acknowledgement

I would like to thank RNDr. Tomáš Gregar Ph.D. for supervising the thesis. His opinions, comments and advising helped me a lot with accomplishing this work. I would also like to thank to Dr. Daniel Sonntag from DFKI GmbH. Saarbrücken, Germany, for the opportunity to work for him on the Medico project and for his supervising of the thesis during my erasmus exchange in Germany. Big thanks also to Jochen Setz from Dr. Sonntag's team who worked on the server background used by my visualization. Last but not least, I would like to thank to my family and friends for being extraordinary supportive.

Abstract

The aim of this thesis is to analyze the visualization of semantic data and suggest an approach to general visualization into the SVG format. Afterwards, the approach is to be implemented in a visualizer allowing user to customize the visualization according to the nature of the data. The visualizer was integrated as an extension of Fresnel Editor.

Keywords

Semantic knowledge, SVG, Visualization, JavaScript, Java, XML, Fresnel, XSLT

Contents

Introduction	3
1 Brief Introduction to the Related Technologies	5
1.1 <i>XML – Extensible Markup Language</i>	5
1.1.1 XSLT – Extensible Stylesheet Lang. Transformations	6
1.2 <i>RDF – Resource Description Framework</i>	8
1.3 <i>Fresnel</i>	8
1.3.1 Fresnel Selector Language for RDF	9
1.3.2 Display Vocabulary for RDF	9
1.3.3 Fresnel Rendering Process	10
2 SVG – Scalable Vector Graphics	11
2.1 <i>Vector vs. raster</i>	12
2.2 <i>SVG Support among Web Browsers</i>	14
2.3 <i>Styling of SVG</i>	16
2.3.1 Presentation Attributes and Styling with XSL	16
2.3.2 Styling with CSS	18
2.3.3 Accessibility of Styles	19
2.4 <i>Representation of Text Content in SVG</i>	19
2.5 <i>Interactivity of SVG Content</i>	20
2.5.1 Events in SVG	21
2.5.2 Scripting in SVG	22
3 Fresnel Editor	25
3.1 <i>Architecture of Fresnel Editor</i>	25
3.2 <i>The Visualization Process</i>	27
3.3 <i>Issues and the Current State of Fresnel Editor</i>	29
4 Batik SVG Toolkit	30
4.1 <i>The Core Modules of Batik</i>	32
4.2 <i>Real World Projects Using Batik</i>	34
5 Data Visualization Approaches	36
5.1 <i>Generating of an SVG Document</i>	37
5.1.1 Transforming of an Intermediate XML Document	37
5.1.2 Building a Document via DOM API	42
6 SVG Visualization in Fresnel Editor	43

6.1	<i>XSL Transformation into the SVG Format</i>	43
6.2	<i>Preprocessor</i>	46
6.2.1	<i>Design</i>	46
6.2.1.1	<i>User Roles</i>	46
6.2.1.2	<i>Use Cases</i>	47
6.2.1.3	<i>State machine diagrams</i>	52
6.2.1.4	<i>Class diagram</i>	54
6.2.2	<i>Implementation</i>	54
6.3	<i>Displaying of the SVG Image</i>	56
7	SVG Tool for Image Annotation	58
7.1	<i>The RadSpeech Project</i>	58
7.1.1	<i>Exhibit-based Facetted Visualization</i>	60
7.1.2	<i>SVG-based Visualization for Image Annotation</i>	61
7.1.3	<i>Benefits And Constraints of the SVG Tool</i>	65
	<i>Conclusion</i>	67
A	Content of the Attached CD	76

Introduction

The major challenge of today's development is to provide information to users in an easy, understandable manner. There are many ways for data presentation; one of the most efficient is to give the data a visual representation.

In case of semantic data is the visualization rather difficult, because of relatively high cardinality of the semantic information set (knowledge base) as well as the number of interrelations per its elements. The problem, how to select and visualize the semantic data has been tackled by proprietary methodologies, which raise the cost for the data presentation – even though the visualization provides more or less similar output, the tools created by different organizations are rarely interoperable, thus (almost) no reusability is possible. The World Wide Web Consortium (W3C) recognized a need for an unified method for the semantic data selection and visualization and introduced the Fresnel language as a solution.

In 2009, Fresnel Editor was developed as a response to a need for an authoring tool for Fresnel Lenses and Formats and visualizing RDF data which was still missing. The tool presented the data in a simple (X)HTML format. In the pursuit for extending Fresnel Editor with additional visualizations, SVG was considered as an output format. SVG provides many useful features and its utilization may be valuable in situations, where (X)HTML format would have proven itself insufficient. Therefore, the main task for the thesis was to extend the Fresnel Editor with an SVG visualization module.

Firstly, an analysis of the domain was carried out and the technologies were explored. This revealed possible issues to deal with and helped to define the tasks to accomplish. The analysis was followed by a design of the module and later on, by its development.

While I was working on the thesis, an opportunity to develop a tool for annotating medical images arose. Also in this case seemed SVG as a reasonable format to use, thus it was possible to profit from the experience gained during the thesis related research and extend it even further.

On the following pages is firstly provided an overview of the technologies used within the projects, emphasizing SVG, Fresnel Editor and Batik SVG Toolkit. Later are discussed issues brought up by the visualization

into the *SVG* format, accompanied with suggestions of the possible solutions, which are implemented in the projects. The description of the projects development deliverables follows.

Chapter 1

Brief Introduction to the Related Technologies

In the very beginning is provided a short description of the fundamental languages – XML, XSLT, RDF and Fresnel – that are closely connected to the thesis. In this chapter, readers are briefly acquainted with these languages, thus the following chapters can build on the basic knowledge.

1.1 XML – Extensible Markup Language

Extensible Markup Language (XML) is a markup language for storing structured data in both human-readable and machine-readable way. XML was derived from Standard Generalized Markup Language (SGML, ISO 8879) which aimed on large-scale electronic publishing and was not well suitable for the need of the World Wide Web and other Internet services. Many features stayed unchanged – the separation of logical and physical structures, the Document Type Definition (DTD) allowing validation, the separation of data and metadata, the separation of processing instructions from data representation and syntax. Removed were the SGML Declaration. XML was influenced by Text Encoding Initiative (TEI) and the language HTML. [1] The final version of recommendation – XML 1.0 – was released on the 10th of February 1998.

Currently, there are two versions of XML. The version XML 1.0 is nowadays available as its fifth edition (released on the 26th of November 2008) and is recommended for general use. [2] The second version – XML 1.1 – was released on the 16th of August 2006. The main differences lies in requirements for characters used in element, attribute names and identifiers and restriction-permission policy for name definition. [3] Regarding future versions, only unofficial notions of XML 2.0 exist.

Generally, the XML specification defines an XML document as well-formed (i.e. the document satisfies syntax rules provided in the specification). Unlike other computer languages, XML does not offer set of fixed, predefined tags; the tags are defined according to individual needs. The

schema describing the structure of an XML file is stored in various formats – DTD, XML Schema and others. The XML schema is used by XML parsers to validate the structure of XML data.

The flexibility of XML led to creating of its extensions: XHTML, SVG, MathML, XSL, RDF, OWL, Atom etc. The primary aim of such formats is to represent text; however, the necessity for accommodation of other data types, such as video, music, vector graphics or web services is growing. XML also brings the opportunity to combine multiple markup languages into single profiles, for example XHTML + MathML + SVG. In such cases, vocabularies of each language are distinguished using the namespace mechanism [4] that provides uniquely named elements and attributes in an XML document.

Namespaces mentioned in the previous paragraph stand for one member of the very broad family of XML markup languages, which also includes XPath, XLink, XQuery, XSLT or XBase etc. Many of them are defined by W3C and belong to essential technology standards.

1.1.1 XSLT – Extensible Stylesheet Language Transformations

Previously various derivatives of XML were mentioned. One of them, XSLT, plays an important role in the SVG Visualizer that will be described later in the thesis. This language was created as one part of Extensible Stylesheet Language (XSL) which eventually split into XSLT, XSL Formatting Objects (XSL-FO) and XML Path Language (XPath). [5] The W3C recommendation XSLT 1.0 was released on 16th of November 1999 and is still used alongside the newer version XSLT 2.0, which reached the recommendation status on the 23th of January 2007. The changes between versions can be found in the XSL Transformations (XSLT) Version 2.0 (under section J – Changes from XSLT 1.0). [6]

We now turn to the purpose of XSLT. It is a declarative language for transforming XML documents. The principal idea of XSLT is to separate the content of information from its presentation; this means that a structured text document recorded in XML can be transformed into (X)HTML document/web page, output for printing (PDF or PostScript) or video display etc. XSLT can be used for transforming XML documents with different XML schemas as well as to make changes in a document itself; in fact, XSLT is able to perform computations together with manipulation or combination of data from different sources. [7]

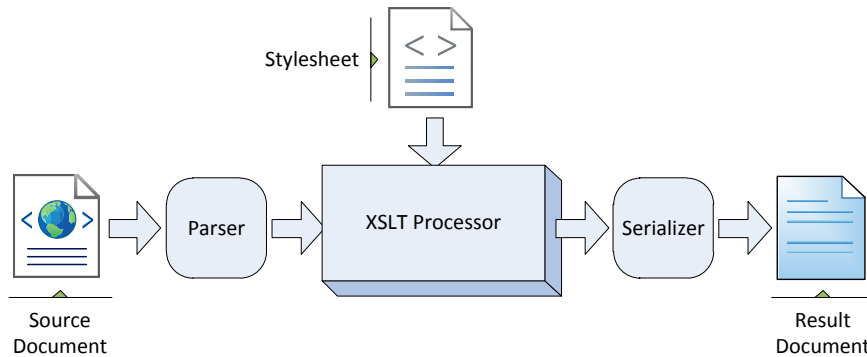


Figure 1.1: Schematic view on the transformation process [7]

There are several approaches to transform a document:

- the result document may be generated dynamically either on server or on client side (both the source XML document and the XSLT transformation are accessed when the result document is requested)
- it may be generated during the publishing process (the result document is published directly).

On the figure 1.1 is illustrated the transformation process:

1. The source XML document is parsed to produce an input tree structure and the XSLT processor reads the XSLT style sheet and prepares template rules.
2. The XSLT processor applies the template rules on nodes of the input tree. The root node of input tree is processed first and evaluates the content according to the best-matched template.
3. Instructions in each template either process other nodes from the input tree or generate nodes in an output tree.
4. The output tree structure is finally serialized into a result (XML) document. [8]

1.2 RDF – Resource Description Framework

RDF is a standard for representing semantic knowledge in an implementation independent manner. Although data can be represented in the XML format, its expressiveness is not strong enough to store all the data semantics.

The RDF format is based on several projects that aimed on the semantic representation: Meta Content Framework, Dublin Core and Platform for Internet Content Selection. The RDF specification became a W3C recommendation on the 22 of February 1999 but later was revised (the latest version of the specification was released on the 10 of February 2004. [9]

The basic concept of the RDF format are statements, so called triplets: *subject* → *predicate* → *object*. The subject represents a resource, the predicate its property or the relationship between the subject and an object. Object represents the value of the property or another resource. As an example might serve: “The sky is (=has color) blue”. In the environment of the Internet, the described entities are usually specified by online available unique Uniform Resource Identifiers (URI).[10]

RDF data models are commonly stored using the RDF/XML syntax or the N3 notation. For their (usual) vastness, the RDF data models required means to query a specific resource to eliminate unnecessary information. For this purpose were created many query languages, such as SPARQL, RDQL, Versa, RQL, or Fresnel.

1.3 Fresnel

When the semantic data stored in the RDF format are to be visualized, two issues have to be addressed to provide a human-readable output. Firstly, it is necessary to specify *what* information should be presented, and secondly, *how* to present the selected piece of information. The mechanisms and vocabularies for presenting RDF data differs between each tool facing this challenge. Consequently, the differences prevent sharing and reusing of the RDF knowledge presentation. This was the reason why W3C created Fresnel language to standardize the selecting mechanism and the vocabulary by putting forward concepts of lenses, formats and groups. [11]

The Fresnel specification consists of two parts: Fresnel Display Vocabulary for RDF and Fresnel Selector Language for RDF (FSL).

1.3.1 Fresnel Selector Language for RDF

FSL is a language inspired by XPath and its purpose is to model traversal paths in RDF graph. FSL does not depend on any RDF serialization; it takes into account RDF models as directed labeled graphs. FSL is compatible with Fresnel's Basic Selectors as they are contained in the display vocabulary part, however FSL is more expressive. [12] The authors of Fresnel summarize the principles of FSL as follows: "An FSL expression represents a path from a node or arc to another node or arc, passing by an arbitrary number of other nodes and arcs. FSL paths explicitly represent both nodes and arcs as steps on the path, as it is desirable to be able to constrain the type of arcs a path should traverse." [11]

1.3.2 Display Vocabulary for RDF

Let us turn to the Display Vocabulary for RDF. It consists of two modules, each of them consists of both lens and format part:

Fresnel Core Vocabulary

Lens-Core: the basic vocabulary for defining and relating lenses

Format-Core: the basic vocabulary for specifying formats

Fresnel Extended Vocabulary Lens-Extended: additional terms for relating lenses and for using lens inheritance

Format-Extended: advanced format selector, media type and format purpose vocabularies [13]

To achieve its goal – the interoperability – it is required from all browsers aiming at Fresnel support to implement the complete core vocabulary. They should also try to implement the extended vocabulary or at least parts of it for the additional functionality to be available.

Lenses define properties of an RDF resource to be displayed, and their order. The selection is carried out by selectors that represent queries in any of the supported query languages; basic selectors based on RDF names (simple test for a resource type), FSL selectors or SPARQL selectors.

Formats are used to assign visual information to RDF resources and properties selected by lenses. The format vocabulary defines how a property is labeled, how property values are displayed and references to CSS classes which define various styling attributes.

Groups are the last of Fresnel basic concepts. They organize lenses and formats so it is easy to determine which lenses and formats work together.

Groups also provide a way to define CSS classes that should apply on every lens and format belonging to the group. [14]

1.3.3 Fresnel Rendering Process

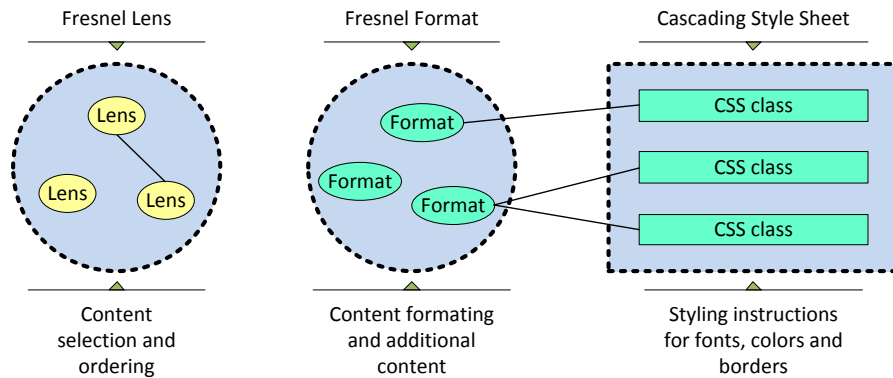


Figure 1.2: Content selection, formatting and styling with Fresnel [13]

Fresnel strictly separate selection and formatting (see figure 1.2). The rendering process can be decomposed into these steps:

1. The Fresnel lenses are applied on a source RDF graph. The result is an ordered tree of RDF nodes without any formatting information
2. The formatting information and hooks to CSS classes are added to nodes of the result tree by applying Fresnel formats
3. The result created in the second step is rendered into an output format (for example HTML, SVG, XML, PDF, plain text documents etc.) [13]

Chapter 2

SVG – Scalable Vector Graphics

SVG is XML-based language developed by W3C for describing 2D graphics in the vector format. It is not only the difference between vector and raster graphics that distinguish SVG from other graphical formats. SVG capabilities are still slightly underrated, therefore the usability of it has not been fully explored. Later in the text are discussed possible use cases, where SVG features might be very useful.

Initially, SVG was influenced by PGML¹ and VML² as well as by previous experience with CSS, HTML or XLink. [15] SVG is being developed since 1999 and the SVG 1.0 recommendation was released on the 4th of September 2001. The most recent version – SVG 1.1 – was released on the 16th August 2011 [16] and was developed in parallel with SVG 1.2, which has still status of working draft. [17] On the 14th of January 2003 was released SVG Mobile recommendation which incorporate two simplified profiles: SVG Tiny and SVG Basic. These lightweighted profiles are aimed for use on devices with reduced computational and display capabilities; the first is defined to be suitable for cellphones, the second for PDAs. [18]

Three types of objects are allowed in SVG: vector graphic, images (in other words raster graphic) and text. Objects included in SVG documents can be grouped, styled, transformed and composited into previously rendered objects; this provides an ability to create not only static drawings but also animated, interactive or both. SVG feature set also includes alpha masks, filter effects, template objects, nested transformations and clipping paths. [16] Nevertheless, one feature of formats offering visual presentation is missing – *z-index*. The drawing order is not separated from the document order. It means that an element overlaps all the elements written in the source code before. However, it is considered to enable this feature in SVG 1.2 because of many requests made in that matter. [17] SVG has a very broad range of

-
1. Precision Graphics Markup Language – a proposal for a vector graphic format inspired by Adobe's PostScript and Portable Document Format
 2. Vector Markup Language – developed from Rich Text Format (RTF) and implemented by a group around Microsoft

features many of which are beyond the scope of this thesis; only the relevant features are described later in the sections 2.3, 2.4 and 2.5.

2.1 Vector vs. raster

At this point it would be appropriate to offer a short explanation about graphic formats. Is vector format better than bitmaps? Are they universal or is each rather suitable for specific purposes?

In vector graphics, the image is described by geometrical primitives such as points, lines, curves and polygons which are based on mathematical expressions. Each of these primitives has so called control point(s) with a definite coordinate in the image; the x - and y -axes position is relative. If a vector is resized by scaling, the primitives are recalculated and displayed with no loss of data or detail. The information about visual style is assigned to every element. The size of a vector image depends on complexity of the drawing, not on its resolution.

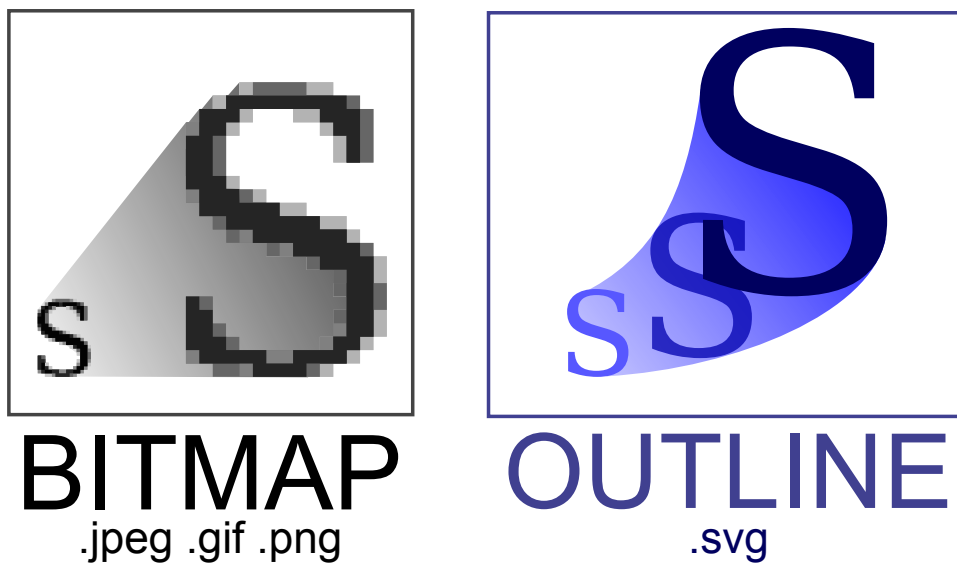


Figure 2.1: Demonstration of the difference between raster image and SVG [19]

Raster graphic image (also known as bitmap) is a two-dimensional grid of dots – pixels. A pixel is a single point bearing information about its color addressable by specific coordinates. The number of pixels in the meaning of

2. SVG – SCALABLE VECTOR GRAPHICS

width and height represents the image’s resolution. The quality and size of the picture are proportional; the higher resolution and the color depth³, the finer details will be distinguishable and the larger the size of the image will be. When a raster image is resized, every pixel is spread over a larger area on the display device which results in the loss of detail and clarity.

HTML5 Canvas	SVG
+ High performance for drawing in 2D	+ Resolution independence – good for scaling for any screen resolution
+ Performance independent on what is drawn; however, resolution increase can cause performance degradation	+ Animation support – directly via SVG syntax or via scripting
+ Result can be save as a raster image (.png or .jpg)	+ Control over each element via SVG DOM API in scripting language
+ Suits well for creating raster graphics, editing images and for pixel-level operations	+ XML origin of SVG makes the accessibility of SVG documents better comparing to canvas
– Absence of DOM nodes for the drawing, only pixels	– The rendering slows proportionally with the increased complexity.
– Absence of API for animation – it is necessary to use timers and other events to update the canvas	– SVG does not suit certain applications, for example games
– Bad capability to render texts	
– Canvas turns everything drawn into a matrix of pixels – accessibility problem	
– Canvas does not offer dynamic behavior – it shows statically the result of user’s interaction. That is the reason why it is not suitable for user interfaces	

Table 2.1: Advantages and disadvantages of HTML5 Canvas and SVG [20]

Which format is better depends on a particular use. The vector format is suitable for drawings, where is vital to have a lossless quality (such as technical drawings and sketches or logos) or for resolution-independent interfaces. The raster format is used for photographs and scanned pictures, image analysis, rendering game graphics etc.

3. The number of bits used to represent the color of a single pixel

With HTML5⁴ establishing itself in the field of web development, the raster format has gotten a new use. The HTML5 brings a concept of a drawing canvas, which allows creating a 2D or a 3D graphics⁵ inside the web browser. The output format for its 2D drawings is a raster image. The same functionality can be provided by SVG. Even though they both do the same, they do not do it equally. [20]

2.2 SVG Support among Web Browsers

The major problem of SVG nowadays is the legacy support of Internet Explorer, which only in the version 9.0 and upcoming 10.0 implemented partial SVG support. Despite the fact that the popularity of Internet Explorer is rapidly decreasing, it is still used by approximately one third of users⁶. However, many web sites provide both raster format and SVG and let the user (or more precisely the browser) to choose the image file.

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
		3.6				3.2		10.0	2.1
	7.0	9.0				4.0–4.1		11.0	2.2
	8.0	10.0	16.0			4.2–4.3		11.5	3.0
Current	9.0	11.0	17.0	5.1	11.6	5.0	5.0–6.0	12.0	4.0
Near future	10.0	12.0	18.0	6.0	12.0				

Supported
 Not supported

Table 2.2: SVG support in browsers (relevant are only features from SVG 1.1 recommendation) [23]

On the basis of the table 2.2 it might seem that SVG is now supported⁷ among the major browsers and as soon as the older version of Internet Explorer will be abandoned, the SVG will be used widely. However, the situation is not so simple as the support is often incomplete or differs in nuances.

4. HTML5 is still under development, the latest working draft was released on the 25th of May 2011. [21] Nevertheless, all major desktop and mobile browsers support HTML5 to some extent. [22]

5. 2D drawing context is relatively well established in browsers whereas 3D drawing context is still in the early stage of definition and development

6. According to statistics published to February 2012 at <http://gs.statcounter.com/>, versions of IE held 35,75% users

7. Green color marks (at least partial) support of SVG

2. SVG – SCALABLE VECTOR GRAPHICS

Internet Explorer: partial native support available since Internet Explorer 9 which was released in March 2011. Before was possible to use Adobe SVG Viewer plugin.

Gecko based browsers (Firefox): incomplete support for SVG 1.1 Full specification since 2005. The support is being improved ever since.

WebKit based browsers (Chrome, Safari): complete support for SVG 1.1 Full specification since 2006.

Opera: since Opera 8.0 was supporting SVG 1.1 Tiny, since Opera 9 SVG 1.1 Basic and partial SVG 1.1 Full. Since Opera 9.5 is partially supported even SVG 1.2 Tiny.⁸

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
		3.6: 50%				3.2: 69%		10.0: 100%	2.1: 0%
	7.0: 0%	9.0: 75%				4.0–4.1: 69%		11.0: 100%	2.2: 0%
	8.0: 0%	10.0: 75%	16.0: 100%			4.2–4.3: 69%		11.1: 100%	2.3: 0%
Current	9.0: 25%	11.0: 75%	17.0: 100%	5.1: 75%	11.6: 100%	5.0: 75%	5.0–6.0: 50%	11.5: 100%	3.0: 75%
Near future	10.0: 50%	12.0: 75%	18.0: 100%	6.0: 100%	12.0: 100%			12.0: 100%	4.0: 75%

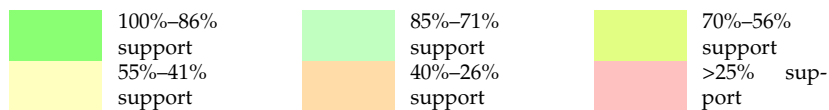


Table 2.3: SVG support of following features: SMIL animation, filters, fonts (all belong to SVG recommendation) [23]

Data presented in table 2.2 and table 2.3 are based on statistics available on <http://caniuse.com>. The criterion for including the browser in the overviews was browser usage higher than 3%⁹. [23]

8. Precise information about support in each browsers can be found on following websites: <http://blogs.msdn.com/b/ie/archive/2010/03/18/svg-in-ie9-roadmap.aspx>, https://developer.mozilla.org/En/SVG_in_Firefox, <http://www.webkit.org/projects/svg/status.xml>, <http://www.opera.com/docs/specs/>

9. Statistics are based on data from <http://gs.statcounter.com/> for February 2012

2.3 Styling of SVG

Let us return to the features of SVG; there are three of them to be introduced more thoroughly. The first one is styling of SVG.

For defining the visual style, according to which is the SVG document rendered, are used styling properties. The styling incorporates parameters affecting visual appearance (*fill*, *color*, *stroke-width* etc.), text styling (*font-family*, *font-size* etc.) and parameters influencing the rendering of graphical elements (*clip-path*, *marker*, *filter* etc.). Many of SVG style attributes are shared with CSS and XSL and are defined in the CSS2 specification¹⁰. The full list of styling properties can be found in chapter *Styling* of the SVG specification. The common scenarios for using SVG styling are following:

SVG content as an exchange format: to ensure interoperability across software tools where the support for a particular style sheet language is not guaranteed, the SVG content has to be fully specifiable without using a style sheet language

SVG content as the output from XSLT: XML data can be transformed with XSLT transformation into an SVG document to provide a graphical representation of the data. The generated output of the transformation has to be a fully specified SVG content.

CSS styled SVG content: CSS is widely used styling language and is for its features very suitable for SVG, thus CSS styling has to be possible to apply on an SVG content [16]

There are two ways how to assign styling properties. Either using presentation attributes or CSS style sheets.

2.3.1 Presentation Attributes and Styling with XSL

Each styling property defined in SVG specification has a corresponding XML (so called “presentation”) attribute with the same name. For example: SVG *stroke* property (defining the color of a line that is painted along the outline of the given graphical element) has a presentation attribute *stroke* that is used to specify the property.

10. Cascading Style Sheets, level 2 is a style sheet language developed by W3C, recommendation since the 12th May 1998 [24]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
  width="200" height="100">
  <circle cx="50" cy="50" r="40" fill="red"/>
  <circle cx="150" cy="50" r="40" stroke="black"
    stroke-width="2" fill="none"/>
</svg>
```

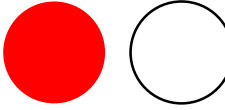
The image shows two circles side-by-side. The left circle is a solid, bright red. The right circle is a black outline with a white fill, and it has a slightly thicker stroke than the first circle.

Figure 2.2: Presentation attributes example

The presentation attributes are simple, widely supported by interpreters and viewers, allow restyling by using XSLT or by adding CSS style to override presentation attributes. They are also easy to use for content generation using XSLT. On the other hand, they bring some limitations: the file size may increase; the styling is attached to the content which makes restyling difficult; the styling rules cannot be abstracted and attached to groups matching specific pattern. Lastly, if an SVG content is embedded into other CSS-styled document, the presentation attributes may collide with the styling defined in the CSS sheet and thus the graphical presentation of the SVG content can deteriorate. They have also the lowest priority while rendering in applications supporting CSS. [16]

While styling with XSL, the presentation attributes are used in the same way as described above. The only differences are that the XSL style sheet contains templates matching the structure of a source and that the presentation attribute is defined with an XSL construction (see figure 2.3).

```
...
<xsl:template match="svg:rect">
  <xsl:copy>
    <xsl:copy-of select="@*" />
    <xsl:attribute name="fill">red</xsl:attribute>
    <xsl:attribute name="stroke">blue</xsl:attribute>
    <xsl:attribute name="stroke-width">3</xsl:attribute>
  </xsl:copy>
</xsl:template>
...
```

Figure 2.3: Example of XSL styling

2.3.2 Styling with CSS

SVG implementation supporting CSS is required to support external CSS style sheets, internal CSS style sheets and inline style. The declaration or style sheet rules has to conform to the specification of Cascading Style Sheets, level 2.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
  width="200" height="100">
  <defs>
    <style type="text/css"><![CDATA[
      circle {
        fill: red;
        stroke: black;
        stroke-width: 2
      }
    ]]></style>
  </defs>
  <circle cx="50" cy="50" r="40"/>
  <circle cx="150" cy="50" r="40"/>
</svg>
```

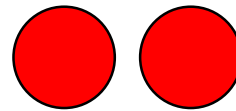


Figure 2.4: Internal CSS style sheet example

Note in the example (vide figure 2.4) that both circles are styled with one CSS rule – the opportunity to style according to a data model is missing within presentation attributes styling method.

The difference between internal and external style sheet is that in case the external style sheet is used, the reference to the CSS style sheet, for example: `<?xml-stylesheet href="foo.css" type="text/css"?>` has to be placed in the preamble of the SVG document. The referred style sheet contains only CSS rules. In the case when internal style sheet is used, the styling rules are strongly recommended to be surrounded with CDATA construct (i.e. `<![CDATA[. . .]]>`). This is necessary as the style sheet rules may contain characters that conflict with XML parsing.

The last method is inlined CSS styles. The rules are declared as pairs “property-name:value” in semicolon-separated list placed in a `style` attribute of an element – the same method that can be used in (X)HTML.

CSS selectors works in SVG similarly to (X)HTML as well. Rules can be attached to SVG element node types or elements with the attribute `class` or `id` assigned. Use of contextual CSS selectors is available as well. For more information about selectors see the CSS2 specifications. [16]

2.3.3 Accessibility of Styles

The use-case scenario of an SVG document can differ and so can the scope where a style sheet is available.

Stand-alone SVG document: any style sheet defined within the document in any way noted in the preceding sections is available across the entire document.

Stand-alone SVG document embedded by reference to a file¹¹: both referencing and referenced documents see only their own style sheet. To achieve the same styling, both documents have to be linked to the same style sheet.

Stand-alone SVG content directly included in XML document: the style sheet of the XML document implicitly affect the included SVG content, even though the content use different namespace than the rest of the document. To get different styling for the SVG part, the `style` attribute should be used. It is also possible to define `id` attribute on the `<svg>` element and then use contextual SVG selectors. [16]

2.4 Representation of Text Content in SVG

The previous section described how to define visual style of an SVG document. Using SVG shapes (`<rect>`, `<circle>` etc.¹²) is quite straightforward, thus it is omitted from this text. However, visualization of text using SVG created an interesting challenge for this thesis. For this reason, the part of SVG specification dedicated to text is briefly described.

To include textual content into an SVG document we use the `<text>` element in which a string of the text is represented as XML character data¹³; this is a great benefit of using text in SVG. Firstly, the text data are readily accessible to the visually impaired, secondly, the text strings are available to reach by searching, as well as select and copy to the system clipboard. The fact that the text content is searchable also enables Web search engines to search throughout the SVG documents.

All the features applicable on graphic elements affect the `<text>` element as well: coordinate system transformations, painting, clipping and

11. Reference created using `img` (HTML, XML), `object` (HTML) or `image` (SVG) elements

12. The full list and description is to be found in the SVG specification, chapter Basic Shapes

13. The description of the format is given in an XML language specification, chapter Character Data and Markup

masking. The obligatory attributes of `<text>` element are `x` and `y` position, which characterize the position of first glyph of the string. Other attributes related to `<text>` are relevant according to the usage context; text is possible to visualize simply in a straight line or attached to a path. For both, SVG supports processing features allowing horizontal and vertical orientation of text and left-to-right or right-to-left text (for example Arabic or Hebrew). [16]

Each `<text>` element contain a single string of text. Thus, visualizing multiline text is achieved using one of following methods:

- Pre-computing the line breaks and using multiple `<text>` elements (one for each line of text)
- Pre-computing the line breaks and using single `<text>` element containing several `<tspan>` child elements. Note: it is necessary to define appropriate start positions for each `<tspan>` element.
- Expressing the text in different language, for example XHTML, which is embedded inline within a `<foreignObject>` element¹⁴. Even though this is part of XML 1.1 recommendation, the exact semantics of this approach is not yet completely defined. [16]
- The text to be rendered is enveloped with `<textArea>` element. This approach is new in SVG 1.2 specification. Implicitly, a `<textArea>` represents a region that is rectangular; nevertheless, in other profiles the region may allow a sequence of arbitrary shapes. [17]

However, the support of `<textArea>` and `<foreignObject>` by web browsers is meagre. SVG developers thus face a challenge with the compatibility among different browsers.

2.5 Interactivity of SVG Content

Creating of a visualization does not only mean giving the data some graphical representation; this would be possible to achieve by using the raster format as well. The advantage of SVG is that we may provide very sophisticated interactive user interface, thus add an extra value on the top of the data presentation simply by exploiting SVG features and/or a scripting language. The features that can be utilized to provide interactivity are following:

14. The use of `<foreignObject>` element is described in the chapter Extensibility of the SVG specification

- triggering execution of scripts or animations by user-initiated actions (clicking mouse button etc.)
- following hyperlinks to the new Web pages by user-initiated actions
- panning and zooming the SVG content
- changing the cursor according to the movement of a pointing device [16]

2.5.1 Events in SVG

In the list above were mentioned user-initiated actions which are represented by events. An DOM2 event listener¹⁵ can be registered using the SVG DOM; when a registered event occurs, scripts or animations can be invoked/ended. SVG also includes event attributes which associate a script to be executed when a given event occurs to an element which owns the event attribute. [16]

The complete set of events available in SVG is listed in DOM and SVG specifications under the chapter Interactivity; events presented in the specifications include document events, pointer events, but there are no key events provided, however, it is planned to include them in future version of the DOM and SVG specifications. It is important to emphasize that the event model used in SVG has slight differences to the model used in HTML; furthermore some events commonly used in HTML are not implemented in SVG (for example double-click event). Also touch-input devices may create a challenge – device specific events may prevent the use of certain pointer events. A pointer event is a representation of an action performed on a pointer device. At any rate, the interfaces of mobile devices are being rapidly developed and the current situation can change in the near future.

The group of pointer events provides a mechanism to emulate the behavior of a pointer device. Each pointer event determines its target element, an element that is the topmost graphics element whose content is under the pointer at the time the event occurs. Interaction behaviors may be either dependent on the type of target element or set explicitly (scripted event listeners, CSS pseudo-classes matches or declarative animation triggered by events. The behavior of pointer events can be more precisely specified using a `pointer-events` property. [16]

15. Document Object Model Level 2 Events Specification is W3C recommendation defining an interface that provides a generic event system to programs and scripts [25]

2.5.2 Scripting in SVG

Scripting over SVG content provides a possibility to create a very interactive environment. The default scripting language used in SVG is ECMAScript¹⁶, although it is possible to specify another scripting language using `contentScriptType` attribute on the `<svg>` element, or by specifying `type` attribute on the `<script>` element. `<script>` used in SVG is equivalent to the same element used in (X)HTML, therefore the scripts are placed within. The scope of such scripts is “global”, i.e. they are applicable throughout the entire document. [16]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
  width="200" height="100">

<script type="application/ecmascript"> <![CDATA[
  function circle_click(evt) {
    var circle = evt.target;
    if (circle.getAttribute("cx")==50)
circle.setAttribute("cx", 150);
    else
circle.setAttribute("cx", 50);
  }
]]> </script>
<rect x="5" y="5" height="90" width="190" stroke="black" stroke-width="1"
  fill="none"/>
<circle cx="50" cy="50" r="40" fill="red" onclick="circle_click(evt)"/>
</svg>
```

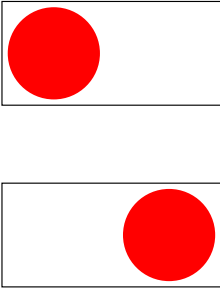


Figure 2.5: The pictures show the change after a click on the circle is performed

The figure 2.5 shows, how a simple script is used within the `<script>`. Note that the construct `CDATA` appears again; the reason for its use is explained in the section 2.3.2. The function `circleclick` from the script is triggered by firing the click user-interface event via `onclick` event attribute on the `<circle>` element. `<script>` can contain a reference to an external file containing scripts; for scripts holds the same as for style sheets: the `CDATA` construct has to envelope scripts only if they are directly in the SVG document. [16]

16. ECMAScript is the scripting language standardized in ECMA-262 specification and ISO/IEC 16262. [26] Its dialect JavaScript is extensively used for client-side scripting on web

Above were mentioned event attributes which facilitate one method of event handling. Their use is not universal, thus they are grouped into:

Document-level event attributes: specify scripts to run when a particular document-wide event occurs

Graphical event attributes: specify scripts to execute on firing a particular user interaction event

Animation event attributes: specify scripts to run for a particular animation-related event

onload event attribute: stands a bit aside as it can be specified on all animation elements and most of the graphical and container elements. The event specifies scripts to run when an `SVGLoad` event is fired on the element where `onload` is specified on.

The full list of event attributes that can be used to provide an SVG graphics with interactivity follows:

- `onload`
 - `onfocusin`
 - `onfocusout`
 - `onactivate`
 - `onclick`
 - `onmousedown`
 - `onmouseup`
 - `onmouseover`
 - `onmousemove`
 - `onmouseout`
 - `onunload`
 - `onabort`
 - `onerror`
 - `onresize`
 - `onscroll`
 - `onzoom`
 - `onbegin`
 - `onend`
 - `onrepeat`
- [16]

While developing an SVG interactive application, apart of event attributes, developers can utilize event listeners described in the DOM specification. For associating an execution of a script with a certain user-interface event, an event listener is registered on a desired element. After the event, for which the event listener was registered occurred, the script is invoked. To allow the registration, DOM introduced and set of interfaces:

EventTarget: is implemented by all nodes in SVG. Using binding-specific casting methods (`addEventListener`, `removeEventListener` and `dispatchEvent`) can be `EventListener` accessed.

EventListener: handles the event itself – after its invoking, a script attached by `addEventListener` method is executed.

Event: provides contextual information about an event to the event handler, for example: `Event.target`, `Event.timeStamp` etc. [25]

DOM Level 2 Event Model allows the definition of new event modules according to requirements. These can be assigned to an element thanks to the event listeners. For the purpose of interoperability, DOM defines a module of user interface events. [25]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
  width="200" height="100" onload="initialize(evt)">

<script type="application/ecmascript"> <![CDATA[
  function initialize(evt){
    var circle= document.getElementsByTagName("circle")[0];
    circle.addEventListener("click",circle_click, false);
  }
  function circle_click(evt){
    var circle = evt.target;
    if (circle.getAttribute("cx")==50)
      circle.setAttribute("cx", 150);
    else
      circle.setAttribute("cx", 50);
  }
  ]]> </script>

<rect x="5" y="5" height="90" width="190" stroke="black"
  stroke-width="1" fill="none"/>
<circle cx="50" cy="50" r="40" fill="red"/>
</svg>
```

Figure 2.6: The example from the figure 2.5 implemented with event listeners

Chapter 3

Fresnel Editor

As the Fresnel language was specified, its implementations emerged, though, not many of them. There are two types of tools: browsers, for example Longwell, Horus or IsaViz, and authoring tools like Cardovan (an internal IBM tool).¹ In December 2008 at the Faculty of Informatics of Masaryk University, Brno a project called Fresnel Editor was started as a part of a lightweight semantic framework. Fresnel Editor is an open source GUI-based tool offering an ability of authoring all Fresnel Core Vocabulary elements for a visualization of RDF data. The focus was given on modularity, to offer an easy way for extending the project with plugins. [27] The intention of this chapter is to introduce Fresnel Editor to a reader unaware of its existence. As a consequence, the description of the SVG visualization module implementation will be easier to follow later in the text.

3.1 Architecture of Fresnel Editor

The whole project is written in Java, more specifically using the Spring framework and for creating the GUI was used the Swing framework. The core libraries used for developing Fresnel Editor are:

Sesame²: providing a local RDF data repository and a support for SPARQL queries.

JFresnel³: is used for loading the Fresnel configuration (Lenses, Formats and Groups) from the RDF repository.

1. The development of the tools seems to be suspended recently. The last updates on projects' web sites are from years 2007 – 2009

2. Sesame is being developed by Aduna, current version 2.6.4 was released on the 14th March 2012 [28]

3. JFresnel current version is 0.3.2 released on the 18th May 2010. Its development runs under INRIA, one of the authors is E. Pietriga, a co-author of Fresnel specification [29]. Some parts were created by students of the Faculty of Informatic at Masaryk University, Brno, namely M. Warchil and J. Horváth

The authors of Fresnel Editor, M. Warchil and I. Zemský divided the whole project according to requirements into the following modules:

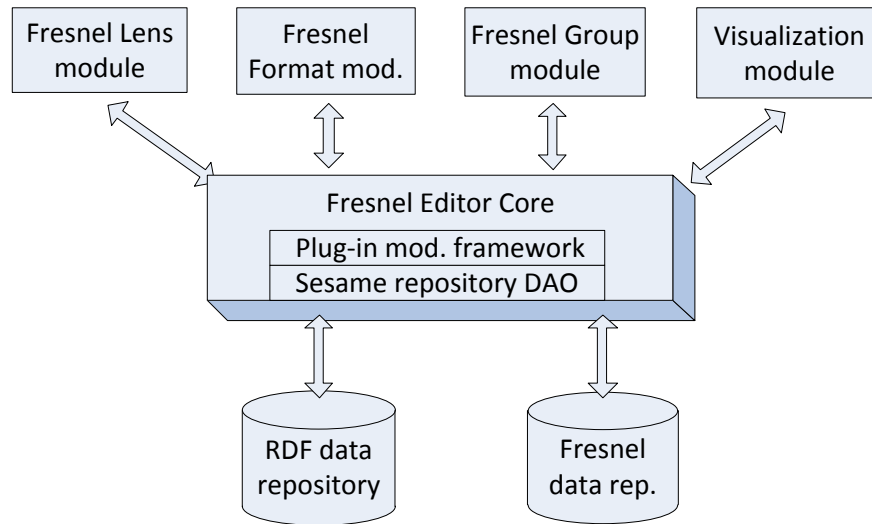


Figure 3.1: Schema of the architecture of Fresnel Editor [27]

Core module: represented by *fresnel-editor-model* and *fresnel-editor-common* sub-modules. The first one encapsulates the access to a semantic repository, the second contains the logical and user-interface core, including the interfaces providing the extensibility of Fresnel Editor

Lens module: represented by *fresnel-editor-gui-mod-lens* sub-module, which provides GUI components for easy and straightforward creation or editing the Fresnel Lenses.

Format module: represented by *fresnel-editor-gui-mod-format* sub-module. This part has basically the same purpose as Lens module but aims on the Fresnel Formats.

Group module: represented by *fresnel-editor-gui-mod-group* sub-module. Again the same purpose as the previous two modules but aimed on the Fresnel Groups.

Visualization module: uses the structures defined in previous three modules, facilitates algorithms for rendering thumbnails and methods for

the visualization of RDF data from the repository. It is represented by *fresnel-editor-gui-mod-vis* sub-module. [14]

Deeper insight into the implementation of Fresnel Editor is possible to gain in the master theses of M. Warchil [14] and I. Zemský [30].

3.2 The Visualization Process

A visualization of the RDF data is among the main goals of Fresnel Editor. The visualization module listed above encapsulates the necessary functionality. There are two manners of the visualization process:

- either all the Fresnel elements from the Fresnel repository are applied on the data stored in the data repository, thus completed visualization is produced
- or only a thumbnail is created while a Lens or Format is edited. Both visualizations means share the same algorithm, but in case of thumbnails preparation steps are necessary. [14]

In the former version of Fresnel Editor, the authors aimed the application at users well aware of the specifications of Fresnel and RDF. Therefore defining the lenses and formats was quite difficult. In 2010, J. Horváth provided Fresnel Editor with an alternative module for defining lenses and formats to simplify use of the tool. However, the visualization algorithm was not changed [31]

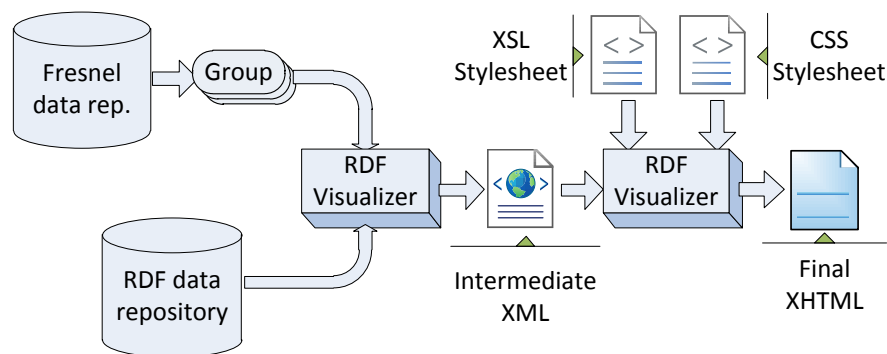


Figure 3.2: Schematic view on the visualization process [30]

1. Classes and their properties that are to be visualized are selected. The properties are ordered and everything forms an RDF node tree. So far, no visual formatting was associated.
2. Visual formatting rules are applied on the RDF node tree. All the selected sources and their properties, together with the visual information are serialized into an intermediate XML document.
3. The intermediate XML document is transformed into the XHTML format using an XSLT transformation style sheet
4. A frame of an internal web browser is instantiated and then navigated onto the previously created XHTML document. During the rendering phase, the associated CSS style sheet is applied on the appropriate elements and the result is displayed in the web browser frame. [14] [31]

Let us take a closer look on the transformation step. In the former version of Fresnel Editor, only transformation to XHTML is mentioned; however, the plan of RNDr. Tomáš Gregar Ph.D. who advised the Fresnel Editor project, was to develop more XSLT style sheets and thus offer more output formats. During the 2010 spring semester course, PB138 – Modern Markup Languages and Their Applications, that was held at the Faculty of Informatics of Masaryk University, Brno, several style sheets were developed as part of the course project. Additional style sheets were also created by S. Petrová within the work on her bachelor thesis [32] – eventually Fresnel Editor can produce for example RDFa, microformats, HTML5 and SVG (the last will be described in this thesis). Besides, the ability to output more formats than XHTML has raised an additional problem concerning the display phase.

The displaying of processed data itself happens in an internal browser; for this purpose the authors of Fresnel Editor used the Lobo browser. It is a pure Java open-source web browser which targets HTML4, JavaScript and CSS standards. The last version of Lobo was published on the 18th of January 2009. [33] At this moment, Fresnel Editor faces one major issue: it is not able to visualize any other format than (X)HTML. From the fact that the most recent update of The Lobo Project web site was in October 2009, we can assume there is no activity, thus the chance for expanding the ability of the Lobo browser is small. Drawing on the bachelor thesis by S. Petrová, the Lobo browser has also discrepancy in rendering CSS styles comparing to the major web browsers. [32]

3.3 Issues and the Current State of Fresnel Editor

Fresnel Editor, a project of the theses of M. Warchil [14] and I. Zemský [30] was not released as a stable version. For the project management was used Apache Maven, a very useful tool when it is well set up and when the Maven library repositories do not change their location. Due to the fact that both authors left the project after finishing their theses, nobody administrated the maven configuration for some time and the Maven management system went out of date; repositories were relocated and some libraries become obsolete.

Additional errors and exceptions are listed in the bachelor theses of S. Petrová [32] and J. Horváth [31], who conducted thorough testing and fixed the errors found.

The most recent official version of Fresnel Editor (published on Sourceforge project website⁴) from the 20th of June 2011 is 1.0. [34]. Separately from the Sourceforge website of the project, J. Horváth maintains a GIT repository⁵ with his development version of the Fresnel Editor project. This version was forked for the purpose of this thesis.

Although the Fresnel Editor is currently in a working state, being maintained and slowly extended, the project would deserve a major cleanup and possibly a reconsideration of the design followed by a reimplementa-tion; actually the Fresnel Editor advisor – RNDr. Tomáš Gregar Ph.D – had such a vision for the future of the project.

4. <http://sourceforge.net/projects/fresnel-editor/>

5. <https://github.com/nodrock/fresnel-editor>

Chapter 4

Batik SVG Toolkit

The need for an SVG visualizer originated in the inability of Fresnel Editor to display a result SVG image. As Fresnel Editor is written in Java, a need for a Java-based tool or library capable of working with SVG emerged. That is where Batik SVG Toolkit (shortly Batik) has proved to be useful – the SVG visualization module of Fresnel Editor is built utilizing Batik’s modules. What does the tool actually offer? This chapter provides a description of Batik and its components.

Batik is Java-based framework (i.e. a set of libraries) that facilitates the handling of SVG documents; the modules can be used for generating, manipulating and transcoding SVG images. The current version – 1.7, released on the 10th of January 2008 – implements almost 94% of SVG 1.1 specification¹ and some parts of SVG 1.2, which makes Batik one of the most conformant implementation of the SVG recommendation. [35]

The modules contained in the Batik’s distribution are logically separated into three layers (see the figure 4.1):

Applications: illustrate how it is possible to use the core modules. It let users evaluate and experiment with Batik’s features.

Core modules: are the libraries developers can use in their projects. These modules provide the main functionality; via them the developers can manipulate, generate, create, convert, render and view SVG content.

Low level modules: are used by core modules. They are rarely used directly for a development.

1. The information is based on the Batik’s implementation status available on <http://xmlgraphics.apache.org/batik/status.html>

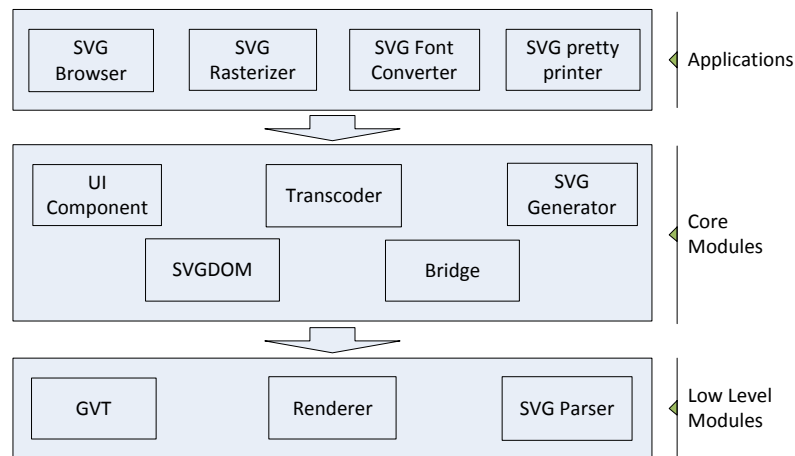


Figure 4.1: The architecture of Batik SVG Toolkit [26]

The application layer contains four ready-to-use tools. The Squiggle SVG Browser (in the figure 4.1 it is referred to as SVG Browser) shows capabilities of the transcoder (represented by a class called `ImageTranscoders`) and UI (`JSVGCanvas`) components; therefore a user can view, zoom, pan or rotate SVG documents and convert them into raster formats. The SVG rasterizer also uses transcoder API, allowing conversions from and to SVG format. Next, the SVG Font Converter is used to convert any True Type Font to an SVG font which are useful in SVG documents. The last one, the SVG pretty printer, helps to tidy and organize potentially disorganized SVG files. [35] These tools should more or less demonstrate the strength of Batik; for further information see the Batik's web page².

The low level layer contains modules that are not usually directly used for a development. That is the reason why they are mentioned only briefly. The module labeled in the figure 4.1 as GVT is the Graphic Vector Toolkit module. It represents the DOM tree in a more suitable way for the rendering and event handling purposes. The module represents the DOM tree as a tree of Java graphical objects. Next is the Renderer module which is responsible for rendering a GVT tree and related tasks. Finally, the SVG parser module facilitates parsing of complex SVG attributes, like `transform`. [35]

The core modules are described in a higher detail in the next section.

2. <http://xmlgraphics.apache.org/batik/>

4.1 The Core Modules of Batik

The most interesting part for developers is without doubts the core modules layer of Batik. In the upcoming paragraphs, the basics of each module are presented. Despite their importance, the description and examples provided in Batik documentation [35] are relatively short; the readers who would like to learn more should refer to a course by Cameron McCormack [36] and Batik Wiki. Another sources are Mailing List [37] administrated by Apache Software Foundation, as well as the Batik's Javadoc API [38] (however, the Javadoc provides merely the raw structure of the source packages).

Firstly we will examine the SVG generator module, which can be used in applications and applets for converting graphics to the SVG format. In fact, the SVG generator refers to `SVGGraphics2D` class; it is an implementation of standard Java `Graphics2D` abstract class accommodated to the generating of the SVG content. `SVGGraphics2D` allows exporting graphics into the SVG format without the necessity of any modification of the graphics code. Other feature it has is the ability to use the DOM API for manipulating the generated content. The principle, how the SVG generator module works is following: it manages a tree of DOM objects representing the SVG content according to the rendering calls coming from the instance of `SVGGraphics2D`. This provides the programmer an ability to access the DOM tree for further manipulation (see below) or writing it directly to an output stream. If it is needed, the module provide a possibility to customize the generation process – from defining the generator context, through style handling to storing the result images. [35]

The DOM API defines an interface `DOMImplementation`, the purpose of which is to bootstrap a particular implementation of an XML DOM. In other words to provide a method to create an `org.w3c.dom.Document` class instance. Then, the particular XML is represented by the `Document` which also acts like a factory for the various DOM objects (`Element`, `Attr` etc.). For the purpose of SVG, Batik introduces an implementation of that interface named `SVGDOMImplementation`; thus developer can cast a document as an `SVGDocument` class instance. However, it is not necessary to represent an SVG document using the `SVGDocument` class because an SVG document can be built using DOM Level2 Core methods [39]. `SVGDocument` is used by Swing and transcoder modules to render an SVG Document. [35]

The task of the thesis was to create an SVG visualizer for Fresnel Editor. The user interface was programmed using the Batik Swing component represented by `JSVGCanvas` class. The Swing component allows displaying the SVG content from a URI or a DOM tree and manipulating it (for example

zooming, panning etc.). [26] The `JSVGCanvas` class is built according to the Swing design rules which made the the class thread unsafe. Thus any usage of that class should be concurrency aware. The class is also a `JavaBean`³, so it can be used in visual application builders. During its runtime, when a `JSVGCanvas` instance displays an SVG content, it performs specific operations, such as parsing, building and rendering or updating. To notify about these operations it is possible to implement a set of listeners that track appropriate events:

SVGDocumentLoaderListener: describes the loading phase: constructing an SVG DOM tree using SVG file.

GVTTreeBuilderListener: notifies about the building phase. In this phase a GVT tree is constructed from the SVG DOM tree. The GVT tree is then used to render the document.

SVGLoadEventDispatcherListener: can be used when processing dynamic documents, where the DOM `SVGLoad` event can be dispatched

GVTTreeRendererListener: describes constructing an image using a GVT tree – the rendering phase. (This event is fired only once for the initial rendering in dynamic documents)

UpdateManagerListener: notifies about the running phase. It follows the changes of a state of an update manager and tracks the updates on graphics. This event is triggered in dynamic documents only.

For every listener class listed above, there is an adapter class available to ease the creation of a new listener. `JSVGCanvas` also provides built-in interactors that let the users manipulate the displayed document by catching the user input to the `JSVGCanvas` component and translating it according to the performed actions.

The next module is the bridge module. Bridge is in control of creating and maintaining objects corresponding to SVG elements; to carry out this task, the bridge translates an SVG document into the Batik internal representation for graphics – Graphic Vector Toolkit (GVT). This module is only seldom used directly.

Lastly, there is the transcoder module providing a generic API for transcoding an input to an output. The transcoder module defines five major classes for its task:

3. JavaBeans are reusable software components written in Java conforming to a particular convention. It encapsulates many objects into the *bean* which can be used as a single unit

Transcoder: defines the `transcode` method, which is used for transcoding a specific input into a specific output.

TranscoderInput: handles the input of a transcoder. The default implementation uses `org.w3c.dom.Document`, `Reader`, `InputStream`, `org.xml.sax.XMLReader` or a URI, nevertheless the list can be extended by implementing this interface.

TranscoderOutput: represents the output of a transcoder. The most common ways to create an output are using `org.w3c.dom.Document`, `Writer`, `OutputStream`, `org.xml.sax.XMLFilter` or a URI. As well as with the input above, other can be added.

TranscoderHints: contains hints that can be used to control the options (parameters) of a transcoder

ErrorHandler: through this class it is possible to get errors and warnings that might occur during the transcoding process. [35]

4.2 Real World Projects Using Batik

So far, we have seen that Batik may be a very powerful tool. In this section some projects that use the Batik libraries are listed .

Apache Cocoon: a component-based web development framework. It uses Batik to rasterize SVG images.

Apache FOP: a Java application for rendering documents in various formats (including SVG). It employs Batik to handle SVG images and to convert them to the PDF format.

GLIPS Graffiti: is a full feature native SVG editor built on Batik.

Lagoon: is an XML-based web-site maintenance framework. Batik is used to render SVG as raster graphics for the purpose of web publishing.

Luxor XUL: is an XML User Interface Language toolkit written in Java. It lets developers to build user interfaces using XML. Luxor also includes an ultra-lightweight multi-threaded web server, a portal engine, a template engine, a scripting interpreter etc.

jDeveloper: is a free development environment created by Oracle for Java-based SOA applications and user interfaces with support for a full development life cycle. It uses batik to export class diagrams into SVG images.

Chapter 5

Data Visualization Approaches

In case of the semantic data, the need for a data presentation in a human readable form is strong. The problem lies in the possible complexity of relations amongst the data; Fresnel provides one way how to solve it. The capability of Fresnel was explored in the Fresnel Editor project, although only the visualization to the XHTML format was delivered with the first version. Together with XHTML as the most common method for data presentation, the SVG format was considered as it might have brought a completely different user experience. It is worth to note that due to the ability of SVG to be interactive, we are able to add an extra value to the visualization. In other words we can develop complex applications allowing users to view the data in context and/or manipulate them.

However, automated generating of SVG graphics has, unlike XHTML, some constraints. According to my experience gained within the domain, the problems are the positioning of the elements, the layout design of the graphics and the text representation.

To explain the issues, let us start with the positioning. The majority of visual elements in SVG (`<rect>`, `<text>` etc.) must have the `x`- and `y`-coordinates and dimensions defined. The value of the coordinates (and lengths as well) can be expressed either by units or by percentages. The units are represented by `px`, `em`, `ex`, `pt`, `pc`, `cm`, `mm` and `in`; all of them has precise values. On the other hand, when coordinates and lengths are represented in percentage, the value is relative to the nearest containing viewport. [16] For the purpose of the visualization, `px` were chosen as the basic unit to keep the layout of elements, in case the viewports would change (for example by resizing the viewer). But this brought the following problem: when the SVG document is generated, it is necessary to know the `width` and the `height` to be able to count the coordinates of the visual elements; the dimensions were omitted as we can assume that the measurements of shapes will not change during the generation process. Having a previously selected finite set of data without endless recurrences helps to compute the coordinates and the dimensions of the SVG document using XPath expressions.

Very closely bound with the previous problem is the layout design issue. In cases when we can assume the structure of the data but not their content in advance, it is difficult to put forward an user friendly and comely graphical representation of the data. Either the format must be very simple (for example a tree), or the desired data must be selected and their visual semantics defined prior the vizualization process starts.

The last difficulty mentioned above emerges from the fact that even though SVG can visualize text, its capabilities are not very convenient. To overcome this inconvenience, the data to be visualized are given to a preprocessor which adds an information on the basis of the text length.

5.1 Generating of an SVG Document

There are two methods how to produce an SVG document from the data. Assuming we have preselected the data and serialized them in a given format, we can either use an XSLT transformation or build the SVG document using DOM API. Both methods have their advantages and disadvantages; their use depends also on the platform which will be used to view the data and on the visualization purpose.

5.1.1 Transforming of an Intermediate XML Document

For the purpose of Fresnel Editor, the first approach was chosen on the basis of the data flow (see figure 1.2) introduced by Fresnel Editor authors.

The constraints of the visualization were listed in the introduction to the chapter. Fresnel Editor's ability to work over general data disables the possibility of the data preselection where would be omitted unimportant or unnecessary data; therefore the visualization has to be general.

At first, the naive idea of the SVG visualization process was to provide only an alternative XSLT transformation, which would simply substitute XHTML elements for appropriate SVG shapes.

Coming from RDF, the structure of the data to be visualized is a set of triplets: *subject* → *predicate* → *object*; hence a tree built from left side, where the *subject* is represented by the root of the tree, *predicates* by edges and *objects* by leafs, was selected for visualization of general data in SVG.



Figure 5.1: An example of basic XHTML visualization

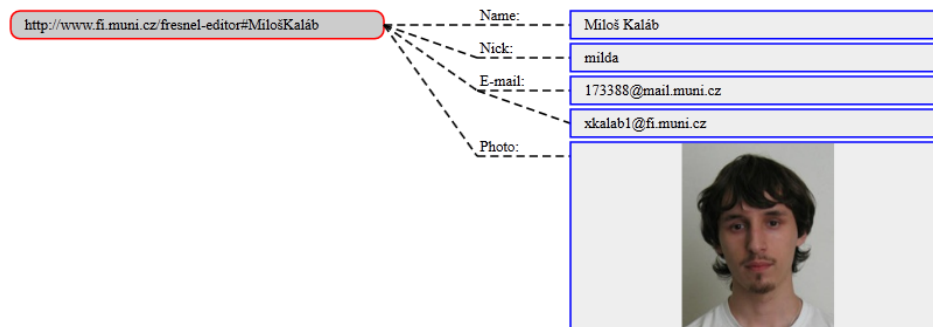


Figure 5.2: An example of basic SVG visualization

In the figure 5.2 only a few pieces of information about the depicted entity are visible. Furthermore, all of them are very simple. In cases with long texts as a value of property, `<foreignObject>` is used to envelope a piece of HTML code which allows the text to be easily wrapped.¹ Even though the SVG recommendation suggests using of several `<tspan>` elements for

1. `<foreignObject>` was preferred over `<textArea>`, since my testing showed better support among browsers

breaking long text [16], this approach was abandoned after a consultation with the thesis advisor, because the text wrapping problem is solved in the draft of next SVG specification [17]. Despite the fact that the long texts are handled by using the `<foreignObject>` element, it is necessary to determine what a long text is. Such a requirement led to changes in the data flow of the visualization process and incorporating a preprocessor of the intermediate data. A detailed description of this component, including the XSL transformation itself, is provided in the chapter 6.

Above, in the list of constraints is stated that the set of the data to be visualized must not contain endless recurrences, which is true. However, a situation where a given entity can be linked with another entity may exist. In such cases, the second entity stands for a value of a property of the first entity (see figure 5.3).

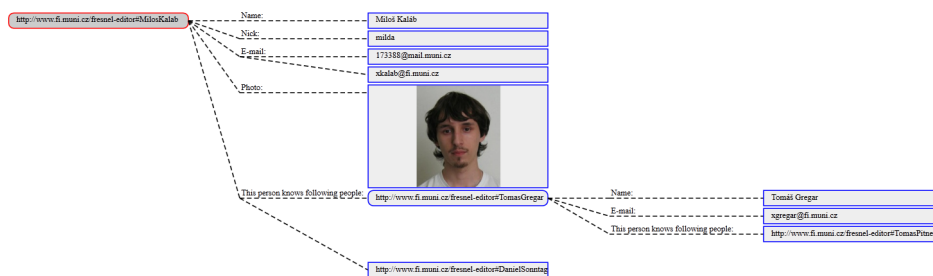


Figure 5.3: An example of SVG visualization with nested resources

Regarding such data structure, it was necessary to change the definition of parameters to make the recurrent matching of entity elements possible (ie. a resource node is present as a value of another resource's property, thus the template rule is matched recurrently). For the purposes that were stated earlier, it is necessary to pre-count the coordinates of all the visual SVG elements; this, in combination with the possible recurrent matching (in cases of multiple nested entities), may cause clarity loss of the resulting SVG graphics (see in the figure 5.4). Another example of the clarity loss happens when too many properties with too many values is visualized (see in the figure 5.5

5. DATA VISUALIZATION APPROACHES

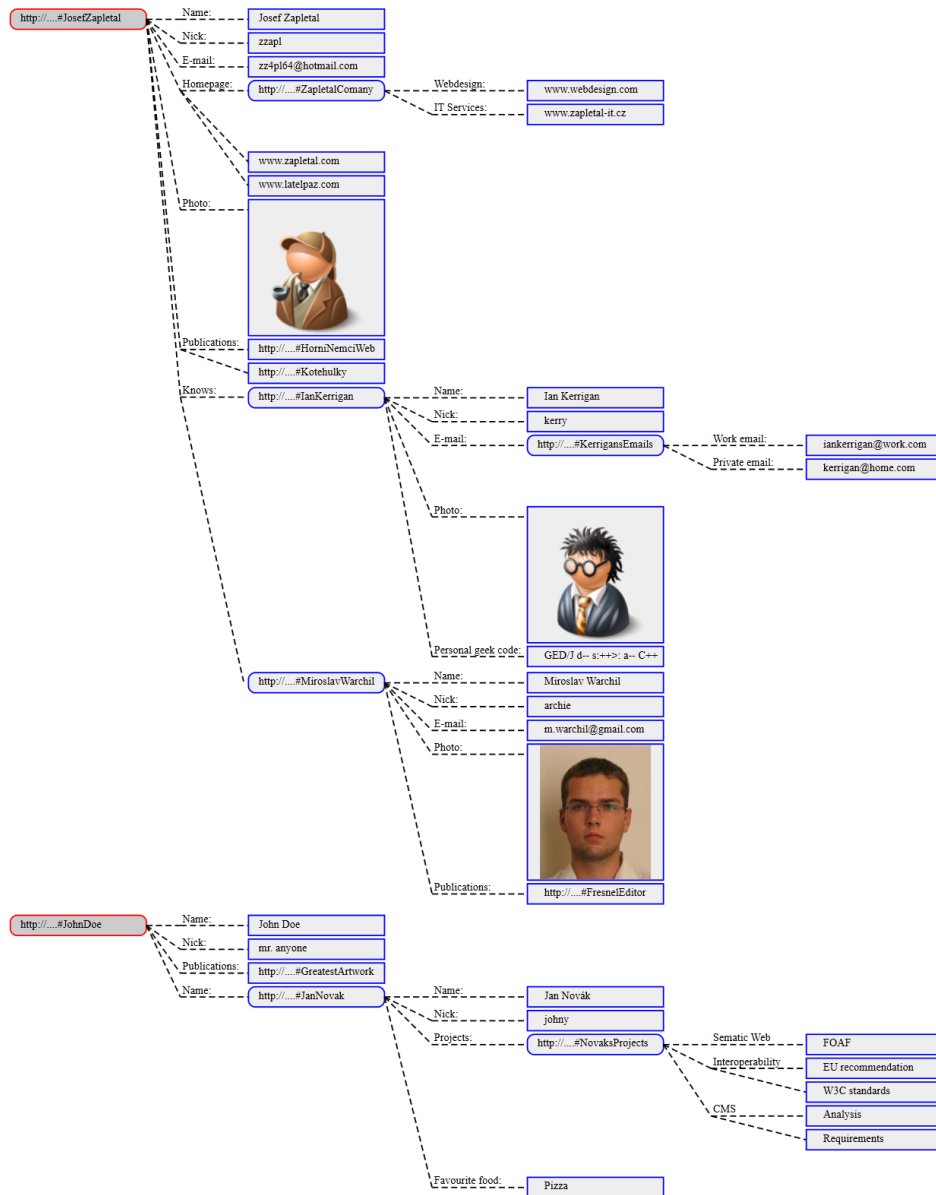


Figure 5.4: Example of the loss of clarity due to nesting of entities

5. DATA VISUALIZATION APPROACHES

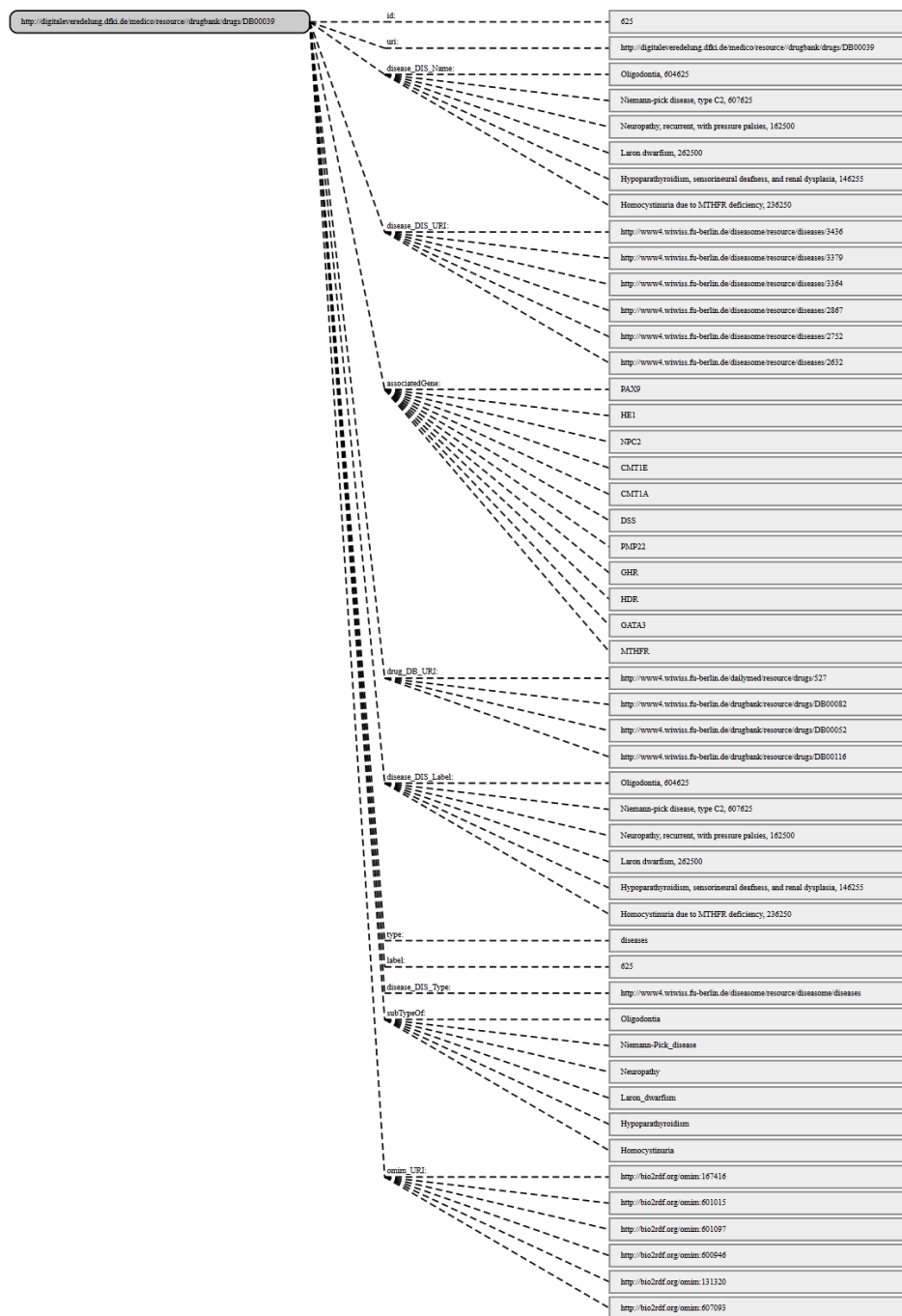


Figure 5.5: Example of the loss of clarity due to displaying too many properties and values

5.1.2 Building a Document via DOM API

Another way to create an SVG document given a set of data is to use a toolkit allowing us to manipulate the DOM tree: the Batik toolkit's DOM API interface or some JavaScript library providing such a functionality, for example jQuery-SVG or Raphaël.²

The XSLT recommendation states: "XSLT does not provide an equivalent to the Java assignment operator `x = "value";` because this would make it harder to create an implementation that processes a document other than in a batch-like way, starting at the beginning and continuing through to the end." [5] Using Java or JavaScript to build an SVG document gives us the advantage of side-effects³ and also the possibility to access the data in an order that is more convenient for us. Actually, if a result SVG image outputted from the transformation process is interactive, scripts providing the interactivity usually modify the DOM tree as well.

Whether to use Java or JavaScript matters only on the delivery method of the visualization. If the data is to be viewed in a web browser, using JavaScript library seems more natural because of the JavaScript support among the browsers (including mobile platform). Although it is reasonable to use Batik's DOM API module to generate the SVG content, it might not serve in terms of cross platform compatibility, as Java applets may not work in certain web browsers. Finally, although it is possible to equip both Batik toolkit and JavaScript for generating and manipulating the SVG document, this approach might lead to a redundancy; it would be necessary to react to the same events in Java code as well as in JavaScript.

A rather complex visualization is described later in the chapter about Medico project (see chapter 7).

2. jQuery-SVG [40] is a jQuery plugin that provides SVG support. Raphaël [41] is a JavaScript framework that allows interaction with SVG

3. the ability to change the values stored in fields of objects or elements

Chapter 6

SVG Visualization in Fresnel Editor

The previous chapter suggested the necessity to preprocess the visualized data. In the context of Fresnel Editor it means to add one additional step after the intermediate XML document is produced as a result from Fresnel lenses.

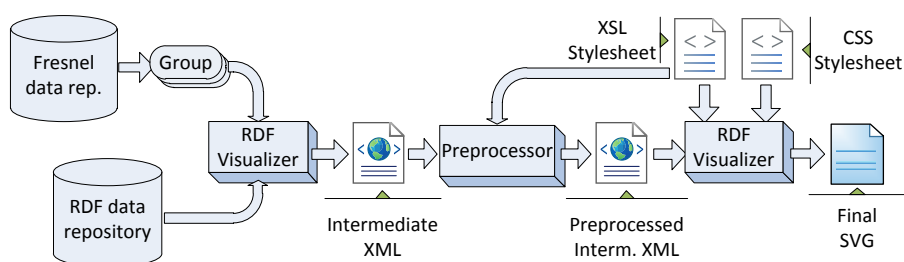


Figure 6.1: A schema of the visualization process with the preprocessor

In the following sections will be introduced the XSL transformation and afterwards the way how the preprocessor works and is implemented.

6.1 XSL Transformation into the SVG Format

As it is noted in the previous chapter, the XSL transformation to SVG format was based on the transformation to XHTML. After the analysis of the XHTML visualization, it was decided that the tree structure would fulfill the requirement to visualize general data. The resources and values are displayed as a text enveloped with a rectangle, the link between the two of them as a line with a textual description (see the figure 5.3. It seems to be a very simple structure on the first look but the features of SVG created a few challenges.

The XSL transformation process matches the template rules on the nodes in an input tree structure starting with the root element, the result tree structure is generated the same way. [8] Firstly, the root (`<svg>`) element is created; within this element, it is necessary to state the `height` and `width` of the document – how to do it when the content of the source document is yet unknown?

These measurements, together with other parameters are computed prior the matching phase of the transformation. Some parameters describe the dimensions of the rectangle, font size, length of lines and various distances between visualized elements, other holds an information about the number of resources, properties and values to be displayed, acquired by XPath queries. The number of the available parameters is relatively high and should contain all the measurements that can influence the final appearance of the result SVG image. Nevertheless, to prevent confusion, only a smaller set can be directly changed by users in the preprocessor; the less important parameters are customizable in the XSL transformation file.

The `height` attribute was simple to count – the number of values multiplied by their heights and indents. However, the `width` attribute is more difficult to compute; counting of resource nodes would not help as XPath would not recognize which resource is nested and which not. Unfortunately, the implemented solution is not completely universal; its issue is that the XPath queries are limited at the moment on maximally four nested resources with their properties and values. The solution of nesting of resources raised the complexity of formulas for counting the coordinates.

A set of XPath queries is evaluated not only at the beginning of the whole transformation, but on the beginning of (almost) every template rule, counting how many of each type of source element have already been processed. The obtained parameters help to count the exact `x`- and `y`- coordinates.

Although the styling attributes of SVG visual elements are provided in an external CSS style sheet, one exception was made: the `font-size` attribute. The reason for such an exception is the use of the `<foreignObject>` element with embedded XHTML code. The styling rule didn't apply on the XHTML code, therefore every `<text>` element has its own `font-size` presentation attribute, and the XHTML code has it defined in a `style` attribute. Eventually, this exception simplified the font size setting in the preprocessor.

The representing of long texts with the `<foreignObject>` element solves the word wrapping, yet, it provides a different challenge in return: assuming the height of the value rectangle with long text. The processing of the text will be explained in the following sections; however, a part of the solution does take place in the XSL transformation. When the value of

a property is matched with a template rule, the source element is tested for a `long-text-rows` attribute. If the value of the attribute is 1, then an ordinary `<rect>` and `<text>` elements are used, `<foreignObject>` with XHTML code otherwise. Still, to avoid big, barely empty rectangles with only two lines of text (for example) in the result SVG image, the height of the `<foreignObject>` is preset to four different lengths of the text. The right height is chosen also according to the value of the `long-text-rows` attribute.

Predefined cases to visualize general text do not provide the ultimate solution. The problem lies in the transformation approach (and SVG text content visualization at all). Building the SVG result image in an imperative programming language like Java would definitely simplify the computing of `x-` and `y-` coordinates of elements as well as their width. However, the DOM API lacks a function that would compute the height of `<foreignObject>` (or `<textArea>`) element; at least in the current version of Batik SVG Toolkit.

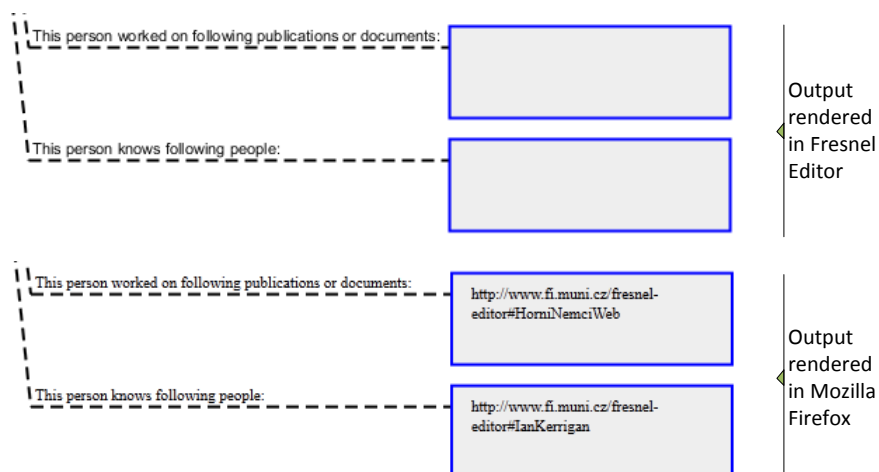


Figure 6.2: The comparison how a fragment of the same SVG document is rendered by Mozilla Firefox and Fresnel Editor

At last it is necessary to note that an issue regarding the displaying of long texts was discovered during the implementation phase. Neither `<foreignObject>` nor `<textArea>` works well in Batik SVG Toolkit. The

first one does not render the HTML content of the element whereas the other is not part of the SVG 1.1 specification, thus it is not supported in the current version. On the other hand, if the SVG document is saved and then opened in a web browser, the long text is rendered as it should be.

6.2 Preprocessor

The reasons for the text preprocessing were explained earlier in the text: for each text to be visualized in SVG format it is necessary to decide whether it would overflow its rectangle and if it would, how high the rectangle needs to be to envelope the text. Another functionality of the preprocessor is to allow a user to specify (some) parameters of the visualization.

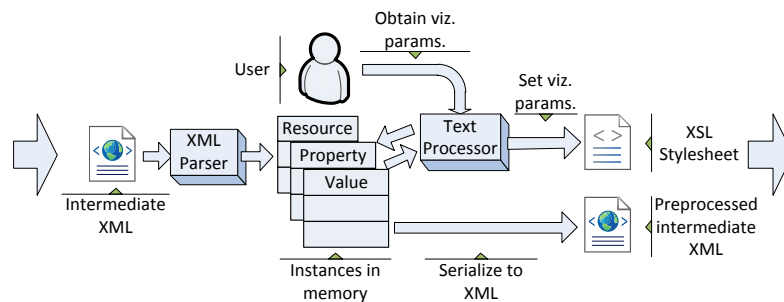


Figure 6.3: Schematic view on the work flow of the preprocessor

6.2.1 Design

Following the analysis of the SVG visualization requirements for the Fresnel Editor, its design was carried out. In the subsections below, there are summarized use cases and other diagrams.

6.2.1.1 User Roles

The user roles have two layers depending on the scope that is taken into the consideration. If is Fresnel Editor considered as a single system, then we can distinguish:

- User
- System (Fresnel Editor)

In a closer look on Fresnel Editor the whole system can be differentiated further:

- Data selector – covers Lens, Formats and Group module, more detailed description is beyond the scope of the thesis
- SVG Preprocessor
- XSL style sheet handler
- XML handler
- XML transformer
- JSVGCanvas – a component to display the visualized data

6.2.1.2 Use Cases

Use Case 1	Visualizing data into SVG format
<i>Level:</i>	User-Goal
<i>Primary Actor:</i>	User, Fresnel Editor
<i>Goal in context:</i>	To set the parameters for the visualization and run the visualization process
<i>Preconditions:</i>	The SVG Visualization tab in Fresnel Editor is opened
<i>Postconditions:</i>	User sees the data visualized to the SVG format
<i>Scenario:</i>	<ol style="list-style-type: none"> 1. User selects a group to be visualized 2. User sets parameters of items in the SVG visualization (optional, user can let the default setting) 3. User specifies the CSS style sheet to be used for styling the SVG visualization (optional, if not set, the default CSS style sheet is used)

6. SVG VISUALIZATION IN FRESNEL EDITOR

4. User specifies output file path where the result SVG image will be saved (optional, if not set, the result is stored in a temporary file)
5. User starts the visualization process by clicking the *Visualize RDF data* button
6. Fresnel Editor process the data and generates the visualization – see Use Case 2
7. Fresnel Editor displays a panel with the generated SVG image

Exceptions:

- 5.a No group is selected for the visualization
 1. An error message is displayed
 2. User returns to the step 1
- 5.b Invalid parametres set – letters or negative values as an input:
 1. An error message is displayed
 2. The part with SVG visualization parameters is highlighted
 3. User returns to step 2
- 5.c An unfitting CSS style sheet is selected:
 1. SVG visualization will not display correctly

Priority: Critical

Frequency of occurrence: Oc- Frequent

Use Case 2	Processing of the visualization
<i>Level:</i>	Subfunction
<i>Primary Actor:</i>	Data selector, SVG Preprocessor, XSL style sheet handler, XML handler, XML transformer, JSVGCanvas
<i>Goal in context:</i>	To generate a visualization according to the parameters provided in Use Case 1
<i>Preconditions:</i>	The <i>Visualize RDF data</i> button was clicked and no exception occurred
<i>Postconditions:</i>	Fresnel Editor generates an SVG image
<i>Scenario:</i>	<ol style="list-style-type: none"> 1. User selects a group to be visualized 2. Data selector selects data to visualize according to the selected group from Use Case 1 – Step 1 3. Data selector serializes the data into a temporary XML document 4. SVG preprocessor is started and instantiated its XSL style sheet handler and XML handler 5. The XSL style sheet handler changes the parameters passed from Fresnel Editor’s GUI in the XSL transformation 6. The XML handler parses the temporary XML document creating in-memory object representation of the data 7. The SVG preprocessor computes the length for every string that is going to be visualized as a text

8. The SVG preprocessor compares the computed length with the parameters of the XSL style sheet and an auxiliary parameter is set to determine the length of the text in rows
9. The SVG preprocessor shortens the text belonging to instances with certain attributes that requires only a single-line item in the result image
10. The SVG preprocessor serializes the in-memory objects into a pre-processed XML document
11. The XML transformer transforms the pre-processed XML document using the XSL style sheet from Step 2
12. The result SVG document is rendered by `JSVGCanvas`

Exceptions:

- 10.a The temporary XML, thus also the pre-processed XML document contains data that `JSVGCanvas` class cannot display
 1. An exception is thrown and logged
 2. An error message is displayed
 3. The visualization stops

Priority: Critical

Frequency of Occurrence: Frequent

6. SVG VISUALIZATION IN FRESNEL EDITOR

Use Case 3	Text augmentation
<i>Level:</i>	User-Goal
<i>Primary Actor:</i>	User, Fresnel Editor
<i>Goal in context:</i>	To show a full text of visualized resource, property or value
<i>Preconditions:</i>	The SVG visualization is displayed in Fresnel Editor
<i>Postconditions:</i>	A dialog window with a full text of visualized resource, property or value is displayed
<i>Scenario:</i>	<ol style="list-style-type: none">1. User clicks on a text on the displayed SVG image2. Fresnel Editor pops up a dialog window with the full text
<i>Priority:</i>	Moderate
<i>Frequency of Occurrence:</i>	occasional

6.2.1.3 State machine diagrams

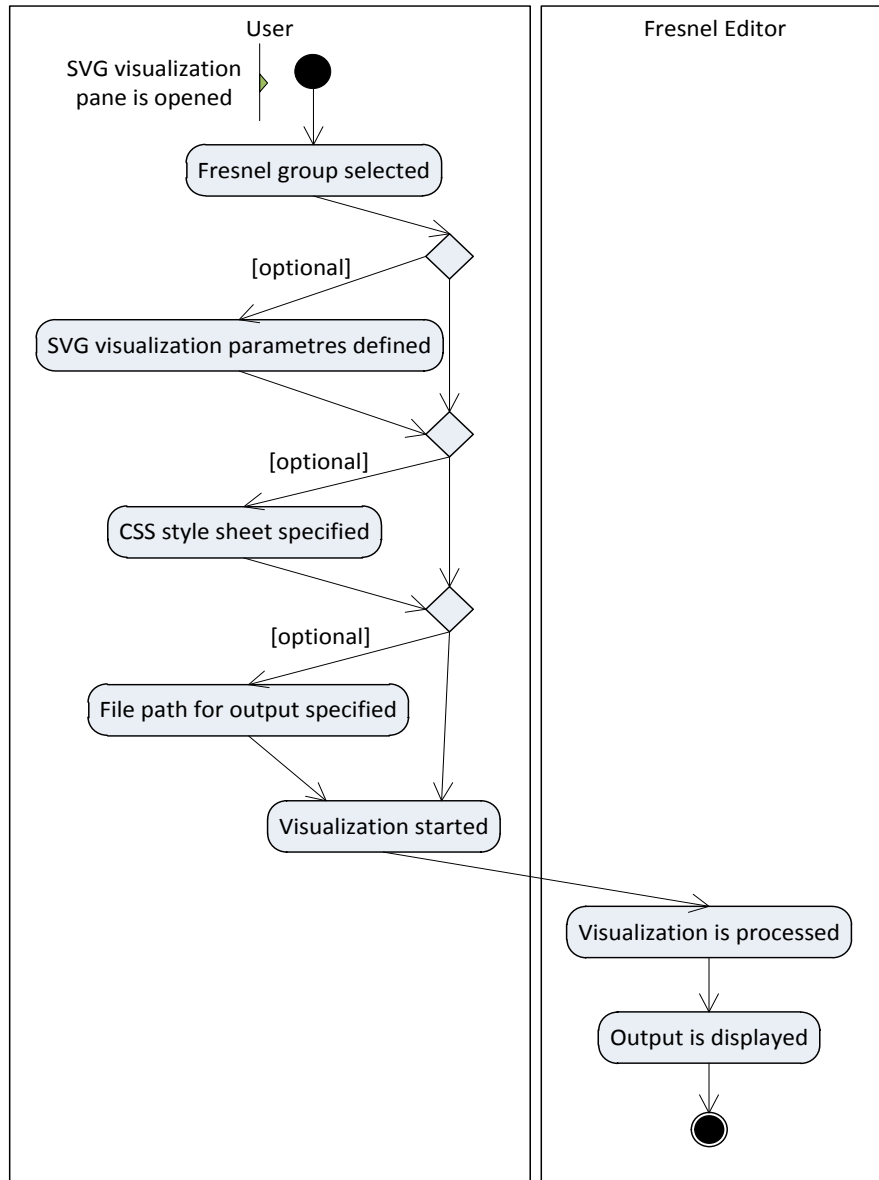


Figure 6.4: The overall SVG visualization scenario

6. SVG VISUALIZATION IN FRESNEL EDITOR

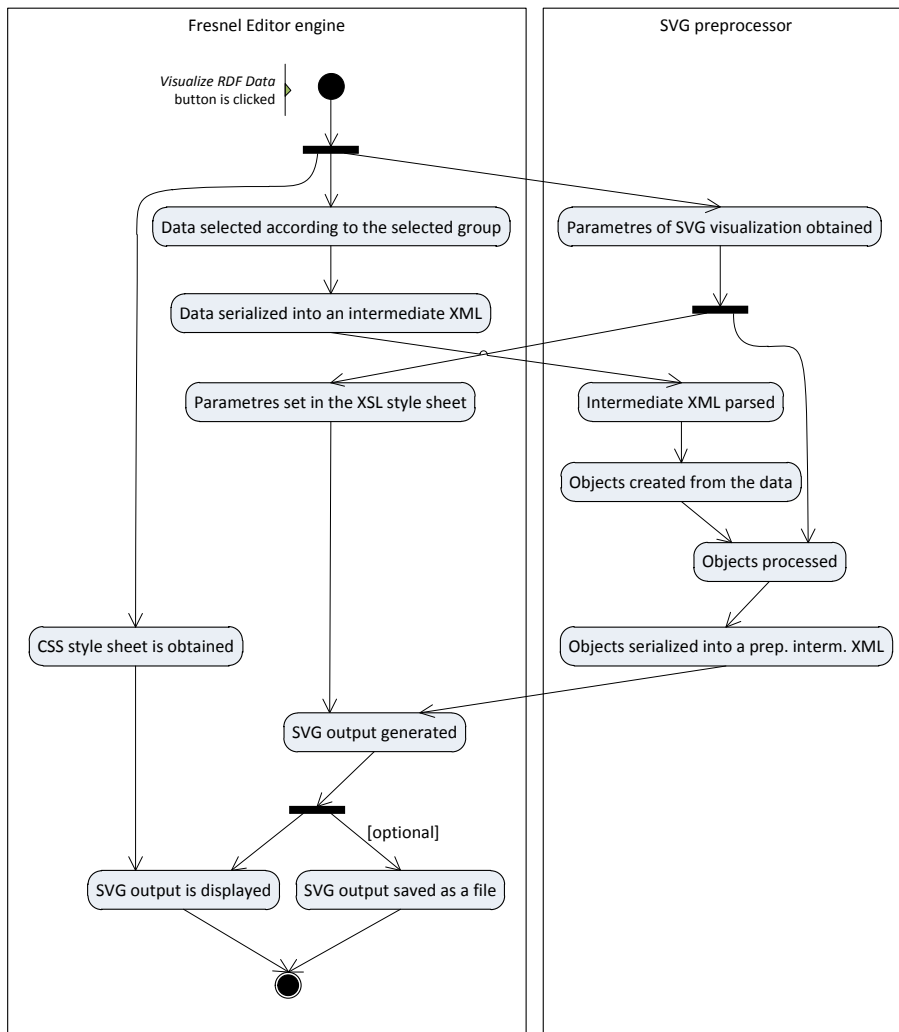


Figure 6.5: Inner states the SVG visualization scenario

6.2.1.4 Class diagram

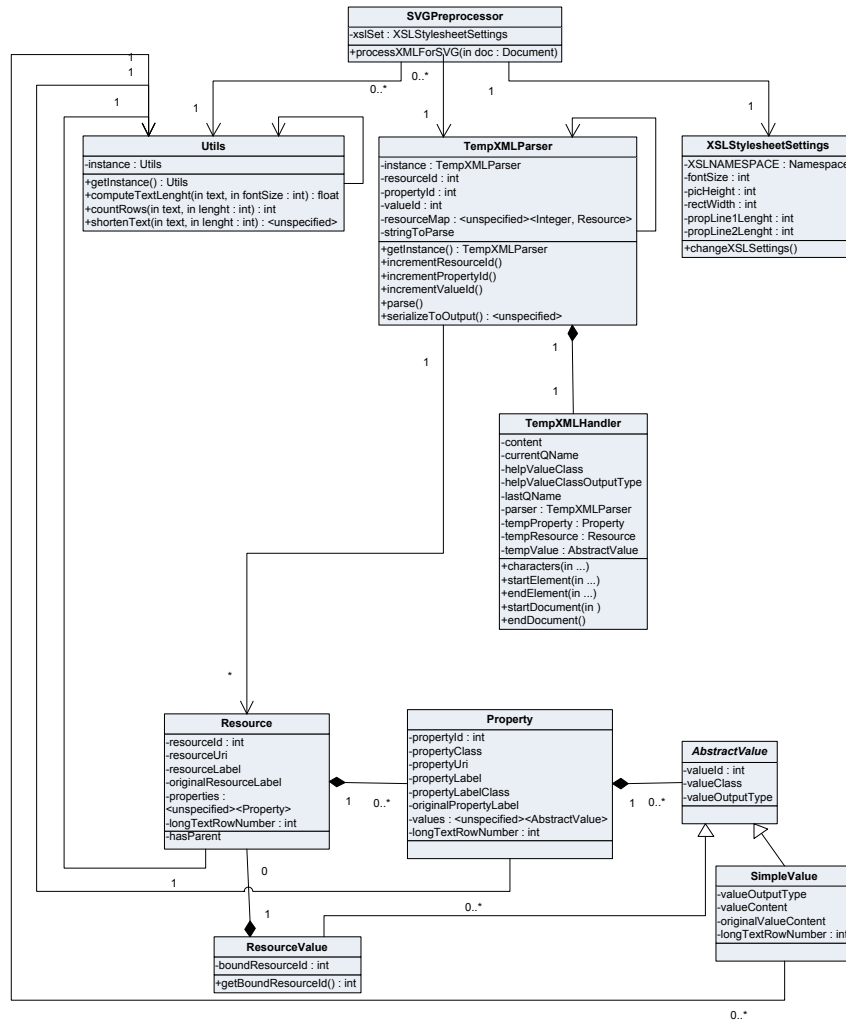


Figure 6.6: Simplified class diagram of the preprocessor

6.2.2 Implementation

The authors of Fresnel Editor expected that the intermediate XML file might be necessary to modify for a various reasons and prepared a method where the change should happen. However, to pass the user’s input from GUI of

6. SVG VISUALIZATION IN FRESNEL EDITOR

Fresnel Editor, a few minor changes had to be made in the original source code.

One visible change was made in the GUI – a SVG visualization parameters frame was added to the SVG visualization tab. After the visualization process is invoked, the parameters are checked and in case they are invalid, the user is prompted to correct them. When is everything all right, the parameters are passed along together with other concerning the visualization.

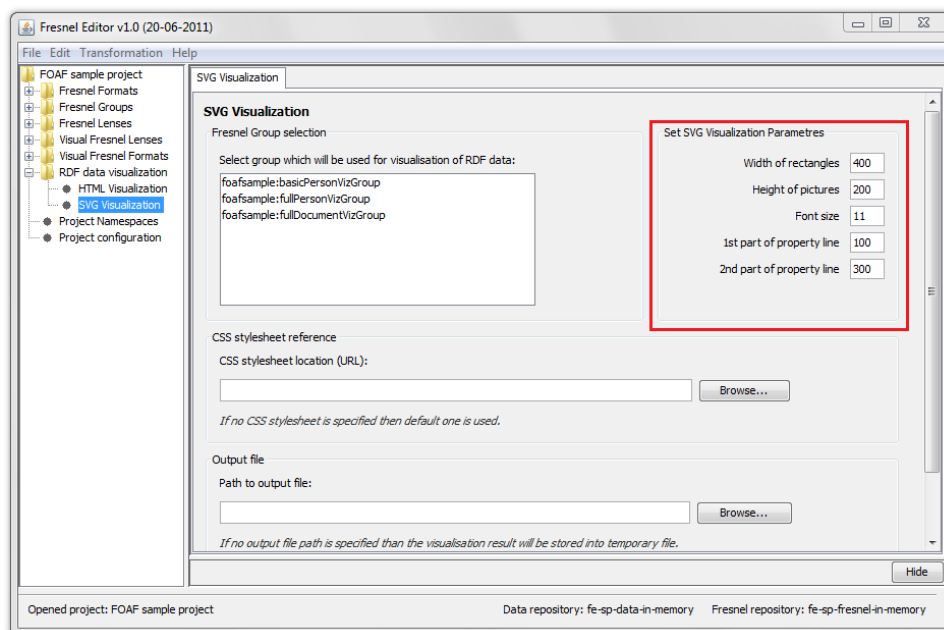


Figure 6.7: SVG visualization parameters panel was added to the SVG visualization tab

When the data to be visualized are selected and serialized into the intermediate XML document, the `modifyXMLdocument()` method is called and the `SVGPreprocessor` class is instantiated; consequently, instances of an XSL style sheet handler and an XML parser/handler are obtained. The parameters passed from the GUI are set to the XSL style sheet which is used later on. The temporary XML document is parsed and resources, their properties and values are represented by instances of appropriate classes.

To be able to compute the length of the text, it was necessary to initialize the SVG and CSS interfaces first. Secondly, simple SVG `<text>` elements with the text to be processed are temporarily created. Only then it is possible

to count the length of the texts and determine how many lines will the text need. In cases, where it is not possible to provide more than single-line visual element for the text, the text is shortened. In order not to lose the former information, an auxiliary attribute holding the copy of the original string is defined. The whole structure is then serialized back to the XML document.

Afterwards, the pre-processed XML is transformed according to the XSL style sheet into an SVG document which is passed to the component responsible for its rendering and displaying.

6.3 Displaying of the SVG Image

The displaying of the generated SVG image utilize Batik Swing component – the SVG rendering is handled by the `SvgShowJPanel` swing component that is attached to the SVG visualization tab of Fresnel Editor. Implementing such a component was originally part of the thesis project, none the less the component was created in paralel as a part of a course project at the Faculty of Informatics at Masaryk University, Brno. At the time the preprocessor has been integrated to the Fresnel Editor, the component for displaying SVG images was already integrated. Therefore I have decided not to integrate other, practically identical solution.

In the visualization component that I had prepared for integrating with Fresnel Editor was a method to display the whole text of a visualized resource, property or value. After a click on a text, a dialog window appears with the text. This feature was added to the `SvgShowJPanel` in Fresnel Editor.

The interaction with `SvgShowJPanel` is performed in the following manner: a general `EventListener` reacting on `onClick` event is attached to all text elements in rendered `SVGDocument`. The important part is that the attachment happens when `SvgShowJPanel` is informed that the GVT tree building is finished, which is performed by `GVTTreeBuilderEvent`. Otherwise the attachment would fail.

Originally, with the idea of creating the component for displaying the SVG content, we intended to provide users with an ability to change the CSS style sheet directly using the graphical interface in a similar way as for displaying the dialog window a with full text of a visualized entity. However, this task has not been fulfilled; problems occurred during the attempts to interact with CSS style sheet utilizing the Batik CSS interface. Unfortunately, the documentation of Batik SVG Toolkit is poor and provided little or no help during the development.

6. SVG VISUALIZATION IN FRESNEL EDITOR

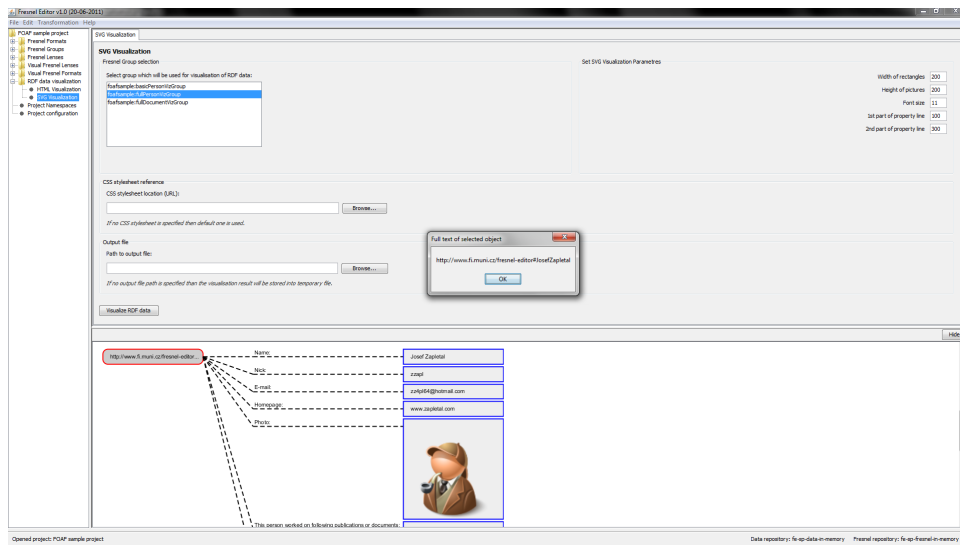


Figure 6.8: The content of the sample project visualized as a SVG image

Chapter 7

SVG Tool for Image Annotation

Earlier in the text were suggested two methods, how can be performed an SVG visualization of semantic data. In Fresnel Editor was used the XSL transformation; a detailed description of the method together with remarks about its implementation within the Fresnel Editor were provided in the previous chapter.

In the year 2011 I got the opportunity to use the knowledge gathered during the research for this thesis and develop a tool for medical images annotating and presenting semantic data. The important fact is that the tool basically uses DOM API to build SVG documents, which is the second possible approach for SVG visualization.

The project is a part of my cooperation with DFKI GmbH¹, which is the leading research institute working on innovative software technology in Germany. It is rated among the most recognized “Centers of Excellence”. [42] My position is a research assistant, working with Dr. Daniel Sonntag’s team on the RadSpeech project which is a part of Theseus-Medico project.

Theseus-Medico is funded by the Federal Ministry of Economics and Technology in Germany. Project’s aim is to facilitate the work of doctors and other healthcare workers by deriving information from image- and text-based findings, putting the relevant pieces together in an intelligent manner. [43] My task is to analyze the possibility to exploit and utilize the SVG format for annotation of medical images. Consequently, the annotation tool is to be developed and attached with an already existing visualization.

7.1 The RadSpeech Project

The RadSpeech project aims to design and implement a multimodal dialogue system for radiologists. The system should provide a dialogue-based semantic image retrieval, which should help in a computer aided diagnosis.

1. the abbreviation stands for Deutsches Forschungszentrum für Künstliche Intelligenz, literally translated The German Research Center for Artificial Intelligence

sis and the related decision making process. The knowledge gained from ontology repositories is used also for the complex natural language understanding and dialogue management process. Semantic information can also help to provide different views on the medical data. The RadSpeech system should eventually streamline the medical finding process and achieve more structured finding reports including semantic image annotations. [44]

RadSpeech allows clinicians to directly annotate images that comes from radiology, CT or MRI; the former approach was to capture the images, provide the content of the annotations (for example by recording on a tape recorder), then assign the annotations to the images. Given the fact that the amount of images is extensive, the task takes a non-trivial amount of time. Methods for creating annotations include also speech input (the annotation is transformed to text by speech-to-text analysis) or "intelligent" pen (handwriting is directly transformed into text using OCR techniques). The project provides also "traditional" user interfaces, in which can users explore patient data. A requirement from the clinicians was to have an ability to view the patient data in the context with other (similar) cases. This should help them with the diagnostic process and treatment decision. [44]

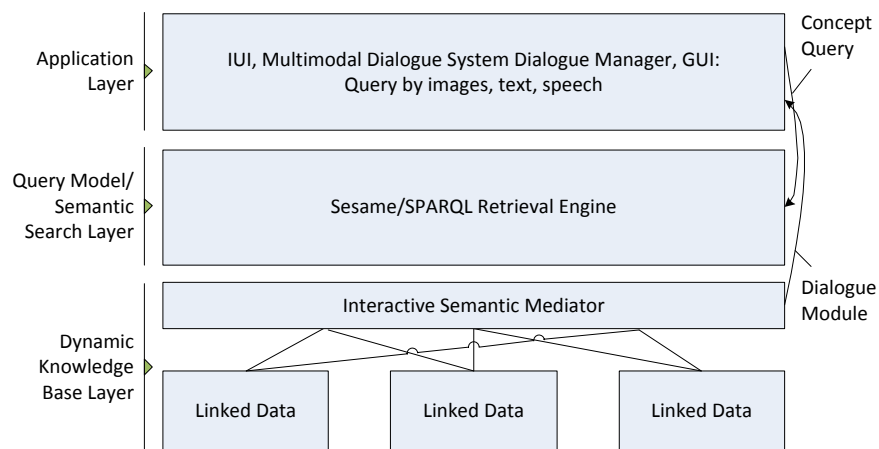


Figure 7.1: Three tier architecture of RadSpeech [45]

The RadSpeech project uses three tier architecture (see the figure 7.1) to query, mediate, retrieve and present the data to users. However, the background of the data selection is beyond the scope of this thesis.

MEDICO Patients | Trial Overview | Radlex | Details | MEDICO RadSpeech

5 searchresult filtered from 23 originally (Reset All Filters)

sorted by: [Linkedct-diseases-name](#), then by... + grouped as sorted

Linkedct-Result: Trial NCT01045928 (external)

Lenalidomide And Rituximab as Maintenance Therapy in Treating Patients With B-Cell Non-Hodgkin Lymphoma

Endpoint Classification: Safety/Efficacy Study, Intervention Model: Single Group Assignment, Masking: Open Label, Primary Purpose: Treatment (Interventional)

Diseases: Cutaneous B-Cell Non-Hodgkin Lymphoma and Adult Non-Hodgkin Lymphoma | Drugs: nucleic acid sequencing, polymorphism analysis, polymerase chain reaction, flow cytometry, laboratory biomarker analysis, lenalidomide, and rituximab

(December 2010) | Case Comprehensive Cancer Center

Linkedct-Result: Trial NCT00720135 (external)

Fusion Protein Cytokine Therapy After Rituximab in Treating Patients With B-Cell Non-Hodgkin Lymphoma

Endpoint Classification: Safety Study, Intervention Model: Single Group Assignment, Masking: Open Label, Primary Purpose: Treatment (Interventional)

Diseases: Cutaneous B-Cell Non-Hodgkin Lymphoma | Drugs: DL-Leu16-IL2 immunocytokine, immunohistochemistry staining method, enzyme-linked immunosorbent assay, reverse transcriptase-polymerase chain reaction, pharmacological study, rituximab, flow cytometry, and laboratory biomarker analysis

(February 2011) | City of Hope MedicalCenter and City of Hope Medical Center

Linkedct-Result: Trial NCT01110135 (external)

Bendamustine Hydrochloride, Dexamethasone, and Filgrastim For Peripheral Blood Stem Cell Mobilization in Treating Patients With Refractory or Recurrent Lymphoma or Multiple Myeloma

Endpoint Classification: Efficacy Study, Intervention Model: Single Group Assignment, Masking: Open Label, Primary Purpose: Treatment (Interventional)

Diseases: Cutaneous B-Cell Non-Hodgkin Lymphoma | Drugs: leukapheresis, bendamustine hydrochloride, dexamethasone, filgrastim, flow cytometry, and laboratory biomarker analysis

(October 2010) | Fred Hutchinson Cancer Research Center

Linkedct-Result: Trial NCT00601718 (external)

Figure 7.2: An example of the faceted browsing (with already applied filters)

7.1.1 Exhibit-based Facetted Visualization

The visualization part of RadSpeech project is a browser application providing the clinicians with a set of inter-related views on the data. The views can be refined by using *lenses* that contain sets of data and *facets* that constrain the number of the datasets (see the figure 7.2).

The visualization is based on the Exhibit API² (see the figure 7.3) which provides JavaScript/ Ajax³ API for queries, widgets for views, facets and lenses. The Exhibit draws the data collection from JSON formatted data,

2. Exhibit is a framework for data-rich interactive web pages developed within the SIMILE project [46]
3. Asynchronous JavaScript and XML

which are generated by the interactive semantic mediator (see the figure 7.1) from results of executed SPARQL queries.

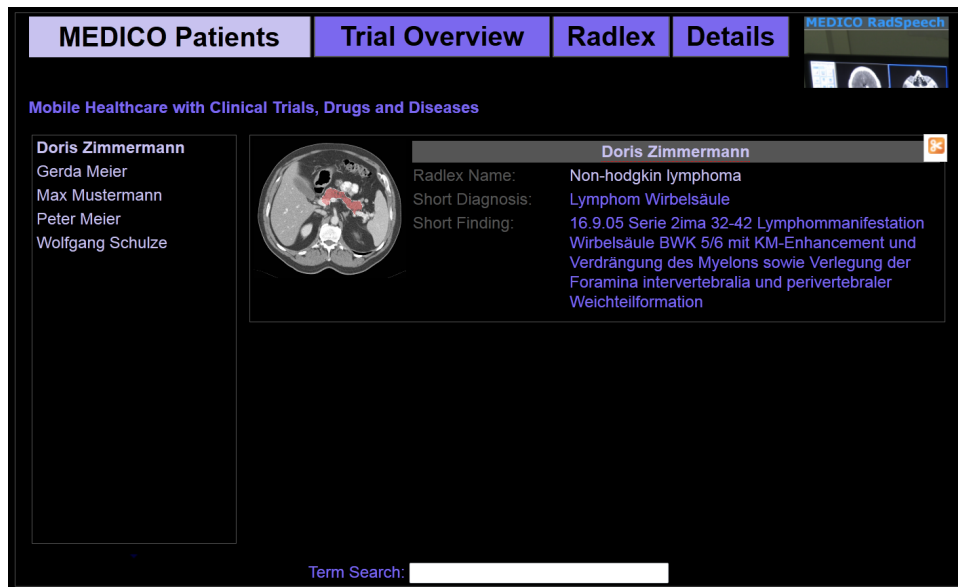


Figure 7.3: Screenshot of the of the Exhibit visualization

7.1.2 SVG-based Visualization for Image Annotation

Through the Exhibit visualization we got to the visualization in SVG that might become a part of the RadSpeech project. Initially it was not absolutely clear where might be SVG features exploited. Several models were created, each time with more interactive interface. Currently, the main goal is to provide a pen-driven annotating tool for the clinicians, where would be possible to draw directly over an image (for example x-ray image) and accompany the drawing with some note. The annotations would be immediately added to the rest of the medical findings in the repository. It would be possible to retrieve the annotations (as well as the other data) for editing or consulting purposes.

At first, the visualization was developed above static data – a set of data that was returned as a result of a SPARQL query on the project-related SPARQL endpoints. The visual style is analogous to the style used for the Exhibit visualization (see the figures 7.3 and 7.4) as the two visualizations are to be connected in later phases of the project. The whole source code of

the scripts was written in JavaScript programming language; to provide the required functionality, a few open-source libraries were used:

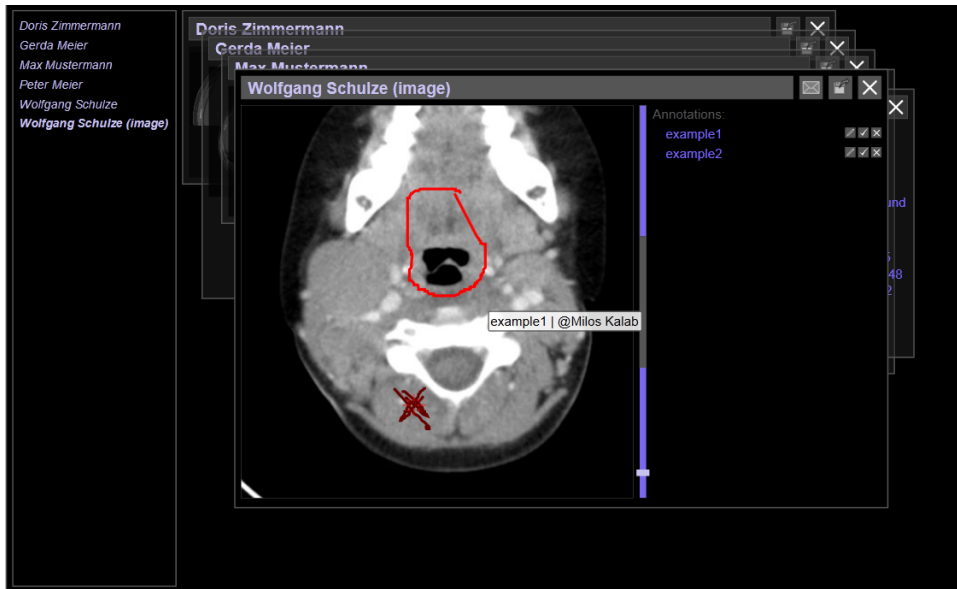


Figure 7.4: Screenshot of the of the SVG visualization

jQuery SVG: a JavaScript library for manipulating with SVG DOM (it is very similar to the DOM API module of Batik – see chapter 4). The library is available on <http://keith-wood.name/svg.html>

Draggable SVG: a library that enable SVG elements to be draggable using the mouse. To allow an object to be movable, the desired element has to have defined a `drag:enable="true"` attribute. The rest (i.e. the changing of transformation attribute) is handled by the Draggable SVG library. The author is Jeff Schiller and the source code is available on <http://www.codedread.com/dragsvg.js>.

SVG slider object: In the early development phase, zooming of the annotated image was performed after a mouse-wheel-rotation event occurred. Due to a requirement, that the visualization should work on hand-held appliances (tablets, smart-phones), zooming of the picture has to be controlled in a different manner. The library has provided a method to add/remove a slider that is responsive to the user commands and through which the user can adjust the zoom

of the image. The project Carto.net was found during the research for examples of interactive SVG documents. Available online on <http://www.carto.net/papers/svg/gui/slider/>

SVG textbox object: SVG support of text is not on the level of for example (X)HTML. The lack of object that would allow inputting text directly in the SVG document (in (X)HTML for example `<textarea>` or `<input type="text" . . .>`) led to acquiring the library simulating the inputbox in SVG. Similarly as the SVG Slider library, the SVG textbox offers simple way to create and remove an inputbox and read the input. The library comes from Carto.net project and is available on <http://www.carto.net/papers/svg/gui/textbox/>

bundled scripts: for proper work of the scripts originated from Carto.net, several additional libraries are required: `helper_functions.js`, `mapApp.js` and `timer.js`. They are accessible from the previously noted web pages.

After a few prototypes were evaluated and the scenario for the use of the tool crystallized, the data source was changed. The visualization was published online (for the purpose of the ongoing development) and a subset of the data used in Exhibit visualization has been utilized. Unlike the first phase, this time the data are accessed over GET requests and new/changed annotations are stored using POST requests and functions available on the Medico server. Both GET and POST are using AJAX's asynchronous processing. The basic scenario of the tool can be shown on the figure 7.5.

In the second phase, the whole library was rewritten using jQuery framework⁴; the code was shortened considerably and some functionality became easier to provide. This change was possible as the visualization were aimed to run in web browsers; the Java based viewer (Batik Squiggle) was rejected for its possible incompatibility with mobile devices.

4. an open source library for simplified event handling, document traversing and Ajax interactions. Available on <http://jquery.com/>

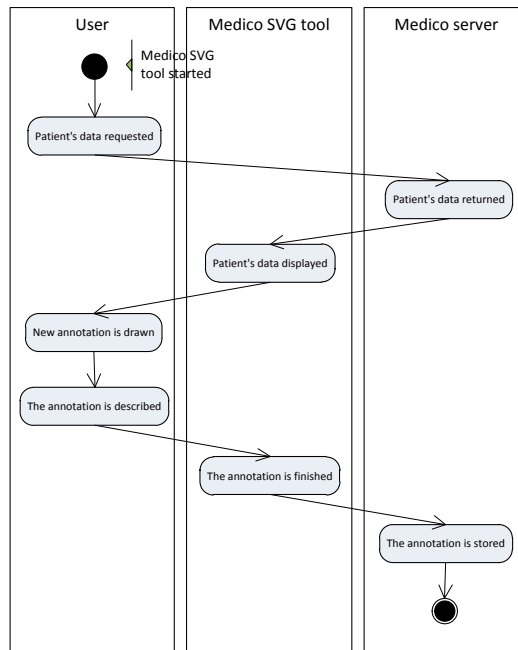


Figure 7.5: State machine diagram of creation of an annotation

In one of the prototypes was explored the possibility of sending an email directly from the SVG visualization. The clinicians would be able to send the annotated image to their colleagues for consultation directly after the annotation was marked. Nevertheless, an email message can contain either plain text (some clients support also rich text format) or HTML. Despite attempts to inline SVG content into a message in HTML format, email clients seem to sanitize it as an unknown and potentially dangerous code. Therefore, the email consists of an URI of the annotated image (or the set of related images), which allows the recipient to open the desired image and its annotations in a new session of the SVG visualization. The desired scenario is shown in the figure 7.6

The SVG visualization can fulfill another requirement – comparison of two or more related images. It may serve for such a task as each patient's image is displayed in a separate panel inside the SVG image. However, this use case is under consideration and the visualization would need some minor alternations. The project is still under development and the final deliverable might differ from the current version considerably.

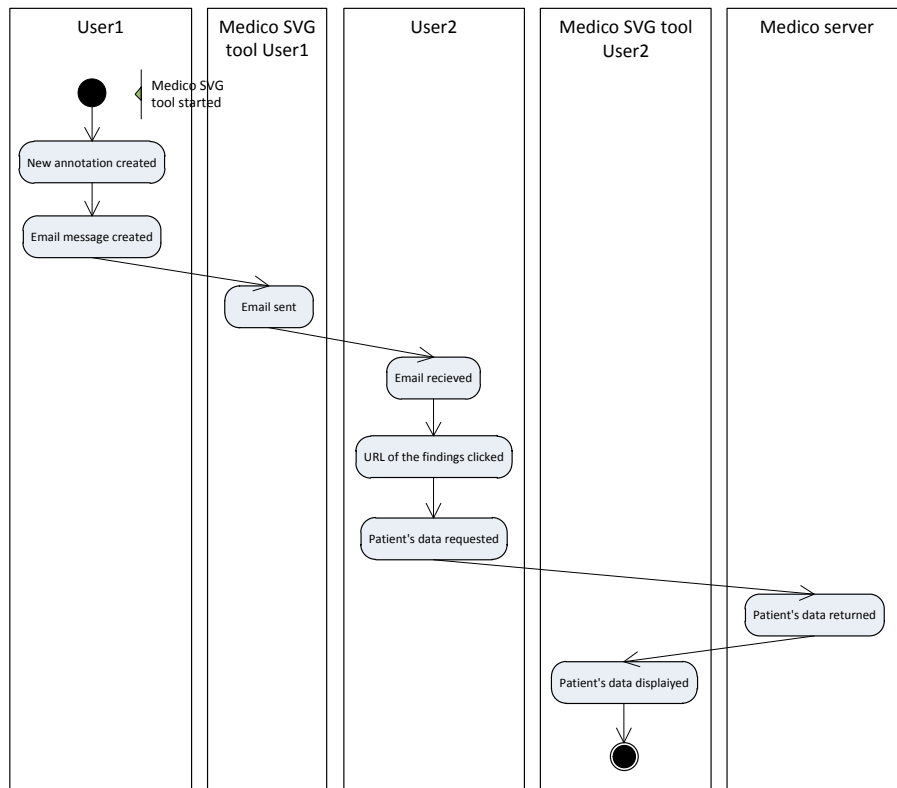


Figure 7.6: State machine diagram of sending an annotation via email

7.1.3 Benefits And Constraints of the SVG Tool

There are several reasons why the medical image data were put to the SVG format. Probably the major benefit is that every drawn annotation is a separate object. If HTML5 `<canvas>` element would be utilized to allow the user to draw on medical images, the annotations would be eventually flattened into an image, indistinguishable one from another (unless an extra effort to analyze the image content would be made). In SVG, each annotation is represented by one or more `<polyline>` elements and a text, that are bound together by an identifier of the annotation. The text can be easily versioned, extended with a contextual information (author, timestamp, version number). The next advantage is the scalability of SVG. However, the medical image is in the raster format so the scalability is limited by quality

of the annotated image, which can be provided in very high definition resolution.

As for the disadvantages, the major one is listed in the table 2.1 – the rendering slows down when an extensive amount of drawings is attached to an image.

In the list of challenges that are to be faced in the future is providing of a support of touch devices – the user's touch interactions are recorded in a different matter than in the case of mouse handling and the pointing is not as precise.

Conclusion

The main topic of the thesis is a SVG-based knowledge visualization. The assignment asked to analyze the relevant technology and suggest a method for visual representation of semantic data. Drawing from the analysis, the method was implemented as an extension of Fresnel Editor – an authoring tool for semantic data utilizing Fresnel language.

The analysis revealed several interesting points. Firstly, let us discuss the SVG standard because providing knowledge to users in a visual manner was the primary goal. SVG is a very powerful technology that can be utilized not only as a format for graphical objects but also as a platform for sophisticated application development. Nevertheless, in my opinion, SVG is generally rather underrated and consequently, the public awareness of its features and capabilities is limited. SVG also suffers from a problematic support in web browsers and image viewers. Because a research and a development of technologies is influenced by their acceptance, it seems that the innovation of SVG has slowed down considerably (SVG 1.2 is still a working draft despite the first version was published in the year 2002). In case of this thesis, one of the problem lies in a representation of long text, which is solved well enough in SVG 1.2, but the feature is badly supported by other tools, viewers and browsers. The lack of new features or changes in the existing ones according to the needs of users may cause an SVG fall-off in the future. The chances are that the situation around SVG will change because of growing use of HTML5, which supports SVG better.

Secondly, there is the Fresnel language specification that was introduced by W3C; the language should lead to better interoperability between different tools/systems for semantic data visualization. Although the standard was published in the year 2005, until now there is only a few tools able to use the language. Furthermore the majority of them seems to be discontinued. One of the tools using Fresnel is Fresnel Editor which has been extended and maintained at the Faculty of Informatics at Masaryk University, Brno. In this thesis was provided a summary about this authoring tool; together with Fresnel Editor – Knowledge Visualization [27], these are probably the only articles about Fresnel Editor written in English. Regarding the tool itself,

after the first version was delivered, several extensions were provided. The GUI was simplified to broaden the base of possible users and several output formats for the visualization of the data were added (one of them is SVG visualization created in this thesis).

The major challenges of the SVG visualization were the positioning of visual elements in the output document and the representation of long texts. Generally there are two approaches to generate an SVG document from the data:

- The data are serialized into a specific XML format and then an XSL transformation is used to transform the source XML to the result SVG image.
- The data are processed by the visualizer and the result SVG image is built using the DOM API according to the processed data.

The challenges are tackled in a similar manner in the both approaches; the positioning of visual elements is computed on the basis of previously generated elements. The computation of the coordinates is rather easy using the DOM API (we can simply hold the coordinates and the length of the last created element in variables), whereas in the transformation, many attributes has to be computed prior to the transformation itself. Furthermore the transformation cannot dynamically react on anomalies in the source XML serialization (actually they can be handled, but the code of the transformation would grown excessively). As for the long texts, in the transformation approach, template rules has to be specified to cover the possible cases; none the less it is impossible to cover all the cases with general data as we do not know whether the text may contain for example a single word/string or a whole text of a book. On the other hand, using DOM API allows us to compute the place necessary for a text element according to the lenght of the visualized string. In both approaches it is necessary to preprocess the texts elements either to know the height of the required place or to provide the transformation with an information which template rule should be used. Both approaches can tackle the challenges stated above, however, the transformation approach is not completely universal.

For implementation of the visualization module in Fresnel Editor, the transformation approach was chosen drawing from the original visualization process of the tool. Its design and development were straightforward thanks the architecture of Fresnel Editor. Its authors prepared the tool well for possible integration of extensions. The visualization process was extended with a preprocessor which adds a control information about lenghts of visualized

texts. In case a user wants to visualize a set of data to the SVG format, the data are selected in an ordinary manner, then they are preprocessed, transformed into the SVG format and eventually rendered by Fresnel Editor. The user is also allowed to specify some parameters of the SVG visualization and interact with the output.

Even though some parameters can be adjusted by the user, the current visualization have a firm structure of a tree built from a left side. This is the prize for visualizing of general data which had to be paid. If the visualization would be applied on more specific data, the output might be adjusted according to their character. Such an example can be observed with the second project included in this thesis.

Paralell to the work on this thesis I was given an opportunity to contribute on a project with a similar goal as the thesis. My supervisor, Dr. Daniel Sonntag saw prospects in exploiting SVG to create a tool for annotation of medical images – Medico SVG tool. In principle, the Medico SVG tool is very similar to the Fresnel Editor visualization process: both draws the visualization on a result set obtained from an RDF repository. The constraints are also the same, nevertheless, the Medico SVG tool is aimed rather on a direct interaction of a user with the visualized data. The main use case is that the user draws annotations over a medical image and accompany the drawings with text. Such annotations are afterwards stored for further use. One of the pursued requirements is to use the tool from web browsers; more specifically, beside the desktop browsers, it should work on mobile devices as well. For this reason, the tool is based on SVG and JavaScript – SVG provides the visual appearance and JavaScript provides the capability to generate the visualization using DOM API, the logic of the tool and its interactivity. This project is based on the second visualization technique, hence complementing the Fresnel Editor visualization.

The fact that the Medico SVG tool should be used by doctors to help them with diagnosis shows the potential of SVG in praxis. What is more, it stresses the value of the semantic data visualization, which is still domain where a lot can be improved. There lies the possibility of a future research: utilization of existing technologies aimed on an interoperability (for example Fresnel language) to create a visualization framework that can process both the general data and domain specific data in an intelligent manner; the visualization (for example to the SVG format) should take into account the nature of the data, the requirements for aesthetics and it should be customizable according to the needs of a user.

Bibliography

- [1] Jon Bosak. The birth of xml: A personal recollection. *Sun Developer Network*, September 2001 [cit. 2012-03-06]. http://java.sun.com/xml/birth_of_xml.html.
- [2] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). Technical report, W3C, November 2008 [cit. 2012-03-06]. <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (xml) 1.1 (second edition). Technical report, W3C, August 2006 [cit. 2012-03-06]. <http://www.w3.org/TR/2006/REC-xml11-20060816/>.
- [4] Leslie F. Sikos. *Web Standards – Mastering HTML5, CSS3, and XML*. Apress, 2011. ISBN: 978-1-4302404-2-6.
- [5] James Clark. Xsl transformations (xslt). Technical report, W3C, November 1999 [cit. 2012-03-10]. <http://www.w3.org/TR/1999/REC-xslt-19991116>.
- [6] Michael Kay. Xsl transformations (xslt) version 2.0. Technical report, W3C, January 2007 [cit. 2012-03-10]. <http://www.w3.org/TR/2007/REC-xslt20-20070123/>.
- [7] Michael Kay. What kind of language is xslt? *IBM Developer Works*, February 2001 [cit. 2012-03-10]. <http://www.ibm.com/developerworks/library/x-xslt/>.
- [8] Benoit Marchal. How an xslt processor works. *IBM Developer Works*, March 2004 [cit. 2012-03-10]. <http://www.ibm.com/developerworks/xml/library/x-xslang/>.
- [9] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). Technical report, W3C, February

- 2004 [cit. 2012-05-21]. <http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/>.
- [10] Miloš Kaláb. Sémantická interoperabilita v rámci iniciativ eu, 2009 [cit. 2012-05-21].
- [11] Christian Bizer, Ryan Lee, and Emmanuel Pietriga. Fresnel – a browser-independent presentation vocabulary for rdf. *ISWC2006 – 5th International Semantic Web Conference*, November 2006 [cit. 2012-03-12]. <http://hal.inria.fr/docs/00/05/61/32/PDF/fresnel.pdf>.
- [12] Emmanuel Pietriga. Fresnel selector language for rdf (fsl). Technical report, W3C, November 2005 [cit. 2012-03-12]. <http://www.w3.org/2005/04/fresnel-info/fsl-20050726/>.
- [13] Christian Bizer, Ryan Lee, and Emmanuel Pietriga. Fresnel – display vocabulary for rdf. Technical report, W3C, June 2005 [cit. 2012-03-12]. <http://www.w3.org/2005/04/fresnel-info/manual-20050726/>.
- [14] Miroslav Warchil. Vizualizace dat pomocí stylového jazyka w3c fresnel. Master thesis, Masaryk university, Brno, Fakulty of Informatics, 2010.
- [15] Chris Lilley and Doug Schepers. Secret origin of svg. W3C, November 2010 [cit. 2012-03-14]. http://www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG.
- [16] Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, and Jonathan Watt. Scalable vector graphics (svg) 1.1 (second edition). Technical report, W3C, August 2011 [cit. 2012-03-15]. <http://www.w3.org/TR/2011/REC-SVG11-20110816/>.
- [17] Dean Jackson and Craig Northway. Scalable vector graphics (svg) full 1.2 specification. Technical report, W3C, April 2005 [cit. 2012-03-15]. <http://www.w3.org/TR/2005/WD-SVG12-20050413/>.
- [18] Tolga Capin. Mobile svg profiles: Svg tiny and svg basic. Technical report, W3C, January 2003 [cit. 2012-03-15]. <http://www.w3.org/TR/2003/REC-SVGMobile-20030114/>.

- [19] Yug. <http://commons.wikimedia.org>, October 2006 [cit. 2012-03-16]. http://commons.wikimedia.org/wiki/File:Bitmap_VS_SVG.svg.
- [20] Mihai Sukan. Svc or canvas? choosing between the two. *DEV.OPERA*, February 2010 [cit. 2012-03-17]. <http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two/>.
- [21] Ian Hickson. Html5. Technical report, W3C, May 2011 [cit. 2012-03-17]. <http://www.w3.org/TR/2011/WD-html5-20110525/>.
- [22] Alexis Deveria (Fyrd). <http://caniuse.com>, March 2012 [cit. 2012-03-17]. <http://caniuse.com/#cats=HTML5>.
- [23] Alexis Deveria (Fyrd). <http://caniuse.com>, March 2012 [cit. 2012-03-17]. <http://caniuse.com/#cats=SVG&statuses=rec>.
- [24] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. Technical report, W3C, May 1998 [cit. 2012-03-21]. <http://www.w3.org/TR/2008/REC-CSS2-20080411/>.
- [25] Tom Pixley. Document object model (dom) level 2 events specification. Technical report, W3C, November 2000 [cit. 2012-03-22]. <http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/>.
- [26] Alexander Kolesnikov. *Java Drawing with Apache Batik – A Tutorial*. Brainy Software Corp., 2007. ISBN: 978-0-9752128-9-9.
- [27] Masaryk University, Brno, Czech Republic. *Fresnel Editor – knowledge visualization*, January 2010 [cit. 2012-03-25].
- [28] Sesame framework for processing rdf data, 2012 [cit. 2012-03-28]. <http://www.openrdf.org/>.
- [29] Jfresnel java library, 2010 [cit. 2012-03-28]. <http://jfresnel.gforge.inria.fr/>.
- [30] Igor Zemský. Vizualizace rdf dat. Master thesis, Masaryk university, Brno, Fakulty of Informatics, 2009 [cit. 2012-03-29].
- [31] Ján Horváth. Vizuální editor formátů standardu fresnel, 2010 [cit. 2012-03-29].
- [32] Silvie Petrová. Tvorba šablon pro w3c fresnel, 2010 [cit. 2012-03-29].

- [33] Java web browser effort (lobo), 2009 [cit. 2012-03-29]. <http://lobobrowser.org/>.
- [34] Fresnel editor, 2011 [cit. 2012-03-29]. <http://sourceforge.net/projects/fresnel-editor/>.
- [35] Batik svg toolkit, 2010 [cit. 2012-03-30]. <http://xmlgraphics.apache.org/batik/>.
- [36] Cameron McCormack. Using the apache batik toolkit for client- and server-side svg processing, September 2007 [cit. 2012-04-02]. <http://mcc.id.au/2007/09/batik-course/>.
- [37] Xml graphics - batik wiki, 2011 [cit. 2012-04-02]. <http://wiki.apache.org/xmlgraphics-batik/>.
- [38] Apache batik api specification 1.8pre, 2009 [cit. 2012-04-02]. <http://xmlgraphics.apache.org/batik/javadoc/>.
- [39] Arnaud Le Hors, Philippe Le Hégarret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (dom) level 2 core specification. Technical report, W3C, November 2000 [cit. 2012-04-02]. <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>.
- [40] Keith Wood. jquery svg, 2011 [cit. 2012-04-10]. <http://keith-wood.name/svg.html>.
- [41] Dmitry Baranovskiy. Raphaël-javascript library, 2008 [cit. 2012-04-10]. <http://raphaeljs.com/>.
- [42] Deutsche forschungszentrum für künstliche intelligenz gmbh, [cit. 2012-04-11]. <http://www.dfki.de>.
- [43] Radspeech (project website), 2012 [cit. 2012-05-24]. <http://www.dfki.de/RadSpeech/>.
- [44] Daniel Sonntag, Christian Schulz, Christian Reuschling, and Luis Galarraga. Radspeech's mobile dialogue system for radiologists, February 2012 [cit. 2012-04-11]. <http://www.dfki.de/RadSpeech/iui-2012-preprint.pdf>.
- [45] Daniel Sonntag. Theseus ctc-wp4.4 interactive semantic mediation – extended linked data access and faceted browsing. Technical report,

7. SVG TOOL FOR IMAGE ANNOTATION

DFKI GmbH., October 2010 [cit. 2012-04-11]. http://www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG.

[46] Exhibit – publishing framework for data-rich interactive web pages, 2009 [cit. 2012-04-12]. <http://www.simile-widgets.org/exhibit/>.

Appendices

Appendix A

Content of the Attached CD

- dp.pdf
- dp.tex
- dp.bib
- pictures – a folder containing all the images included in the thesis
- medico_svg (data from JSON) – a folder containing the a current version of the Medico SVG tool that obtains the data dynamically from the JSON file
- medico_svg (hardcoded SVG) – a folder containing the an earlier version of the Medico SVG tool with hardcoded data in the SVG document
- transformation – a folder with an example of the XSL transformation and sample data
- fresnel editor – a folder containing the Fresnel Editor with the implemented SVG visualization