A feasibility study of building Set-top box user interfaces using Scalable Vector Graphics
A feasibility study of building Set-top box user interfaces using Scalable Vector Graphics

Examensarbete utfört i Informationskodning vid Tekniska högskolan i Linköping av

Fredrik Vinkvist

LITH-ISY-EX--08/4011--SE

Handledare: Niels Bosma
Motorola, Inc

Examinator: Ingemar Ragnemalm
isy, Linköpings universitet

Linköping, 4 June, 2008
A feasibility study of building Set-top box user interfaces using Scalable Vector Graphics

Due to the cost-sensitive market of IPTV STBs the preferred platform to develop the user interface is the HTML browser as it allows for fast development times and low costs. As a W3C standard it also offers high portability and hardware abstraction making it easy to use more than one STB vendor. The cons of HTML based GUIs are low performance and lacklustre graphics.

This thesis aims to find out if SVG can be used to achieve rich, scalable and animated graphics with high performance and still keep the attractive characteristics of HTML.

To do this much effort was put into identifying the strengths and weaknesses of SVG. The lessons learned resulted in an SVG AJAX framework called TOIXSVG making it possible to develop SVG GUIs in the same manner as modern Rich Internet Applications, enabling component reuse to make sure development time scales preferably with the scope and complexity of the user interface. Along with the framework several new widgets had to be developed to achieve the targeted functionality. As a proof of concept a mock-up GUI was created with the framework and widgets.
Abstract

An IPTV Set-top box enables the possibility of doing much more than decoding television content. Its Ethernet interface gives it the same possibilities to communicate with the outside world as any network device. This enables a wide range of services from internet radio to acting as a digital media receiver in your home network. These highly interactive services increase the demands for responsive and visually attractive user interfaces.

Due to the cost-sensitive market of IPTV STBs the preferred platform to develop the user interface is the HTML browser as it allows for fast development times and low costs. As a W3C standard it also offers high portability and hardware abstraction making it easy to use more than one STB vendor. The cons of HTML based GUIs are low performance and lacklustre graphics.

This thesis aims to find out if SVG can be used to achieve rich, scalable and animated graphics with high performance and still keep the attractive characteristics of HTML.

To do this much effort was put into identifying the strengths and weaknesses of SVG. The lessons learned resulted in an SVG AJAX framework called TOIXSVG making it possible to develop SVG GUIs in the same manner as modern Rich Internet Applications, enabling component reuse to make sure development time scales preferably with the scope and complexity of the user interface. Along with the framework several new widgets had to be developed to achieve the targeted functionality. As a proof of concept a mock-up GUI was created with the framework and widgets.
First of all I want to thank my examiner Ingemar Ragnemalm and Motorola for giving me the chance to pursue this thesis. Something I’ve valued alot during this project is the continious input from my supervisor Niels Bosma. I also would like to thank my family for their support, and Sara Bagger for putting up with me during all hard days’ nights when I’ve been working like a dog.
Contents

1 Introduction .......................................................... 1
  1.1 Background .......................................................... 1
  1.2 Problem description ................................................. 1
  1.3 Purpose .............................................................. 2
  1.4 Delimitations ......................................................... 2
  1.5 Structure ............................................................ 2

2 IP-STB Environment ................................................. 5
  2.1 IP-STB .............................................................. 5

3 SVG ................................................................. 7
  3.1 History ............................................................. 7
  3.2 Rendering .......................................................... 7
  3.3 Document structure ................................................ 8
  3.4 Graphic primitives ................................................ 8
    3.4.1 Vector graphics ............................................... 8
    3.4.2 Text support .................................................. 9
    3.4.3 Raster images ............................................... 10
  3.5 Useful concepts ................................................... 10
    3.5.1 Transformations ............................................... 10
    3.5.2 Grouping ...................................................... 10
    3.5.3 Use ............................................................ 11
    3.5.4 Defined graphics ............................................. 11
    3.5.5 Inheritance of painting properties ......................... 12
  3.6 Filter effects ...................................................... 12
  3.7 SVG DOM .......................................................... 12
  3.8 SVG uDOM ........................................................ 13
  3.9 Dynamic SVG ...................................................... 13
  3.10 OpenVG ............................................................ 14
  3.11 Graphic authoring tools ......................................... 14
    3.11.1 Adobe Illustrator ............................................ 15
    3.11.2 Inkscape ...................................................... 15
  3.12 Summary .......................................................... 15
## 10 Implementation

10.1 Overview ................................................. 51
  10.1.1 Widgets ............................................. 51
  10.1.2 Packets and classes ................................. 52
10.2 The layout manager ........................................ 54
  10.2.1 SvgWidget ............................................ 54
  10.2.2 The bounding box ................................... 54
  10.2.3 Transforming widgets ................................. 54
  10.2.4 Aligning widgets ..................................... 55
  10.2.5 The widget tree ...................................... 56
  10.2.6 Algorithm ............................................ 56
10.3 Additional layout functionality ............................ 58
  10.3.1 Update with constraints .............................. 58
  10.3.2 Transforming widgets ................................. 59
10.4 Artistic Resizing .......................................... 60
10.5 Mockup GUI .............................................. 61
  10.5.1 Sun selection ......................................... 62
  10.5.2 Flash selection ....................................... 62
  10.5.3 Videotape selection .................................. 63
  10.5.4 TV selection ......................................... 63
10.6 Summary ................................................... 64

## 11 Evaluation

11.1 Conclusions .............................................. 65
  11.1.1 TOIXSVG requirements ............................... 65
  11.1.2 Rich graphics ......................................... 65
  11.1.3 Browser environment characteristics .................. 66
  11.1.4 Performance .......................................... 66
  11.1.5 Scalability ............................................ 66
  11.1.6 Dynamic content ...................................... 67
  11.1.7 Text support .......................................... 67
  11.1.8 Authoring ............................................. 67
  11.1.9 Preferred version ..................................... 67
  11.1.10 Artistic Resizing .................................... 67
11.2 Future developments ....................................... 68
  11.2.1 Layout ................................................. 68
  11.2.2 Skinning .............................................. 68

Bibliography ................................................. 69
Chapter 1

Introduction

1.1 Background

A set-top box (STB) is a small computer specialized at decoding video data streams (typically TV channels) and presenting the content on some kind of monitor. An IP-STB is a set-top box that receives this information over a high speed IP-based network rather than ordinary methods like terrestrial broadcasting and satellite TV. Being connected via a two-way connection allows additional services like web surfing, internet radio, video on demand, games etc. which encourages a high level of user interaction.

1.2 Problem description

The limited hardware of STBs introduces conflicts between performance, complexity and the richness of graphics when developing Graphic User Interfaces (GUIs). The limitations promote the use of lower-level languages like C to keep performance up and the memory footprint low. But this is often not a viable solution due the time and costs required to develop GUIs this way. The time factor associated also limits the possibilities to do rapid development of new multimedia services that the providers can offer their customers. Content providers and operators also prefer standard based solutions which abstract the hardware and ensures high portability.

To cater ease of development and portability, an HTML browser environment is used today. This allows the content-creators to create user interfaces with web-based techniques, combining ECMAScript and HTML and thereby significantly cutting development costs. The downside is worse performance and limiting the designers to simple HTML based graphics. As it builds on raster image techniques, the user interface has to be adapted to each resolution it is rendered in. A problem highlighted by the current SD-HD transition.
The goals are:

- Rich graphics.
- Keep the positive time and cost characteristics available in a browser environment.
- Good performance.
- Less scale sensitive graphic user interfaces.

1.3 Purpose

The purpose of this thesis is to see to what extent Scalable Vector Graphics (SVG) can be used to solve the problems described above. This includes researching the functional capabilities of SVG (what can be done?) and the performance in various aspects to asses how SVG GUIs should be designed. I should then use the knowledge gained to overcome eventual problems that may stand in the way of creating a working prototype that can show off the graphical capabilities of SVG user-interfaces. To cover the whole chain when creating SVG GUIs the thesis will also cover available authoring tools for creating SVG graphics.

1.4 Delimitations

This thesis will limit itself to cover functionality of SVG that I find suitable for use in GUIs. The functionality will be prioritized in this regard and the prototype will focus on establishing a sound concept to build further functionality upon. However some concepts implemented are experimental.

1.5 Structure

Chapter 2 describes the system targeted and the area of interest.

Chapter 3 gives an introduction to SVG and an insight into its capabilities. It also covers the different specifications, available tools, and concepts interesting from a GUI perspective.

Chapter 4 discusses the problem at hand and what has to be solved to make SVG useful for authoring user interfaces on a set-top box.

Chapter 5 is an overview of the requirements that have to be fulfilled to solve the problems highlighted in chapter 4.

Chapter 6 offers a short introduction to the AJAX framework currently used in the browser environment.

Chapter 7 gives a thorough description of a scaling technique called 'Artistic Resizing'.
Chapter 8 consists of several benchmarks covering different parts of the SVG specification. The aim is to draw conclusions on how to best author SVG user interfaces to achieve good performance.

Chapter 9 describes the solution aimed at solving SVG’s shortcomings from a GUI perspective. It contains several use cases describing the supposed behaviour of the prototype framework developed.

Chapter 10 gives an insight into how the most important parts of the framework were implemented.

Chapter 11 covers the lessons learned about SVG and evaluates the prototype framework to see if it lives up to the targeted functionality. Future developments of the framework are also discussed.
Chapter 2

IP-STB Environment

The main purpose of all STBs is to decode some external signal carrying video information. Normally this signal is distributed via satellite or terrestrial broadcast. The STBs covered in this thesis use an IP-based network (e.g. the Internet) to receive this signal.

This allows for a two way communication which makes it possible to provide a wide range of services previously not available. More importantly it adds more services where the user has to interact with the STB like chatting, web browsing, shopping etc. It also makes old services more useful. For example, video on demand now can be unicast to a user at anytime (instead of certain fixed times) and allow for pausing of the stream without buffering it locally.

2.1 IP-STB

An IP based set top box is a piece of computer hardware designed to decode video transmissions sent via an IP based network and to provide the means to interact with it. The computationally demanding video decompression is done with a dedicated Digital Signal Processor (DSP) which makes it possible to use slower (and cheaper) hardware for the other applications. Thus the cost sensitive market for STBs indirectly leaves little processing power to use for high fidelity graphics in user-interfaces. The interaction with the STB is usually done via a graphic interface and a remote-control.
The STB in question run on a GNU/Linux operating system which has the advantages of being free and offering a familiar programming environment. As seen in the picture there are several layers of abstraction. With SVG we will be able to continue the practise of coding the GUI on top of the ECMAScript [1]/JavaScript [2] interface. Because of this only the top layer is in our area of interest.

Figure 2.1. STB software architecture
Chapter 3

SVG

Scalable Vector Graphics is an XML-based specification by W3C for two-dimensional vector graphics. As a vector format it is scalable in the sense that the image isn’t bound to a certain discrete resolution to look good. Instead it can be rendered client-side in the appropriate resolution (typically the screen resolution). SVG supports all three types of graphic objects: vector graphic shapes, raster images and text. This allows combining raster images with vectors like when a photo is displayed in a vector graphic user interface. The SVG specifications also allows a compressed variant called SVGZ (as it is nothing else than a gzipped SVG file) [3](ch.1.2).

3.1 History

The W3C SVG Working Group started working in 1998 and SVG 1.0 became a World Wide Web Consortium (W3C) recommendation on 2001-09-04. 2003-01-14 SVG 1.2 was introduced as a W3C recommendation. The difference to SVG 1.0 is limited, with the most important change being the modularization of the standard [3](ch.1.1.1). This allows subsets to be defined as profiles which resulted in SVG Tiny (SVGT) and SVG Basic (SVGB) being introduced as SVG 1.1 profiles and W3C recommendations. SVGT was targeted for highly restricted mobile devices like cell phones and SVGB intended for more competent devices like PDA’s.

The latest SVG recommendation is SVG 1.2 Tiny (SVGT 1.2) which is the result of the lessons learned by SVGT and SVGB and became a W3C recommendation 2006-08-10. SVG 1.2 Full is being built upon SVGT 1.2 and is not yet a W3C recommendation. It is still being worked on by the W3C SVG Working Group and is only available as a working draft.

3.2 Rendering

SVG uses the painter’s algorithm [3](ch.3.2) when rendering meaning that there is no depth values (z-value) associated with the graphic objects. Instead the order of
rendering is based upon when the objects appear in the document. The first object is rendered first and so on. As the graphics are rendered client-side anti-aliasing can be applied allowing crisp non-jagged lines independent on what resolution the image is rendered in.

### 3.3 Document structure

The first tag of a document is the `svg` tag. It contains basic information about the document like namespace, version, size and coordinates. The `defs` tag contains graphic information that has to be referenced by other parts of the document to be visible. What then follows is the graphics seen on screen. This basic example below draws a red rectangle and a green circle. As seen they are displayed according to the painter’s algorithm described earlier.

```xml
<svg xmlns="http://www.w3.org/2000/svg" version="1.0" width="640" height="480">
  <defs>
  </defs>
  <rect fill="red"/>
  <circle fill="green"/>
</svg>
```

![Figure 3.1. SVG document structure](image)

### 3.4 Graphic primitives

As mentioned SVG allows us to mix vector graphics, text and raster images. This section describes how these three kinds of graphics are achieved when creating SVG content. The limited examples only show a small subset of the available functionality as they are only intended as an introduction to the reader.

#### 3.4.1 Vector graphics

SVG is able to render basic geometrical primitives like rectangles, circles, ellipses, lines and strokes. By manipulating the attributes the appearance can be modified. This extends further than changing the shape and position. You can also affect the filling and stroking of the object via attributes or styles. Figure 3.2 shows a subset of what can be done.
3.4.2 Text support

SVG 1.1 text support is limited in a document layout perspective. To achieve text that span over several rows you must either specify one text tag for each row or use one text tag in conjunction with several tspan elements. The SVG player offers no help so line breaks have to be precomputed [3](ch.10.1). This flaw has been corrected in SVGT 1.2 and remains so in the SVG 1.2 working draft. SVGT 1.2 offers the textArea element [4](ch.10.11) for wrapping of text inside a box while SVGF 1.2 is supposed to offer more advanced functionality like text wrapping inside arbitrary shapes [5](ch.3.1). SVG offers font manipulations

### Figure 3.2. Vector graphic examples

### Figure 3.3. Text wrapping examples
features like font family and font styles. On top of that non Tiny versions of SVG allows additional manipulation of the text like for example the textLength attribute which controls the length of a text string. Something with could be useful in a situation where a variable amount of text have to fit inside a limited space like an electronic program guide (EPG). You can also use SVG fonts [3](ch.20.2) to bundle your own fonts with the SVG file to guarantee platform independent rendering of text.

3.4.3 Raster images

SVG’s raster image support is pretty straightforward and in the lines of HTML. A drawback is that according to the specification you have to supply the image tag with a width and height attribute. Leaving out these attributes does not render the image in its native resolution. If you don’t know its exact size you can at least force the aspect ratio to be correct by using the preserveAspectRatio attribute and leave out either width or height. Something positive compared to its HTML counterpart is that it can be transformed like any other SVG element.

3.5 Useful concepts

This section describes concepts that I have found especially useful when experimenting with GUI creation in SVG.

3.5.1 Transformations

One very useful feature is the transformation attribute. It allows you to hook on affine transformations to SVG elements. You can do it either by applying rotation, translation, scaling and skewing separately or by combining them into one matrix. It also offers a homogeneous way to control the geometric appearance of all SVG elements that support the SVGTransformable interface [3](ch.4.3).

\[
M = \begin{bmatrix}
a & c & e \\
b & d & f \\
0 & 0 & 1 \\
\end{bmatrix}
\]

Figure 3.4. SVG graphics transformed with a SVGMatrix M.

3.5.2 Grouping

Grouping via the g element is a great way to get a good structure in the document. It also makes it easier to control the layout of the document via combining the nested data structure it brings with transformations. A more specialized use case is inheritance of attributes (see section 3.5.5).
3.5.3 Use

The use element is powerful combined with transformations as it allows effective reuse of graphics. By doing so something like a logo can be rendered at several places at different scale with the same source graphic.

```
<use transform="translate(50,60)" xlink:href="#myLogo"/>
```

Figure 3.6. Reuse of SVG graphics.

3.5.4 Defined graphics

By storing graphics inside the `defs` tag, they can remain invisible until referenced (with the `use` element). This makes it possible to organise the data instead of scattering the graphics all over the document. The `defs` element is also the place where you specify graphics like gradients and filters and then reference them from the rest of the document.

```
<svg>
  <defs>
    <g id="icons">
      <g id="tv"></g>
      <g id="radio"></g>
    </g>
    <use xlink:href="#tv"/>
  </defs>
</svg>
```

```
<svg>
  <defs>
    <linearGradient id="myGradient">
      <stop offset="0%" stop-color="#000000"/>
      <stop offset="100%" stop-color="#FFFFFF"/>
    </linearGradient>
  </defs>
  <rect width="100" height="40" fill="url(#myGradient)">
  </svg>
```

Figure 3.7. First: reusing defined graphics. Second: using a predefined gradient.
3.5.5 Inheritance of painting properties

By leaving out attributes in child nodes they will inherit these from their parents if available and supported [3](ch.11.8). This is useful if you want to style graphics with cascading style sheets (CSS) [6] as you can leave out the painting properties in most nodes and let them inherit the properties from a node that they share. It also results in smaller files and less DOM operations (see 3.7) to do when you want the change the appearance of the graphics via scripting.

```xml
<rect stroke="black"/>
<g fill="red">
  <rect stroke="black"/>
  <rect stroke="black" fill="blue"/>
</g>
```

Figure 3.8. Inheritance of painting properties.

3.6 Filter effects

SVG Full allows raster image effects like blur or 'drop shadow' SVG supports filter effects [3](ch.15). You can choose to use already defined filters or create your own convolve matrices. These can then be combined and applied to the document (or parts of) just like when you use filters in graphic editors like Photoshop. Filters are absent in SVG Tiny versions.

Figure 3.9. Filter effects (http://www.w3.org/TR/SVG/images/filters/filters00.svg)

3.7 SVG DOM

The SVG document object model (SVG DOM) is used to access the elements in the SVG document to allow interaction and animation. It builds upon, and is therefore compatible with the Document Object Model level 2 (DOM2) specification [7]. SVG DOM requires complete support for the DOM2-core and DOM2-views. It also requires support for relevant aspects of the DOM2 event model. Implementations of the SVG DOM that support CSS requires complete support of the DOM2 style sheets and all interfaces in DOM2-CSS required for aural and visual media. See SVG 1.1 specification [3](Appendix B) for more detailed information.
3.8 SVG uDOM

During the late stages of the development of SVG Mobile 1.1 the need for a more lightweight SVG DOM became clear. This resulted in SVG Micro DOM (SVG uDOM) [5](Appendix A), in which some features were sacrificed to achieve a smaller run-time footprint. According to the SVG Tiny 1.2 specification:

“The goal of the uDOM definition is to provide an API that allows access to initial and computed attribute and property values, to reduce the number of interfaces compared to the traditional SVG DOM, to reduce run-time memory footprint using necessary features of the core XML DOM, as well as the most useful SVG features (such as transformation matrices)”

It also states that the uDOM is used successfully in various Java Specification Request 226 (JSR-226) implementations. JSR-226 is a scalable 2D vector graphics api for Java Platform, Micro Edition(J2ME). SVGT 1.2 extends the functionality of the uDOM introduced in JSR-226 without making it too bloat ed for SVG Tiny implementations. It is also designed to be backwards compatible with the JSR-226 uDOM subset.

In SVG uDOM you are only allowed to navigate the element nodes of the DOM tree. This can be done in two ways, either by accessing elements by their id-attribute with the `getElementById` method on the `Document` interface, or by traversing the tree with the `ElementTraversal` interface which provides: `firstElementChild`, `lastElementChild`, `previousElementSibling` and `nextElementSibling`. This is the preferred method on constraint devices due to its simplicity.

The SVG uDOM includes functionality to modify the DOM tree just like the full SVG DOM. Introduced by SVGT 1.2 is the ability to access XML attributes and CSS property values with SVG uDOM through the concept of traits [5](Appendix A, sec.2.6). Unlike the use of `getAttributeNS` which always returns a string, traits offers direct access to the actual values with functions like `getFloat-Trait`. This is achieved through the `TraitAccess` interface with its trait facilities. It also offers a more strongly-typed access to element attributes. E.g. an element has a well-defined set of traits and you are only allowed to access traits which are defined. According to the working drafts SVG 1.2 Full DOM also implement traits which should allow applications which only use a subset to be backwards compatible [8](ch.3.1).

3.9 Dynamic SVG

Animation in SVG can be achieved in two ways. You can use declarative animation with SVG’s animation elements, or by using ECMAScript in combination with the SVG DOM.

SVG’s animation elements are developed in collaboration with W3C Synchronized Multimedia (SYMM) Working Group who is behind the Synchronized Multimedia Integration Language (SMIL) [9]. SVG include animation features defined
in SMIL and incorporates some SVG specific extensions as well. The animation elements allow some basic time-based animation like moving elements and changing colours and transformations. The animations can be controlled via ECMAScript allowing us to add control mechanisms on top of them making them more useful.

By scripting you can modify the SVG document in anyway you want on the fly. Every attribute and style is accessible via the DOM tree. By adding or removing nodes to the SVG DOM and modifying existing ones you can essentially build the SVG document as it is used. The downside of this is the computational weight that comes with it. Due the nature of the two ways I’ve found that scripting is more useful for gluing GUIs together while native SVG animations are more suitable for visual effects.

### 3.10 OpenVG

OpenVG [10] is a low-level open API for hardware acceleration of vector graphics authored by the Khronos Group [11]. It targets limited devices which need to provide high-quality user interfaces. It can be use to accelerate any vector graphics (flash, fonts etc.) but is specifically developed with SVG in mind:

“OpenVG must provide the drawing functionality required for a high performance SVG document viewer that is conformant with version 1.2 of the SVG Tiny profile. It does not need to provide a one-to-one mapping between SVG syntactic features and API calls, but it must provide efficient ways of implementing all SVG Tiny features.” [10]

The Khronos Group is a member-funded industry consortium (which includes chipsets manufacturers) focused on the creation of open standard APIs aimed at media services [12]. And as chipsets are becoming more available it might be sensible to have OpenVG in mind when developing new set top boxes.

Hardware with OpenVG support is already used in high end mobile devices. One example is Imagination technologies’ [15] PowerVR MBX chipset which is used in devices such as Apple Iphone and Sony Ericsson P1 [16]. They also offer an even more potent successor called SBX [17]. It boosts fully accelerated OpenVG 1.1 and extensive 3D capabilities like OpenGL 2.0 and OpenGL ES 2.0. They specifically mention high definition STB user interfaces as a use case. Another thing to consider is to request software implementations on current chipsets used. This way SVG players developed now can be ported with ease and take advantage of hardware acceleration in the future.

### 3.11 Graphic authoring tools

An essential and time-consuming part of GUI creation is graphic authoring. Designers and artists must not be limited by a subpar toolset. It is therefore time and cost efficient if they can use tools that they are familiar with. There is a range of options available both commercial and Open Source.
3.12 Summary

3.11.1 Adobe Illustrator
Adobe Illustrator [13] is more or less an industry standard when it comes to vector graphics for design and GUI related graphics. As an alternative to its own proprietary format it gives the user the choice to output in several SVG versions. It also has support for inserting simple scripting into the SVG document.

3.11.2 Inkscape
Inkscape [14] is a free Open source alternative for creating vector graphics with a SVG derivate as its native format. It also allows you to save in plain SVG 1.1 format. It offers most of Adobe Illustrators functionality but has a slightly different GUI. One useful feature is the ability to directly edit the SVG source inside the editor.

3.12 Summary
After this brief analysis of SVG the reader should now be familiar with the different concepts in SVG and its abilities. As described SVG is quite potent graphically but has some shortcomings when it comes to layout. This is natural as it isn’t tailored for document layout. But it still has to be adressed as a user interface requires both. By combining scripting and DOM tree manipulations with the concepts introduced in this chapter we can extend SVG and overcome its shortcomings.

Of the different SVG specifications available SVGT 1.2 should be a sound choice. Its focus on performance and extended text layout makes it a perfect fit for our need. If combined with hardware acceleration via OpenVG an attractive development platform could be achieved. The excellent availability of SVG authoring tools adds to this in a designer perspective as well.
Chapter 4

Problem analysis

This chapter highlights and discusses the problems that must be solved in order for SVG to achieve the demands for ease of use, rich graphics, performance and scalable GUIs.

4.1 Component reuse

While researching how SVG is used today in user interfaces it became clear that its use was quite limited. The GUIs were mostly meant for mobile phones and therefore limited in scope. They also seemed to be created in a "draw graphics and then insert some script"-fashion. And when they referenced other parts of the GUI (like after a menu selection) a new SVG file would load, much like an old scripted webpage.

This way of creating GUIs has two major drawbacks. First and foremost, it scales bad as there is not much room for component reuse as much of the logic becomes tied to the graphic presentation. Secondly, loading "sub-portals" by parsing and loading new files result in delays. These are likely to create annoyances and break the feeling of rhythm when navigating. Component reuse doesn’t only include GUI components but also graphic effects like animations and filters and importing of external SVG.

4.1.1 AJAX

These issues are something that’s being addressed in modern web applications with the adoption of AJAX technologies [18]. Even more so with the emerging use of Rich Internet Application (RIA) frameworks used to create web applications more similar to desktop applications than web pages [19]. Both open AJAX based ones and proprietary alternatives like Adobe’s Flex [20] are becoming more adopted with the recent trend of RIAs. By leaving the thin-client model and moving towards doing most computations on the client, delays are minimized resulting in better performance in the form of faster response times.
The company’s current browser based GUIs are developed in a declarative AJAX based framework on top of Mozilla to cope with these issues. It is called TOIX and draws inspiration from other AJAX frameworks like DOJO Toolkit [21] and Prototype [22]. By continuing on this path but on top of SVG one could combine SVG’s graphic palette with efficient component reuse. This way the demand for rich graphics, ease of use and fast development times should be fulfilled.

4.1.2 Layout

One of the most fundamental problems to solve in order to secure effective component reuse is to abstract and extend SVG’s lacking layout capabilities. The transform attribute offers a consistent way to control the appearance of graphics but it has to be abstracted and controlled in some way to make it useful.

Adobe Flex

Adobe Flex is interesting as it is vector graphic based with features similar to SVG as it is built on top of Flash [23]. It is declarative and uses an XML based language to lay out the application. To build new components and extend existing ones ActionScript 3.0 [24] (which is based on the ECMAScript version 4 proposal [25]) is used. Its combination of vector graphics and modern web-based technologies is in line with an AJAX based SVG framework. And as a high profile successful commercial product it should be an ideal source of inspiration on how to solve features as layout and animation to make the framework easy and powerful to work with. Adobe Flex allows for both absolute and automatic positioning [26]. Automatic layout is used when components reside inside of a container with a specific layout pattern. One example may be several components being arranged in a horizontal row inside of the container. Absolute positioning can either be done by specifying the position of the component in the coordinate system it resides in or by using constraints and attaching the component, with or without an offset, to a side, or centre of a container.

Constraint based layout

To achieve this kind of layout, constraint based programming is useful as described in Ultra-Lightweight Constraints [27]. Specifically, one-way constraints where for example the position of a component depends on the size of another component. But the size of the second component wouldn’t depend in any way on the position of the first component, hence, one-way constraints. These types of constraints can be formulated $u = C(p_1, p_2, \ldots, p_n)$, where $u$ depends on $p_1$ to $p_n$ according to the constraint function $C$. But no $p_i$ depends on $u$ [28]. By keeping a dependency graph on how layout attributes are related in-between graphic components, an algorithm is used to keep track of which order they have to be calculated, and how changes to one attribute affects other attributes to know which ones that have to be updated.
The need for constraints in SVG is nothing new and has been discussed in A Constraint Extension to Scalable Vector Graphics [29]. They developed an extension of SVG called Constraint Scalable Vector Graphics (CSVG) to solve this problem. Though their implementation solves the layout problems and on top of that interesting new scaling capabilities, it’s not appropriate in my case. There are two reasons for this. First of all, it is an extension of SVG and therefore isn’t standards compliant. Two, they don’t abstract the layout enough. The constraints work on a low level and are directly hooked on to the attributes of the SVG elements. The layout could be abstracted to the level of Adobe Flex without breaking standard compliance with the use of scripting.

4.2 Adaptive graphics

More often than not the graphics you see in a GUI isn’t just a static image. For example: Buttons sometimes have to adjust its size to cope with text labels of variable length. Or, you want to be able to show the button in several different sizes. Below is an example of wanted scaling behaviour. As seen the outline of the button is constant in width, the label isn’t distorted (either by scale or position) by the varied aspect ratio of the button, and the label is scaled down when necessary.

![Done](image1.png)

![Done](image2.png)

![Done](image3.png)

**Figure 4.1.** Wanted button scaling behaviour.

Normally creating a specific image for each button isn’t a viable option. So instead it is often solved with the help of a programmer and some image tiling method as the ones described in Providing Visually Rich Resizable Images for User Interface Components [30]. The button example could be solved (except for the smallest size due to radius of the button) by dividing the button into 9 boxes which are controlled by logic.
Problem analysis

Figure 4.2. Slicing images for 9-part tiling.

With vector graphics this can be solved in more powerful ways by manipulating the graphics by scripting. I.e. supplied parameters are translated into manipulations of the graphics.

**Drawbacks with the approach covered above:**

- The logic has to be re-written for every kind of behaviour wanted.
- It’s a waste of resources to have a programmer assist the designer.
- More complex behaviour (more dynamic parts) requires more logic.
- Limitations in what kind of behaviour that can be achieved.
- Graphic designers usually work by the eye. That is, they try something, evaluate the result, make changes and repeat the process until the desired result is achieved. Relying on a programmer makes this method slow and unintuitive [31].

I wanted to try and solve these problems by implementing a concept called Artistic Resizing (see chapter 7) in the SVG framework.

### 4.3 Implementation alternatives

Two possible paths existed. Either a new JavaScript framework just for SVG could be developed, or the existing in-house TOIX framework (see chapter 6) could be extended. TOIX was chosen as its hierarchical widget system and modularity made it possible to extend it with functionality like layout handling without having to rely on any HTML specific functionality. With TOIX fundamental functionality like input handling and Ajax requests already existed which made it possible to focus on SVG specific functionality. A possible future need to mix SVG widgets with HTML widgets also favoured the TOIX solution. As no STB based SVG-player is available, the prototype will be implemented on top of Firefox (PC).
4.4 Validation

The SVG framework will be used to create a mock-up STB GUI. This way it will evaluate if the SVG framework solves all problems it set out to do. In addition to implement the rudimentary components needed fulfill the requirements (described in the next chapter) a menu widget should be created. Due to it being such a fundamental part of a STB GUI it is a good validator. The mock-up GUI should make use of all these components.
Chapter 5

Requirements

This section covers all requirements on the SVG framework. The main purpose is to solve the problems discussed in the previous chapter. But as it is an extension of an existing framework, the company required that the extensions should be in line with the existing HTML functionality. Each requirement has an identification number and a motivation.

5.1 Basic requirements

These are the basic requirements the SVG framework needs to fulfill to overcome the problems mentioned in the problem description, and to be considered a success.

R1 As I extend the existing framework the basic functionality of the SVG components should be the same as for HTML components. Hence, making porting of existing portals easier (required by company).

R2 The framework should make it possible to use constraint based layout. That is, you should be able to align GUI components that reside inside container components to the sides or centre of the container by using `valign` and `halign`.

R3 It should be possible to transform all graphic components. Transforms should be abstracted and used by supplying the components with the parameters: `skew`, `scale` and `rotate`. If no point is specified, the transformation is applied around the components centre.

R4 When applying the transformations they aren’t allowed to upset the layout functionality specified by requirement R2. E.g. a graphic object that is scaled by a factor 2 and rotated by 45 degrees should still be aligned to the bottom of the screen if it has `valign="bottom"` as an attribute.

R5 The only transformation that is allowed to upset the layout is a translation. This way you can use the `translate` attribute to adjust the placement of the SVG widgets.
R6 Containers should have the capabilities to override the absolute positioning of child components according to layout rules like flow layout. This way the layout parameters of the child widgets can be left out as they will be aligned automatically.

R7 By being able to associate SVG widgets with animations you can for example trigger an animation of a widget on an event to offer feedback to the user.

R8 All SVG widgets should behave in the same way in response to their shared parameters. That is, you should be able to apply a layout parameter, add an animation, transformation etc. to any SVG widget and get a consistent result regardless of the widget being a menu, image or container.

R9 The framework should add text wrapping inside a box that is aligned to the left for SVG 1.1 viewers.

R10 It should be possible to import external SVG data and attach to a chosen place in the DOM tree.

R11 It should be possible to control the graphic style of a SVG widget by feeding it with a class parameter specifying its CSS class.

R12 You should be able to apply filters to all SVG Widgets and thereby achieve raster effects like blur if supported by SVG player.

R13 Support for adaptive graphics by implementing a SVG widget that brings Artistic Resizing support.

5.2 Extra requirements

This section lists extra functionality that would be nice to have but might not be feasible to implement in the timeframe of the thesis.

E1 Make it possible to control the layout parameters via a CSS-like interface.

E2 Make it possible to "deep skin" [32] the GUI. Add support to allow changes to large parts of the graphic by just manipulating skin files.

The extent to which the framework fulfills the requirements will be evaluated in chapter 11.
Chapter 6

TOIX

TOIX (TOI with AJAX) is a JavaScript framework which draws its inspiration from other HTML RIA frameworks like DOJO Toolkit. It extends JavaScript with a more Java-like Object Orientated Programming (OOP) model with support for packages, classes and inheritance. Other packages included add the most needed functionality like Ajax requests, DOM manipulation, widgets, and abstracting the STB environment. As it was developed specifically for the STB environment, small memory footprint and high performance (in browser GUI perspective) has been its focal point. It is developed with Mozilla as the targeted platform but it also works on Opera [33]. Below I describe the fundamentals needed to understand the modifications done to add SVG support.

6.1 The widget concept

A widget is usually a graphic component that can be seen on screen like a menu but it can also be logic used to make the GUI behave in a certain way. A visual widget has both an internal ECMAScript side and an external HTML representation controlled by its internal representation. What sets widgets apart from an ordinary class it that they can be created using the TOIX mark-up language. In the menu example the HTML would make up the visuals of the menu and the ECMAScript would manipulate the HTML depending on user-input.
6.1.1 Example

The following TOIX code for a clock-widget is interpreted resulting in a HTML clock that is controlled (refreshed, activated etc.) by an ECMAScript object.

**TOIX code:**

```
<toix:clock class="clockRoot" halign="left"></toix:clock>
```

**Resulting HTML:**

```
<div align="left" class="clockRoot">07:58 AM</div>
```
Chapter 7

Artistic Resizing

Artistic Resizing [31] is a technique that allows the designer to control the behaviour of the image by providing examples of the vector graphics. Then by interpolating between the transformation matrices that define the different sample images you get an image suitable for the current case. We return to the button example to show how we can control the scaling behaviour with this method.

Figure 7.1. The designer has provided three sample images. The lower three buttons have been calculated from these via interpolation according to the chosen width and height variables (w and h).
7.1 Orthogonal interpolation

We use an interpolation method that allow us to specify the width \((w)\) and height \((h)\) values separately. This is called orthogonal interpolation and is based upon that matrix elements of affine transformations can be separated into those who impact \(x\)-values and those who impact \(y\)-values in the output.

\[
T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \quad v = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad Tv = \begin{bmatrix} a_{11} \cdot v_x + a_{12} \cdot v_y + a_{13} \\ a_{21} \cdot v_x + a_{22} \cdot v_y + a_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}
\]

As seen only \(a_{11}, a_{12}\) and \(a_{13}\) affects the \(x\)-value. Let’s say we have two matrices \(T_1\) and \(T_2\) and that the only thing that differs between them is an affine horizontal translation. That is, only \(a_{13}\) differs. Now, if we want the resulting translation to be a value "between" them, the resulting \(a_{13}\) should be: \(a_{13} = a_{13} \cdot (1 - w) + a_{13} \cdot w\), \(0 \leq w \leq 1\). This is nothing more than a linear interpolation and can be written: \(I_{a_{11},a_{13}}^{lin}(w)\). It’s obvious that the same can be done with \(a_{11}\) and \(a_{12}\) which affect scaling and shearing. And that the same goes for the elements that affect the \(y\)-values. A rotation depends on both those values which impact \(x\)-values and those which impact \(y\)-values. And as the relationship between \(x\) and \(y\) isn’t linear the rotation will be interpreted as one horizontal and one vertical shear when interpolated.

The definition for the orthogonal interpolation between two transformations \(T_1\) and \(T_2\) then is as follows:

\[
I_{T_1,T_2}^{orth} = \begin{bmatrix} I_{a_{11},a_{12}}^{lin}(w) & I_{a_{12},a_{13}}^{lin}(w) & I_{a_{13},a_{23}}^{lin}(w) \\ I_{a_{21},a_{22}}^{lin}(h) & I_{a_{22},a_{23}}^{lin}(h) & I_{a_{23},a_{33}}^{lin}(h) \\ 0 & 0 & 1 \end{bmatrix}
\]

This can be adapted to the case where we have more than two examples (and therefore more than two transformation matrices for each object) via piecewise linear interpolation [34]. Note that the matrices we interpolate between with the \(x\)-value don’t have to be the same ones used with the \(y\)-value. If we have three transformation matrices \((T_1, T_2 \text{ and } T_3)\) to interpolate between and choose the values: \(w = 1.5\), \(h = 2.7\) the calculated matrix would be:

\[
\begin{bmatrix} I_{a_{11},a_{12}}^{lin}(0.5) & I_{a_{12},a_{13}}^{lin}(0.5) & I_{a_{13},a_{23}}^{lin}(0.5) \\ I_{a_{21},a_{22}}^{lin}(0.7) & I_{a_{22},a_{23}}^{lin}(0.7) & I_{a_{23},a_{33}}^{lin}(0.7) \\ 0 & 0 & 1 \end{bmatrix}
\]

Notice how element \(a_{11}, a_{12}\) and \(a_{13}\) depend on elements from matrices \(T_1\) and \(T_2\) and element \(a_{21}, a_{22}\) and \(a_{23}\) depend on elements from matrices \(T_2\) and \(T_3\).

7.2 Use case

Artistic Resizing can be used for more than the simple button resizing. Below I present a different use cases to show what more can be done.
7.2.1 Variable icon size

It could be useful to have a variable icon size that depends on the number of items presented or user input. It would be ugly to just scale everything. First of all you might want lines and text to be the same like the button example. But an icon can also be complex graphics where different parts might have to scale independent of others to please the eye. You might also have variable details so some parts of the graphics disappear completely when the icon gets too small or that new items appear. All this can be done with Artistic Resizing.

![Figure 7.2. An icon with advanced scaling behaviour.](image)

The example above is supposed to show an icon for scheduling (for example a TV-show). This icon makes use of several of the techniques covered above. First of all the width of the icon’s border remain the same between all four example frames. The exceptions from normal affine scaling between frame 1 and 2 are that the clock is scaled less or else it would look too small and that "07" isn’t scaled as it would become hard to read. Frame 3 is a transition frame that removes parts that shouldn’t be displayed (by setting the scaling elements in the transformation matrix to 0), adds the new part of the simplified clock and re-aligns "July". As it doesn’t make sense to interpolate between frame 2 and 3, functionality to handle this gracefully have to be added on top of the purely linear calculations. From frame 3 to 4 we see that the upper red and white parts of the icons remain the same height, and that the clock isn’t scaled as much as the rest of the image. In the icon example you typically only have one scaling value that is applied to both $w$ and $h$.

7.3 Summary

Artistic Resizing is an interesting concept which allows designers to work more independent. Its generality makes it applicable in a wide range of different uses cases and with the ones described here we have only scratched the surface. The key in an implementation is to make it easy to use so it can be exploit to the fullest.
Chapter 8

Benchmarks

The performance aspects covered are divided into three categories. First we investigate the SVG to SVGZ compression ratio\(^1\). Secondly we see how file size and compression affect the loading times when importing external SVG. And lastly we analyse how different parts of the SVG specification affects performance. These benchmarks doesn’t pit SVG against another solution for comparison. The purpose is to find out how to make the best use of SVG and to become aware of eventual pitfalls in SVG GUI development. The browsers used where Mozilla Firefox 2.0.0.5 and Opera 9.22. Firebug \([35]\) was used to collect data import statistics.

8.1 SVG Compression(SVGZ)

The variation in the compression from SVG to SVGZ varies a lot (between about 2-10 times). The reason for this can be found in that a SVGZ file is nothing more than a gizado SVG file \([3]\)(ch.1.2). Gzipe \([36]\] uses a deflation algorithm that is a variation on LZ77 \([37]\) (Lempel-Ziv 1977). Simply put it works by finding duplicated strings in the input data. When a match is found it is replaced with a pointer to the first occurrence. In the case of SVG this means that files with simple elements that occur often and don’t contain a lot of unique data (like \(<rect x="x_1" y="y_2" width="w" height="h"/>angle\) compress several times. But files with a lot of data heavy elements (typically \(<path d="M x_1,y_1 c x_2,y_2 x_3,y_3 x_4,y_4 x_5,y_5 \ldots "/>\)) compress less. To test this a Python \([39]\) script that creates a requested number of SVG files containing a specified amount of random elements was used. The four kinds of documents randomized were:

1. 10 random elements including \(rect\), \(circle\), \(ellipse\) and \(path\)
2. 10 random elements including \(rect\), \(circle\), \(ellipse\) but \(no path\)
3. 100 random elements including \(rect\), \(circle\), \(ellipse\) and \(path\)
4. 100 random elements including \(rect\), \(circle\), \(ellipse\) but \(no path\)

\(^1\)compression ratio = uncompressed size / compressed size
Each type of file was randomly created 100 times to get a good mean. Another script was then used to gzip the files and collect the statistics.

<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Paths</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Yes</td>
<td>1.90</td>
</tr>
<tr>
<td>10</td>
<td>No</td>
<td>2.05</td>
</tr>
<tr>
<td>100</td>
<td>Yes</td>
<td>2.37</td>
</tr>
<tr>
<td>100</td>
<td>No</td>
<td>3.71</td>
</tr>
</tbody>
</table>

Table 8.1. Compression ratio dependence, considering file-size and type of SVG elements.

Now what would be really interesting to see is what level of compression that can be achieved with the kind of graphics used in user-interfaces. To test this Open Clip Art Library 0.18 [38] was used. It includes a number of files under: /openclipart-0.18-svgonly/clipart/computer/icons/etiquette-theme/ which contains graphics typical to a GUI (according to my subjective preference). After using the compression and statistics script once again the following was gathered:

<table>
<thead>
<tr>
<th>Folder</th>
<th>SVG size (bytes)</th>
<th>SVGZ size (bytes)</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>1030781</td>
<td>298533</td>
<td>3.45</td>
</tr>
<tr>
<td>/actions</td>
<td>14163</td>
<td>40017</td>
<td>3.54</td>
</tr>
<tr>
<td>/apps</td>
<td>316894</td>
<td>100120</td>
<td>3.17</td>
</tr>
<tr>
<td>/devices</td>
<td>83420</td>
<td>28639</td>
<td>2.91</td>
</tr>
<tr>
<td>/emblems</td>
<td>14631</td>
<td>3031</td>
<td>4.83</td>
</tr>
<tr>
<td>/filesystems</td>
<td>151633</td>
<td>43989</td>
<td>3.45</td>
</tr>
<tr>
<td>/mimetypes</td>
<td>436880</td>
<td>117137</td>
<td>3.73</td>
</tr>
<tr>
<td>/stock</td>
<td>573882</td>
<td>186941</td>
<td>3.07</td>
</tr>
<tr>
<td>/stock/generic</td>
<td>12370</td>
<td>3805</td>
<td>3.25</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>2762128</strong></td>
<td><strong>822212</strong></td>
<td><strong>3.36</strong></td>
</tr>
</tbody>
</table>

Table 8.2. Compression ratios of GUI like SVG content.

### 8.2 Importing SVG

To avoid storing all SVG graphics in one file we can import the graphics needed from external SVG formatted files with the help of Ajax requests (XMLHttpRequest). So it is useful to research the general performance of importing SVG and if gzip compression has any influence on the results. The first thing to test is if there are any difference in performance between importing a few big SVG files compared to splitting up the SVG data in smaller chunks. The table shows the import of a 136kB large randomized SVG file compared to the import of ten randomized smaller files of 139kB combined. So the data size should be about the same. Each case was tested five times to get a good mean.
8.2 Importing SVG

<table>
<thead>
<tr>
<th>Test case</th>
<th>Test1</th>
<th>Test2</th>
<th>Test3</th>
<th>Test4</th>
<th>Test5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 136kB</td>
<td>1.74</td>
<td>1.77</td>
<td>1.74</td>
<td>1.74</td>
<td>1.72</td>
<td>1.74</td>
</tr>
<tr>
<td>10 x ~14kB</td>
<td>1.69</td>
<td>1.71</td>
<td>1.69</td>
<td>1.68</td>
<td>1.69</td>
<td>1.69</td>
</tr>
</tbody>
</table>

*Table 8.3. Import time in seconds (small versus large files).*

Based on the negligible differences this is a non-issue and the chosen solution should be decided by other factors like achieving a sound data structure of the GUI.

The next thing to test was if gzip compression has any influence on the loading time of the files. To test this I once again used the script that creates randomized SVG files. As I wanted to cover all bases files with 10, 100 and 1000 elements were tested. 25 files of each type were generated to get a good mean. The result in the table 8.4 shows the mean time to load each type of files. As seen the difference is minuscule. There seems to be a slight advantage in favour of SVGZ especially when the amount of elements increases. Probably because of the higher data loads required for the SVG files as they are about 3 times as large. As vector graphics files SVG is still small compared to pixmap graphics so there probably isn’t a great need to make them smaller too fit on the STB platform despite its limited memory. Bear in mind that at runtime the footprint would still be the same. On the other hand as the performance difference is negligible there is no reason not to use SVGZ. It would also be a bandwidth saver when the content is downloaded from a remote server to the STB.

<table>
<thead>
<tr>
<th>Test case</th>
<th>10 elements</th>
<th>100 elements</th>
<th>1000 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVG</td>
<td>0.025</td>
<td>0.180</td>
<td>2.05</td>
</tr>
<tr>
<td>SVGZ</td>
<td>0.024</td>
<td>0.180</td>
<td>1.76</td>
</tr>
</tbody>
</table>

*Table 8.4. Import time in seconds (SVG versus SVGZ).*
8.3  Graphic performance

To achieve responsive GUIs performance is essential. If “Everything is for the best in the best of all possible world” we should be able to construct the most elaborate user interfaces with complex dynamic graphics and still get excellent performance. As we know this is not the case, especially on a limited STB platform. Optimizations can, of course, be done by enhancing the algorithms used in the SVG viewer. Another approach is content control. The later can be done in two ways:

1. To conform to the SVG specification(s) SVG viewers are more general than we need them to be. We are more interested in getting good performance with graphics typical to a GUI. By analysing our needs we could add functionality to the SVG viewer to speed up the targeted content.

2. Another approach is to analyse the cost of doing certain things and adapt to this when creating the content. The sections below are supposed to cover the most important considerations to take with the graphical content.

As there is no available SVG viewer on the STB platform at the moment the tests have to be carried out on a PC. It is no secret that you won’t get the same performance on the STB and that you can’t expect a linear relationship between the cost of doing something on the PC and on the STB. But as we are only interested in trends, not exact numbers it doesn’t matter. We just want to know what considerations to take in order to get the most out of SVG.

8.3.1  Test description

The testing is done with an ECMAScript file that renders the content 50 times with a 100 ms delay resulting in an optimal time of five seconds. But as the DOM operations themselves (and other operations) take some time running the test without rendering takes a little over 6 seconds both for Mozilla and Opera. Each test was done three times resulting in a mean value which was adjusted by subtracting the time spent not rendering.

8.3.2  Path complexity

To assess the impact the complexity of a path has on the performance I used modifications of the file fern_mo_01.svg from openclipart-0.18. First of all the file was saved in four different levels of complexity:

1. **Leaf1**: Curves consisting of 2379 nodes.
2. **Leaf2**: Curves consisting of 1867 nodes.
3. **Leaf3**: Curves consisting of 1514 nodes.
4. **Leaf4**: Curves consisting of 2379 nodes.
Each type was rendered in three different ways:

1. Filled with uniform colour.

2. No filling but stroked.

3. Filled with a uniform colour and stroked

![Diagram showing different versions of a fern SVG with test cases for Leaf 1, Leaf 2, Leaf 3, and Leaf 4 with times forFilled, Stroked, and Filled & Stroked versions.]

**Figure 8.1.** The different versions of fern_mo_01.svg.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Leaf 1</th>
<th>Leaf 2</th>
<th>Leaf 3</th>
<th>Leaf 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filled</td>
<td>3.0</td>
<td>2.0</td>
<td>1.3</td>
<td>1.6</td>
</tr>
<tr>
<td>Stroked</td>
<td>17.2</td>
<td>11.3</td>
<td>6.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Filled &amp; stroked</td>
<td>19.9</td>
<td>12.9</td>
<td>7.3</td>
<td>2.9</td>
</tr>
</tbody>
</table>

**Table 8.5.** Rendering time in seconds depending on path complexity and type. Browser: Mozilla
As seen the path complexity impacts both the filling and stroking performance. The stroking seems to suffer more, which is natural as the area is more or less the same but with a more complex border while stroking has a much longer path to paint in the complex case. Another thing that stands out is that rendering lines is much faster than rendering curves as can be seen from the result of Leaf 4.

To find out how much of the performance gain going from Leaf 1 to Leaf 3 that depends on the lack of nodes and not the graphic complexity of the curve, Leaf 3 was rendered again but with the double amount of nodes while keeping the curve’s appearance.

![Figure 8.2. Same visual appearance but different path complexity.](image)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Leaf 3 (1514 nodes)</th>
<th>Leaf 3 (3028 nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filled</td>
<td>1.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Stroked</td>
<td>6.1</td>
<td>10.3</td>
</tr>
<tr>
<td>Filled &amp; stroked</td>
<td>7.3</td>
<td>12.1</td>
</tr>
</tbody>
</table>

Table 8.6. Rendering time depending on the amount of nodes (in seconds). Browser:Mozilla

The difference in performance show that the amount of nodes in itself, not the increased graphical complexity, contributes to the performance loss. The bottom line is that a lot can be gained by being cautious when creating the content. Avoid using too much detail and get rid of unnecessary nodes. In the end you have to weigh aesthetics versus performance so in some cases you might even be able to use lines instead of curves to get a performance boost. And of course there are situations where you can’t control path complexity like when you render text. If it’s static (e.g. a header) you might consider swapping it with a pixmap but otherwise you just have to avoid doing anything heavy in the meantime like animating it.
### 8.3.3 Filling

When playing around with different kinds of SVG graphics I have experienced that gradients tend to slow things down. To assess to what extent this is true different rectangles with different fillings were rendered to see if there was a difference.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Solid (2 stops)</th>
<th>Linear (2 stops)</th>
<th>Linear (4 stops)</th>
<th>Radial (2 stops)</th>
<th>Radial (4 stops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla</td>
<td>0.0</td>
<td>0.6</td>
<td>0.7</td>
<td>24.7</td>
<td>27.6</td>
</tr>
<tr>
<td>Opera</td>
<td>0.0</td>
<td>1.6</td>
<td>2.2</td>
<td>5.6</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Table 8.7. Rendering time difference (in seconds) between solid fill, linear gradients and radial gradients.

As seen in the table, when doing a simple colour fill, the rectangles has no noticeable influence at all. The linear gradients had a clear performance impact but adding more stops to the gradient didn’t influence the result that much. Because of the extremely bad results of radial gradients on Mozilla I decided to also test it on Opera. The same pattern can be seen there. But the bad result of radial gradients in Mozilla seems to be an implementation issue. The results suggest that performance can be gained by showing some restraint when using gradients, especially radial.

### 8.3.4 Pixmaps

Often there is no need for graphics to be dynamic, hence, pixmaps [40] can be used. Of course there are other drawbacks such as loss in quality due to resampling artefacts if scaled. The following tests are supposed to give hints about the behaviour of pixmaps when mixed with SVG.

#### Scaling

In this test the file bitmapLeafLarge.png (631*600) is rendered at different sizes to see how it affects the performance. Scale 1 is equivalent of rendering the image in its native resolution 631*600.

<table>
<thead>
<tr>
<th>Scale 0.2</th>
<th>Scale 0.5</th>
<th>Scale 0.9</th>
<th>Scale 1</th>
<th>Scale 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>8.0</td>
<td>36.4</td>
<td>4.0</td>
<td>56.4</td>
</tr>
</tbody>
</table>

Table 8.8. Pixmap rendering time in seconds. Browser:Mozilla

To no surprise it’s faster to render the image on a smaller scale. What’s more interesting is that when the image is rendered in its native resolution a performance boost is evident both in Mozilla and Opera. This is likely to do with that the browser doesn’t have to resample the image to chosen scale. So with a proper SVG implementation (on that doesn’t resample when rendering a pixmap in its native resolution) this could be used to ones advantage when creating a GUI. But
it should be noted that this limits what can be done with the pixmap. As soon as the image is geometrically transformed or *translated to a sub-pixel coordinate* there is no performance gain.

### Resolution

To see what impact the source image’s resolution has I render the same picture saved in three different resolutions at the same size (that is non native for all of them).

<table>
<thead>
<tr>
<th></th>
<th>Large 631*600</th>
<th>Medium 315*300</th>
<th>Small 63*60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering time (in seconds)</td>
<td>14.6</td>
<td>14.1</td>
<td>13.8</td>
</tr>
</tbody>
</table>

**Table 8.9.** Pixmap rendering time (in seconds) depending on resolution. Browser:Mozilla

The resolution clearly makes a difference but it’s not nearly as influential as the scale. So there isn’t much point in basing the decision of pixmap resolution on performance. For example, tweaking by scaling up a low resolution image and get bad quality probably isn’t worth the trade-off. Note that scaling down a high-resolution pixmap also can result in image degradation depending on the scaling algorithm used. Unfortunately this is out of our control so choosing the proper resolution is key.

### 8.3.5 Area

Today HDTV’s are becoming commonplace which puts pressure on STB makers to rise up and offer HD services. This shouldn’t only be done by offering HD decoding capabilities but also by enhancing the GUI. A higher resolution allows you to display more information which would improve the usability in a wide range of functionality. This is supposed to give a hint on the increased demand when rendering in HDTV resolutions instead of the standard NTSC and PAL resolutions. The test consists of a mock-up GUI rendered 50 times just like the previous tests. The GUI itself consists of some pixmaps and vector graphics that I composed in Inkscape.

<table>
<thead>
<tr>
<th></th>
<th>NTSC 720*480</th>
<th>PAL 768*576</th>
<th>720p 1280*720</th>
<th>1080p 1920*1080</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering time (in seconds)</td>
<td>6.0</td>
<td>8.0</td>
<td>12.6</td>
<td>27.4</td>
</tr>
</tbody>
</table>

**Table 8.10.** Rendering time measured in seconds in correlation with resolution with GUI like content. Browser:Mozilla
8.4 Summary

This chapter has described several ways how to optimize the content to gain performance. As these tests have been carried out on a PC it would make sense to do further analysis on a STB when a compatible SVG viewer is available in order to get more accurate results. We will return to this performance evaluation in chapter 11.

As seen the impact the area (number of pixels) has on performance is slightly less than linear. But, in reality it isn’t that bad. With a properly implemented SVG viewer you would only re-render parts of the screen which have been updated resulting in a lot less pixels to draw. Even so it clearly gives a hint about the increased demands when drawing vector graphics at higher resolutions.
Chapter 9

Solution

This chapter describes the solution to the issues brought to light in the problem analysis. It resulted in an extension of the TOIX framework called TOIXSVG. The extension includes both changes to the framework and additions of new widgets needed to implement the new features. Functionality and widgets needed to implement the mock-up GUI are also described.

9.1 TOIXSVG

The framework extension is called TOIXSVG and as it is bound by Mozilla’s and Opera’s capabilities it is limited to their SVG implementations which are subsets of SVG 1.1. The emphasis for the framework has been to solve the layout functionality as it is fundamental to make component reuse possible.

9.1.1 Layout

A widget’s geometrical shape (after rotation etc.) and spatial position can be controlled with the SVG transform attribute. But without added functionality this is too cumbersome and not powerful enough when creating user interfaces. Therefore functionality must be added that let us modify a widget’s geometrics and alignment in a more effective way. We have roughly two types of SVG widgets to consider.

1. Container widgets: Container widgets can be used to group other SVG widgets and their purpose is to control the layout of the GUI. Children in a container can use constraint based layout to align themselves in it. On top of that containers also include support for laying out child widgets according to some layout pattern like flow layout.

2. Ordinary widgets: This category includes everything from basic text boxes to menus. These are typically atomic and therefore don’t have any child widgets. But there are exceptions like the menu widget which uses child widgets as menu selections.
Transformations

All graphic components can be transformed. In the case of containers the transformation also affects the appearance of its children due to SVG nested transformation structure.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>'&lt;number&gt;'</td>
<td>Scales the widget uniformly</td>
</tr>
<tr>
<td></td>
<td>'&lt;number&gt;,&lt;number&gt;'</td>
<td>Scales the widget non uniformly</td>
</tr>
<tr>
<td>skew</td>
<td>'&lt;number&gt;'</td>
<td>Skews the widget horizontally</td>
</tr>
<tr>
<td></td>
<td>'&lt;number&gt;,&lt;number&gt;'</td>
<td>Skews the widget horizontally</td>
</tr>
<tr>
<td></td>
<td>and/or vertically</td>
<td></td>
</tr>
<tr>
<td>rotate</td>
<td>'&lt;number&gt;'</td>
<td>Rotates the widget</td>
</tr>
</tbody>
</table>

Table 9.1. Types of transformations

Positioning

The layout for all graphic components can be controlled either via absolute or automatic positioning. For absolute positioning both constraint based layout and manually supplying a translation is allowed. A widget position itself according to size of the container it resides in.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halign</td>
<td>'left', 'center'</td>
<td>Controls the horizontal alignment of the widget.</td>
</tr>
<tr>
<td></td>
<td>'middle', 'right'</td>
<td></td>
</tr>
<tr>
<td>valign</td>
<td>'top', 'center'</td>
<td>Controls the vertical alignment of the widget.</td>
</tr>
<tr>
<td></td>
<td>'middle', 'bottom'</td>
<td></td>
</tr>
<tr>
<td>translate</td>
<td>'&lt;number&gt;,&lt;number&gt;'</td>
<td>The supplied x and y values are added to the current position of the widget.</td>
</tr>
</tbody>
</table>

Table 9.2. Attributes for absolute and constraint based positioning.

When using automatic positioning the layout depends on the type of the parent container. As you shouldn’t have to specify the absolute positions of each widget the layout manager supports layout patterns inspired by Adobe Flex [26]. The patterns are implemented as extensions of the basic container widget.
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hbox</td>
<td>The widgets are distributed horizontally in the container.</td>
</tr>
<tr>
<td>Vbox</td>
<td>The widgets are distributed vertically in the container.</td>
</tr>
<tr>
<td>FlowBox</td>
<td>Equivalent of a vbox with hboxes as child widgets. The height of a row depends on the highest widget that reside in it.</td>
</tr>
</tbody>
</table>

**Table 9.3.** Container types with different layout patterns.

**Use cases**

The following use cases show how the layout manager is expected to behave. They are presented with an explanation, pseudo TOIX code and a picture showing of the end result. The dashed border marks the width, height and position of the widget.

**Use case 1:** A container widget with a widget aligned to its bottom right corner.

```
<container width="200" height="200">
  <widget halign="right" valign="bottom"/>
</container>
```

![Figure 9.1. Use case 1](image)

**Use case 2:** Same alignment as last case but the widget’s geometrics has been changed via a rotation of 10 degrees. As seen the rotation isn’t allowed to disrupt the alignment and the widget is still perfectly aligned despite becoming larger due to the rotation.

```
<container width="200" height="200">
  <widget halign="right" valign="bottom" rotate="10"/>
</container>
```
Use case 3: Same as use case 2 but with an extra widget. As seen the *translate* attribute is allowed to adjust the spatial position after alignment.

```xml
<container width="200" height="200">
  <widget halign="right" valign="bottom" rotate="10"/>
  <widget halign="right" valign="bottom" rotate="10"
    translate="-100,0"/>
</container>
```

Use case 4: You are able to use nested containers. The inner container (taken from use case 3) is aligned at the top vertically and in the middle horizontally. It is also rotated -10 degrees to spice things up. The grey dashed line is the inner container and the inner black dashed rectangle the new position, width and height that must be taken into account when positioning it.
Use case 4: This example shows how the flowbox container is used to layout widgets automatically. As seen the attributes hpad and vpad are used to control the spacing between the widgets.

<flowbox width="200" height="150" hpad="20" vpad="30">
  <widget/>
  <widget scale="0.7,1.3"/>
  <widget scale="1.3,0.6"/>
</flowbox>

Figure 9.4. Use case 4

Use case 5: This example shows how the flowbox container is used to layout widgets automatically. As seen the attributes hpad and vpad are used to control the spacing between the widgets.

<flowbox width="200" height="150" hpad="20" vpad="30">
  <widget/>
  <widget scale="0.7,1.3"/>
  <widget scale="1.3,0.6"/>
</flowbox>

Figure 9.5. Use case 5
9.1.2 Importing SVG

The framework makes it simple to import external SVG data via XMLHttpRequest to the current DOM tree so it is simple to divide the GUI into separate files. A problem when importing SVG is that it can be drawn anywhere in the canvas. On top of that the canvas can be of a totally different size than the one we use for our GUI. The layout functionality described earlier makes it easy to transform and place the imported SVG where you want when referencing it. You are also able to attach the imported SVG wherever you want in the DOM tree.

Use case

Lets assume that you have created a collection of icons, each with a unique root node id in the file myIcons.svg. The first step would be to import it and place the data in an appropriate place. As we don’t want it to be visible this would be some node under the defs tag. If we recall section 3.5.4 it is then easy to reference it via a use tag and the root id (in this case oneIcon). The following two lines of code do what was just described:

```xml
<import svgFile="myIcons.svg" attachNode="icons"/>
<graphic srcNodeId="oneIcon"/>
```

9.1.3 Pixmaps

Implementing support for raster images it straight forward as the widget only has to mirror SVG’s image-tag.

Use case

```xml
<image uri="myImagePath.png"/>
```

9.1.4 Animation

Animation support is a bit primitive at the moment. The reason for this is that Mozilla lacks native SVG animations support. The purpose of the animation implementations is that the framework should have the same animation interface independent of native SVG support or not. Currently this means that it maps all animations to DOM manipulations. But future versions of TOIXSVG should automatically choose the best available option for the current case, thus relying on both native SVG animations and DOM manipulations. Basic functionality has been implemented for all geometric transformations. On top of that some additional effects are available. All animations allow you to specify the duration and transition type (ease in, linear etc.). The functionality has been implemented in separate ECMAScript classes instead of being fully fledged widgets. This way they can be used when programming a widget. To make them available in the declarative layer as well, widgets which maps directly to the animation classes have been implemented.
Use case

In this case the "animation target" (the widget that is animated) is set explicitly as an argument to the animation. If not set, the animation will automatically target its parent just like native SVG animations do.

```
<move id="anim1" targetId="menu" duration="500"
    transition="txEaseIn" translate="0,-100"/>
$getWidget("anim1").play();
```

9.1.5 Text handling

To be able to present text that isn’t already formatted a very basic text wrapping widget was implemented. It supports specifying a width on the textbox and font specific attributes like size and colour.

Use case

```
<textbox font-size="14" font-family="Sans-serif" width="50">
  This text should be formatted in a somewhat decent manner.
  Hopefully...
</textbox>
```

9.1.6 CSS

The framework let you associate widgets with CSS classes. This way you can use style sheets to control all styles supported by the SVG specification.

Use case

```
<graphic ...class="paintItRed"/>
```

9.1.7 Filters

SVG filters can include a lot of different tags and attributes. As I didn’t want to cripple this functionality or add unnecessary overhead to define filters I decided that the best way to make use of them was to import them just like graphic SVG content. On top of that a widget what can be used to clone an imported filter and adapt its effects region to the current graphic target [3](ch.15.5) was implemented. The filter can then be referenced by all graphic components.

Use case

```
<svgimport svgFile="myFilter.svg" attachNode="filters"/>
<svgfilter id="filter1" srcFilterId="myFilter" x="0" y="0"
    width="100%" height="100%" filterUnits="userSpaceOnUse"/>
<graphic ...filter="filter1"/>
```
9.1.8 Artistic Resizing

To make use of the Artistic Resizing widget implemented one must first create the source images in accordance to the description in chapter 7. The files should then be named: fileName1.svg . . . fileNameN.svg, where N is the number of files. By supplying the filename, number of files, an id, and the scale value used when interpolating between the supplied source files, a new SVG image is generated according to the rules of Artistic Resizing. It is then referenced like any other SVG graphic via its id attribute.

Use case

```xml
<artistic id="refId" nrOfFiles="2" svgFile="fileName"
aScale="1.2,1.7"/>
<graphic srcNodeId="redId"/>
```

9.2 Concept GUI

The goal of the concept GUI is both to validate if TOIXSVG fulfils the requirements placed upon it and to show off the rich graphic capabilities of SVG. Because of this it makes extensive use of the features implemented in TOIXSVG.

9.2.1 Menu

The menu system is the most fundamental part of a GUI. Therefore the concept shows a menu that can handle a variable amount of selections. It also makes use of the TOIXSVG animation support (requirement R7).

9.2.2 Dynamic graphics

To test the added possibilities (in contrast to HTML) of doing dynamic graphics a simple widget that updated the background graphics according to the time of the day was created.

9.2.3 Layout

As the layout is a fundamental part of the TOIXSVG framework extensions the GUI makes extensive use of the added layout capabilities (requirement R2-R6).

9.2.4 Text handling

It displays flowing text to show of the extended TOIXSVG text-handling capabilities (requirement R9).

9.2.5 Importing SVG

The GUI is divided into separate files which are imported into the framework (requirement R10).
9.2.6 CSS
CSS is used to control graphic appearance (requirement R11).

9.2.7 Artistic resizing
Control scaling behaviour with the help of Artistic Resizing is showcased (requirement R13).

9.3 Summary
The solution described is sufficient to fulfill the requirements put on the framework. In other words it is possible to create the mock-up GUI in the framework. Now when its clear what the added functionality is, we can move on to how it has been implemented.
Chapter 10

Implementation

The first sections of this chapter describes the design and implementation of the layout functionality which is the most fundamental part needed to make TOIXSVG behave as specified in the solutions chapter. It also include a description of the Artistic Resizing implementations as it is one of the more advanced features added. Finally there is a section describing the resulting mock-up user interface built to test and validate the framework.

10.1 Overview

This section covers widgets and packets currently available in TOIXSVG.

10.1.1 Widgets

As described earlier all graphic components are based upon SvgWidget.js so that they all share the same layout behaviour. A second group of widgets are the ones which only are functional and lack a visual representation. They all inherit directly from Widget.js.

![Widget overview diagram](image)

**Figure 10.1.** Widget overview.
Graphic components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SvgWidget.js</td>
<td>Parent to all visible widgets. Contains the layout functionality shared by all graphic components.</td>
</tr>
<tr>
<td>SvgContainer.js</td>
<td>Container widget used to organize other widgets.</td>
</tr>
<tr>
<td>SvgFlowBox.js</td>
<td>Container with a flow layout pattern.</td>
</tr>
<tr>
<td>SvgHBox.js</td>
<td>Container that lays out its children on a horizontal row.</td>
</tr>
<tr>
<td>SvgVBox.js</td>
<td>Container that lays out its children vertically.</td>
</tr>
<tr>
<td>SvgGraphics.js</td>
<td>References SVG graphic in the current document.</td>
</tr>
<tr>
<td>SvgImage.js</td>
<td>References external raster images.</td>
</tr>
<tr>
<td>SvgShiftMenu.js</td>
<td>A sliding menu that can contain any SVG widget as a selection.</td>
</tr>
<tr>
<td>SvgText Box.js</td>
<td>Used to display text. Has support for rudimentary text wrapping.</td>
</tr>
</tbody>
</table>

Table 10.1. Graphic components.

Functional widgets

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SvgImport.js</td>
<td>Imports external SVG data and store it in the current document.</td>
</tr>
<tr>
<td>SvgArtistic.js</td>
<td>Sets up SVG graphic according to the rules of Artistic Resizing.</td>
</tr>
<tr>
<td>SvgFilter.js</td>
<td>Clones imported SVG filter data so it can be adapted to a specific purpose.</td>
</tr>
<tr>
<td>SvgMove.js</td>
<td>Animation which translates a widget.</td>
</tr>
<tr>
<td>SvgRotate.js</td>
<td>Same as SvgMove but rotates.</td>
</tr>
<tr>
<td>SvgScale.js</td>
<td>Same as SvgMove but scales.</td>
</tr>
<tr>
<td>SvgSkew.js</td>
<td>Same as SvgMove but skews.</td>
</tr>
<tr>
<td>SvgEffect.js</td>
<td>Contains predefined animations like &quot;shake&quot;.</td>
</tr>
</tbody>
</table>

Table 10.2. Functional widgets.

10.1.2 Packets and classes

The table below covers packets which have been created or extended with SVG specific functionality. Their classes are also described.
<table>
<thead>
<tr>
<th>Packet name</th>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toix.math</td>
<td>name</td>
<td>Contains SVG specific mathematical extensions.</td>
</tr>
<tr>
<td></td>
<td>Matrix</td>
<td>Mirrors SVG 1.1 matrix functionality in ECMAScript. Created to make porting from SVG 1.1 to SVG 1.2T easier as they differ in their Matrix interfaces.</td>
</tr>
<tr>
<td></td>
<td>Point</td>
<td>Same as for Matrix. Mirrors SVG 1.1 functionality.</td>
</tr>
<tr>
<td></td>
<td>BoundingBox</td>
<td>Extends SVG 1.1’s representation of a bounding box as a SVGRect. It can be transformed with a matrix or translated and the new bounding box would be calculated.</td>
</tr>
<tr>
<td>toix.fx</td>
<td>name</td>
<td>Has been extended with classes that allow SVG to be animated via DOM manipulations.</td>
</tr>
<tr>
<td></td>
<td>SvgMove.js</td>
<td>Class which handles an animated translation by manipulating the DOM tree.</td>
</tr>
<tr>
<td></td>
<td>SvgRotate.js</td>
<td>Same as SvgMove but scales.</td>
</tr>
<tr>
<td></td>
<td>SvgScale.js</td>
<td>Same as SvgMove but scales.</td>
</tr>
<tr>
<td></td>
<td>SvgSkew.js</td>
<td>Same asSvgMove but skews.</td>
</tr>
<tr>
<td></td>
<td>SvgEffect.js</td>
<td>Contains predefined animations like &quot;shake&quot;.</td>
</tr>
<tr>
<td>toix.svg</td>
<td>name</td>
<td>Includes a wide range of SVG specific functionality. Among the most important are:</td>
</tr>
<tr>
<td></td>
<td>Artistic Resizing</td>
<td>Handles all setup work and calculations needed to create an artistic resizing image. Also manages created images to make reuse fast and efficient.</td>
</tr>
</tbody>
</table>

Table 10.3. Packets and classes.
10.2 The layout manager

It is now time to look at the algorithms behind the extensive layout enhancements. All layout problems mentioned boils down to knowing the position and size of the graphic component we are working with. With this information and the proper algorithms we can manipulate the graphics to our liking by controlling the transformations associated to the SVG elements.

10.2.1 SvgWidget

To understand how the layout works we must first look at the most important parts of SvgWidget. The allowed layout attributes halign, valign, translate, scale, rotate and skew are all internal variables which are used to calculate the layout. In addition to them SvgWidget contains a representation of the bounding box in form of a bounding box object and a matrix object representation the current transformation of the widgets root node. The matrix object is the end result of the layout algorithm when it combines the supplied layout attributes with the bounding box of the widget. When the matrix has been calculated it is applied to the transformation attribute of the widget root node.

10.2.2 The bounding box

The solution chapter described how the size of the graphic components affects the layout. We will from now on refer to the size of the component in combination with its position as its bounding box. That is, the smallest rectangle covering the widget.

The bounding box can either be specified or calculated depending on widget type. A basic container’s bounding box is directly related to its width and height (specified or inherited). Where as a widget like SvgGraphic which references vector graphics calculates its bounding box by calling the function getBBox from the SVGLocatable interface [3](ch.4.3) on its root node. Some widgets might rely on other widgets which is the case with SvgShiftMenu. It uses its width parameter to setup its bounding box width while the bounding box height is decided by the size of the largest (height wise) child it contains.

10.2.3 Transforming widgets

As mentioned briefly in the solution chapter the width, height and position changes when a transformation is applied on a widget. This is fine as long as we update the internal bounding box of the widget accordingly. So if we modify the current matrix of the widget by applying a matrix M, we must also apply this matrix to the current bounding box. To allow this a BoundingBox class was implemented which in addition to the variables x, y, width and height, also contains the methods transform and translate (a special case of transform to avoid unnecessary calculations).
10.2 The layout manager

10.2.4 Aligning widgets

With the knowledge of the position and size of the widget we now only have to decide where it should be placed. As described in the previous chapter we want to implement constraint based layout which allow use to place it in relation to the sides and centre of the container it resides in. Again with the knowledge of the bounding box of the component and the width and height of the containers canvas this is straight forward to calculate the necessary translations. The table and picture shows the resulting translation that must be applied to the transformation matrix of the component.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Axis</th>
<th>Resulting translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>halign</td>
<td>origin</td>
<td>X</td>
<td>(-\text{widgetBB.x} - \text{widgetBB.width}/2)</td>
</tr>
<tr>
<td></td>
<td>left</td>
<td>X</td>
<td>(-\text{widgetBB.x})</td>
</tr>
<tr>
<td></td>
<td>middle/center</td>
<td>X</td>
<td>(-\text{widgetBB.x} - \text{widgetBB.width}/2 + \text{container.width}/2)</td>
</tr>
<tr>
<td></td>
<td>right</td>
<td>X</td>
<td>(-\text{widgetBB.x} - \text{widgetBB.width} + \text{container.width})</td>
</tr>
<tr>
<td>valign</td>
<td>origin</td>
<td>Y</td>
<td>(-\text{widgetBB.y} - \text{widgetBB.height}/2)</td>
</tr>
<tr>
<td></td>
<td>top</td>
<td>Y</td>
<td>(-\text{widgetBB.y})</td>
</tr>
<tr>
<td></td>
<td>middle/center</td>
<td>Y</td>
<td>(-\text{widgetBB.y} - \text{widgetBB.height}/2 + \text{container.height}/2)</td>
</tr>
<tr>
<td></td>
<td>bottom</td>
<td>Y</td>
<td>(-\text{widgetBB.y} - \text{widgetBB.height} + \text{container.height})</td>
</tr>
</tbody>
</table>

Table 10.4. Attributes for absolute constraint based layout.

![Diagram of alignment](Figure 10.2. Aligning widgets with absolute constraints. (The picture shows the widget in a state where it has been translated to the upper left corner by applying: \(x = -\text{widgetBB.x}\) and \(y = -\text{widgetBB.y}\)).)
10.2.5 The widget tree

As the declarative XML like GUI code describing the widgets is interpreted, an internal ECMAScript tree is built in accordance to their hierarchical positions. Mirroring this is the resulting SVG DOM tree where the nodes correlates to the internal ECMAScript widget tree. By taking advantage of this structure and making use of nested transformations (described in section 3.5.2) and the positioning technique covered earlier we can achieve the layout behaviour described in the previous chapter.

Declarative GUI code:                             Widget tree:                             SVG:

<container id="root" valign="middle">
    <hbox id="myHBox">
        <graphic .../>
        <menu id="myMenu" .../>
    </hBox>
</container>

<g id="root" transform="matrix(a b c d e f )">
    <use .../>
    <g id="myHBox">
        <use .../>
        <g id="myMenu" .../>
    </g>
</g>

Figure 10.3. The relation between widget structure and SVG.

10.2.6 Algorithm

Instead of a separate layout manager that manages each widgets position "from the outside", this functionality was implemented as an integrate part of SvgWidget. This way the functionality is inherited by all SVG widgets but at the same time they can override parts of it and adapt it to their needs as long as they follow its basic structure.

Initialize layout

When all widgets have been loaded the layout toix.svg.initializeLayout is called with the root widget as argument. It traverses the entire widget tree and when ascending it calls setupWidgetLayout on all widgets which are instances of SvgWidget. It is the method inSvgWidget that is the heart of the layout functionality. The reason for calling it on the way up instead while descending will become clear. Before starting to traverse the widget tree suspendRedraw [3](ch.5.17) is called on the document root node. This way no graphic is updated while the layout manager is busy, thus stopping the SVG viewer from interrupting the script execution temporarily to be able to render. When finished, unsuspendRedraw is called and everything is updated. By doing so the setup time becomes significantly shorter.
10.2 The layout manager

Figure 10.4. Parsing the widget tree to initialize the layout.

Setting up widget layout

`setupWidgetLayout` is divided into several sub-methods which are meant to be overridden instead of the method itself. The purpose is that it should act as a mould guaranteeing a sound structure while still being flexible enough to adapt to each widget’s need. Below is a step-by-step walkthrough to iron out the question marks.

```javascript
setupWidgetLayout : function()
{
1    this._overrideChildren();
2    this._initBBox();
3    this.currentMatrix = toix.svg.createSVGMatrix();
4    var gM = this._calculateGeometricMatrix();
5    this.currentBBox.transform(gM)
6    var a = this._calculateAlignment();
7    this.currentBBox.translate(a.x, a.y);
8    this.currentMatrix = toix.svg.createSVGMatrix()...
        translate(a.x, a.y).multiply(gM);
9    toix.svg.SVGTransform(this.domNode, this.currentMatrix);
}
```

Table 10.5. Function setupWidgetLayout.

1. This is the reason for doing the layout when ascending. In `SvgWidget` the method `_overrideChildren` is abstract. But some widgets like `SvgFlowBox` and `SvgShiftMenu` want to control the layout of their children. By calling `setupWidgetLayout` on the way up the children are allowed to layout themselves before the parent steps in and finish the job. Doing it in the opposite order the child would sabotage the layout set by its parent.

2. Initializes the bounding box of the widget. How this is done depends on the widget type as described in 10.2.2.

3. Resets the transformation matrix of the widget.
4. Calculates the geometric transformation that should be applied on the widget in accordance with supplied scale, rotate and skew values.

5. Updates the bounding box in accordance to the calculated transformation.

6. Calculates a translation in accordance to the alignment attributes as described in section 10.2.4.

7. Updates the bounding box in accordance to the translation.

8. Multiplies the current transformation matrix of the widget with the calculated geometric transformation and then translates it according to the the calculated alignment values.

9. Updates the actual transformation matrix on widgets root node.

10.3 Additional layout functionality

To be able to modify the layout of a widget in "runtime" extra functionality had to be implemented.

10.3.1 Update with constraints

Unlike traditional constraint systems TOIXSVG doesn’t have a dependency graph to keep track on relations between layout attributes. Instead it limits itself to the local neighbourhood of the widget, a strategy described in Ultra-Lightweight Constraints [27]. That is, constraint relations can only exist between the widget and its children, parent and siblings. This of course limits which kinds of constraints that can be expressed, but it is sufficient to achieve the functionality targeted in this thesis. TOIXSVG goes further and uses this on a widget level instead of attribute level. For example, if changes to a widget may affect one or several of its children, they are all automatically updated. As the widget hierarchy also reflects the constraint dependencies between the widgets we don’t need a dedicated dependency graph to make sure that all widgets are updated appropriately when refreshing the layout.

This is handled by the SvgWidget method updateLayout. By passing it layout parameters it refreshes the layout according to the rules described in chapter 9 and section 10.2, but with the new layout values. The work needed to achieve this is really limited and can be summed up in four steps:

1. Update all layout parameters passed.

2. Call setupWidgetLayout for the current widget.

3. If the widget is a container. Call setupWidgetLayout on all children. By doing so we make sure that constraint remains sane if the size of the container has been changed.
4. If the current widget has a parent widget which controls the layout of its children (like `SvgFlowBox`), it is called so it can update the layout of all its children as they may have been affected.

### 10.3.2 Transforming widgets

Sometimes we just want to do simple manipulations of the layout while ignoring constraints. Therefore `SvgWidget` contains additional methods for translating, scaling, skewing and rotating the widget. But when doing these geometric transformations we still want to have full control over the positioning of the widget. As these operations are applied in relation to the origin of the coordinate system, measures must be taken to control the behaviour. Therefore these methods let you specify the origin of where you want to apply the transformation. If left out, the methods make sure to apply the transformation to the bounding box centre of the widget.

**Controlling geometric transformations**

The solution is simple and standard computer graphics practise. To apply a transformation around a point \((cx, cy)\), we first apply a translation on the widget with the values \(-cx\) and \(-cy\). After that we apply the transformation and then do the opposite translation with the values \(cx\) and \(cy\). In the picture \(cx\) and \(cy\) is the bounding box centre of the widget.

![Figure 10.5. Rotating a widget around its centre.](image-url)
10.4 Artistic Resizing

The decision to implement an artistic resizing widget was based on that I wanted to investigate its usefulness in controlling advanced scaling behaviour, not that it was a necessary component of the SVG framework. The core functionality was implemented as a separate `toix.svg.ArtisticResizing` class. The following steps are done internally when setting up an instance of an artistic resizing image:

1. All files’ node trees are loaded via Ajax requests.

2. The node trees are traversed and the transformations are stored in a tree structure identical to the node tree.

3. Attach one of the node trees to the document. When doing so remove the local `defs` element and attach it to the `defs` element of the main document.

4. Calculate the new transformation matrices by doing an orthogonal interpolation between the extracted matrices based on the supplied scale values.

5. Set the newly calculated transformation matrices to the nodes of the attached tree.

6. A reference to the attached node tree and the extracted transformations are stored by the `ArtisticResizing` instance. This way another copy of the image can be created by just cloning the node tree and calculate new matrices with a new set of scale values, thus leaving out the heavy data-loading and matrix extractions. And as the `defs` element was extracted from the sub-tree it won’t be duplicated when a new instance is created.

When all this is done the Artistic Resizing image is left in the DOM tree ready to be referenced.
10.5 Mockup GUI

When the GUI is started only the top menu is visible. It contains five possible selections but only four are visible at the same time where the current selected item is the second from the left. This is indicated by it being larger than the other icons. The icons are all references to imported vector graphics. By pressing left or right key the icons slide and the selection changes. Pressing the menu key (m) results in an animation that hides the menu and deactivates all widgets. If pressed again another animation is triggered which makes it visible again.

By pressing the ok key (enter) the current selection is animated and an expression associated with that selection is executed. Described below are the action taken for the different selections. All selections that activates a non-fullscreen widget also activates a corresponding Artistic Resizing widget which makes up the border around the activated widget. The "surfer" selection toggles the resolution the GUI is rendered in.
10.5.1 Sun selection

When the sun icon is pressed a widget which adapts its appearance according to the current time is activated. It does so by interpolating colour values which are set to the gradient that makes up the sky.

10.5.2 Flash selection

The flash selection displays a container with flow layout. It contains several widgets of different sizes which all have been laid out by the flow box.
10.5 Mockup GUI

10.5.3 Videotape selection

This selection displays the text box widget which contains some "Lorem ipsum" text that has been wrapped by the widget.

![Videotape selection GUI](image)

10.5.4 TV selection

When the TV icon is pressed all widgets except menu are hidden and a TV stream dummy is displayed (a raster image with non copyrighted 'black and white' Gaussian noise).

![TV selection GUI](image)
10.6 Summary

We have now covered the more advanced modifications done to support the features described in the solution chapter. The framework has also been used to implement a simple mock-up GUI which shows how TOIXSVG was used to create a component based user interface. It is now time to evaluate to what extent TOIXSVG lives up to the requirements put on it and draw conclusions on SVG’s suitability as a technological base for STB GUI development.
Chapter 11

Evaluation

11.1 Conclusions

The following section contains the lessons learned from SVG and TOIXSVG during my work on this thesis.

11.1.1 TOIXSVG requirements

As demonstrated by the mock-up GUI all requirements placed on TOIXSVG were fulfilled, although to different degrees. The most fundamental and important requirements like the ones placed on the layout (R2-R6 and R8) were favoured on the expense of less important ones.

One important requirement that wasn’t fulfilled to the extent I wanted was that of animation (R7). The reason for this was the lack of native support in Mozilla. But the animation structure implemented should make sense when incorporated with native SVG animations later.

Text wrapping (R9) was implemented but more or less only to the extent of a primitive proof of concept. This as it is something that should be implemented natively in the SVG player.

None of the extra requirements were implemented as it wasn’t possible in the timeframe of this thesis.

11.1.2 Rich graphics

The first goal stated in the introduction was the need for rich graphics. This is something that is fulfilled due to the graphical properties of SVG. All desirable kinds of 2D graphic possibilities are there and it just a matter of using them in an efficient way.
11.1.3 Browser environment characteristics

As shown we can keep the positive time and cost characteristics available in the HTML browser environment. By fulfilling its requirements, TOIXSVG has proven that SVG can be combined with efficient component reuse to offer an attractive development model in line with modern RIA frameworks.

11.1.4 Performance

As it wasn’t possible to evaluate SVG on a STB there aren’t any hard facts to draw conclusions regarding performance. But compared to HTML which is used today it should have several advantages.

First of all, its document structure is much simpler. It only uses absolute positioning, changes to a node can’t result in changes to neighbour nodes and there are no z-values as painter’s algorithm is used. In addition to this there is the uDOM (for SVG Tiny versions) which should be more efficient. All these things should reduce the complexity of drawing graphics.

One important property mentioned in the SVG 1.2 working drafts is the addition of rendering hints [8](ch.11.20) aimed at giving more control to the content creator. By setting certain attributes the SVG player is given hints when it might be useful to cache certain parts of the graphics on reuse or when it can assume that the graphics will be modified continuously. Being able to cache large parts of the graphics would be useful when doing transition effects like fading away the GUI or making a new menu slide in.

As described in chapter 8 optimizing the content can make a huge difference. In addition to the methods mentioned, there are fundamental ones like avoid drawing several layers on top of each other as this is equivalent of rendering the same pixel several times. You should also avoid using opacity to a large extent as this also requires extra processing of each pixel (i.e. reading the memory, blending and then writing to the memory).

11.1.5 Scalability

One of the most powerful aspects of SVG is the flexibility vector graphics offer. The ability to reuse graphics with different transformations applied is very useful in user interfaces. This as you often has the graphics like icons appearing in different places with different sizes or colours. In a pixmap based environment like HTML, this use case would require one image for each unique instance of the item while SVG only would require one. This as with the inheritance of painting properties, we can combine transformations and CSS classes with use elements, allowing us to change the appearance of graphic content from instance to instance. This allows you to change the entire colour scheme of the icons in a GUI, something which in a pixmap case would require changing the colour of each icon and saving it in every resolution it is used in.

The possibility to keep user interfaces scale insensitive is especially useful during the current transition from SD to HD (and from HD ready to full HD). Instead
11.1 Conclusions

of creating different versions for each option we can now use the same GUI independent of rendering resolution.

11.1.6 Dynamic content

SVG offers two powerful tools for run-time changes of the content. First of all, the SVG DOM, which allows unlimited customizations of the content in run-time. This is the preferred method when doing updates of the layout (both inside widgets and in between). It could be used to animate the graphics as well but native SVG animations are a better choice. The native SVG animations allow all animations necessary for user interfaces like colour transitions and animated geometrical transformations. Except from setting up an animation and telling it when to start playing, the animation is played natively in the SVG player without any ECMAScript execution involved.

11.1.7 Text support

Even though text wrapping can be implemented via scripting it is a slow and inefficient approach. If SVG 1.1 is used instead of SVG Tiny 1.2 (or 1.2 Full when available) the used SVG player really should be extended with text wrapping support even if makes it non standards compliant to a certain degree. This as the amount of dynamic text (e.g. EPG data) that is displayed in a modern GUI is significant.

11.1.8 Authoring

The availability of tools for SVG user interface creation is scarce except for graphic authoring tools where there are excellent alternatives like Inkscape and Adobe Illustrator. Inkscape as a free open source alternative which could be modified to suit special needs (e.g. Artistic Resizing), and commercial alternatives like Adobe Illustrator which offer a familiar environment for graphic designers.

11.1.9 Preferred version

Among the currently available SVG versions SVG Tiny 1.2 is the best choice for the targeted purpose. Its focus on performance in combination with the extended text layout possibilities and added media capabilities [5](ch.12) makes it ideal. The only major feature it lacks is filter effects. But this functionality isn’t very useful as it is computationally heavy and many effects interesting in a GUI (like drop shadow) can be faked.

11.1.10 Artistic Resizing

Artistic Resizing can be done completely inside a SVG editor like Inkscape by copy-pasting and transforming the graphic content. When doing this you have to make sure that the SVG editor does so by changing the node’s transformation matrix instead its x and y values. Fortunately Inkscape can be configured to
do exactly this. If not available in the editor it have to be solved by doing extra post-processing where the x and y values are added to the transformation matrices.

All in all Artistic Resizing offers a way to control the behaviour of graphics without any programming, leaving it all in the hands of the graphic designer. This way graphics can be created for widgets allowing them to be rendered in any size without having to redo the graphics for each case.

11.2 Future developments

11.2.1 Layout

Many layout features implemented would be useful on a node level. By using the bounding box of the nodes controlled geometric transformations and functionality to align nodes could be implemented just like in the widget case (though extra constraint management have to be added if the constraints should update correctly according to their dependencies.). Layout functionality on a node level would be useful to control layout and graphics inside widgets, making authoring of new graphic components easier.

A possible addition would be to use Artistic Resizing for this purpose. This would allow sharing the same applications between different platforms with vastly different available screen space and have them adapt their appearance accordingly.

11.2.2 Skinning

Being able to control the layout via CSS would make it possible to rearrange the layout of an entire GUI by just changing a CSS file. In addition to this the ability change the graphical presentation of widgets by decorating them with arbitrary SVG graphics would be a useful feature. Combined these features would make the development process of a GUI more designer friendly.
Bibliography


