Master thesis

Investigation and Integration of a Scalable Vector Graphics Engine on a Set-Top Box

by

Fredrik Johansson

LiTH-ISY-EX--08/4091--SE

2008-04-22
A set top box is an embedded device, much like a computer with limited capabilities. Its main purpose is to decode a video signal and output it to a TV. The set top box market is constantly growing and to be competitive in it, a set top box has to be able to do more than only TV. One way to make an attractive product is to give it an appealing user interface. This thesis is a part of a larger work at the company to find new ways to create graphical user interfaces. Its goal is to investigate what SVG implementations that exists, which one that is most suitable for an integration attempt and then perform the integration.

Several SVG engines were investigated and one provided by the company was selected for integration. Three ways to integrate the SVG engine were identified. One of these alternatives was to extend the callback interface between the engine and the underlying platform. Because of the good fit with the current architecture this alternative was chosen and implemented. As a part of this investigation a demo application suite of SVG content was also constructed.

This investigation resulted in a working integration of the chosen SVG engine on the platform. It has also showed that SVG is a suitable language to build graphical user interfaces on set top boxes.
Master thesis

Investigation and Integration of a Scalable Vector Graphics Engine on a Set-Top Box

by

Fredrik Johansson

LiTH-ISY-EX--08/4091--SE

2008-04-22

Supervisor: Niels Bosma
Examiner: Ingemar Ragnemalm
Abstract

A set top box is an embedded device, much like a computer with limited capabilities. Its main purpose is to decode a video signal and output it to a TV. The set top box market is constantly growing and to be competitive in it, a set top box has to be able to do more than only TV. One way to make an attractive product is to give it an appealing user interface. This thesis is a part of a larger work at the company to find new ways to create graphical user interfaces. Its goal is to investigate what SVG implementations that exits, which one that is most suitable for an integration attempt and then perform the integration.

Several SVG engines were investigated and one provided by the company was selected for integration. Three ways to integrate the SVG engine were identified. One of these alternatives was to extend the callback interface between the engine and the underlying platform. Because of the good fit with the current architecture this alternative was chosen and implemented. As a part of this investigation a demo application suite of SVG content was also constructed.

This investigation resulted in a working integration of the chosen SVG engine on the platform. It has also showed that SVG is a suitable language to build graphical user interfaces on set top boxes.
Acknowledgements

I wish to thank Motorola for giving me the opportunity to carry out this thesis. Special thanks to Niels Bosma who has been my supervisor at the company and who has given me good guidance and a good background to GUI’s. Others at the Linköping office that has helped me a lot during my time there are Erik Nilsson, Robert Johansson and Thomas Hässler. I especially enjoyed the, sometimes intense, discussions between Thomas and Niels of how to best create GUI’s.

Another person at the company to thank is Andrew Dickersson. Without your help in the integration part of this thesis work, the project would probably never have been successful.

I would also like to thank my wonderful girlfriend Maja Beckman, who has supported me during the whole time. Your proofreading towards the end of this project was also invaluable. It is amazing how many times you can read something and still miss obvious misspellings and errors. For this, Dr. Charles Blake and Frida Beckman also deserves to be thanked. You certainly found your fair share of these mistakes!

Finally I would like to thank my examiner Ingemar Ragnemalm and my opponent Jimmie West. Your suggestions for improvements have made this thesis a lot better.
Contents

1 Introduction ........................................... 3
   1.1 Background ........................................ 3
   1.2 Problem Description ............................... 4
   1.3 Delimitations ...................................... 4
   1.4 Structure .......................................... 4
   1.5 Reading Instructions .............................. 5
   1.6 Method ............................................ 5
   1.7 Glossary ........................................... 5

2 Architecture ......................................... 7
   2.1 Overview .......................................... 7
   2.2 Hardware .......................................... 8
   2.3 Software .......................................... 8
      2.3.1 The main layers ............................... 8
      2.3.2 Graphical User Interfaces ................... 8
      2.3.3 The Platform’s Graphic Layer ............... 9

3 SVG and GUI’s ...................................... 11
   3.1 Graphical Systems ................................. 11
      3.1.1 Vector Graphics ............................... 11
      3.1.2 Raster Graphics ............................... 11
   3.2 SVG - an Introduction ............................ 12
   3.3 SVG Implementations .............................. 15
      3.3.1 Commercial Engine A ......................... 15
      3.3.2 Commercial Engine B ......................... 15
      3.3.3 Open Source Engine A ....................... 15
      3.3.4 Open Source Engine B ....................... 16
      3.3.5 Browser Alternative A ....................... 16
      3.3.6 Browser Alternative B ....................... 16
      3.3.7 Company Engine ............................... 16
## 4 Problem Analysis

4.1 Preferred Environment ........................................... 17
4.2 Integration ......................................................... 18
  4.2.1 Commercial Engine A ........................................ 18
  4.2.2 Commercial Engine B ........................................ 18
  4.2.3 Open Source Engine A ...................................... 19
  4.2.4 Open Source Engine B ...................................... 19
  4.2.5 Browser Alternative A ....................................... 19
  4.2.6 Browser Alternative B ....................................... 19
  4.2.7 Company Engine ............................................. 19
  4.2.8 Choice of Engine ........................................... 20
4.3 Requirements ..................................................... 21
  4.3.1 Basic Requirements ......................................... 21
  4.3.2 Additional Requirements .................................... 22
4.4 Conclusion ......................................................... 23

## 5 Design

5.1 Components ........................................................ 25
  5.1.1 The Make Systems .......................................... 25
  5.1.2 Graphics and Video ......................................... 26
  5.1.3 Scripting Support .......................................... 26
  5.1.4 Font Rendering ............................................. 26
  5.1.5 Demo Application Suit ..................................... 27
5.2 Design Concepts ................................................... 27
5.3 Design Solutions .................................................. 27
  5.3.1 Using an Adapter Layer .................................... 27
  5.3.2 Extending the Callback Interface .......................... 27
  5.3.3 Direct Modification of the Engine ......................... 29
  5.3.4 The Chosen Design ......................................... 29

## 6 Implementation

6.1 The Make System .................................................. 31
6.2 The Engine Libraries ............................................. 32
6.3 The Viewer ........................................................ 33
  6.3.1 Being a Good Citizen ...................................... 34
  6.3.2 Dealing with Input ......................................... 34
  6.3.3 The Rendering Loop ....................................... 35

## 7 Prototype

7.1 Proof of Concept .................................................. 37
7.2 Different Content .................................................. 37
  7.2.1 A Mini Portal .............................................. 37
  7.2.2 Declarative Animation Content .............................. 38
  7.2.3 Electronic Program Guide .................................. 40
8 Evaluation 43
  8.1 A Systematic Approach 43
  8.2 Shifting Focus 43
  8.3 Requirements 44

9 Further Work 45
  9.1 Video and Audio 45
  9.2 Hardware Acceleration 46
    9.2.1 With the Current Blitter 46
    9.2.2 OpenVG 47
  9.3 EcmaScript 48
  9.4 Middleware 48
  9.5 Performance 48

10 Conclusions 51
  10.1 The Importance of Content 51
  10.2 Perceived Performance 52
  10.3 An Engine with Potential 52

A Glossary 55

Bibliography 57
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A set-top box with a digital video recorder</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Platform overview</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>A SVG polygon</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>Conceptual image of an adapter layer solution</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Conceptual image of a callback solution</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Conceptual image of a direct modification solution</td>
<td>29</td>
</tr>
<tr>
<td>7.1</td>
<td>Declarative animation</td>
<td>39</td>
</tr>
<tr>
<td>7.2</td>
<td>Declarative animation 2</td>
<td>39</td>
</tr>
<tr>
<td>7.3</td>
<td>Declarative animation with animated text</td>
<td>40</td>
</tr>
<tr>
<td>7.4</td>
<td>Electronic Program Guide (normal mode)</td>
<td>42</td>
</tr>
<tr>
<td>7.5</td>
<td>Electronic Program Guide (information mode)</td>
<td>42</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter will provide the background to the intentions behind this thesis, why it was wanted by the company and which questions it tries to answer.

The thesis work was done as a partial fulfillment of a Master of Science in Information Technology (four and a half years) at Linköping University. Ingemar Ragnemalm from the Department of Electrical Engineering and Niels Bosma from the company were responsible for supervising this thesis.

1.1 Background

The IPTV market is a steadily growing and evolving market [Thu07]. To be competitive, the IPTV vendors needs to be innovative and constantly supply their end users with new attractive products. One big part of creating an attractive product is to give it an appealing graphical user interface (GUI). Traditionally these have been created in a browser environment at the company and by its customers. It has been the first choice mainly because of its flexibility and portability. This environment is, however, not without disadvantages. A big problem is that the Mozilla browser that has been used generally is slow at rendering graphics. When trying to do for example animated GUI’s, this becomes a great obstacle and limitation. Another disadvantage is the high memory usage of this browser. The desired solution is to have a way to create GUI’s that have the benefits of flexibility and portability, but at the same time is fast and memory efficient. This thesis fits into the work that the company does at looking for new ways to create their GUI’s and reach the desired solution. Trials are done with a number of techniques, one of them being Scalable Vector Graphics (SVG) that this thesis centers around. [Bos07b]
1.2 Problem Description

The company wanted this thesis to investigate whether SVG is a suitable language for creating GUI’s on set top boxes (STB). As a step in this process the purpose of this thesis is to investigate the different SVG engines available and if found possible, to integrate one of them. This purpose can further be divided into the following sub questions:

- How does a SVG engine fit into the current graphics stack? Which are the interfaces between the graphic layer in the engine and the lower layers in the stack?
- Which engine alternatives exist? How well are they suited for integration?
- Can hardware acceleration be used to faster render SVG graphics on the STB?

To answer these questions this thesis will first determine if it is possible to integrate a SVG engine into the STB platform. If so, the integration work will expose which interfaces that has to be established between the graphic layer in the engine and the graphic layers in the rest of the platform. The possibilities of using hardware acceleration will then also be discussed.

1.3 Delimitations

This thesis is not intended to provide the company with a ”ready to launch” solution. It is intended as an overview of the possibilities and limitations when it comes to using SVG as a way to do GUI’s for STB’s. From what is discovered during this work recommendations will be presented to what needs to be done in order to create a complete product. This will be presented in chapter 9.

1.4 Structure

Chapter 2 gives a general overview of the STB architecture.

Chapter 3 provides a background to graphical systems in general and SVG in particular.

Chapter 4 breaks down the problem description and discuss it in more detail.

Chapter 5 describes design choices and the chosen design.

Chapter 6 follows on where chapter 5 finished and describes the implementation of the design.
Chapter 7 is about the prototype portal that was developed in SVG on top of the SVG engine.

Chapter 8 provides an evaluation of the work done.

Chapter 9 points out how further efforts in the line of this thesis work should be focused.

Chapter 10 draws some final conclusions about the result of this thesis.

1.5 Reading Instructions

Readers who do not want to read the entire thesis back to back, but instead are interested in parts of it, are here given a few recommendations.

- Those interested solely in an overview of the subject will find chapter 2 and 3 most rewarding. Although this thesis is not aimed at teaching SVG, these chapters will give the reader a general overview of the language and how it fits into the STB environment.

- For those interested in how the SVG engine was integrated into the STB platform, chapter 5 is a good entry point.

1.6 Method

When working on this thesis I have tried to adopt an iterative approach to the development. This is true for the writing of the final report but even more so for the implementation part. Doing this has enabled me to constantly reassess risks and to improve the design and functionality in small incremental steps. The motivation for using this method came from the fact that it is risky to integrate two entirely separate softwares with each other. If it would prove to be impossible I wanted to know as soon as possible. [Wik07a]

This thesis is based on an empirical investigation. The background to this thesis is based on literature. A big part of it will however describe practical problems with my investigation and argue for different possible solutions to these problems.

1.7 Glossary

As always with a technical report this thesis will contain a lot of abbreviations. They will be explained when they are introduced. The most common ones will also be explained in appendix A.
Chapter 2

Architecture

This chapter provides an insight into the world of STB’s, or more specifically, IP-STB’s (that is STB’s connected to an IP based network). Since this investigation is done on IP-STB’s only, the term STB will hereafter be considered a synonym for the more specific term IP-STB. In this chapter a background of how the hardware and software architecture of the STB looks like will be given.

2.1 Overview

In short, a STB is a device that makes multimedia services available to a TV receiver. Although there are other use cases, for example video on demand (VoD) or voice over IP, the main purpose of the STB is to decode a video stream and display it on a TV. Looking at the STB from a technical standpoint, it is basically a small computer. With that being said, it does have some specialized hardware for dealing with video streams and other, in our case, IPTV specific tasks. [Wik08d]

![Figure 2.1: A set-top box with a digital video recorder](image)

The way the STB is designed and the fact that it basically is a computer, offers interesting opportunities for application developers. As the boxes at the company are running Linux, most Linux applications are possible to run on the box.
2.2 Hardware

This investigation was done using the Motorola VIP1920 STB which has a 300 MHz processor, 128 Mb of system memory and 32 Mb of video memory. The hardware also consists of a flash memory, allowing the STB to boot from and also keeping user configurations saved. Other special hardware for IPTV specific tasks exists, for example a digital signal processing unit (DSP) for decoding media streams. Interesting from a graphical point of view is the blitter. This special hardware makes it possible to draw (blit) to the screen without having to use the CPU. Graphic operations done with the blitter and directly to the graphics memory, is said to be hardware accelerated operations. If they are done without the blitter and instead by the CPU, they are said to be done in software. [Inc06]

2.3 Software

This section will give a brief overview of the software layer used by the STB’s at the company.

2.3.1 The main layers

The software of the STB is layered in its design. The bottom layer, counting from the hardware and up, is the hardware abstraction layer (HAL). It enables the rest of the software to run without having to be aware of the hardware on the specific box. On top of HAL lays the platform layer. This layer offers services such as the media streamer for handling video content and an application manager that controls the lifespan of applications. Although platform services are invisible to the end users, they are often the ones doing the most work. The final main layer is the application layer. All the applications that are visible to the end user resides in this layer. A schematic view of the main layers can be seen in figure 2.2. [Inc08]

2.3.2 Graphical User Interfaces

So where does the GUI’s fit into this picture? It is entirely possible to create your own applications with their own user interfaces directly in the application layer. However this is not the most common method of doing it. When creating a GUI, say a menu for handling a VoD service, this is often done on top of the web browser Mozilla. This is done even though creating them directly in the application layer would be sounder performance wise. One reason for this is that GUI creators working outside of the company are more used to working with web techniques than to create native C++ applications. Using standard web techniques also makes rapid prototyping more easy and a lot of tools to ease the content creation are also available.
So, when using the browser for creating GUI’s, looking at the layered architecture with GUI eyes we need to add a layer. The GUI’s that run on top of Mozilla are often called portals. They are scripted using standardized web techniques such as XHTML and Ecmascript. The portal gets access to platform services, such as the media player, via a custom made implementation of Ecmascript. This interface is called TOI/JS. With the browser in mind and with GUI eyes, the architecture in figure 2.2 can be seen as four layered rather than three layered (the three main layers). [Inc08]

### 2.3.3 The Platform’s Graphic Layer

An important part of this thesis has been the graphical layer of the platform. This is a library designed to replace the X window system on the company’s STB’s. X has been known to be slow and the introduction of a home grown graphic library is an attempt to provide a smarter, faster and more direct way to handle graphics on the STB. It provides support for transaction based graphics and is the interface towards which an integration of a SVG engine would be done. The layer also provides the interface towards the blitter, mentioned in the hardware section. It is crucial to mention that this graphical library is not a toolkit that can be used to directly create GUI’s. It is basically a low level layer that provides you with surfaces to draw on and makes sure that these surfaces are then displayed properly on the screen. The idea is to either create new applications directly on top of this graphical layer or port old X applications to use this library instead. To continue using the HTML/CSS/Ecmascript solution on top of this graphical
layer, a browser would have to be ported to it. [OM07]

Pixmaps

An important aspect of the platform’s graphic layer, for the implementation part of this thesis, is how it handles pixmaps. When programming against this graphical library, it is possible to declare a lot of pixmaps. It is, however, first when you lock a pixmap that a piece of memory is given to it. The graphic layer does this in order to utilize the video memory as efficient as possible. So in order to actually draw to a pixmap, you have to first declare it and then lock it. When you are done and you have committed the transaction, you should unlock the pixmap. This allows the graphical library to use the video memory for other pixmaps. [OM07]

Asynchronous Blitting

In the platform, the blitter hardware is managed in an asynchronous manner. This means that when you ask the graphic library to perform a blitting operation, you cannot be sure precisely when it is going to be executed. A consequence of this is that if you mix hardware and software rendering, you can not be sure of the order in which these operations are performed. It is done in this manner because the platform wants to have the opportunity to batch several blitting operations together into a single big one. This is simply because initiating times of the blitter hardware makes it more efficient to do a few big jobs, than doing a lot of small ones. If synchronous blitting is required a wait command on the blitter is used. A call to this method will be blocking until the blitting operation has been fully completed. [OM07]
Chapter 3

SVG and GUI’s

This chapter will provide an overview of both the foundations for SVG graphics and for SVG used in GUI’s. The overview begins with an introduction to graphical systems in general and then moves on to SVG and different implementations of this standard.

3.1 Graphical Systems

In this section an overview of different graphical systems will be given to better understand where SVG fits in.

3.1.1 Vector Graphics

In the early days of computer graphics vector graphics was the industry standard. Vector graphic systems in the mid sixties typically consisted of: a display processor, a CPU, a display buffer memory and a CRT. The buffer contained instructions for what to draw and between which coordinates. Instructions were provided by the CPU and used by the display processor to draw to the CRT. The CRT was updated through a technique called random scan. This means that the electron beam was moved between arbitrary coordinates specified in the display buffer. This way of handling the electron beam is completely different to the way it is handled by raster graphics. [JDFH95]

3.1.2 Raster Graphics

Modern computer graphic systems are based on raster graphics. In raster graphics, a central concept is the bitmap or pixmap. A bitmap is a per pixel representation in a bilevel graphical system (that is white on black or green on black for example). For each pixel on the display there are a bit in the bitmap that indicated whether if it is colored or not. A pixmap
is the corresponding construct in a modern, multiple-bit-per-pixel system. Since bilevel systems are rarely used today the term pixmap will hereafter be used although the rendering techniques are equivalent to those used with bitmaps. [JDFH95]

A row of pixels on the display is called a *scan line*. When an image is to be displayed onto the screen, the graphics hardware will do this scan line by scan line, reading the pixel values from the pixmap. The electron beam will move from left to right, line by line until it has finished scanning the entire display. After that it will retrace to the top left and repeat the process. [JDFH95]

One major reason to why raster graphics took over from vector graphics is that the hardware needed for rendering was much less expensive then the one for vector graphics. Another reason for the raster graphics breakthrough was that relatively inexpensive RAM memory became available in the early seventies. Since a pixmap stores the information about every pixel it contains, these requires more memory than equivalent vector graphics, which was then no longer a major problem. [JDFH95]

### 3.2 SVG - an Introduction

The main focus of this investigation is not the SVG language itself but rather implementations of it. However, by explaining the basics of the language and its history, the implementations are put into context.

SVG is a XML based language for describing two dimensional vector graphics. Several flavors of the specification exists which are tuned to fit different types of devices. The focus for this thesis will be *SVG Tiny 1.2*, the mobile version of the specification [Gro06c]. SVG allows for three types of objects: vector graphic shapes, multimedia (for example audio and video, or raster graphics) and text. It supports interaction via a well specified event model and content can also be animated, either via declarative animations or scripting. Further, GUI logic can be created by using EcmaScript, which is tightly integrated into the SVG specification. [Gro06b]

**Graphical Primitives**

SVG contains a set of basic shapes, or graphical primitives. These are: rect, circle, ellipse, line, polyline and polygon. Each of these SVG tags also provides the GUI creator with a set of attributes to alter the shapes appearance. [Gro06c]

An example of how a polygon is constructed is given below, and the result can be seen in figure 3.1.
Example of polygon primitive

```xml
<?xml version="1.0"?>
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.2" baseProfile="tiny">
  <desc>Example polygon01 - star and hexagon</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2" />
  <polygon fill="red" stroke="blue" stroke-width="10"
           points="350,75 379,161 469,161 397,215
                   423,301 350,250 277,301 303,215
                   231,161 321,161" />
  <polygon fill="lime" stroke="blue" stroke-width="10"
           points="850,75 958,137.5 958,262.5
                   850,325 742,262.6 742,137.5" />
</svg>
```

Figure 3.1: A SVG polygon

Document Object Model

The Document Object Model (DOM) or the similar micro Document Object Model (uDOM) are platform and language independent standards for representing XML formats. One can say that the DOM is an internal representation of the (in our case) displayed SVG. The DOM allows EcmaScript to dynamically inspect and alter the XML document, which can be used for animations or for creating GUI logic. uDOM is developed out of the regular DOM and focuses on mobile devices with limited capabilities. Therefore some of the more demanding operations of the DOM are removed in uDOM and other more efficient ways of accessing the content are introduced. [Wik08a]
Animation

SVG content can be animated in two different ways. Animations can either be done through the use of declarative animations or scripted animations. SVG Tiny 1.2 supports declarative animations through implementation of SMIL 2.1 [Gro05]. Declarative animations is simply put animation support built into the language. Scripted animations are supported through the uDOM. With access to the uDOM a scripting language such as EcmaScript can access and manipulate the SVG objects. [Gro06c] [Gro06d]

Example of an Declarative Animation

<!-- Example of an expanding rectangle -->
<svg xmlns="http://www.w3.org/2000/svg"
     version="1.2" baseProfile="tiny"
     xmlns:ev="http://www.w3.org/2001/xml-events">
    <rect x="0" y="0" height="576" width="0" fill="red">
        <animate attributeName="width" dur="4s"
                 from="0" to="720" fill="freeze" />
    </rect>
</svg>

This chunk of code first creates a red rectangle with a height of 576 pixels and a width of 0 pixels. It is placed with its top left corner on the coordinate x=0, y=0. Embedded in the rect is the animate tag. In this particular case, the animation specifies that the rect will expand from 0 pixels to 720 pixels in width during a time period of four seconds.

Scripting

The SVG language does not provide you with many facilities to program the application logic of your GUI. SVG is in its core a language in where you can draw, using graphical primitives. But with the use of XML Events one can connect events happening on the canvas to GUI logic written in EcmaScript [Gro06a]. With this mechanism a GUI can handle input data from for example a remote control. Other applications of the scripting support may be to use an AJAX solution for SVG, enabling asynchronous request to the server that delivers SVG content. [Gro06c]

Example of XML Events and Scripting

<!-- Example of another expanding rectangle -->
<svg xmlns="http://www.w3.org/2000/svg"
     version="1.2" baseProfile="tiny"
     xmlns:ev="http://www.w3.org/2001/xml-events">
    <rect x="0" y="0" width="10" height="576" fill="red">
        <handler type="application/ecmascript" ev:event="click">
            
        </handler>
    </rect>
</svg>
This example is collected from the SVG Tiny 1.2 specification document [Gro06c]. The code is an example of how EcmaScript can be connected to SVG. With SVG a red rectangle is specified, almost identical with the one defined in the previous animation example. Each time the rectangle is clicked it will expand by ten pixels. The events are captured using the mentioned event model of SVG, XML Events.

3.3 SVG Implementations

The main purpose of this section is to give an overview of the different SVG implementations that exists today and what features they support. For quite some time, SVG has been the big thing in the mobile industry. This has attracted a lot of companies to the market and there now exists several commercial SVG engines and a couple of open source alternatives. By request from the company the actual engine names has been anonymized.

3.3.1 Commercial Engine A

This engine is an engine built to specifically target the STB market. They support SVG Full 1.1 and its DOM, as well as EcmaScript. As the engine is a commercial product the source code is closed. [Ref]

3.3.2 Commercial Engine B

This engine is one of the commercial engines from the mobile industry, here named Commercial Engine B. The company behind the engine have had the approach to focus on the SVG Tiny 1.2 specification (as opposed to Commercial Engine A’s focus on Full 1.1). It implements the uDOM specification and has scripting support via EcmaScript. The engines source code is closed. [Ref]

3.3.3 Open Source Engine A

This engine consists of a library that enables applications to render SVG graphics. The source code is distributed under a free license and is thus Open source. Open Source Engine A is based upon one of the most popular 2D graphic library available to Linux platforms. It is mainly created to give SVG support for the GUIs created on one of the biggest desktop environments used on Linux and Unix. The library does not support rendering via a
3.3. SVG IMPLEMENTATIONS

DOM tree and, as a consequence, does not support scripting via EcmaScript or similar. Only static SVG can be rendered via the library. [Ref]

3.3.4 Open Source Engine B

Open Source Engine B is a Java based toolkit for applications or applets wanting to use SVG for various reasons. The toolkit is not limited to browsing SVG. It is also possible to, for example, generate SVG as a part of a export functionality in a drawing program. At the moment when writing this thesis (September 2007) the latest version of this engine claims to be a conformant static SVG implementation. Although it does not have a complete implementation of declarative animation it does support a big part of it. Open Source Engine B also supports the interactivity, linking and scripting features of the SVG 1.1 specification [Gro03]. [Ref]

3.3.5 Browser Alternative A

This browser today offers a partial support for SVG Full 1.1. Their goal is to support the full specification but they still lack support for filters, SVG defined fonts and declarative animations among other things. It does have support for all basic shapes, gradients, events and much of the DOM however. This is also a commercial product with closed source. [Ref]

3.3.6 Browser Alternative B

Browser alternative B is probably the web browser that supports the largest part of the SVG standard. It supports a superset of SVG Basic 1.1 and SVG Tiny 1.1, which maps to a partial support of SVG Full 1.1. Some support for SVG Tiny 1.2 features are also included. Although Events are supported, some events are not sent in the browser. This includes the focusin, focusout and activate events. It is possible to, via EcmaScript, script SVG content in the solution offered by Browser alternative B. [Ref]

3.3.7 Company Engine

This engine is a product supplied from another business unit of the company. Although this is a commercial engine just like Commercial Engine A and B, the fact that it is owned by the company leaves the source code available for this investigation. The engine supports SVG Tiny 1.2 and has a uDOM implementation with wrappers to both the Spidermonkey [Moz07] and kjs [Wik07b] EcmaScript engine. [Ref]
Chapter 4

Problem Analysis

The purpose of this chapter is to further break down the problem description given in section 1.2. At the end of the chapter requirements for the implementation part of this thesis is also given.

4.1 Preferred Environment

As discussed earlier the browser environment is often the preferred environment when developing applications for the STB today. It is flexible and offers portability and a rapid development cycle, but lacks in performance and is memory hungry.

What one would like is an environment in which applications could be developed in the same way as is the case for the browser environment, but with better performance and lower memory usage. The reasons for this are the ones just mentioned, the browser environment is flexible and content created on it portable (for example between the STB and a regular PC). Content creators are also used to develop with web techniques such as HTML and SVG and thus have experience with the tools that are used for this. But the current browser solution is simply too slow and too memory hungry so something new is needed.

One of the main questions of this thesis is to investigate if SVG is a suitable environment for creating GUI’s for STB’s. That means that the thesis has to explore if SVG GUI’s could have a good performance, and provide a good user experience to the end user. To test this, the implementation part of the thesis work will be to integrate an existing SVG engine into the platform.

To emphasize and once and for all clarify why SVG might be a preferred environment for GUI development some of the advantages of the solutions are outlined below:

- By using SVG we get the rapid development cycle that this environment enables.
• GUI’s should be designed, not engineered. SVG enables this by living high up in the GUI stack. [Bos07a]

• The solution is portable. SVG is a well accepted W3C standard and many middleware providers already produce content in SVG.

• The use of SVG would enable the company to smoothly transfer from creating applications in the browser environment to creating them for a SVG engine. This is because both development approaches uses W3C standards and are very similar. This is true also for other companies that are currently working with HTML/CSS/Javascript today.

• When designing and creating GUI’s, using a technique that is less time consuming, the cost for producing GUI’s are reduced. Even though this does not affect the company where this thesis was done directly it does provide them with an argument to why operators and other customers should chose their solution.

4.2 Integration

One of the goals with this thesis is to get a working integration of a SVG engine up and running on the platform. In this section, the different engines are analyzed. The analysis is based upon section 3.3. An engine to integrate is then chosen in section 4.2.8.

This analysis is done not only on a technical basis, but to also incorporate business arguments in it. This has been done to maximize the possibility that the solution will be found useful by the company. In a commercial environment, attention has to be given, not only to the technical aspects but also to more ”soft” issues as economical and sometimes the internal political decision of the company.

4.2.1 Commercial Engine A

Preliminary tests of this software showed that this is a potent engine that is able to render SVG graphics fast. It is a commercial company so the source code for the engine is closed. This is also why porting efforts are usually handled by the companies themselves. The engine both has a DOM implementation and support EcmaScript, enabling GUI developers to easily create middleware upon the engine.

4.2.2 Commercial Engine B

As with the first commercial alternative, the first tests of this engine showed that it is a capable SVG engine, although seemingly not as fast as Commercial Engine A. It is a commercial product and porting jobs are done by the company behind the engine itself. The engine supports DOM through EcmaScript.
4.2.3 Open Source Engine A

This engine does not support dynamic SVG that can be altered through Ec- 
mascript via a DOM or animated via declarative animations. This severely 
limits the GUI designer in his or her options when it comes to creating an 
appealing user interface. It is, however, a project driven in an open source 
manner so the source code is available.

4.2.4 Open Source Engine B

No tests have been made with this engine on the STB platform. The engine 
does not offer any SVG Tiny 1.2 support although it does offer close to full 
SVG Full 1.1 support with scripting and DOM interface possibilities. Open 
Source Engine B is built in Java and therefore needs a Java JRE to work. 
To use an engine on top of a Java JRE also raises concerns when it comes 
to performance and memory consumption. Generally Java applications are 
slower and more memory hungry than native applications.

4.2.5 Browser Alternative A

This browser is appealing due to the fact that the source code for the project 
is available and it is therefore possible for a integrator to have good control 
over the integration. Although it does not support declarative animations 
at the moment, it does support much of what you would like as a GUI 
developer. It has support for a DOM and it supports Events. However, 
previous experience with this browser is that it is slow at rendering graphics 
and memory hungry. [Bos07b]

4.2.6 Browser Alternative B

Browser alternative B offers scriptable SVG via a DOM interface as well as 
support for Events. It is however a commercial browser. This means that 
the integrator does not have access to the source code of the browser.

4.2.7 Company Engine

Experiences of this engine from other projects, in other business units of 
the company, shows that it has the potential of being a good engine that 
obtains most of the features one needs to create appealing GUI's for STB’s. 
The engine supports manipulation via a uDOM interface and scripting via 
EcmaScript. As it is a project maintained within the company all of the 
source code is also available. The fact that it is the company’s own project 
also makes it an appealing choice business wise.
4.2.8 Choice of Engine

The previous section was used as a base for deciding which engine to chose. The criteria’s used for the selection was:

- **Rendering capabilities**
  For the end users it is important that the engine can render an appealing GUI and that the performance of the rendering is good.

- **Implementation of a DOM**
  For the GUI developer it is necessary to have some form of DOM structure that is possible to modify on the fly. This is one of the things needed to enable a rapid development cycle.

- **Code availability**
  For the integrator this is essential. To create a port that is efficient and possible to fine tune, the source code has to be available.

- **Money**
  Although this is a technical investigation, care also has to be taken to the soft issues. If it is possible to earn more money by integrating a certain engine, this is of course in favour of the concerned engine.

From the first criteria show that Open Source Engine A cannot fulfill the requirements for this project. It is only capable of rendering static SVG and has no concept of a DOM or scripting possibilities. Previous experience with Browser Alternative A also shows that it is slow at rendering graphics and the browser can therefore not meet the first criteria. The last criteria rules out the commercial engines as well as the commercial browser (Browser Alternative B).

The final choice was between Open Source Alternative B and Company Engine. The latter was chosen partially because of the fact that it was implemented in C++ and not Java (which would have required a JRE running on the box), but also because of “political” reasons within the company. By using a product that the company owns the full rights to, the company will, upon success of this project and further development of its results, be able to make money by selling the SVG solution to operators using their STB’s.

So to summarize this section, one can say that the company engine was chosen because of its promising technical features and because it provides the opportunity for the company to make a good profit.
4.3 Requirements

This section outlines the requirements for the implementation part of this thesis work. By providing clear and detailed requirements of what the implementation part should focus on the evaluation of the prototype is much simplified.

---

R0  Example requirement

This is an example of what requirement looks like. It consists of a requirement number, a title and a description. The description includes an explanation of what the requirement actually means and sometimes the motivation behind having it.

---

4.3.1 Basic Requirements

These are the requirements that need to be fulfilled in order to be able for the prototype to be useful as a demonstration of SVG functionality in a STB environment.

---

R1  Rendering of Static SVG

The integrated SVG-engine should be able to render static SVG files as specified by the SVG Tiny 1.2 standard.[Gro06c]

---

R2  Interactivity

It should be possible to interact with the SVG content through the SVG engine via remote control and a USB connected keyboard.

---

R3  Animation

Some kind of support for animation should be provided by the prototype in order to compare rendering speeds with other engines.

---

R4  Prototype Application

A GUI written in SVG displaying the functionality of the different requirements.
## 4.3.2 Additional Requirements

The requirements listed in this section will be implemented if time permits. Roughly, these requirements are placed in descending order from the most to the least important.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R5</strong></td>
<td>DOM Manipulation through EcmaScript</td>
</tr>
<tr>
<td>The content creator should be able to use EcmaScript in order to create application logic in his or her GUI's.</td>
<td></td>
</tr>
<tr>
<td><strong>R6</strong></td>
<td>Video</td>
</tr>
<tr>
<td>A content creator should be able to play a video stream from a multicasted channel in the network using the media player in the platform.</td>
<td></td>
</tr>
<tr>
<td><strong>R7</strong></td>
<td>Font Rendering</td>
</tr>
<tr>
<td>The engine should be able to render fonts via an external font library. This is placed as a additional requirement just because of the external property. It could be that the SVG engine is portable but the font engine or font layout engine library is not. In that case it would not make sense to halt the investigation all together, but instead continue without font support.</td>
<td></td>
</tr>
<tr>
<td><strong>R8</strong></td>
<td>Advanced Prototype Applications</td>
</tr>
<tr>
<td>Enable the engine to run a more advanced prototype application. This would preferably be content provided from a third party. By using external content, the SVG engine would be useful in a more &quot;sharp&quot; environment. Instead of producing a home grown advanced prototype for this thesis, which takes time, more time can also be spent on the actual engine and the integration of it into the STB platform. This requirement depends on all the basic and other additional requirements.</td>
<td></td>
</tr>
</tbody>
</table>
4.4 Conclusion

The requirement list in the previous section was not only a mean to simplify evaluation. It also serves as some kind of summary over the first part in this thesis. Now the background of the environment in which this investigation was done is layed out and SVG is explained. The different engine alternatives have also been presented and one to integrate have been chosen. Following this background part is a description of the design and implementation of the integration. This part is in turn followed by a end phase where an evaluation is done, further work presented and conclusions are drawn. So stay tuned, there are exciting times ahead!
Chapter 5

Design

This chapter describes the considerations taken into account when the design for the practical part of this investigation was done. It starts by identifying the different components that needs work in this area. The chapter continues with laying out different ways of designing the software and finally a design is chosen for the implementation part.

5.1 Components

A number of sub components were important in the work of integrating the chosen SVG engine with the STB platform. These will be identified in this section.

5.1.1 The Make Systems

In order to integrate the SVG engine into the platform its make system had to be altered. This was necessary in order for the make system to be able to build it. It was also necessary to identify which components were needed for this. The components that were found to be needed for SVG viewing are listed below. Each item in the list is a separate shared library in the system.

1. The main SVG engine
2. The rasterization graphics library
3. A font rendering library
4. A font layout library
5. A library for EcmaScript support

How and where these components were built will be explained later in this chapter.
5.1.2 Graphics and Video

Perhaps the most important part of the integration was to enable the two graphic stacks to communicate with each other. The higher level part of the stack (the SVG engine) had to be able to talk to the lower level part of the stack (the platform). Not only is this true for the graphical part of the integration, it is also true for video and audio. To be able to show video and listen to audio playback through SVG, the engine has to be interfaced with the video and audio layer of the underlying platform.

5.1.3 Scripting Support

Section 4.1 showed that using common web standards (including EcmaScript) is a way of getting a good, flexible and powerful environment for creating attractive and functional GUI’s. Therefore, support for EcmaScript needs to be available to the SVG engine through the platform. Two alternatives were available when choosing which EcmaScript engine to use. These were KJS and Spidermonkey. [Wik07b] [Moz07] Both engines can provide the environment with the similar EcmaScript support. In this design, the Spidermonkey library has been chosen. The reason for this is that the engine already has binding to Spidermonkey plus that Spidermonkey has been used in the platform before in conjunction with Mozilla. Although it has been used before, Spidermonkey is tightly bundled with Mozilla when it is used this way. For it to work with the SVG engine, a separate and independent library has to be built.

Another advantage with building Spidermonkey this way, although it is not within the scope of this investigation, is that this would give the opportunity for a generic EcmaScript solution in the platform. This means that any application could hook up with the same Spidermonkey library to parse and execute its EcmaScript. Examples of such applications could be browsers, other SVG engines, home grown layout engines or other generic applications that could take advantage of the flexibility that EcmaScript offers. More discussion on a generic EcmaScript solution can be found in chapter 9.

5.1.4 Font Rendering

To render fonts, the SVG engine takes help from external libraries. These will have to be integrated into the platform if we want the engine to be able to render text. The engine uses two external libraries for font handling, one for the actual font rendering and another for the font layout. These two libraries connect to the engines graphic library which in turn connects to the engine itself. This setup makes it possible to use different font and font layout libraries as long as they are wrapped into the graphics library in a proper way.
5.1.5 Demo Application Suit

It was important for this investigation to create a proof of concept to show what the SVG engine integration could do on a STB. For this purpose, a demo application, or prototype, needed to be created. Although an important part of the project, it was also considered a separate part from the actual integration. Therefore, the work with the prototype has been given its own chapter, so for more information on this topic see chapter 7.

5.2 Design Concepts

When designing software it is important to keep different components decoupled. By keeping the components as separate as possible and the interfaces between them constant we allow the individual components to change but their cooperation to continue to work. That is - we encapsulate the things that change. In this specific case, that means that the SVG engine and the platform should be allowed to evolve separately but still be able to communicate. With this in mind, the design for this integration is inspired and influenced by the adapter pattern. [FF04]

5.3 Design Solutions

Before going ahead and integrating the engine, a study of how this is best done was performed. The basis for this study was the ideas presented in section 5.2. Three major designs were considered. They are all described in this section, followed up by the motivation for the final choice.

5.3.1 Using an Adapter Layer

In this design solution a dedicated class or a set of classes for handling interaction with the platform is introduced. The idea is to include layer in "engine space", that is, make it a part of the engine source code tree and its compilation procedure. With this follows that the engine source code and its make system would have to be made aware of platforms source code and headers. However, the actual intrusion in the engine source code would be small and only work as a minimal layer towards the adapter layer.

In SVG engine classes that needs to interact with the platform, an instance of this adapter class is kept and all interaction with the platform is done through it.

This design is heavily inspired by the Adapter pattern. [FF04]

5.3.2 Extending the Callback Interface

In the current design, the SVG engine already uses a callback mechanism to interface out functionality such as image decoding, resource handling, time
management and so forth. This is done to make it easy to port the engine to various platforms. At a more technical level the callback interface works this way:

- A function pointer is stored in the class that needs to use a platform callback.
- A struct exists in the SVG engine to store all the platform callbacks. This struct is a public data structure that the platform application is aware of. When initializing the engine in the application an instance of the struct is created and the appropriate function pointers in it are set.
- This struct instance is passed to the engine via a function in the engine that takes care of the callback registration. The function then copies the function pointers from the struct into the object in the SVG engine that needs the callbacks.

One solution on how to integrate the engine into the platform would be to extend this callback interface with callbacks for playing video, managing audio, doing blitter operations etcetera.

This means that extending the callback interface is a four step process:

- Add the function pointer as a member of the class that needs it.
- Add another function pointer with the same signature to the callback struct.
- In the application - add the appropriate assignment of the struct function pointer.
- In the engine function for registering callbacks - add an assignment of the objects function pointer member that reads from the supplied struct.

The cooperation between the engine libraries and the platform would in this solution, look like shown in figure 5.2.
5.3.3 Direct Modification of the Engine

The two previous design solutions presented in this section have suggested an architecture where the two softwares are kept apart by establishing clear interfaces between them. This, of course, is not the only way in which it could be done. Another way would be to make the engine more aware of which environment it lives in. We would then have to make it aware of the graphics stack of the platform. It would also have to know which interfaces that are used to play video, audio and so forth.

5.3.4 The Chosen Design

When choosing which design solution to use, several factors have to be taken into account. Some of these are:

- How well does the solution fit into the current architecture?
- How much effort would have to go into implementing the solution?
- Is the solution easy to extend and maintain?
By examining the source code of both the engine and the platform the answers to these question can be found. Both the engine and the platform are today built in a modular fashion. This makes them easier to maintain and less cumbersome to extend. By building a software system this way we also avoid the need of having to know every little detail before being able to implement new solutions within it. As long as the modules have clear interfaces of how they should communicate it is relatively easy to implement new functionality in such a system. This goes well with the ideas presented in section 5.2.

Thus, based on the reasoning above the design solution involving direct modification of the engine’s source code is deemed not to be an alternative. Still left in the match, then, is the solution with an adapter layer and the solution of extending the callback interface. For the implementation part of this thesis the decision was to go with the latter alternative. Motivating this is the first question asked in the above set of questions: How well would the solution fit into the current architecture? As the SVG engine as it is today is designed with a callback interface looking like it does, extending it would fit good into the current architecture.

A design solution as the one chosen also has the advantage of the two softwares being clearly separated. This allows the engine and the platform to grow and evolve independently. As long as the interface between them remains intact the two softwares will still be able to work together.
Chapter 6

Implementation

In this chapter, the design solution chosen in chapter 5 will turn into an actual implementation. The structure of this chapter reflects the three main parts of the implementation. First of all, everything had to be placed into the right locations in the make system. Modifications to the engine libraries were needed to allow the SVG engine to use functionality from the platform. A viewer application that used the libraries were also needed to get a fully functional SVG prototype solution.

6.1 The Make System

In section 5.1.1, a number of different modules were identified. All of these have to be placed somewhere in the STB source tree. Two questions guided the placement of the modules:

- To what extent is this module connected to the rest of SVG engine?
- Can this module be used by other subsystems in the platform?

Source code in the STB source tree is placed either as applications or platform services (see chapter 2). In this case, all the modules are of an application kind and should therefore be located among the other applications. Software is further divided into third party modules and home grown solutions. The font rendering engine, the font layout engine and the Ecmascript (SpiderMonkey) library are clearly third party modules since these are developed outside of the company. These modules will from now on be referenced as the "true" third party modules. The two remaining modules (the main engine and the rasterization graphics library) were also placed as third party modules. This decision was made because even though the modules are written and maintained by the company, they are developed at another division than the STB division. These two modules will from now on be referenced as the engine modules.
The three true third party modules were placed in three separate locations in the source tree. Here, the second question above comes into play. Each of these modules could be used by other applications in the platform. Therefore they should reside in a space of their own, to allow removal of the other parts of the SVG engine while keeping the functionality of these third party modules (in case that another application would start to use them).

When it comes to the two engine modules described above, these were put together in a single space in the source tree. Looking at it from a technical point of view, this was not necessary. Each of these modules could be built on its own, with the right settings done in the make system. However, it is very unlikely that you would need to use the graphics library of the SVG engine without using the main SVG engine library, or vice versa. These two libraries are therefore built as an entity, looking at it from the platform’s make system perspective.

Another component that needed to fit into the source tree was the actual viewer application. This application is responsible for initiating and using the engine libraries (which in turn use the other third party modules). Since this application is created solely for this investigation, and is therefore clearly home grown at the company, it was placed among the other home grown applications.

Since the true third party modules were broken out and built on their own, dependencies had to be created between the engine modules (built as one) and the true third party modules. The SVG viewer is also (obviously) dependent on the engine libraries. These dependencies were created with a standard approach taken from the platforms make system. By specifying which other modules a particular module depends on in its make file, the make system makes sure that these modules are built before the module depending upon them.

### 6.2 The Engine Libraries

The implementation of this part followed the designed laid out in section 5.3.2. Callbacks for video playback and for blitter operations were added to the callback structure. Proper modifications to the affected classes were made, that is, function pointers were added for the callbacks, and the necessary assignments of these pointers were added in the callback register function (also described in 5.3.2).

The callbacks for dealing with the pixmap that was added were:

- **FlushRegion** - Before drawing to a pixmap the SVG engine will erase previous content of that region. This callback allows the engine libraries to use the viewer (which is aware of the blitter) for this.

- **ColorBlit** - Similar to the callback described above, although in this case an arbitrary color can be blitted.
• **WaitForBlitter** - As described in section 2.3.3 the blitter is asynchronous and this poses a challenge when mixing hardware and software rendering. This callback is supplied so that the engine libraries can use the Wait command on the blitter via the viewer if hardware operations are used in the rasterization methods.

The callbacks for dealing with video that was added were:

• **OpenVideo** - Initiate a video source
• **PlayVideo** - Starts playing a video
• **PauseVideo** - Pause a playing video
• **StopVideo** - Stop a playing video

As described earlier, all of these has to be callbacks since the engine libraries has no concept of how video playback is handled in this specific platform. It is up to the platform to handle this, and the viewer (through the callbacks) is the one that communicates with this platform.

### 6.3 The Viewer

Much of the work with integrating the engine into the platform was done in the, for this investigation, created viewer application. As was described earlier, this is an application that is linked against the different engine libraries and then used functionality from them to render SVG. The viewer has a number of responsibilities, each described in this section. Along with the description of the responsibility, an explanation of how the viewer met up with it will also be given.

Before delving into the details of the responsibilities of the viewer application, its general layout will be presented. The viewer was designed in an object orientated fashion with different classes for handling different responsibilities. One class - one responsibility was a principle guiding the design. A listing of the different classes and their tasks in the viewer is given below:

• **TSVGEngine** - The main entry point for the callbacks. Also responsible for the main rendering loop and for dealing with interaction.
• **TSVGCanvas** - Responsible for drawing to the canvas
• **TSVGVide** - Handling communication with the platforms media player.
• **TSVGAudio** - Does the same for audio as TSVGVide does for video.
• **TSVGTTimer** - Helper class to handle time, used for animations.
6.3.1 Being a Good Citizen

The platform is constructed in a way so that several applications can run simultaneously. For this to work, the platform layer has an application manager to monitor and control the life cycle of the different applications. To do this, it has to know how to communicate with the applications. The way it knows this is by knowing that each application that wants to run in the platform has to obey a certain interface. That is, the main class of the application has to inherit the application interface. By inheriting this interface, the class is forced to implement certain methods. Among these methods are the ping method. This is a method that the application manager calls on a regular basis to make sure that the application is alive. If the ping method should not return, this means that the application has failed (died) and that the application manager then can clean up after the application that has failed. Also the SVG viewer has to be a good citizen. Therefore, the TSVGEngine inherits the application interface. Except for dealing with ping requests from the application manager, this interface also makes sure that the application is capable of handling interaction via the remote control.

6.3.2 Dealing with Input

One part of being a good citizen, as an application in the platform, is to handle input in an appropriate way. In the case of the SVG viewer, dealing with the input is simple. The SVG engine is the one that should be responsible for taking care of the input which means that all the viewer really has to do is to forward the input to the SVG engine via the appropriate function calls for this. More specifically, the application interface that TSVGEngine inherits (described in the section 6.3.1) forces the application developer to implement a method for dealing with remote input. Because this method is a part of the main SVG viewer application, it is aware of the engine libraries and the initiated SVG engine. The engine contains functions for reporting both key downs and key ups. The method in TSVGEngine will therefore blindly pass through, to the engine, each keycode that is retrieved. If there was a need for it, the keycode could be examined and certain codes could be handled directly in the viewer without having to alert the engine libraries. However, in this implementation there were no such need so things were kept as simple as possible.
6.3.3 The Rendering Loop

The rendering loop is one of the most important components of the TSVGEngine class. Its purpose is to keep the rendering of SVG content going. Pseudo code of the loop is given below:

Pseudo Code for the Rendering Loop

```c
while(/*should not quit rendering*/) {
    BeginGraphicUpdate();
    LockPixmapToGraphicsLibrarySurface();
    UpdateTime();
    CallSVGEngineRenderingMethod();
    CommitToGraphicsLayer();
    usleep(/*amount possible*/);
}
```

As described in chapter 2, the graphics library upon which the viewer is built uses transaction based graphics. This means that the rendering loop has to start with the beginning of a graphics update. After this, the engine pixmap is locked to the pixmap of the platforms graphic layer. This step might require some explanation.

As mentioned previously, the engine is designed in a way that enables it to easily be ported to different platforms. When designing the engine it was recognized that a pixmap structure may look very different depending on which platform that is used. So instead of trying to chose "the best" one, the engine was designed so that it could be interfaced against many types of pixmaps. This mechanism works by using a struct acting as a glue between the platform specific pixmap type and the engine. When initiating the engine, it is feed with an instance of this struct. The struct has the basic information needed to perform operations on a pixmap, including:

- Height and width
- A pointer to where the pixmap memory is located
- The row stride
- Whether it contains an alpha channel or not

A pixmap in the platform’s graphic library contains methods for retrieving all of these properties. This means that in order to supply the engine with an instance of the struct all we have to do is to extract the information from the platforms graphics library pixmap, assign this to the struct and then start the rendering loop.
Pseudo code for the ”glue struct” initiation

// The declarations of the pixmaps has previously been done
PlatformPixmap myPlatformPixmap;
EnginePixmap myEnginePixmap;

// The graphics layer of the platform
// will write to these variables
int height, width, rowStride;
bool hasAlphaChannel;

myPlatformPixmap->GetHeight(height);
myEnginePixmap.Height = height;

myPlatformPixmap->GetWidth(width);
myEnginePixmap.Width = width;

myPlatformPixmap->GetRowStride(rowStride);
myEnginePixmap.RowStride(rowStride);

myPlatformPixmap->GetAlpha(hasAlphaChannel);
myEnginePixmap.HasAlpha = hasAlphaChannel;

// continue and eventually start the rendering loop

The observant reader might have noticed that the second property described above, the pointer to the memory buffer, is not initiated above. This has to do with the locking/unlocking mechanism of the platforms graphics library. Before a pixmap is locked we can not be guaranteed that a memory buffer exist. So before we assigning this to the ”glue struct” we have to lock the pixmap. Since it is good practice to unlock a pixmap after you have used it, we have to do this lock-assign-unlock cycle on each iteration in the rendering loop. The lock-assign step is done in the LockPixmapToGraphicsLibrarySurface method. Unlocking is done in the CommitToGraphicsLayer step.
Chapter 7

Prototype

This chapter will describe how a prototype to test the SVG engine integration was developed. It will explain the motivation behind creating it as well as a description of the actual functionality of the prototype.

7.1 Proof of Concept

An important goal when choosing how to design the prototype was to make sure that it needed all the features from the requirements, described in chapter 4. As with the entire integration of the SVG engine, this prototype is not intended to be a "ready to launch" solution. It is rather built to be a proof of concept and a demonstration of what the integration is capable of.

7.2 Different Content

As mentioned above, the prototype should use all the features covered in the requirements for the SVG engine integration. This means that all the basic functionality needed to be tested as well as the implemented additional requirements. To do this in a way that felt natural, it was decided that the prototype would consist, not only of a single big prototype, but of a set of demonstration content. This also had the advantage that this investigation could use content created by the company and others right away, without having to make it a part of a home grown prototype application. Doing this allowed more time to be spent on other parts of the prototype set and on the actual investigation and integration. The rest of this section will give a description of this set of demonstration content.

7.2.1 A Mini Portal

The first part of the prototype suite was a scaled down portal (the concept of a portal is described in section 2.3.2). This portal consisted of a menu
where the user chooses different features of SVG for demonstration. The four different features was:

- TV
- Declarative animation
- Scripted animation
- Complex static SVG (SVG art)

As further explained in chapter 8 the video layer of the SVG engine integration was never made fully functional. This made the first feature impossible to demonstrate in this portal and the video window was therefore made up of a static screenshot of a video stream.

The primary objective for creating this portal was to test interaction. This was accomplished by using the remote control to choose between the different menu options (the different features to be demonstrated). Another objective with this prototype was to test more extensively the Ecmascript part of the engine integration. For this purpose, GUI logic was written for handling the menu and to take care of the "wrap around" of it (that is, when you press down in the lowest menu item, focus is set to the top option, and vice versa for the top option). Ecmascript was also used for turning on and off the different demonstrations.

Apart from these two main objectives, this prototype also tested animated content and more advanced static SVG art (the classic SVG tiger [cro08]).

### 7.2.2 Declarative Animation Content

Even though the mini portal described above did test some parts of the animation support, more extensive tests of this were required to fully assure that the declarative animation support worked. For this purpose three different SVG’s were borrowed from the company. These were all rather basic, although they stressed the implementation and were therefore suited as a kind of benchmark content.

**Bouncing Balls**

In this demonstration, four circles (the balls) were animated with regard to size and position on top of a blue rectangle. This mainly tests the speed at which the STB is able to redraw the canvas in. The content is made up entirely of declarative animations, no Ecmascript is used to control it. Figure 7.1 shows a screenshot of the animation.
7.2. DIFFERENT CONTENT

CHAPTER 7. PROTOTYPE

Stress Test

Even though this test require full screen updates as well, it adds another dimension. The content consists of a total of 2852 circles that are being animated with regard to size and color. Because of the number of circles, the DOM traversal now becomes an issue.
Halloween

This content is designed as a Halloween card. It contains a lot of animations that require full screen updates. One thing that singles it out from the rest of the animated content described in this chapter is the text. This Halloween card contains animated text, which stresses the font engine a lot.

Figure 7.3: Declarative animation with animated text

7.2.3 Electronic Program Guide

An Electronic Program Guide (EPG) is a on-screen guide to the (in this case) TV programs broadcast to the STB. The EPG allows the user to navigate through the guide selecting different times, dates, TV channels, genres and so forth. Interaction with the EPG can be performed in various ways, for example via remote control or a keyboard. [Wik08b]

In the STB industry, the EPG has become almost a holy grail when it comes to portal content. The reason for this is that an EPG contains a lot of the features that may be difficult to do when it comes to GUI creation on a STB. A common way to design an EPG is to construct it as a table. On the horizontal axis, time is represented and on the vertical, the programs are listed. This means that you have to scroll sidewise to show programs that does not fit into the current view, and up or down to view the programs of TV channels that are not currently visible. One of the challenging things about an EPG is exactly this scrolling. If the EPG is designed in the traditional way described above, a scroll means that you have to rerender almost the entire screen, which is ”expensive” and takes time. Another challenging task when creating an EPG is how to handle the EPG data. The amount of data that needs to be stored quickly grows quite
large when channels are added or the time horizon is widened.

As a part of the prototype application suite an example EPG has been created. It has been designed in the way described in the previous paragraph, as a table of information. The main objective has been to get a feel for the rendering speed when browsing through the programs (and scrolling the EPG). With this purpose in mind it has not been necessary to store and deal with data in the same way as a true EPG solution probably would have. However, it should be noted that how the data is updated when the EPG is scrolled is an important part of how the rendering speed of the SVG viewer is perceived. In this sense some considerations have been made when designing this prototype EPG. The easiest way to construct an EPG in the form of a table would be to create cells for all the TV programs that the application has data for, and then only display the time frame currently chosen by the user (and keeping the rest hidden, but still constructed by the EcmaScript). However, this would be too much data to handle efficiently. In this prototype, a compromise has been made. It will populate more cells with data than what is most often needed, but it will be within a limited time frame and will therefore handle a lot less data than the simplest approach. However, on each scroll all of this data will be updated, which takes time. A better solution would probably have been to populate cells covering three times the visible table (for the current time frame, one step to the left and one step to the right). Then, when scrolling, a translation of the group of cells would first be made (no updating of the information needed) and then when the translation is done, the cells would be updated to prepare for the next scroll. This way, the time to perform the first step, which is from the users perspective the entire operation he or she wanted to do, would only involve the actual rendering. With the solution taken in this prototype EPG (updating the information, not translating it and then update) the time to scroll is the sum of the time it takes to update the information and the time it takes to do the graphical rendering.

As described above, the prototype was designed in a traditional way, laid out as a table. Five channels were used and displayed at the same time and as a result no scrolling up or down was possible or needed. The data consisted of the TV programs for these five channels during one entire day. It contained the start and stop time for each program, its title and a shorter description of the particular TV show. A user of the EPG uses his remote control or the keyboard to navigate through it. The selected program is always highlighted and to access the short information about the selected program, the ”Ok” button on the remote (or a key on the keyboard) is pressed. This information is then presented in a window above the EPG table. The EPG is still navigatable in this mode and when a new program is selected its description is updated and displayed automatically. With a press on the ”Ok” button the EPG returns to its normal mode. Figure 7.4 shows this normal mode while 7.5 shows the information mode.
Figure 7.4: Electronic Program Guide (normal mode)

Figure 7.5: Electronic Program Guide (information mode)
Chapter 8

Evaluation

In this chapter it is time so summarize what has been done in this investigation and the outcome of it. A general evaluation of the method used and the results of this investigation will be given. How well the project meet the requirements put on it are also discussed.

8.1 A Systematic Approach

As described in section 1.6 the method for this thesis has been an iterative development model. This has allowed risks to be continously reevaluated and requirements to be adjusted in time. However, at times this had lead to a somewhat unsystematic investigation. It is the authors advice, to someone that is planning to use a method similar to the one used in this investigation, to carefully plan each iteration. Even though whole idea is to be able to alter the plan this does not mean that it is useless to create one at the first place. By always having a plan, that is finetuned or perhaps rewritten altogether at times when this is necessary, your work stays focused.

8.2 Shifting Focus

From the beginning the purpose of this investigation was, to a large extend, to compare different SVG engines with eachother. This was supposed to be done in order to provide the company with a recommendation of which engine that should be chosen. As the thesis work carried on it became clear that the focus had to be shifted. It has shifted from a pure comparison of engines to a more general investigation of how to integrate a SVG engine on the targeted platform. This also strengthens the results of this investigation. With this new focus the results are to a larger extent possible to apply on integrations of other types of graphic engines, not just SVG engines. This shift in focus was also sound in an academic view. To just list the features
of a set of engines, compare them and choosing one would not have added a lot of academic value to this thesis work.

8.3 Requirements

At the start of this thesis project, the actual requirements were at first a bit unclear. Once the initial stages of the thesis work were passed, the requirement picture grew more complete. The aim was high, but the requirements were set at a more cautious level to give the project flexibility in case of unexpected problems.

A look at the requirements in section 4.3.1 shows that all of the basic requirements of the implementation part were fulfilled. The SVG viewer is capable of viewing both static and animated SVG content and it is also capable of handling interaction via a remote control or a keyboard. A set of demo applications was also put together to fulfill the requirement of a prototype application. This requirement also works as a verification of the other basic requirements.

A big part of the additional requirements has also been fulfilled. In the solution, it is possible to manipulate the DOM tree via uDOM binding to Ecmascript. A more advanced demo application has also been created in the EPG content, described in chapter 7. However, the requirement to be able to run a more advanced prototype applications (preferably from an external middleware company), is not fully fulfilled. This is partly because of the fact that video playback via the SVG content was not implemented. The EPG has been created which is an advanced application, but it still lacks video.

Despite the fact that not all requirements were fulfilled, the prototype and investigation is a success from a requirement cover perspective.
Chapter 9

Further Work

In this chapter, suggestions for further work will be provided. A thesis work is a project with fixed and tight time frames. Because of this, there are subjects that this investigation has not been able to look into. But with this report as a basis, further work could later be made in the field. Before further work is done with the integration of the company’s engine this section should be consulted.

9.1 Video and Audio

In the current prototype, it is not possible to play a video stream via the SVG content (the SVG video tag). Preparations for enabling this have been done in the implementation part (described in chapter 5 and 6 of this thesis, but lacking support in the engine libraries made the work to integrate video a bigger challenge than what was first estimated).

The preparation made was the insertion of the necessary callbacks for video playback into the callback mechanism. These are also implemented in the viewer, so that this application is capable of communicating with the media player in the platform. What remains to be done is to implement the video layer inside the engine libraries. An implementation for the Linux platform exists, but it has to be changed so that is can be platform independent, much like image decoding and resource handling works today.

In the case of audio playback, the same work as with the video layer has to be performed. Here, the callbacks and the support in the viewer application also have to be implemented, but these will be almost identical to the ones implemented for Video.
9.2 Hardware Acceleration

One of the original ideas with this investigation was to explore the extent to which hardware acceleration was possible to use in an integration of a SVG engine. This is, however, a task that needs further work. What was done and what is left are discussed in this section.

9.2.1 With the Current Blitter

During the investigation, it was decided to focus on the integration part rather than on hardware acceleration. This decision was taken both to keep the report stringent and to manage time spent in the implementation part. However, some experiments and investigations around hardware acceleration were done. These are presented here, in an informal way, so that these results can be used by others, doing more structured experiments on hardware acceleration.

As described in section 2.3.3 and further in section 6.2, blitting with hardware on the platform used is done in an asynchronous manner. Therefore, if software operation is to be mixed with hardware operations, one has to be sure that the blitting operation is completed before continuing with software operations. When the SVG is rasterized and painted on the canvas the painters algorithm is used. In short this means that painting is done from the background, layer by layer, until the final picture is composed. Take an example where a red circle will be drawn on top of a blue rectangle and then within this circle another rectangle will be painted. Also picture that the rectangle drawing has been hardware accelerated while the circle drawing is done with the CPU (the likely scenario on this platform). When drawing this example according to the painters algorithm, the blue rectangle will be drawn first. Even though the rectangle painting is done by the blitter the SVG engine has to wait for the blitter to be ready before going ahead and doing the circle painting, for the reasons already described. Then when the circle is drawn the second rectangle can be blitted. If more content will lay on top of this we again have to wait on the blitter for it to be ready. Practically, the asynchronous blitting means that before any software rendering operation is done, the WaitForBlitter callback has to be used. If no blitting operations are queued, this call will return immediately and if there is blitting operations waiting the call will be blocked until these are performed.

The problem with this approach is that the blitter will seldom will be able to batch several jobs together but will, instead, do them one by one. Initiating times on the blitter hardware are relatively high, so a large number of pixels has to be blitted in order to do the blitter operation more efficient than the corresponding operation done with the CPU. Examples of when the blitter becomes a great asset are easily created, but harder to find in any realistic portal content. One example used when experimenting was a pulsing rectangle. This was a rectangle with a height of 576 (full screen...
height) pixels, and a animated width of between 0 and 720 (full screen width) pixels. The animation used was causing the rectangle to oscillate in width, from 0 to 720 and then back again, in an everlasting loop. To force repainting of the entire rectangle at all times a color animation was also created, oscillating the color between red and blue. When using the blitter, this example performed well and smooth rendering was achieved, even at high animation speeds (around one second for the 0-720-0 animation). The same content, but while animated using the CPU, resulted in a considerably less smooth animation. However, this kind of content is not realistic and when the blitter was used to handle rectangle rendering in a more realistic setting any performance gains could not be perceived.

Your author is pessimistic to whether the current blitter, in the setting studied for this thesis, can be used to achieve greater performance wins. The impression is rather that the blitter is a great tool for dealing with raster graphics, where a lot of operations can be batched together, but harder to utilize in SVG. However, what advantages and disadvantages there are with using the blitter will need further investigation. A better integration of the blitter in the rectangle rendering part of the graphical library is also to recommend if this track is further studied. The current implementation only blits pure rectangles, while rounded ones still are done in software. Support for the styling of rectangles, like borders and gradients, also has to be taken care of.

9.2.2 OpenVG

OpenVG is an initiative taken by a gathering of major firms, many from the mobile industry, including the company where this investigation was performed. It is a standard API for doing hardware accelerated 2D graphics. It has similarities with the 3D library OpenGL and some OpenVG implementations are even implemented on top of OpenGL. Even though it is fully possible to do a OpenVG implementation that only uses software rendering, the idea with the standard is to make it easier to use hardware acceleration. With a standard API that is supported by a large group of companies it is easier for the hardware suppliers to provide the hardware needed. [Wik08c] [Gro08]

With the current hardware and platform the only hardware acceleration that was possible to use is rectangle blitting with the blitting hardware. An OpenVG implementation in hardware would open this up and allow more graphical primitives to be drawn by specialized hardware instead of by the CPU. This would release some of the pressure on the CPU and probably result in great performance gains in the rendering step. It is therefore the authors opinion that the development of this standard is something that has to be carefully watched and, if possible, steps to get hardware into the STB’s that support OpenVG should be taken.
9.3 Ecmascript

One of the corner stones when creating portals with the current Mozilla browser solution at the company (described in chapter 2) is TOI/JS. This extension to regular Ecmascript is used to control and interact, via Ecmascript, with other components in the platform. It is implemented as a XPCOM component [Moz08]. XPCOM is a part of the Mozilla framework. When moving from using the entire Mozilla to only using the reference version of the Spidermonkey Ecmascript engine, a different implementation of TOI/JS has to be created.

It has been out of scope of this investigation to fully explore the possible options on how to create the new implementation. However, preliminary studies have shown that this is not a trivial task if one wants the same functionality from TOI/JS as is supplied today.

A major advantage with a generic Ecmascript solution like this (a TOI/JS implementation without using XPCOM) is that the bindings to Mozilla are broken. This means that another web browser could be used while maintaining TOI/JS support, home grown layout engines can be used as long as they are Ecmascript based and so forth.

9.4 Middleware

This investigation has again and again stressed the importance of the content. If the integration of the chosen SVG engine is to be continued, and then matched towards other SVG engines, the comparisons need to be made with the same content. So many tricks can be performed by smart GUI creators that it is virtually impossible to compare the "feel" of a GUI to the other. And yet, it is the "feel" that needs to be evaluated. Hardcore statistics only get you so far; important for the overall experience of a GUI is the "feel". So in order to be able to compare this, the same portals need to be developed for both the SVG engine integrated in this thesis and its competitors.

9.5 Performance

The lack of formal and structured studies on the performance of the SVG engine on the STB platform is described in section 10.2. This is something that requires further work. More formal studies of the performance of the engine libraries have to be executed. This is true both towards the competition (the commercial SVG engines) and when deciding whether or not to continue trying to use the existing blitter hardware to accelerate rendering of SVG. During this investigation more time had to be spent in the integration part than what was initially expected and this is the reason why this report does not present the reader with these formal experiments on the solutions.
performance. How fast other parts of the engine are is of course of great importance as well. The impression the user gets the perceived rendering speed, is made up of more than just the actual rendering speed in terms how fast the pixmap is painted.
Chapter 10

Conclusions

A story is not complete without an end, a wise examiner once said. This is of course true for a thesis report as well. In this chapter some final conclusions regarding this investigation are drawn. The story is rounded off with a couple of suggestions to the company on how to proceed from the work done here.

10.1 The Importance of Content

As has been obvious during the entire investigation, the importance of the content cannot be underestimated. What becomes clear when comparing different engines are the small differences between the engines and what flavor of SVG they are able to interpret and display. Some engines are standard compliant and therefore only accept SVG that is written in a standard compliant way. Other engines have implemented their own namespaces where they have added their tags, attributes and such like. This has made it hard to compare more advanced content written by external partners on different engines. The engine chosen for integration has suffered in particular in this regard since the integration was not complete and therefore could not be distributed to the third party content creators.

That content was less portable than what was expected is a disappointment. One of the great advantages of using SVG is otherwise that it is a standardized language and that a portal created in standard compliant SVG should be possible to run on different engines. However, if content creators are aware of this and knows the differences between the engines, this obstacle is not impossible to overcome. The creator can chose features in the language that are common to the different implementations. If SVG content is created and inserted into the DOM from Ecmascript, the content creation can also be conditioned on the engine that is the current target. So although it is unfortunate that content proved not to be fully portable, portals will still be able to run on different engines as long as the content creators have been aware of the differences in implementation. Another
possibility, of course, is that more pressure is put on the engine providers to create standard compliant implementations and to discourage the use of special tags.

10.2 Perceived Performance

No formal measurements on the performance of the integrated SVG engine have been made, so no hard facts will be presented here. This has to be further worked with. However, during the time of this project, informal comparisons has been made with commercial SVG engines that also has integrated against the platform. These engines have been Commercial Engine A and B. Although the comparisons has been informal and that they therefore should be taken with a grain of salt, it is the author’s opinion that the alternative of the chosen engine has good potential against these solutions. Of the commercial engines Commercial Engine A (see section 3.3.1) has performed best. It has to be admitted that right now, this engine is a faster engine than the engine supplied from the company. However, it is the belief of the author, a belief that is supported by leading developers within this SVG engine project, that steps can be taken to optimize the graphics rendering and thereby make the company’s engine as fast as Commercial Engine A.

The impression is also that Commercial Engine A consumes less CPU power and less memory than the company’s engine. But as with the observations concerning the rendering speed, this impression about CPU load and memory usage comes from informal measurements.

When talking about performance, it is important to mention content. This will be further discussed in this chapter, but in short it can be mentioned that the perceived performance is not always the true performance (of the isolated SVG engine). With smart coding of the SVG content it is often possible to create the illusion of faster rendering. This can be done by introducing timers on when to render certain parts of the content, preload parts of the content etcetera.

10.3 An Engine with Potential

A final recommendation that I would like to give the company is to keep track of how the SVG engine project within its own organization is developed. That engine has shown to be an engine with a lot of potential. Owning the source code would be a big advantage, compared to having to rely on a third party provider. The experience from this investigation is, however, that this engine currently is not the fastest one. Related to this it is also very important to keep in mind that GUI’s created for the selected engine are made in a fashion that makes them portable. If too much proprietary extensions are used the content becomes ”locked in” to that specific engine.
By creating the content to be standard compliant the possibilities to switch engine are kept open. Content wise the experience from this investigation is that SVG is a very suitable language for creating GUI’s on STB’s.

Another recommendation is to watch the development of the OpenVG standard closely. If hardware that supports this standard can be used in the STB the speed of the graphics rendering could probably be increased a lot. This would also put a lot of pressure off from the CPU.
Appendix A

Glossary

Blitter    A specialized hardware used for memory data transfers when working with bitmapped graphics.

DSP       Digital Signal Processor, a processor used to decode audio and video streams on the STB.

EPG       Electronic Program Guide, interactive TV guide.

HAL       Hardware Abstraction Layer, see section 2.

HTML      Hyper Text Markup Language, a language designed for the creation of web pages.


IP        Internet Protocol, a data-oriented protocol used for communicating data across a packet-switched network.

IP-STB    IP set top box, a STB connected to an IP based network.

IPTV      Television sent over an IP network.

OpenVG    A 2D graphics API designed to ease hardware acceleration.

SMIL      Synchronized Media Integration Language, the standard used in SVG to allow for declarative animations.

STB       Set top box, see section 2.1.
SVG
Scalable Vector Graphics, a XML language for describing 2D vector graphics, see chapter 3.

VoD
Video on Demand, a system giving a user the possibility to select a video movie to be displayed at any selected time.

XHTML
Extensible Hypertext Markup Language, a hybrid of HTML and XML designed to display web content.

XML
eXtensible Markup Language, a general markup language.
Bibliography


[OM07] Motorola Inc Oskar Malm. Graphic layer design notes, May 2007. [Internal Motorola documentation].

[Ref] Secret Reference. [Due to the non disclosure agreement signed with Motorola this reference can not be publicly displayed].


58