filtered – a tool for editing SVG filters

Master's Thesis

Kiia Kallio

Master´s Degree Programme in New Media, Department of Media
School of Arts, Design and Architecture, Aalto University

Abstract

Vector graphic image files defined in SVG format can contain filters, graphical effects that can be used for modifying the image pixels algorithmically. filtered is an open source tool for visual editing of these filters.

Although the process that eventually led to filtered started over ten years ago, filtered is still a relevant tool for SVG content development, as no other tool supports visual editing of SVG filters to the same extent. During the past ten years, SVG has also become an integral part of the WWW infrastructure, supported by all major web browsers. However, filters are still used rarely in the SVG content, as tool support for editing filters has been poor.

This is a production-based thesis. The written part describes the user interface design process of filtered. The outcome of the production, filtered software, is freely available for download from http://filtered.sourceforge.net.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope

This thesis is a production-based thesis, and consists of two parts: a computer program called filtered and this written report.

filtered is a tool for graphic artists for developing image filters for vector graphic content in Scalable Vector Graphic (SVG) format. filtered is published as an open source project, and is freely available for download at http://filtered.sourceforge.net. filtered is written in Java and uses Apache Batik programming library (http://xmlgraphics.apache.org/batik/) for SVG rendering.

The written part describes filtered and its development process. Instead of describing the whole development process – the bulk of which goes into the field of software engineering and outside the scope of Master of Arts degree – this thesis focuses mainly in the following areas:

- Defining the context of use and requirements for the software
- Designing the main user interface components
- Evaluating the usability of the interface

This introductory chapter focuses on the key concepts, contributions of the presented work, and explains the background of the project.

1.2 Key Concepts

In this section, some key concepts related to filtered are explained. Readers who are familiar with concepts of bitmap and vector graphics, image filtering, and SVG file format may skip this section.

1.2.1 Bitmap Graphics

Bitmap graphics define an image as a two-dimensional array of color values stored in the computer memory. Each of these values represents a pixel in the image. The dimensions of the memory array match the dimensions of the image.

Typically, each pixel is described by intensities of three primary colors, red, green and blue. In addition, each pixel can also contain an opacity value, known as alpha.

The color intensity value is usually represented by a number in the range of 0.0 – 1.0. The value is mapped to a sequence of bits, and the scale of the mapping depends on the number of bits used for each color channel. Typical number of bits for each channel is eight bits, i.e. one byte. With eight bits, the largest representable number is 255. Value 1.0 – the highest

intensity – is mapped to this. With four channels – red, green, blue and alpha – each pixel in the bitmap therefore consumes 32 bits, i.e. 4 bytes.

Figure 1.1 illustrates a bitmap image of 12×8 pixels. At four bytes per pixel, this image would consume 12×8×4=384 bytes of memory.

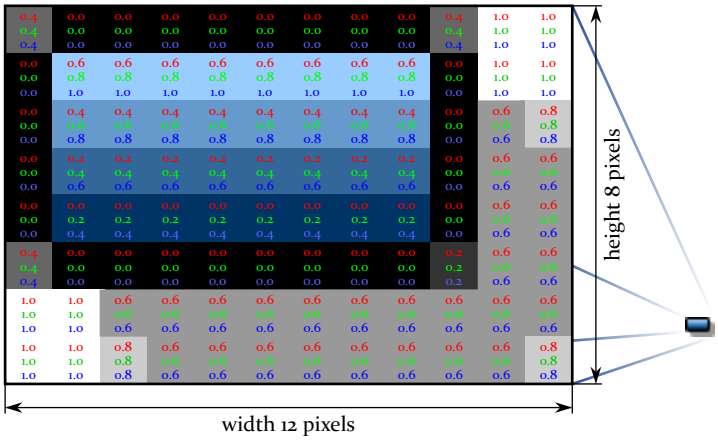


Figure 1.1: A bitmap image.

Bitmap images tend to consume large amounts of memory, as the memory consumption depends solely on the dimensions of the image and number of bits stored per pixel. Bitmaps are therefore often stored in compressed formats such as PNG [27] or JPEG [24].

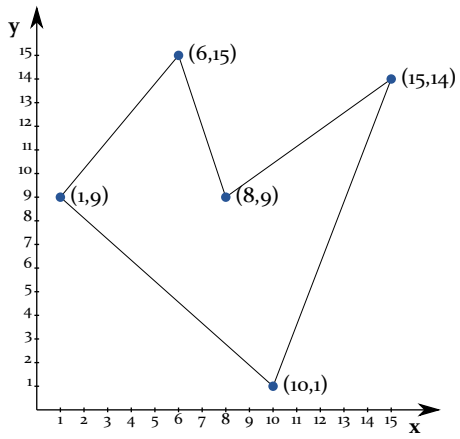
Computer display hardware reads bitmap data from the computer memory and forms the image on the screen. All images viewed on the computer screen are therefore stored as a bitmap at some stage.

1.2.2 Vector Graphics

Bitmap graphics define an image as a two-dimensional array of color values. In vector graphics, images are defined as a sequence of mathematical expressions. These expressions can be viewed as instructions for constructing a bitmap image.

The image data in vector graphics typically consists of sequences of 2D points, and instructions related to the points. Figure 1.2 illustrates a simple shape and commands defining the shape. The example uses only lines; the vector graphic formats typically include also other primitives such as curves and arcs.

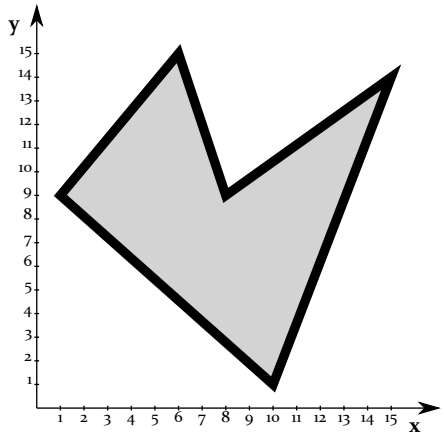
The plain mathematical definition of the shape outline is not enough for generating a visual representation of the shape. Various visual attributes are needed as well, for instance color for the filled interior of the shape and thickness and color for the outline stroke. Often the vector graphic rendering systems offer a variety of paint alternatives in addition to solid colors, such as gradient paints. Also stroke can be defined with a wealth of attributes, for instance strokes can have dash patterns and there are various ways for handling the joins and ends of the strokes.



instruction	x	y
move pen to	1	9
draw line to	6	15
draw line to	8	9
draw line to	15	14
draw line to	10	1
draw line to	1	9

Figure 1.2: A simple vector graphic shape.

Figure 1.3 illustrates the example shape with rendering attributes.



attribute	value
fill color	lightgrey
stroke color	black
stroke width	2 mm

Figure 1.3: Fill and stroke definitions of a shape.

Vector graphic files consist of a collection of vector graphic shapes. Typically, the shapes in the file are arranged to groups of shapes. This way the artist can build elements in the image from a collection of shapes. When e.g. moving such a group, all shapes in the group are moved together. Groups may also contain other groups, thus forming a hierarchy of graphical elements in the file.

Transformations such as moving, rotating and scaling the shapes are often also defined as instructions in a vector graphic file. When an artist moves a shape in a vector graphic editing tool, the coordinates in the shape definition are not altered, but a transformation instruction is added to the shape description or the group being edited. Transformations can usually be animated to create movement of shapes and groups – provided of course that the vector graphic file format supports animation in the first place.

Figure 1.4 illustrates a sequence of shape transformation instructions.

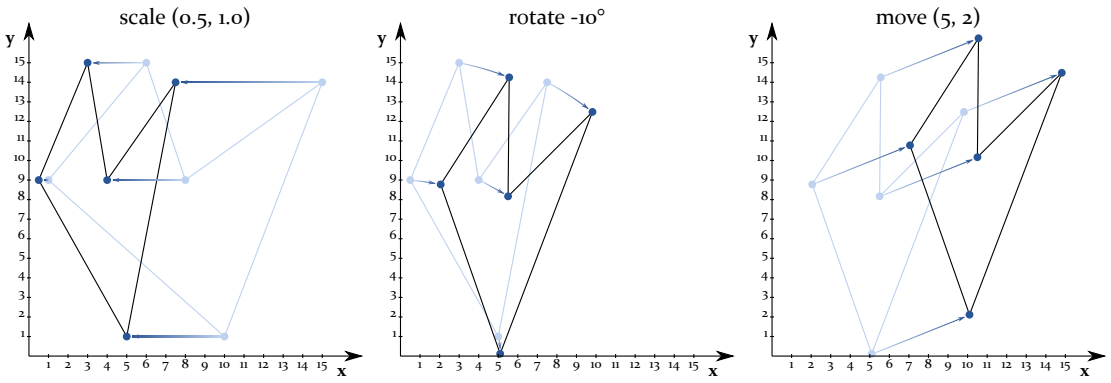


Figure 1.4: Transformations of a shape.

The storage needed for vector graphic images can be considerably smaller than with bitmap images. For instance, the example shape is defined by a sequence of six entries, each consisting of an instruction and two coordinate values. Assuming 1 byte for the instruction, 4 bytes for each coordinates, and 4 bytes for each of the three rendering properties, the result storage size for the example shape would be (1+4+4)×6+4×3=66 bytes. This is just a fraction of the storage needed by a tiny 12×8 bitmap image in figure 1.1.

Vector graphic images are converted to bitmap images for display in a process called rasterization. Rasterization algorithms process the mathematical description of the shape and set colors in the destination bitmap according to the rendering properties of the shape. Modern rasterization algorithms also perform antialiasing when rasterizing the shapes [30], thus avoiding stair-stepping artifacts sometimes visible in bitmap graphics.

Vector graphic images can be rasterized to a bitmap in any size or zoom level by applying a suitable scaling transformation to the whole image. This means that vector graphic images are always utilizing the full precision of the display, and do not become blurred or pixelated like scaled up bitmap images. This means that vector graphics are resolution-independent and can be shown across a range of various devices.

The rendering model of vector graphics is based on seminal work by Warnock and Wyatt published in 1982 [48], and has been since then incorporated into various standards and programming interfaces such as PostScript [1], PDF [28], PostScript Fonts [2], Flash [4], SVG [16] and OpenVG [43].

Resolution-independent rendering has made vector graphics popular especially in the printing industry. In addition, Adobe Flash has been a popular format in the web because of its small file size and scalability.

1.2.3 Image Filtering

Image filtering is a central concept in filtered. Image filters operate on bitmap images. Even in the context of vector graphics, the vector graphic images are rasterized to bitmap images for viewing. Once the vector shapes have been rasterized to bitmaps, it is possible

to apply image filters to them.

Image filtering is a process where pixels of one or more input images are processed by an algorithm, and a new value for each pixel in the output image is calculated based on the inputs. Image filters can also be generators, in which case they don't need any input images and produce just an output.

The filtering algorithm may access just a single input pixel for each output pixel, or a larger set of pixels. For instance, a simple blur filter could calculate the average of each input pixel and its eight neighbors, and write the result to the output pixel. Another example is a darkening filter that reads in a single pixel, multiplies its color value by 0.5 and writes the result out.

Figure 1.5 illustrates the filters explained above.

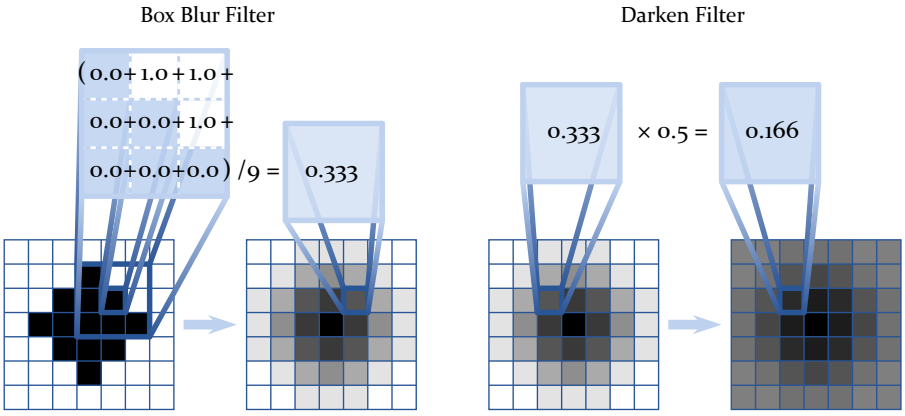


Figure 1.5: An example of two simple image filters.

Typically, filters have a set of parameters that can be used for adjusting the effect of the filter. For instance, the simple blur filter in figure 1.5 could have a "size" parameter which defines the size of the blur, i.e. how large area of input pixels it accesses when calculating the effect, and the darken filter could have a "multiplier" parameter that allows also other darkening values than 0.5.

Filters can be combined by executing another filter on the output image of the previous filter. This way a set of basic filter primitives can be used as building blocks for more complex filter operations. For instance, the sequence in the figure 1.5 first blurs the image and then darkens the result.

1.2.4 Scalable Vector Graphics (SVG)

Scalable Vector Graphic (SVG) [16] is an open vector graphics file format endorsed by World Wide Web Consortium (W3C). SVG is based on Extensible Markup Language (XML) [13], and is therefore compatible with other XML-based web formats.

SVG has been around since 2001 [21], and it has been gaining more momentum during the years. It is one of the core web image technologies, being integral part of the new HTML5

specification [10], and is supported by all modern web browsers.

SVG has a thorough imaging model supporting gradients, transparency, pattern fills, antialiasing and various other rendering features required by modern vector graphics applications. SVG files can be interactive and contain animation. SVG is scriptable with ECMAScript (JavaScript) [19] and has a document object model (DOM) [23] accessible from other languages such as Java. SVG also has support for image filters.

Similar to other XML formats, SVG files are text files and can be edited with a text editor, but usually files are edited with a vector graphics editor, such as Adobe Illustrator or Inkscape. As text-based files, SVG files can be verbose, and therefore a gzip-compressed variant, SVGZ, is defined in the standard. For mobile use, there are two subsets of SVG, SVG Tiny (SVGT) [6] and SVG Basic (SVGB) [20].

1.3 Contribution

Traditionally vector graphic images have a distinct 'clean' visual style, and in this respect limit the possibilities of artistic expression.

Vector graphic files in the SVG format can contain filters, graphical effects that can be used for modifying the image pixels algorithmically.

Filters can be used for achieving a richer visual style. They can be used for e.g. imitating natural materials such as wood and rock, adding dirt, wear and tear, adding shadows and lighting effects and for imitating painting and drawing materials such as ink or water color.

SVG content creation tools have limited support for defining and editing SVG filters. In practice graphic artists haven't been able to utilize the possibilities of SVG filters, as editing them has been difficult, typical tool for that being a text editor.

filtered enables a richer visual style in SVG-based vector graphic images, as it empowers the artists with a graphical user interface (GUI) for defining and editing filters in SVG files. This kind of visual interface for filter editing is missing from the current selection of vector graphic content creation tools. filtered fills this gap.

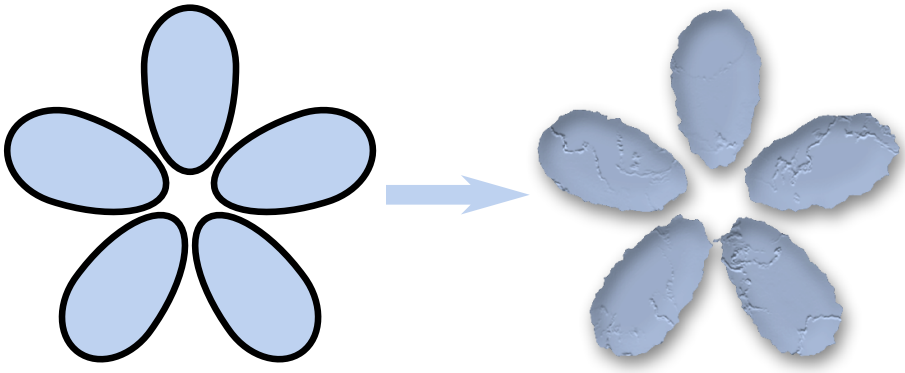


Figure 1.6: An example of a vector graphic image with filters applied.

Figure 1.6 displays a plain vector graphic image on the left, and a version of the same image with filters applied on the right.

SVG is one of the core web image technologies, and is supported by all modern web browsers. However, SVG filter support in web browsers is still relatively weak. Since there haven't been any proper content creation tools for SVG filters, there's no web content pushing the browsers to comply with the SVG standard. filtered can drive also the quality of SVG implementations in the web browsers forward, as artists can use it to create content that utilizes SVG filters to the full extent.

1.4 Project Background

The development of filtered originally started during years 2002-2003, under the title "SVG Filter Editor". During the years 2012-2013, the program was reconstructed to what is now known as filtered.

Originally, I developed "SVG Filter Editor" for a mobile game startup called Fathammer. I never completely finished the tool – nor my studies – during my time at Fathammer, and the software was forgotten for years. However, I did write an article for SVG Open 2003 conference. This paper is available at http://www.svgopen.org/2003/papers/UsingSVGFor2DContentInMobile3DGames/index.html, and in order to preserve it in the case the web site gets updated or disappears, it is included also here as Appendix C.

The original motivation for the tool was to allow creation of high-quality texture images for mobile 3D games with tightly restricted storage space. Vector graphics can be stored very tightly, but plain vector graphic images often have a distinct 'clean' look that is not suitable for 3D games aiming at realistic visual style. Various texture generation techniques such as procedural textures or texture synthesis on the other hand can produce more realistic texture images, but with these techniques, the texture artist has only a limited control over the result image.

One practical technique for texture generation is to define the image as a sequence – or a graph – of image filtering operations. This approach allows combining shapes drawn by the artist with generated image content. The content created by the artist can be in vector format as well. This technique can be used for combining accurate artistic control of vector graphics with the richer visual style of texture generation, while maintaining the small file size of texture images.

The initial idea for the project was to develop a proprietary image format that has vector graphics rendering capabilities along with a basic set of image filtering operations. However, SVG vector graphics format already had a set of image filters (in addition to a wide selection of vector graphics capabilities). Even if the set of filter operations was not optimal for texture generation, it was good enough for the purpose. Using a common standard instead of a fully proprietary format made more sense, as it allowed e.g. wider selection of content creation tools.

However, SVG file format [16] is complex to process, and has dependencies to a vast amount of other technologies, such as XML, Cascading Style Sheets (CSS), Synchronized Multime-

dia Integration Language (SMIL), JavaScript) etc. This means that the infrastructure required for processing SVG files is relatively heavyweight, leading to large application program file sizes. Today, this entire infrastructure is available as a natural part of the web browser – or even the operating system – but in 2003, the situation was different, and the game application had to contain all this functionality. This was prohibitive from the point of view of the delivery channels, as the application size was strictly limited.

In order to solve this problem, we developed a compressed binary file format that removed the dependencies to external technologies and allowed small executable file sizes. Files stored in XML-based SVG format were converted to this proprietary format using an offline conversion tool. Naturally, we also implemented a C++-based vector graphics rasterizer with full SVG feature set in order to handle the files on the device. This part of the framework is completely left out from the discussion of this thesis, as it is proprietary technology developed for Fathammer.

1.5 Project Revisited

The original goals for developing an editor for texture creation in mobile games are hardly relevant as of today. First, the storage requirements for mobile games are totally different – for instance total application size limit for Apple's iOS devices in 2 GB, and for over-the-air downloads 50 MB [7, p. 208]. This makes using bitmap textures much more viable option. Second, efficient usage of SVG content creation tool chain requires appropriate SVG rendering capabilities on the device. These were provided by the game engine developed at Fathammer, but this engine is no longer available. If SVG files were to be used for textures in mobile games today, some modern game engine should first provide support for SVG rendering.

However, I still see the open source release and further development of filtered justified in the year 2013. There are areas of visual design where a tool like this can be valuable. Below are listed some of them.

Web design

During the past 10 years, SVG has become an integral part of the web infrastructure, and it is supported by all modern web browsers. When SVG content creation tools still lack adequate capabilities for editing filters, filtered is a nice addition to the toolbox of any web graphic designer working with SVG content.

Smaller file size of SVG files is still a benefit over bitmap files when making responsive web design or mobile web pages, although the file size is not such a critical issue in this use case.

However, SVG has also several other benefits in web page environments. In addition to being scalable to various display sizes, SVG has fully scriptable XML DOM, allowing complex interactions with the HyperText Markup Language (HTML) content. SVG files can use CSS styling – considering e.g. the color schemes and other visual elements – and thus allows constructing content that can be easily adjusted and configured to follow specific visual style.

Despite the benefits, web designers may not still want use vector graphic images, as they don't necessarily suit the desired visual style. filtered enables larger variety of visual styles by making SVG filtering functionality accessible for designers.

Texture creation

Originally, the main benefit of SVG-based textures was the compactness of the vector-based data. It is however still possible to use filtered for offline creation of bitmap textures, as using SVG-based tool chain for texture creation does have other benefits as well:

- Resolution independence. SVG textures can be rendered to very high resolution to be used e.g. in movies.
- Automation support. SVG files can be created and modified by scripts, thus allowing creation of texture collections with just slight variation in each. (Snowflakes! Tree leaves! All different!)
- Possibility to cross-reference images so that a single SVG source is used for creating all texture variants needed by the shader program, e.g. color maps, normal maps, displacement maps etc.

filtered can therefore still have value in 3D production, both in 3D games and in movies.

There are multiple tools for texture generation on the market today. Most successful ones are probably Substance Designer (http://http://www.allegorithmic.com/products/substance-designer), Genetica (http://www.spiralgraphics.biz/genetica.htm) and Filter Forge (http://www.filterforge.com).

These solutions are definitely superior feature-wise when compared to filtered, as they all have a large collection of proprietary filter primitives to use for texture generation. However, the fundamental difference with these solutions and filtered is the approach to use an open data format. Since filtered sticks to SVG with all its functionality, it allows construction of a content creation tool chain that can be scaled to large productions where tool integration, automation and content revision management are important factors.

Even if filtered doesn't directly support editing of vector shapes, it is meant to be used as a companion of a vector graphics package. This allows superior control when combining content drawn by the artist with generated image content, and blending of these two.

Photo effects creation

In addition to texture creation, filtered can also be used as a tool for creating effects in digital photographs. The filters operate identically on bitmap and vector image data, and therefore it's possible to load up bitmap images to filtered as well – although currently it requires wrapping the bitmap in an SVG file.

Although there are excellent tools for this, such as Adobe Photoshop, filtered has slightly different usage philosophy that has advantages in certain use cases. As SVG defines filters as a set of filter elements each with different parameters, the filters are never permanently

applied to the bitmap layer, freezing its pixel color values. All filter primitives remain editable all the time, and any changes are reflected to the output image. This allows more freedom in tuning the filter parameters for a specific image.

The benefits of automation and scripting apply also to photo effects creation. Although currently not in the scope of filtered, it is also possible to animate all properties in SVG files, thus making it possible to create animated filter effects on series of images.

2 Design Process

This chapter briefly describes the theories behind user interface design process, and the process used in the project.

2.1 Design Process Theory

The design process used in the development of filtered was influenced by various sources.

Figure 2.1 illustrates the design process according to ISO-13047 standard "Human-centred design processes for interactive systems" [25].

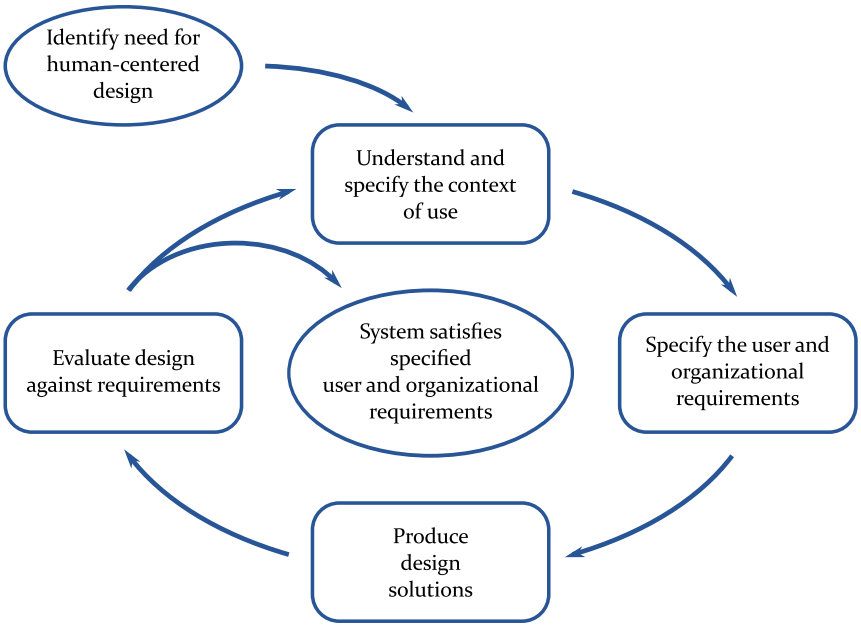


Figure 2.1: Human-centered design process according to ISO-13407 standard.

In "Task-Centered User Interface Design" [31], Lewis and Rieman describe the steps in the task-centered design process:

1. Figure out who's going to use the system to do what
2. Choose representative tasks for task-centered design
3. Plagiarize
4. Rough out a design
5. Think about it
6. Create a mock-up or prototype
7. Test it with users
8. Iterate

9. Build it
10. Track it
11. Change it

In "Usability Engineering" [38], Jacob Nielsen describes the stages of usability engineering life cycle model:

1. Know the user
    a) Individual user characteristics
    b) The user's current and desired tasks
    c) Functional analysis
    d) The evolution of the user and the job
2. Competitive analysis
3. Setting usability goals
    a) Financial impact analysis
4. Parallel design
5. Participatory design
6. Coordinated design of the total interface
7. Apply guidelines and heuristic analysis
8. Prototyping
9. Empirical testing
10. Iterative design
    a) Capture design rationale
11. Collect feedback from field use

All these examples – as well as most of the other literature – emphasis few key issues: knowing the user, iterative design process, and user testing.

However, there are other aspects in design in addition to these approaches mostly based on principles of usability engineering. In "Design Thinking" Nigel Cross points out that attempts to impose technical rationality on design processes have failed, as cognitive processes of design are different from purely rational mental processes [15]. Still, some successful designers he uses as examples in the book are also highly skilled mechanical engineers: mindsets of designers and engineers do not exclude each other out but can also be complementary.

2.2 Design Process in Practice

The development process of filtered was not implementing any interface design process strictly. The design goal was constrained heavily by the technical requirements of mobile game content creation, and the targeted user group was restricted to a specific audience, game graphic artists.

The implemented design process consisted of following parts:

1. Defining the users and context of use
2. Defining the technical requirements

3. Analyzing the existing solutions
4. Initial design
5. Implementing the tool in an iterative process
    - Designing
    - Implementing
    - Evaluating
6. Releasing the tool in the Internet
7. Collecting feedback from the users

Stages 1-4 of the design process were done in 2002. Stage 5, the actual implementation, was initially done in 2002-2003, and then repeated in 2012-2013. The last two stages were done in 2013, and are still in process.

Initially the attention in the process was focused on searching creative ways to solve the technological challenges, and focus was shifted on user interfaces design when a satisfactory solution was formed. Although the first part of the process can be viewed purely as solving an engineering problem, it had a lot in common with design processes; lots of alternative solutions were explored and the final solution was a synthesis of the results of the exploration.

2.3 Design Process and Agile Software Development

The software development process of the filtered was based on agile software development processes [9], most notably Extreme Programming [8] that was gaining large traction back in 2002.

Unlike traditional waterfall design of software, agile processes implement iterative software development process. This process can be interleaved with an iterative user interface design process. [12]

As this was a one-man project, and I was doing both the interface design and software development, this interleaving of processes was even easier. There were some practical impacts of the agile software development process on the design process; most notably the need for non-functional prototypes was diminished, as implementing a functional prototype was possible within the same schedule.

This thesis focuses on the interface design process rather than software development process, and therefore a detailed description of the software development process is considered to be outside the scope of the thesis.

3 Context and Requirements

This chapter documents the process of the requirement definition. The requirement definition was done in 2002, with the goal of producing a technique and tool chain for texture generation in mobile 3D games. Although this goal is no longer relevant regarding the use cases of the resulting tool, it is important part of the development process and thus described in this chapter.

3.1 Users and Context of Use

In 2002, I worked for a company called Fathammer. Fathammer was a mobile startup founded in the year 2000, building 3D game technology and 3D games for new mobile devices emerging at the time.

Up to that point, mobile games had been very simple and technically restricted "casual games". New personal digital assistants (PDAs) and cell phones with large color displays and operating systems supporting native C++ applications were emerging back then, and the founders of Fathammer saw the technological opportunities in the new environment The background of the people behind Fathammer was from the personal computer (PC) and console game business. These people saw that the mobile devices could become a platform for games visually and technologically similar to PC and console games, and decided to build a game engine for licensing – along with a portfolio of showcase games.

3.1.1 Context of Use

The intended customers for the product of Fathammer, X-Forge™ game engine, were "serious game studios". The goal was to enable those studios to scale their development from high-end PC and console 3D platforms towards emerging low-end platforms.

The game engine included also a suite of tools, containing e.g. a 3D content exporter and a 3D particle system editor. The target for the suite of tools was to enable a content creation tool chain similar to those suites the game artists had been used to have when developing content for PC and console games. The content creation tools suite was targeting Windows-based PCs, as those were – and still are – the dominant platform in game development.

3.1.2 Users

The targeted users for the software were professional game texture artists. Based on my experience from game industry (since circa 1994), graphic artists working in game development are not only highly skilled in arts, but also very knowledgeable in game technology.

For this kind of user demographic, the challenges in the user interface design were not in trying to simplify the design by restricting available technical choices, but in structuring the interface so that it enables the artist to build appropriate conceptual model of the process. Providing a good conceptual model is a fundamental principle of designing for people. [41, p. 13].

As noted by Edward R. Tufte in "Envisioning Information" [47, p. 51]: "Simpleness is another aesthetic preference, not an information display strategy, not a guide to clarity.''

In order to be able to have control over highly technical process, the artists had to be enabled with accurate control over the minute details of the process, while being able to operate in a visual editing environment different from text-based tools programmers are used to have for controlling processes like this.

3.2 Technical Requirements

3.2.1 The Problem Setting

In 2002, high-end mobile devices of the time, typically PDAs and high-end cellphones, were becoming a new platform for 3D games.

These devices had processing power comparable to PCs of mid-1990's. They had memory in the range of 4 MB – 32MB – including storage, and small color displays, with resolutions ranging from 176×208 to 320×320. The display controllers of the devices were typically minimal, with no hardware accelerated 2D or 3D rendering capabilities.

Within these limitations, it was still possible to create 3D games that competed with the quality of console and PC games of the 1990's. Development in tight mobile environment required however different approach from the methods used in desktop or console systems.

The physical design of the devices along with the varying usage situations gave some new challenges to the game design. As the primary use of the devices was making phone calls, the controls for playing games were often far from optimal. The size of the display and its poor resolution were also limitations. From the technical point of view however, the lack of memory and storage space was the biggest problem.

Prior to the arrival of compact disk read-only memory (CD-ROM), storage was also a problem for PC and console gaming. As one CD-ROM can store approximately 700 MB of data – almost 500 times the capacity of the prior standard, 1.44 MB floppy disk – arrival of CD-ROM removed the storage problems. One of the most successful systems using CD-ROM storage was Sony PlayStation, released in 1994 [44]. On desktop PCs, CD-ROM quickly became the standard media for distributing games in mid-1990's as well.

In respect to the 3D rendering quality, mobile devices of 2002 were rivaling the PlayStation. In the amount of content however, the situation was even worse than it was in the times before CD-ROM. Most mobile devices for instance didn't have any removable media whatsoever. No diskettes or cartridges – let alone anything as massive as CD-ROM.

The games were crammed to the device memory along with all the other applications and data, and there should have been some memory left for running the applications as well. For devices with as little as 4 megabytes of total memory, this was clearly a problem. For comparison: PlayStation had total of 3.5 MB of runtime memory (2 MB main memory, 1 MB video memory, 0.5 MB audio memory) [45] devoted for one single game at a time, and the whole CD-ROM for storage. Clearly mobile games required some clever strategies to overcome the situation.

About half of the content in mobile 3D games made by Fathammer was 2D graphic content, one third was 3D models and the rest was mostly audio – see appendix C for more accurate figures. The 2D content consisted of in-game textures and user interface (UI) graphics. As increase in display resolutions was also anticipated, the relative quantity of 2D graphic content was projected to be on the rise.

Therefore, it was obvious that reducing the size of the stored 2D content would result in biggest savings in the overall storage space consumption.

3.2.2 Possible Solutions

Natural solution for the problem with storage space was to seek methods for storing 2D graphics more efficiently. However, as visuals have very important role in games, the solution should neither degrade the image quality nor set limitations for artistic expression in content creation. Usability of the content creation tools was also important; artists are not programmers, and too complex implementations or tool chains slow down the work and cause confusion.

There are two important use cases for 2D graphics in 3D games: user interface graphics and in-game textures. The solution was required to be such that it can be used for both of these. The requirements for reduction of file sizes were also radical; several hundred megabytes of bitmap graphics should have been stored into just a few hundred kilobytes.

Compression

There are some methods developed in the computer graphics that were considered as a solution for the problem. First and most obvious was heavy image compression. The X-Forge™ engine did not support JPEG [24] or other lossy compression formats at the time. Implementing JPEG or wavelet-based compression would have allowed tighter compression rates for the images. However, this approach could have resulted in problems with image quality. In textures some compression artifacts may have been acceptable, but not in the user interface graphics. Compression also has its limits, lossy compression may well reduce the file size to half when comparing to lossless compression, but for more extreme results, compression was not an option.

Texture generation

There are various techniques for generating textures algorithmically, such as procedural texturing and texture synthesis.

In computer science, adjective *procedural* is used for entities described by program code rather than by data structures. "Procedural texturing" is a term covering a variety of techniques for generating textures with procedural techniques. Principles of procedural texturing are well covered in the book "Texturing and Modeling: A Procedural Approach" [18].

"Texture synthesis" on the other hand refers to a set of methods that start from a sample image and attempt to produce a texture with a visual appearance similar to that sample. A review of various texture synthesis techniques can be found in the report "Concise Texture Editing" [14, p. 34-46].

These techniques use algorithmic methods for creating images based on some parameters. The result images are especially suitable to be used as textures representing natural phenomena.

There are various methods for generating textures procedurally, and in general these methods have a wealth of benefits: the texture definitions are extremely compact, there is no fixed resolution, there is no fixed size for the texture, and textures can be parameterized for generating variations of the same texture. However, procedural textures have also disadvantages: they are difficult to build and debug and may create unpredictable results. The results may look unnatural and mechanical as well. Each procedural algorithm can produce only a limited range of images, and to get satisfactory results, a large collection of generation algorithms is required, which just moves the storage requirements from the content to the executable application binary.

There is also wide variety of methods for texture synthesis. The typical problems with texture synthesis are related to the performance and complexity of the analysis and synthesis algorithms. Texture synthesis methods are also usually restricted to a specific class of textures, typically images representing various natural phenomena. In addition, the required sample images consume storage space – even if they can be considerably smaller than the resulting textures.

Both of these techniques have also restrictions regarding the artistic control of the result images, and thus they are not suitable for constructing UI images, or even textures that have accurate representational detail.

Because of these issues, generative texture creation methods were not considered as a satisfactory solution. However, procedural texturing was seen as a method that could be used for a partial solution.

Vector graphics

Bitmap format for storing graphics is only one solution. The other solution is to store the graphics in vector format, where the image file stores only the drawing commands required for re-creating the visual appearance.

With a tight vector format, it is possible to define high-resolution images, even animation, using only a few kilobytes of memory. However, as the complexity of images grows, the file size grows as well. Typically, a vector image does not only define shape outlines and strokes, but allows for some limited visual effects such as gradients and transparency. With the help of these features, it is possible to define images that are closer to the visual richness of bitmap images but have the size and scalability of vector images.

Vector graphics work best when creating crisp and accurate images, logos and text for instance. This makes them ideal for creating user interfaces. For textures, however, vector graphics do not provide enough feeling of real surfaces, but are too geometric and clean. They however can provide something that texture generation can't: accurate visual control over surface details. Vector graphics also require sophisticated rendering algorithms in order to get good display quality. All edges for instance should be anti-aliased in order to avoid the "staircase" effect.

Conclusion

All solutions had their benefits and drawbacks. An optimal solution would combine the good features of the candidates and avoid the problem areas.

All methods listed above rely on completely different principles in their operation. Compression takes the original image, analyzes it, throws away unnecessary information, and packs the remaining information as tightly as possible. Procedural textures are closer to programming than drawing; various algorithms are adjusted and combined until a desired result image forms. Vector graphics have a third approach: instead of storing the final image, they store the "drawing commands".

The conceptual differences of these approaches can be combined however. It is possible to embed bitmap graphics to vector graphics, and this allows tightly compressed bitmaps with overlay of crisp vectors for providing missing detail. The same applies for procedural textures, it is possible to create images procedurally and overlay other images on top of them.

3.2.3 Towards the Solution – Filter-based Texture Generation

As none of the possible solutions above was satisfactory alone, there was one more alternative for consideration. Even back in 2002, this practical method was used by some texture generation packages, and nowadays it seems to be the main mechanism for most of the texture generation software on the marked. It is interesting to notice that even if there's a lot of academic research on procedural textures and texture synthesis, this method is hardly mentioned in the literature.

In traditional procedural textures, the input for the procedural algorithm is the texture space or world space coordinate, and the output is the calculated color at that coordinate. The procedural texture generator is the function that does the mapping from a coordinate to the pixel color.

The input for the texture function doesn't have to be just the input coordinate, but it can be also a color value or a set of color values calculated by other procedural texture functions. This approach allows creating graphs that describe the procedural calculations of a pixel in smaller steps instead of a single large procedural program.

One problem with procedural texturing is the limited locality of the sampling process. The texture function is sampling just a single point in the coordinate space, and operations that require knowledge about neighboring values – such as surface normal calculations or convolution filters – can be very heavy, as they need to recalculate these values for the surroundings of each processed pixel.

Procedural textures are usually resolution-independent, i.e. it is possible to zoom in into a procedural texture and it always remains completely crisp. However, if the content is such that full resolution-independence is not needed, it is possible to render the procedural texture into a bitmap texture instead. This usually gives also better performance, as usage of bitmap textures is computationally lighter process for the graphics processing unit (GPU).

When using bitmaps for textures, the straightforward way for procedural generation is to operate in the same way as with 3D space: generate color values based on the 2D texture coordinate. The only difference is that in 2D space, the texture coordinates are evenly spaced, when in 3D space the distribution of the sampling points depend on the relationship of the textured surface and the virtual camera.

Because the texture space is evenly spaced, it gives a natural coordinate frame for operations like convolution filters. Furthermore, it is not necessary to calculate these evenly spaced values of neighboring pixels for each processed pixels, but intermediate results can be stored as full bitmap images.

Conceptually, when operating in 2D texture space, the input for a procedural function is not a single coordinate or color, but a two-dimensional array of these values. The process is identical to filtering concept in image processing: the input for the operation is a buffer or buffers of pixel colors, and the output is another buffer. I refer to this method with term "filter-based texture generation".

This approach gives ability to look around in the input buffer for larger areas of pixels, and to perform operations such as blurring, lighting calculations and convolution filtering, thus allowing wider variety of processing operations than traditional procedural texturing.

Even if the technique is largely ignored in academic research, it has been known in the demoscene – a loose community of artists and coders – for years [22].

Texture design with filters

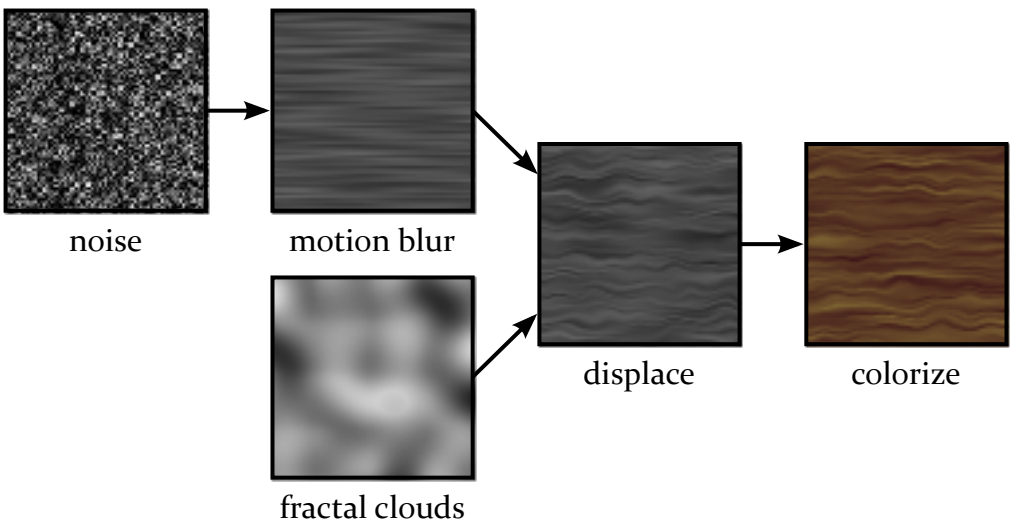


Figure 3.1: Texture generation process.

Texture design with this method also differs from traditional procedural textures, as the texture is perceived as an image that can be manipulated, not as a mathematical description

for the color of an individual point in 3D space. This gives the artists who are familiar with image processing packages an intuitive interface to texture generation. Artists can compose a texture by applying various image processing commands, but instead of saving the resulting bitmap, the process of generating it is saved. When the application loads the texture the generation process is then replicated to construct the bitmap again. See figure 3.1 for an example of the process.

This may still generate dull and artificial looking textures, as the generated textures lack any special structures – it's easy to create a concrete wall or a rock surface in this manner, but adding windows to the wall or moss to the stone would be difficult or not possible at all.

However, when this approach is combined with vector graphics and bitmap images, the situation is different. The filters can add richness and natural feeling to the vector images, and vector images can be used for adding accurate detail to the textures. Because the intermediate steps in the filtering process are handled as bitmaps, not as collection of mathematic formulas as in traditional procedural textures, it is possible to combine vector or bitmap graphics with generated graphics at any point.

There are vector graphics packages that allow – at least limited – use of filters to add visual richness to images. The use of filters in these programs is rather primitive however, mostly just adding simple effects such as drop shadows or embossing effects to vector shapes. The nature of filters allows much more, but the programs lack a good interface for utilizing filters at their full extent.

3.2.4 Technical Requirement Summary

In 2002, the result of the requirement definition process for the image generation technique was the following list:

- Images should consume very little space
- Images should have good visual quality
- Image generation should be fast
- Images should be suitable for all uses, both user interfaces and textures
- The technique should be lightweight
- The technique should allow font definitions in addition to image definitions

The tool should fill the following requirements:

- The tool should support the technique fully
- The tool should fit the usability requirements of the target user group of professional game artists
- The tool should be native part of X-Forge™ tool chain
- The tool should be able to read and write the format used in the games

In addition, the following features could be utilized in mobile environment:

- Scalable size
- Animation support
- Interactivity

3.3 Analysis of Existing Solutions

When the requirements were defined and potential solutions evaluated, the next step was to do research in order to avoid 're-inventing the wheel'. An existing application or an existing file format suitable for this purpose would reduce the workload of creating everything from scratch.

3.3.1 Existing Applications in the Area

In 2002, there were various texture generation packages available in the market, and a survey of those alternatives was made. None of the packages provided mechanisms that could be used in mobile environment for storage savings. The packages did not provide the image-generation programming library that would have been needed, as the applications were used only for generation of texture bitmaps. Some of the packages did however utilize a method based on filtering, and thus acted as a proof of concept.

The report of analyzed applications is included in Appendix B.

In addition, the available options of vector drawing packages were briefly evaluated. Back in 2002 Adobe Illustrator was the best option, as Inkscape was not available yet. Today Inkscape is probably the preferred tool for SVG development, as it's using SVG as the native format.

3.3.2 File formats

Various vector graphic file formats were also evaluated regarding their suitability for the technique. As vector graphics can provide small storage for content drawn by the artist, a vector format was seen as a necessary part of the solution.

Most notable candidates were Shockwave Flash (SWF) and SVG.

SWF

SWF [4] – Macromedia Shockwave Flash back in 2002, now Adobe Flash – is a format introduced by Macromedia, and was used widely on the web to create vector-based interactive animations. Back in 2002, Flash was at version 6; as of February 2013, it is in version 11.5 [5].

SWF is a compact chunk-based binary format and as such suits mobile use. SWF files are relatively easy to parse and process, thus allowing lightweight implementation. SWF is a based on vectors and thus is scalable. SWF also has support for image quality features such as antialiasing.

SWF can be used for storing fonts – although the file usually stores only the letters used in that file and font extraction is difficult. SWF can be used also for defining other graphical objects – such as buttons – that could be used outside the actual file.

The biggest problem was that SWF didn't provide mechanism for filters back in 2002. This would have meant that SWF was usable only as a partial solution for storing the vector images.

SVG

SVG is a vector graphics file format endorsed by W3C. For mobile use, there are two subsets of SVG, SVGT [6] and SVGB [20]. Back in 2002, SVG was in version 1.0 [21]; in 2013, the latest W3C recommendation is (still) 1.1.

SVG provides a rich set of rendering functionality. Furthermore, SVG defines also animation and interaction, and even sound. SVG also has filter support, which is a big benefit. SVG can be used also for font definitions, and SVG fonts can be used outside the defining document as well.

However, back in 2002 SVG was an emerging format and as such did not have good content creation tools. SVG is also complex to parse; parsing SVG requires an XML parser, which would increase the size of the application binary. As a text based format SVG files are big – although they compress well, and there even exists a standard for gzip-compressed variant of SVG, SVGZ. Then again, adding gzip-support would again increase the size of the application.

Supporting full feature set of SVG is also a huge task, and the resulting SVG viewer is a complex piece of software, easily eating up several hundred kilobytes. For mobile use, full support was out of the question. Mobile profiles, SVGT and SVGB, provided a limited subset of features, but they did not directly support all the features required, and had some unnecessary features as requirements. However, as an XML-based format, SVG has mechanisms that were reducing these limitations: it is possible to define a limited subset of SVG as long as the content creation tools can conform to said subset. It's also possible to extend SVG with custom XML tags for providing functionality missing from the format definition.

The bigger problem was XML parsing. However, it is possible to pre-parse the XML-format and save it in tokenized binary format that is easy to read and small to store. The tokenization process can also perform necessary checks for the file format. This is exactly how Wireless Application Protocol (WAP) [49] behaves: it converts XML to WAP Binary XML (WBXML) [33] that can be transferred to target devices and parsed there easily. Using WBXML or a custom tokenized XML format can therefore solve the problems related to XML parsing and text format.

PostScript

PostScript (PS) [1] and its variants, Encapsulated PostScript (EPS), Adobe Illustrator (AI) and Portable Document Format (PDF) are vector formats based on PostScript page description language. PostScript has been a standard in the printing industry from the 1980's, and has a good tool support. However, PostScript is better suited for publishing than mobile

use; PostScript is a complex, text-based file format that actually defines the source code for a stack-based programming language. PostScript interpreter is an implementation of this language, containing all the required rendering functionality. The rendering model is also more suited to printing than display, having features such as raster control etc. Plain-text PostScript files are also large, although there are also binary variants of the format.

CGM

Computer Graphics Metafile (CGM) [26] is a format originally designed for distribution of computer-aided design (CAD) drawings. CGM is however not a drawing exchange format, but rather an export format for displaying the finished works. It is a binary format and rather easy to decode. CGM was originally designed in the 1980's, but it has been revised several times after that. Current revision includes features – such as gradients – that make CGM usable for visual arts as well, and CGM is encouraged by W3C as well in the form of WebCGM [11] standard. However, CGM lacks features such as transparency and filters, and CGM export in drawing packages is generally poor, supporting only the early versions of the standard.

Other formats

There are also other formats for vector graphics, such as AutoCAD Drawing Exchange Format (DXF) or Windows Metafile (WMF). These formats don't however serve the requirements of mobile gaming: DXF is targeted for CAD use, and WMF is basically a format for storing Windows graphics device interface (GDI) commands, thus being dependent from the imaging model of Microsoft Windows.

3.3.3 Conclusion

Although SWF and SVG were strong competitors, SVG seemed to be better choice because of its filter support. SVG is more complex format though, and requires special processing of XML. In 2002, tool support for SWF was better than for SVG. However, the tools for SWF creation didn't have all the required functionality, especially they lacked filtering support.

Another benefit of SVG was that it is editable with a plain text editor. This allowed prototyping with the filtering features even without any graphical tool support. In addition, interactivity and scripting support of SVG would have enabled construction of functional prototypes for interface testing, although this path was eventually not taken in the design process.

4 Tool Design

In this chapter, the initial design process of filtered tool is documented. The result of this process was the first version used in the usability evaluation. Subsequent design iterations were interleaved with usability evaluation cycles, and they are described in the next chapter.

4.1 Design Constraints

As it was obvious that there's no readily available tool supporting all the required functionality, there was no other option than building and designing a new tool for creating the images. Considering the tool, there were few important decisions to make. First, should the tool be a stand-alone application or an extension for an existing drawing program?

Stand-alone application would allow for complete control of the results: the interface can be restricted to contain only supported features, and exported file format would be exactly what is needed. On the other hand, stand-alone application requires also programming all "ordinary" functionality, basic drawing commands, user interface, etc., not just the special features that are missing from current tools.

If the tool was to be an extension plugin to an existing application, there are different aspects to consider. First, the program to be used should be easily extensible. It is not enough to make just some filter designer extension, but the new functionality should be integrated tightly to the software. Also exporting and importing should work flawlessly and use an interface that does not perform crippling conversions to the image content. The host software should also have native SVG support, so that adding SVG functionality – such as new filters – is a WYSIWYG operation. It should also be possible to disable features that are not supported from the host program, or at least convert them to the supported functionality.

There were also few things to consider from the point of view of technical implementation. First, the rendering engine; should it be the same engine used in the games themselves, or should it be some other engine with full SVG support, for instance the engine of the host application? Using the same rendering engine as with the actual game would mean real "what you see is what you get" (WYSIWYG) editing. However, as the engine is mostly designed for game use, it is not easily embedded into another applications, especially from the point of view of the user interface.

4.1.1 Selecting the Host Tool

As building a full vector drawing application is a tremendous task, it seemed necessary to find an existing drawing program that can be extended with the desired functionality.

From the evaluated artists' tools, Adobe Illustrator [3] – at version 10 back in 2002 – was a clear winner: it had built-in support for SVG, thus requiring only conversion tools for binary format, and was easily extensible by using its thorough plugin-architecture. (Actually, most of the native functionality in Adobe Illustrator is built using the same plugin architecture

that is available for external development.) Adobe Illustrator was also one of the most popular vector graphic packages among graphic designers, which was also considered as a good thing. Customers of Fathammer were game studios, so if we were to force them to use a specific graphics application, it had better to be a popular one.

The biggest flaw with Adobe Illustrator was the lack of proper support for SVG filters. It was possible to use SVG filters in projects, but the filters had to be defined in a separate SVG file and edited manually as text, and there was no GUI available for this. (Incidentally, in 2013, the situation is still the same.)

The feature set of Adobe Illustrator was calling for a solution divided in two parts: a filter creation plugin that could be used for defining textures, and an export plugin that could be used for previewing and exporting drawings to tokenized binary SVG format supported by the game engine.

However, a plugin-based solution would generate a vendor lock-in, and this raised some questions. A stand-alone application that could be used as a companion for any vector graphics package was also a valid option.

4.1.2 Filter Creation

The actual image was to be constructed using the drawing interface of Adobe Illustrator. The role of the filter editor was to define SVG filters that could be used for modifying the elements of the image with algorithmic methods, or for creating imagery from scratch. This meant that the filter editor itself did not create finished images; it was to be used for creating components that can be used for image creation in Adobe Illustrator.

The open question was the level of integration with Adobe Illustrator. If the editor were to be implemented as a plugin, it would allow smoother editing and adjustment of the filter parameters, but result in vendor lock-in.

Fortunately, the mechanism for handling filters in Adobe Illustrator was such that it was easy to generate an external file containing the filters, and import the filters from the file to be used in the drawings.

It seemed therefore feasible to start with a separate filter editing application, and possibly only later integrate this into Adobe Illustrator as a plugin. This decision would also allow using the editor application with other vector editing packages.

4.1.3 Image Conversion

The images used in the game engines were not to be in plain XML-based SVG format, but in a proprietary tokenized binary format that was more compact and simple to parse.

The initial plan was to implement the conversion from SVG to this proprietary format as an export plugin for Adobe Illustrator. However, as SVG itself was already an export format in Illustrator, the process of exporting the Illustrator file to SVG would have to be implemented first – as codebase for the SVG export was not open source – and on top of that, the binary conversion from SVG to the proprietary binary format.

It seemed therefore more straightforward to make just a separate conversion application, taking SVG as input and producing the binary format as output. This was eventually implemented as a command-line converter, and is out of the scope of this thesis.

The plan was to integrate the conversion also to the filter editor, but this functionality was never completed. In filtered there's no need for such functionality.

4.1.4 Programming Architecture

There were some requirements for the filter editing application: it should be able to load and save SVG files, as well as render them fully. The editor should also have an intuitive graphical user interface for filter creation. A secondary requirement was the possibility to connect the tool to Adobe Illustrator plugin interface at a later point in the development.

The two options when choosing the programming architecture were either to use the game engine itself, or an external SVG toolkit. Only real option for external toolkit back in 2002 was "Batik" [17], an open source Java-based SVG toolkit, available at http://xmlgraphics.apache.org/batik/index.html.

Use of the game engine would have provided the same rendering engine as used in the games, better performance and better integration to Illustrator because of the C++ programming language used both for the engine and Illustrator plugin application programming interface (API). However, it would have required implementing a parser for SVG. The UI would have been problematic as well; the engine did not have native user interface components, and the look and feel of the interface would have been different from the GUI provided by the operating system.

Using Batik as the toolkit would have provided access to the UI components of Java, readily available SVG loading and saving and faster development. However, the performance of Batik was not the best possible, and integration with Adobe Illustrator would have been tricky because of the different programming language.

Therefore, the choice was not easy to make. It seemed however that the greater versatility of the user interface and SVG features provided by Java and Batik are more important than the performance. The integration with Adobe Illustrator can be done by a C++ gateway to the Java application, and the requirement of an editor plugin was anyway secondary; a stand-alone application would suffice as well.

4.1.5 SVG Features

In order to understand the design decisions made during the tool development, it is essential to have basic knowledge of SVG filtering model.

The SVG filtering model defines various filter primitives. Filter primitives are arranged into groupings of filters, which then can be used for processing graphics components inside the drawing. Primitives inside a filter grouping can use various inputs and render filtering results into temporary bitmaps, thus defining a complex flow of operations that take place inside that specific filter.

Each SVG filter component has a set of parameters. As everything within SVG, these parameters are defined as text. Some of the filters, such as feColorMatrix, have generic nature, combining together operations that would be presented with separate interfaces in paint programs. In the editing tool, it is not absolutely necessary to combine these operations either even if the operation is the same from the technological point of view, but implement several different interfaces for various uses of the operation if usability so requires.

Generally, each SVG filter has two input images, source and background. "Source" is the SVG object where the filter is attached, and "background" is the area of the screen where the object will be drawn. SVG filtering model also allows use of other source images by embedding them within filters. This doesn't necessarily mean separate bitmaps or referenced files, but also images completely embedded inside the filter definition.

Table 4.1 contains a listing of filter primitives defined in SVG, along with their short description.

Filter primitive	**Description**
feBlend	Blends together two images using various blending modes.
feColorMatrix	Makes a matrix transformation for color values.
feComponentTransfer	Makes a component-wise remapping of color values.
feComposite	Combines two input images using Porter-Duff compositing operations.
feConvolveMatrix	Applies a convolution matrix filter to the image.
feDiffuseLighting	Calculates diffuse lighting using the alpha channel as a bump map.
feDisplacementMap	Uses bitmap values from one input image to spatially displace pixels in the other input image.
feFlood	Fills the filter area with constant color.
feGaussianBlur	Performs a Gaussian blur on the input image.
feImage	Renders external graphics to the image.
feMerge	Allows collapsing several layer images into one output image.
feMorphology	Performs "fattening" or "thinning" of the artwork.
feOffset	Offsets the image in the image space by a vector.
feSpecularLighting	Calculates lighting with Phong lighting model, using the alpha channel as a bump map.
feTile	Fills the target image with repeated, tiled pattern of the input image.
feTurbulence	Creates an image using Perlin turbulence function.

Table 4.1: SVG Filter Primitives

4.2 Prototyping

The design process was based heavily on functional prototypes. The first design mockups were rough schematics (see figures 4.2, 4.3, 4.4, 4.6 and 4.8). After that, a transition to a functional prototype was made as quickly as possible, instead of prolonged development with non-functional prototypes (e.g. paper prototypes).

The main reason for this was that it was quickly apparent that the design for controlling the filter generation was such that producing a non-functional prototype would have been very heavy process.

Non-functional prototypes work best with interfaces based on navigation through screens and dialogs where it's easy for the test moderator to change the screens based on the user interactions. The design of the interface here was not based on such structure, but the modification of the filters was utilizing dynamic dragging mechanism and routing of connections between filter primitives. Such dynamic interactions are difficult to implement and clumsy to use in non-functional prototypes.

Furthermore, the visual feedback of the interface is a change in the image being edited. The space of choices affecting the resulting image is very large, and modeling this to any sensible extent that would give illusion of real functionality would have resulted in an unrealistically large collection of prebuilt UI screens.

Therefore, the conclusion was that with the same effort that a non-functional UI mockup would require, it is possible to build a functional prototype. As I was both doing the UI design and the implementation, there wasn't even any conflict in the availability of resources.

4.3 Tool UI

The two high-level tasks of the artist using the tool are editing filter definitions of an existing SVG file, and creating filter definitions from scratch to a 'library' file, to be used in the vector graphics package.

Although the filter editing part of both use cases can be the same, the essential difference is that in the first case, the filters are already applied to a specific instance of a graphical object, and the artist is modifying a specific visual instance, whereas in the second case it is not yet known how the filter would be used.

At a lower level the artist's tasks are:

- Loading and saving of files containing filters or filtered images.
- Managing the collection of filters contained by the file.
  - Identifying and choosing a specific filter for editing.
  - Adding and removing filters in the file.
- Visually evaluating the filter being edited, either in the context of the original image, or with content representing possible use cases of the filter.

- Editing the filter primitives within the filter.
    - Adding filter primitives.
    - Removing filter primitives.
    - Reordering filter primitives.
    - Reconnecting filter primitives.
    - Modifying primitive parameters.

The main tasks when designing the interface were building the model that was used for defining the relationships between the components of a filter, defining the visual representation used in the visual evaluation, and to create graphical user interfaces for the filter primitives.

4.3.1 UI Metaphor

SVG filers are handled as series of commands inside the SVG renderer. For artists however, the texture creation process should be as visual and intuitive as possible. The foundation of the creation process is based on the technical principles of image processing. It is therefore necessary to present the process as clearly as possible, without hiding things behind "magic" operations that take the control away from the user. At the same time, the details of the technical implementation should be hidden, so that the artists don't have to care about image buffers, command parameters and XML tags, but the system takes care of these automatically.

According to Jef Raskin, in the context of user interfaces, "intuitive" equals familiar [42], so creating an intuitive tool requires knowledge of the users. Although I've worked as a graphic designer for several years, my approach may be more technical than average, as I'm a programmer as well.

As Fathammer was not only a game technology company, but it was doing also game development, several graphic designers were working there. Opinions of these persons were important for the initial design of the tool, and we had many fruitful discussions considering what kind of interface they would find as "familiar".

Initially there were three alternative possibilities for the user interface for filter creation:

Script	The most straightforward approach from the technological point of view is to record everything the artist does into a script. This would however limit the possibilities and responsiveness of the editor a lot when modifying the texture, as the script is linear, and changing something in the beginning of the script would mean full playback of the rest of the script in order to preview the resulting image. Although the script would follow the structure of SVG filters rather well, things like references to earlier stages of the process are difficult to implement, visualize and comprehend.

Graph	Another approach would be to use visual graph for describing the process. Each operation could be presented as a box, and these boxes would be connected with arrows showing data flow between operations. Modifying these diagrams would be easy, but the graph concept may prove to be difficult for artists to grasp, as it is completely different from interfaces they are used to. However, graph metaphor doesn't reflect the SVG filtering model accurately. As the traversal order of the graph is undefined, constructing filters with optimal performance and memory usage is difficult, as the artist doesn't have complete control over the rendering order of the components.
Layers	The third alternative is to use layer approach, familiar from photo editing packages such as Adobe Photoshop. The interface would be easy to grasp, but it would limit the construction of textures, as things would be stacked on top of each other, allowing only one input and one output for each layer. Creating groups of layers or links between them could reduce this, but it could also clutter the interface so that the intuitiveness of the interface is lost. However, by adding visual links between the layers, it is possible to make the layer metaphor to match the filtering model of SVG extremely well. The filter primitives inside an SVG filter are defined just like that: as a list of actions that happen in a sequential order, where each action can reference to the output image of any action that has already been performed.

Figure 4.1 visualizes these three alternatives.

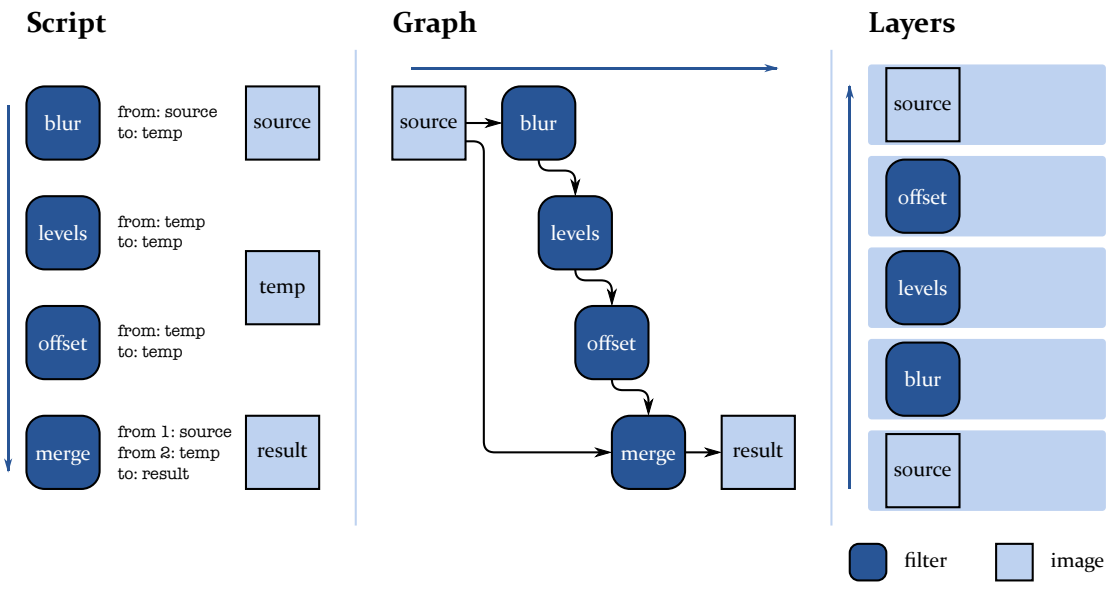


Figure 4.1: Initial alternatives for filter GUI.

Chosen metaphor

Based on the familiarity and good representation of the conceptual model of SVG filters, layers with links between them were chosen to be the interface metaphor for the tool. The challenge with the interface was then to minimize the clutter of the interface that linking might cause.

The filter primitives are connections between them are constrained by the SVG definition to form a directed acyclic graph (DAG). DAG is a directed graph with no directed cycles. It is formed by a collection of vertices (filter primitives in this case) and directed edges (links between primitives), each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again. [46, p.118]

Because of this graph property, term "layer graph" is used for the representation of the filter.

As the actual drawing operations are not included in the filter construction tool but are part of the Adobe Illustrator suite, the interface does not need direct interaction with the image. The editable thing in the interface is the layer graph, and changes in the graph are reflected to the image.

4.3.2 Overview of the GUI

Within the main tool application window, there are two windows: preview image and layer graph. All common operations, such as load, save, etc., are located in the top menu of the main window. The application does not allow editing several files at once. However, it is typical that one SVG file contains several filter definitions. It is therefore possible to switch between editing various filters within the file. The active filter can be changed from the layer graph window. There can be any number of preview windows displaying various filters, but only one layer graph window.

Figure 4.2 contains the overall schematic of the GUI.

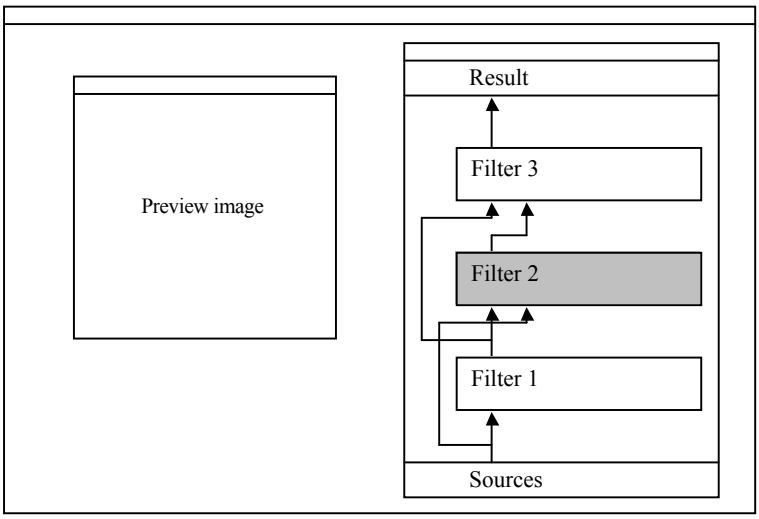


Figure 4.2: Schematic of the tool GUI.

Menu commands

The drop-down menus of the main tool window follow the common layout of windowed applications. The menus and their commands are:

- File
  - New
  - Open...
  - Save
  - Save As...
  - Exit
- Filter
  - New Filter...
  - Duplicate Filter
  - Delete Filter
  - Filter Settings...
  - Add primitive
    - List of available primitives
  - Duplicate Primitive
  - Delete primitive
  - Primitive Settings...
- Window
  - New Preview
  - Layer Graph (with a checkbox indicating if the window is open)
- Help
  - Help Topics
  - About

Additionally, a design for Edit menu with Undo, Redo, Cut, Copy, and Paste operations was made. However, these features didn't seem to be top priority: the assumption was that for undo and redo there's not that much need, as the changes in the filter primitive settings are typically easily reversible – exception being removal of filters or filter primitives, which however requires confirming. For cut, copy and paste there was also a little conceptual problem: is the operation aimed at filters or filter primitives? Therefore, "duplicate" was a better metaphor that provides most of the functionality of copying and pasting.

Editing layer graph

When editing a filter, there is always one active filter primitive. When executing commands such as "Add Primitive" or "Remove Primitive", the active filter primitive is affected. In "Add Primitive" command for instance, the new filter primitive is inserted above the active filter primitive.

The commands can be executed from the "Filter" menu, or from the layer graph window. The top of the window contains the selection combo box for choosing the current filter, and commands for creating, deleting and duplicating filters and for editing the filter settings. The bottom of the window contains buttons for adding, removing and duplicating filter primitives.

Each filter primitive has exactly one output, and usually one or more inputs. When filter primitives are added to the graph, all their inputs are automatically connected to the primitive below them. The output is connected to replace the connection that was between the closest connected primitive above them and the primitive below them. This way it is ensured that all primitives are connected, and that connections are valid.

The semantics of the SVG format define the valid connections between the primitives. Each input of a filter primitive can be connected to exactly one output. Each output, however, can be connected to any number of inputs. Inputs can be connected only to filter primitives below them, and outputs can be connected only to filter primitives above. Each input is always connected to some output; there are therefore no loose inputs in the graph. It is possible to create a situation where an output is not connected anywhere though. Filter primitives can also be rearranged by dragging them, these operations may need to disconnect and reconnect primitives so that the connections stay valid.

The connections can be edited by starting a drag from the input handle of a filter primitive. The drag is released over the other end of the "pipe", i.e. some output handle below the filter in the graph. During the drag, a line from the starting position to the current mouse position is drawn to indicate the forming pipe. If the place where the drag is released is valid – i.e. there exists an output handle at the position and the position is below the starting position of the drag – the old pipe going to the input is removed and a new pipe is formed. The color of the dragged pipe indicates if the connection is legal: when there's no legal connection, the pipe is rendered in red, and when there's a legal connection forming, the pipe is rendered in green.

Filter primitives can be temporarily disabled, i.e. "hidden". For each filter primitive, there is an eye-shaped icon for this. Each filter primitive has also a thumbnail preview representing the output image of that filter. Filter parameters can be edited by double-clicking the filter primitive, or by selecting "Primitive Settings" from the "Filter"-menu.

The pipes connecting the filter primitives are automatically arranged so that the connections remain visible. The pipes may cross each other, but the amount of crossings is minimized. The pipes coming from one output may also branch to several inputs. The branching happens immediately after the output. This leads to a greater amount of pipes, but makes following the pipe configuration easier. The pipes are rendered using gray color, except for the pipes that are connected to the current active primitive the color is blue. This makes it easier to see which pipes are currently active, even if the active primitive has been scrolled out from the window.

Figure 4.3 displays a schematic of a single filter primitive in the layer graph.

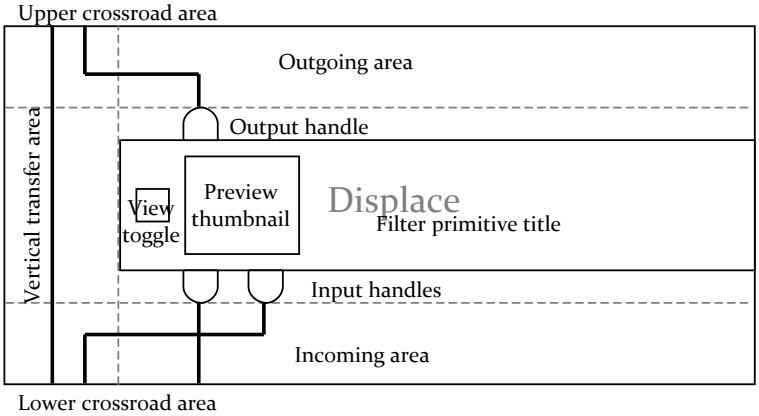


Figure 4.3: Graph of functionality in the UI of a filter primitive.

From the programming perspective, each filter primitive in the layer graph takes care of arranging and rendering the pipe connections relevant to the primitive. Generally, pipes are positioned to the left of the filter primitives. Each pipe travels vertically from its starting position, until it reaches the filter primitive to which it is connected. Those pipes that are unrelated to the primitive, i.e. which are not connected to any inputs or outputs of the primitive, just travel through "lower crossroad area" to "vertical transfer area" and continue over "upper crossroad area".

Those pipes that go to the inputs of the primitive turn right at the "lower crossroad area" and then up at "incoming area". Some incoming pipes may come directly from the primitive below; these travel directly over the "incoming area". Pipes going through the incoming area are arranged so that pipes going to furthest on the right are lowest ones at the left edge.

In a similar fashion, pipes on the "outgoing area" enter the area from the output handle of the filter primitive, and turn left to go to the "upper crossroad area", where they turn up. As the output may be connected to several inputs, the branching of the pipe happens on the "outgoing area". Those pipes going directly to the primitive above go through the outgoing area straight up, or turn first left to travel to the correct horizontal position and then go up.

Vertical pipes on the left side are packed as tightly as possible. The left side where the pipes are positioned is divided into pipe slots that can be either used or unused. For each pipe, starting from the bottom of the stack, a slot is calculated by searching for the rightmost slot that is unused for the whole span of the pipe.

Horizontal position of the pipe in "vertical transfer area" depends on the slot it uses. In the "incoming area", the vertical position of the pipe entering the area depends on the input to which it is connected. Therefore, in the "lower crossroad area", the pipe just makes a 90-degree turn connecting these two constrained positions.

In the outgoing area, the order of pipes is determined only by the order of pipes in the "vertical transfer area". When the pipes come from the outgoing area and turn up, they are arranged in such fashion that they do not cross each other, i.e. the pipe that is furthest on the right is the lowest one that comes from the outgoing area.

The sizes of the "incoming area" and the "outgoing area" depend on the amount of pipes travelling through the area.

Figure 4.4 displays a detail of the layer graph window, a mockup from the design phase.

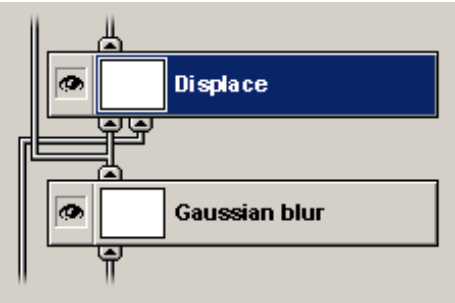


Figure 4.4: Detail of the layer graph window.

In addition to the filter primitives, the filter model in SVG has six global inputs for each filter. These inputs are represented as six output handles at the bottom of the layer graph, and they can be connected to the inputs of any of the filter primitives. The inputs are:

- Source graphic
- Source alpha
- Background image
- Background alpha
- Fill paint
- Stroke paint

Source graphic is the actual SVG object to which the filter is applied. Source alpha is the alpha channel of this graphic. Background image is the background behind the graphic, and background alpha is its alpha. (Background doesn't necessarily exist; it requires the "enable-background" attribute to have value of "new" in an ancestor element of the filtered element.) Fill paint is the paint used for filling at the element where the filter is applied, and stroke paint is the paint used for stroking there.

The SVG filter has always only one output. This is the output from the topmost filter primitive, and therefore the connection between global output and the topmost filter primitive can't be edited.

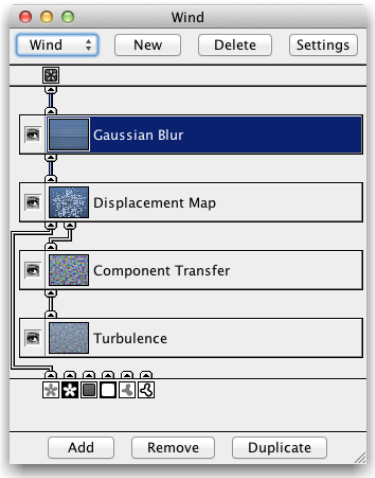


Figure 4.5: Initial implementation of the layer graph window.

Preview window

The preview window is used for displaying the previews of filters. Since the edited file can be the SVG filter definition file of Adobe Illustrator, there isn't necessarily any original content in the file but just the definitions for various filters. It is therefore necessary to have some graphical content that can be used for testing the effects of the filter.

However, it is also possible to edit the filter definitions of existing SVG files, so the original graphic should be viewable as well.

To accommodate both of these requirements, the preview window has a selection of preset graphics that can be used for the preview, as well as the original graphic. The presets can be used for displaying either the current filter or any other filter in the file. With the original graphic, naturally only those filters that are used in the first place are included in the preview.

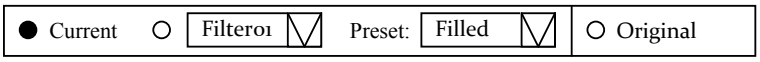


Figure 4.6: Schematic of the UI of the preview window menu bar

The presets are loaded from SVG files that have specific format – they contain one empty filter with name "CurrentFilter" – stored in the "Presets" directory in the program path.

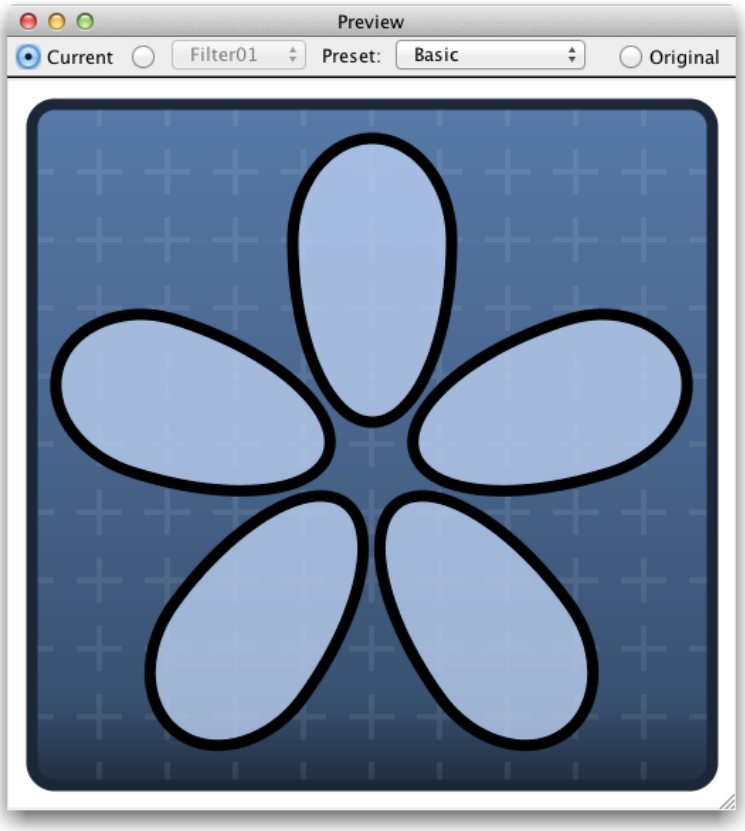


Figure 4.7: Implementation of the preview window.

In the original design, the default preview image was a brownish red star with black outline on a green gradient background. The basic preset, called "Filled", was varied with presets called "Gradient", "Inverted", "Multicolor", "Outline" and "Tiled". "Gradient" used a gradient for filling the star, "Inverted" had inverted color scheme, "Multicolor" had several fill colors

in the star, “Outline” was not filled but has a double stroke as the edge and “Tiled” had a repeating tile of small stars.

These were revised for the final design in order to be aesthetically more pleasing and have more variation. The star shape was replaced with a simplified “flower” – the same shape as used in filtered logo. The preset selection was enhanced by adding a couple of presets using photographs, one with text and one with complex geometry. See figure 4.9 for more information.

The user can create new presets if the default selection is not satisfying by just adding new files in the same format to the Presets directory.

Filter primitive settings

Each filter primitive has settings that are viewed when the primitive is added to the graph, and when the user double-clicks the primitive or selects “Primitive Settings...” from the “Filter” menu.

SVG format defines filter primitives as XML elements, and each element can have a number of attributes. These attributes are in the format of `key="value"`, for instance `width="100%"`. The attributes are different for each filter primitive, although there are some attributes common to all filter primitives.

The variety of attributes for filter primitives is wide, but supporting the full feature set defined by the standard is important. The software should be able to load and handle files that are originally created elsewhere, so all attributes defined in the standard should be supported. The program should also try to preserve the structure of the original file as well as possible, so that it is possible to continue editing the file in source code format.

The first visual HTML editors for instance had a problem of supporting only a subset of HTML and destroying information stored in a way unknown to them. They also produced HTML code that was non-readable for a human, although the whole point in markup languages (such as HTML and XML) is that they can be read both by humans and by machines. It was necessary for the software to avoid these pitfalls.

The first working prototype of the software had a simple interface, where the attributes were in a scrollable list on the left of the settings dialog and the value of the selected attribute was displayed in a text area on the right. This made it possible to very quickly provide some sort of interface for editing the primitive values so that it was possible to test the other aspects of the program.

This solution for the interface was not really usable, and had a number of issues:

- There was no indication about the acceptable values, whether it should be a number, some keyword, a list of numbers or something else.
- There was no validation of the entered values, so by entering illegal values it was possible to jam the preview and get a load of error messages from the render.

- There was no indication about the importance of the values. Some filter primitives have a huge amount of attributes, most of which are irrelevant for most of the tasks.
- Some filter primitives (such as feComonentTransfer, feDiffuseLighting and feSpecularLighting) also have some child nodes that define the behavior of the filter primitive. With an interface handling only attributes, it was impossible to handle these filter primitives.
- Most of the attributes are optional; some are required. The interface made no difference whether the attribute existed or not, and whether it was required to exist.
- Attributes have names defined in the SVG "code language" that is sometimes not so obvious. (For instance feGaussianBlur has an attribute called "stdDeviation" which is the technical term for defining the range of the blur effect, but not really familiar for artists using the program.)

This required designing a new system for editing the filter primitive attributes. Further requirements for the design were:

- Modular structure, so that adding new types of filter primitives is easy.
- Possibility to add functionality, such as previews or graphical user interface components, at a later phase.

The basic editing interface for attributes has a name, a checkbox for enabling the attribute, and a text field for the value. The checkbox can be locked, so that the attribute becomes always enabled. The value for the text field can be validated, so that it is not possible to enter illegal values. The name of the attribute is not the actual SVG name, but more descriptive one. In most cases, the difference from SVG code is minor – "lighting-color" becomes "Lighting Color" – but for some attributes with non-descriptive names, the displayed name is completely different.

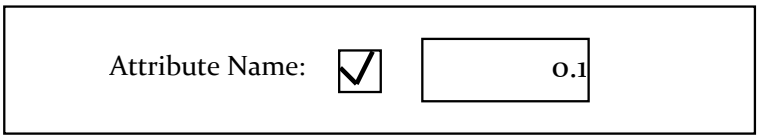


Figure 4.8: Schematic of the basic attribute UI

However, the attributes rarely follow the format of having just one value. There are at least following possibilities for attributes:

- In addition to the value, it is possible to choose units for the value from a set of predefined unit identifiers.
- The value is a textual value from a predefined selection.
- There is more than one value – usually two – and optionally some of the values can be left blank.

- There is more than one value, but the amount can be adjusted freely by the user.
- There is a predefined amount of values, but the amount depends on other attributes of the filter primitive.
- There are a number of values and the first value is a fixed textual value.
- Attribute can be some predefined value – for instance “inherit” or “none”, or some value entered by the user.
- A child element is used as a parameter for the filter primitive instead of an attribute. The interface should display the UI to the attributes of the child element in a manner similar to the filter element. The child element is either one from a selection of elements, or there can be several child elements from the selection active at the same time.

Furthermore, the validation of the value should be able to handle following situations:

- The value is a numeric value.
- The value is an integer.
- The value is limited to some range, either from both ends or just required to be greater than something.
- The value is not zero.
- The value is a textual value in some predefined format, such as color using any color notation valid for SVG, or a universal resource locator (URL).
- The value is constrained by the values of other attributes of the filter primitive. Changes in attributes are reflected to the constrained attributes as well.

To make the interface more legible, attributes can be grouped together and displayed in sets using tabbed panes or dialogs.

Building a set of Java classes following the requirements above enabled a modular system for building user interfaces for the filter attributes. The interface is not programmed for each filter primitive separately; instead, it consists of smaller blocks designed as an interface for each attribute type. By combining these together, it was possible to create an interface for each filter primitive in one factory class. This approach also enables incremental programming: for instance, the color attribute had a normal, un-validated attribute UI for entering the color value first, and a color picker and textual color validator was built only later.

4.3.3 Graphic Design

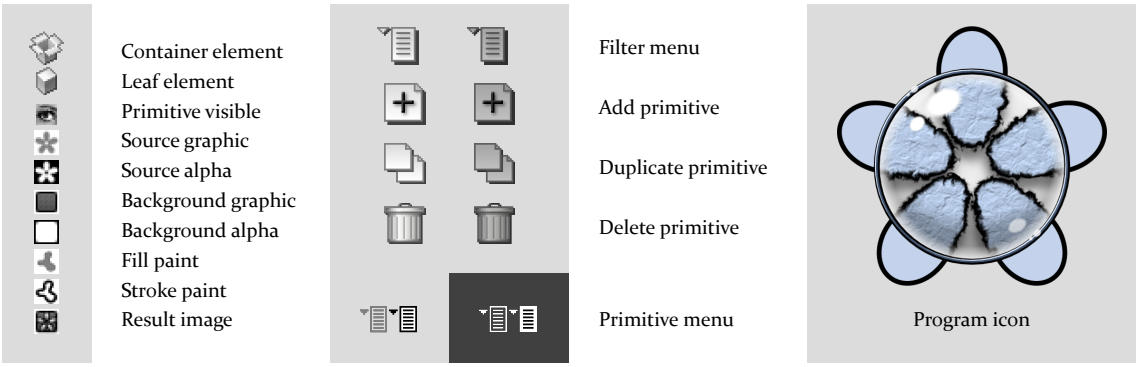


Figure 4.10: Icons used in filtered.

One aspect of the tool UI is the graphic design of the user interface. Since filtered is a cross-platform tool, a large portion of the look-and-feel is defined by the platform operating system. However, filtered still contains a collection of icons used in various places in the interface, as shown in figure 4.10.

Some preliminary design was done in 2002-2003, but everything was redesigned in 2012-2013. Buttons with text were replaced with icons in many places, which resulted in larger amount of icons.

Since filtered is a cross-platform tool, and the platform color schemes may differ from each other, or be adjustable by the user, all in-program icons in filtered are in gray scale and have transparent background. This makes them more adaptive for varying color schemes.

The icon design aims at simplicity while still having contemporary look-and-feel. The icons are e.g. not designed to work on 8-bit color palette – as filtered is a graphics tool, it is assumed that the user doesn't attempt to use the software on an old low-end display.

The graphic design also included "brand design" of filtered, including the "flower" symbol and filtered logo. These are used extensively in all material related to filtered, including this thesis.

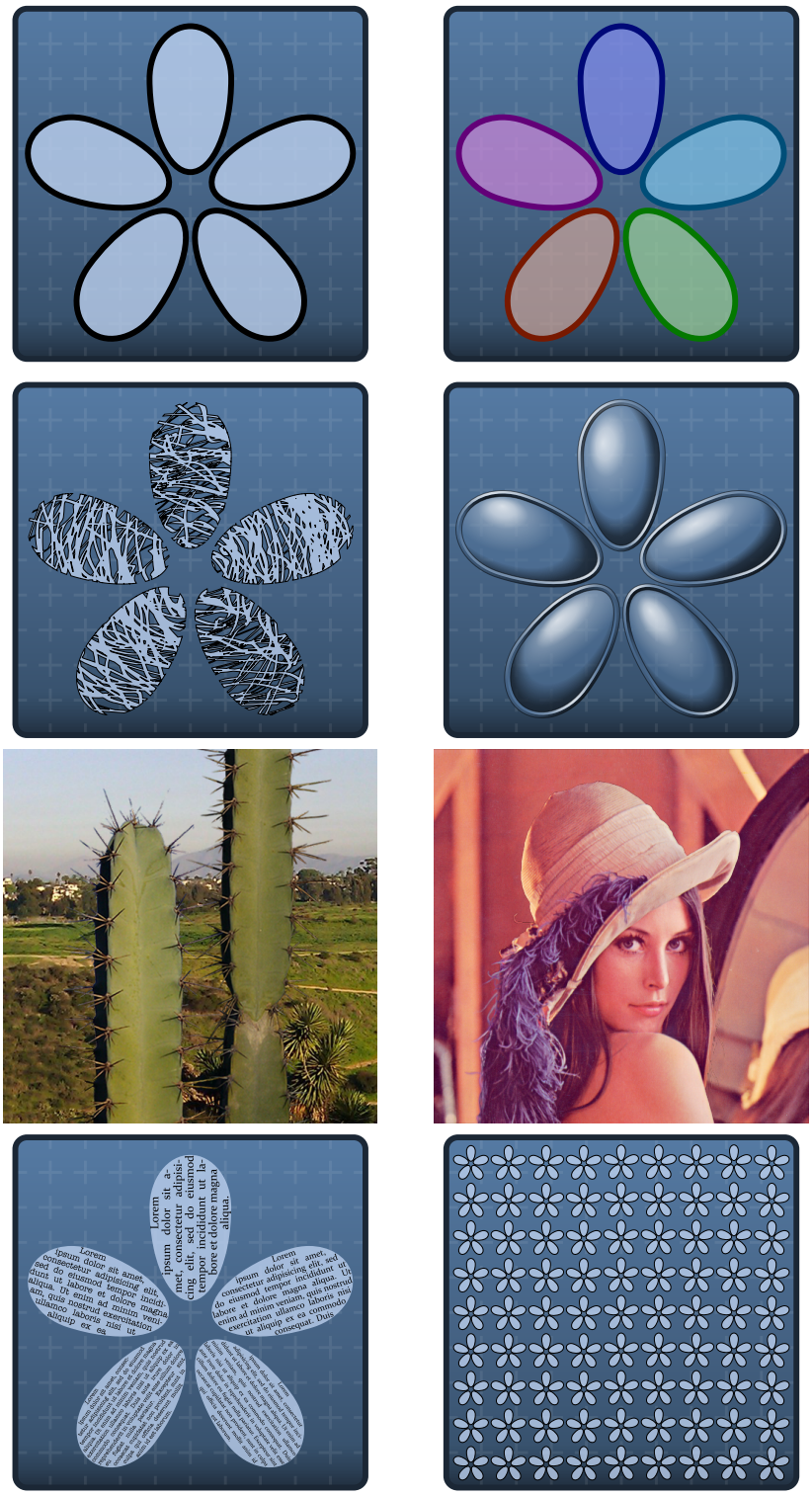


Figure 4.9: The preset images for the preview.

5 Usability Evaluation

This chapter documents the usability evaluation process and the design iterations based on the results of the usability evaluation.

Apart from some small non-structured test sessions, the bulk of the usability evaluation happened during 2012-2013, when I was developing the tool on my own time and resources.

Because there was no funding for the project at that stage, the usability testing was based on "discount usability methods" as described by Jacob Nielsen. [38, p.17]

The usability evaluation was mainly done using heuristic evaluation. The group of test subjects for each evaluation round was deliberately kept small (three evaluators), as even this small group of people will find majority of usability issues. It is better to use small group of subjects for several test iterations rather than large group just once. [39].

The focus of the evaluation was strictly in the usability of the interface, not in exploring the possibilities of artistic process the tool enables.

5.1 Heuristic Evaluation

Heuristic evaluation [40] is a method for finding usability problems in user interface designs. A small set of evaluators systematically examines the interface and judges its compliance with recognized usability principles – the "heuristics". Heuristic evaluation is less formal than other usability inspection methods, can be considered as "discount usability engineering" method [37]. Heuristic evaluation has great value in cases where time and budget are limited, as research has found it to be extremely cost-efficient [29].

5.2 Implementing Heuristic Evaluation

Heuristic evaluation was chosen as the primary usability evaluation method, since it was possible to do with very lightweight setting; I contacted some of my former colleagues, and asked them to perform evaluation of the software.

As numerous inspection rounds are a better alternative than one round with large number of testers, the absolute minimum of three evaluators per round was used. Even if it is not strictly forbidden to use the same evaluator multiple times at different stages of the evolution of the interface, the evaluators do "worn out" during the testing and don't stay representative of novice users [38, p.107]. Therefore, it was better to use as small group of evaluators as possible in order to avoid exhausting the resources of potential evaluators early on.

Heuristic evaluation is typically implemented in a moderated setting, but here unmoderated remote evaluation was used. Some of the evaluators were living abroad or in another cities, and remote evaluation enabled the participation of these evaluators. In addition,

doing unmoderated evaluation allowed the evaluators to freely choose the time and place for the evaluation.

Once enough people were recruited for the first round, they were sent the practical instructions in the form of following letter:

> You are asked to do a heuristic evaluation of filtered. Installer for the program is at available for download at http://filtered.sourceforge.net.
>
> The purpose of the program is to allow construction of effects that make vector graphics richer and more bitmap-like. In the real production situation, filtered is typically used in companion with Adobe Illustrator or Inkscape. This evaluation focuses only to the usability of the editor, and hence not on production of a finished image, so using the editor with a vector graphics editor is not in the scope here.
>
> You are asked to develop your own task for using the program. The task could be for instance decorating a vector image with some effects such as drop shadow, glow, emboss etc. or to create a texture effect for the vector image, some sort of stone or wood for instance. It is preferable that you have a clear goal in your task – for instance "creating granite texture" – instead of just fiddling with the software and seeing what will happen. The main point is however not in testing the software, but in evaluating it. You are not the user, but an evaluator evaluating how an user would behave when given this task and what problems there are on the way.
>
> The help files and examples provided with the software should give you some sort of start. They are not completely finished, so if you can't get a grip of the software, I can give you a short tutorial of it over Skype for example. However, information about requirements for the documentation and examples is also valuable and part of the heuristics.
>
> The software is mostly functional, but there are some minor unimplemented features (such as About-box).
>
> Use the attached list of heuristics in your evaluation. You should look for points in the prototype you feel confused, or you think the user would be confused. You should describe the point, evaluate its severity and extent, and record the heuristic that was violated. If you have a suggestion for a solution, it's more than welcome!
>
> Don't waste too much time on this. Two hours should be enough. However, go through the interface at least twice. You could for instance first get a general feeling of the software, then go through the task you have developed.
>
> Do not discuss with other evaluators about your findings.
>
> Record your findings on a paper or a text file. Do it immediately when you find a problem. When you are finished with the evaluation, send the report to me. If you wish, you can also do an audio recording and speak out the problems while you are using the software. I can do the transcription of the audio file.
>
> After all evaluators have done their evaluation, I will send a summary report for you all.
>
> On the next page is a list of heuristics (by Jacob Nielsen) you should look for. Print it out and have it at hand when doing the evaluation.
>
> Have fun!

The second page of the letter contained Jacob Nielsen's usability heuristics. [36]

Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

Recognition rather than recall

Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use

Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognize, diagnose, and recover from errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

5.3 Issues Identified before Heuristic Evaluation

The software was at beta testing stage (version 0.8) when the first heuristic evaluation round was performed. A number of issues were already known prior to the evaluation, and there were tasks on the to-do-list that had not made it to the 0.8 beta release.

Below is a list of unfinished user-interface related tasks known prior to the evaluation:

- About-splash is missing
- Filter Settings, New, Duplicate, Delete should be in a drop-down menu instead of buttons in the Layer Graph.
- Interface of Tile-primitive is unclear; add explanation about use.
- Add help-button for each filter primitive interface.
- Implement zooming of preview.
- Improve error handling; catch all exceptions.
- Add options for Layer Graph windows (change thumbnail image, thumbnail size)
- Replace primitive Add, Remove, Duplicate with icons. Delete by dragging to trashcan.
- Add preview option to filter primitive dialog
  - Small preview window in the dialog.
  - Show the effect in actual preview windows.
- Implement interface for editing lighting primitives visually.
- Implement interface for editing component transfer primitive visually.
- Improve color matrix primitive: add preset matrices, matrix transforms and a preview using a color wheel.
- Add presets for arithmetic blending.
- Add presets for convolution matrix.
- Add bitmap export.
- Add undo and redo.
- Allow comments in filters.

5.4 Results of Heuristic Evaluation

Two rounds of heuristic evaluation were performed in the scope of this thesis. The tool has not yet reached version 1.0, and not all known issues are fixed.

Both evaluation rounds were performed using the same method: the evaluators did the evaluation by themselves, and reported the results by e-mail. The results received from the evaluators were in free-form language, quite much in a speak-aloud format. Extracting the concrete violated heuristic and accurate description of the problem from the reports required some work, but this was no different from post-processing the results that would

have been gathered from moderated evaluation sessions. No evaluator opted for capturing the evaluation session to an audio recording even if this possibility was offered as well.

Major usability issues identified after the first evaluation round were fixed, and the second round was performed with a different set of evaluators. Further testing rounds are still possible; however, as the software is reaching a higher level of maturity, it is more useful to let the software gather a larger user base of real users and then collect feedback from this group.

5.4.1 First Evaluation Round

The first heuristic evaluation cycle resulted in identification of a set of design problems. While some of the problems were already identified and included in the to-do-list, there were also some surprises. Below is the list of identified issues, sorted by number of evaluators reporting the problem and then by category.

Violated Heuristic	#	Problem Description
Flexibility and efficiency of use	3	Dragging is restricted to be from destination to source. It would feel more natural the other way round, or both ways.
Visibility of system status	2	Preview window should show changes immediately when editing filter primitives, not only when the window is closed.
Recognition rather than recall	2	Flow direction (from bottom to top) is not immediately obvious.
Visibility of system status	1	Relation of layer graph window title and preview filter names is not clear. The layer graph title could contain e.g. “Now editing:”.
Visibility of system status	1	Filters with complex connections are difficult to understand.
Visibility of system status	1	A mode for viewing just the alpha channel would be useful.
User control and freedom	1	Esc-key should work as emergency exit from filter primitive editing.
User control and freedom	1	Undo would be useful in cases where filter connections are messed up.
Consistency and standards	1	Window resize may scroll out layer graph window. This should rather be dockable window.
Consistency and standards	1	Handling of key focus in filter primitive settings is not following the platform conventions (on Windows) consistently.
Consistency and standards	1	Save-functionality doesn’t add .svg automatically to the end of the filename on Windows.
Consistency and standards	1	Save leaves the file in locked state, preventing viewing in other programs until the program is closed.

Consistency and standards	1	UI settings (window positions etc.) are not saved between usage sessions.
Error prevention	1	Renaming filter will render it useless in the original image. The references to filter should be (optionally) changed as well if the filter is renamed.
Recognition rather than recall	1	Editing filter primitives by double clicking is not obvious. Add edit-icon or right-clickable context menu with options (enable/disable, settings, remove, duplicate).
Recognition rather than recall	1	Most of the filter primitive settings are not obvious if the user is not familiar with SVG filters.
Recognition rather than recall	1	Options for adjusting the size of the filters are difficult to locate.
Recognition rather than recall	1	Functionality of color matrix and component transfer primitives is difficult to understand. The interface doesn't give enough hints about what various SVG parameters do.
Recognition rather than recall	1	Mechanism for showing just a single filter primitive (by hiding the others) is not immediately obvious.
Recognition rather than recall	1	Filter inputs and their relations to each other are not obvious.
Recognition rather than recall	1	The "original" view of the preview window is not obvious (in a case when there's no original.)
Flexibility and efficiency of use	1	Add "edit" icon or right-clickable context menu with options (enable/disable, settings, remove, dublicate).
Flexibility and efficiency of use	1	Dragging filter primitives to "trashcan" for removal is desired (instead of clicking "remove").
Flexibility and efficiency of use	1	It's not possible to attach filters to the elements in the destination SVG file.
Flexibility and efficiency of use	1	Switching of input order requires dragging of pipes around. An accelerator is needed.
Help and documentation	1	Installer on Windows doesn't create a link to help file to the start menu folder.
Help and documentation	1	Example filters are too complex to understand.
Help and documentation	1	Example filters don't contain files that actually use the filters; all examples are "filter libraries".

Top three issues

The most surprising result was that the dragging direction was considered counter-intuitive by all evaluators. However, there is a clear reason why it's implemented that way: an output

can connect to any amount of inputs, and therefore dragging from an output would just add new connection going out from the output, not really re-route the existing connection. When dragged the other way round, the route is always fully reconfigured, as there can be just one source for an input.

It is technically possible to implement the dragging the other way round as well. It may however result in a situation where the dragging doesn't produce desired results, as the old connection would not be removed, just a new one created. Probably the best solution is to allow dragging both ways. This enables the more intuitive dragging direction when appropriate, while still allowing fast reconfiguration of the route with the opposite dragging direction.

The lack of preview functionality when editing the filter primitives was reported by two evaluators. This was also a known issue prior to the evaluation. There are some concerns regarding the performance of the preview, but in general, the feature should be implementable.

The third issue – difficulty in understanding the flowing direction – is an interesting case. This was reported by two evaluators, both of them being mainly game software developers, not graphic artists. The only full-time game artist did not feel this was an issue.

This result demonstrates the effect of "intuitive" being in fact "familiar". For the graphic artist the direction was intuitive, as the UI is built to resemble a photo editing software. In such software, images are typically constructed from multiple layers, and the layers are visualized in the order from bottom to top.

For the software developers, however, the mental model of a filter clearly resembled the model of program code. Computer programs are usually edited from top to bottom order. This is natural for programming languages where the program description is text-based, but the same applies also to visual programming languages such as MIT's Scratch [32].

5.4.2 Fixing the issues

The biggest issues in the user interface were related to the behavior and visualization of the layer graph. Following changes in the layer graph interface were implemented in order to improve usability:

- Editing of links was improved so that it's possible to drag the connecting link either way.
- Arrowheads were added to the ends of connecting links. The purpose is to improve the visibility of the direction of the data flow in the graph.
- Filter primitives can be removed by dragging them to a trashcan. As this required changing the remove-button to a trashcan icon, also other buttons were changed to icons.
- An icon for a drop-down menu was added for each filter primitive. The menu improves visibility of some functionality, e.g. primitive settings were earlier accessible by just double-clicking the primitive (or from the main menu).

- Functionality for showing and hiding multiple primitives was added. This is accessible through the aforementioned drop-down menu.
- Shortcut functionality for reversing the order of filter inputs was also added through the drop-down menu.
- The tile of the layer graph window was changed to indicate that the visible filter is the one currently being edited.
- Buttons for filter creation, deletion and settings were removed and a single drop-down menu accessible from an icon was used instead.

Figure 5.1 shows the layer graph window after usability improvements.

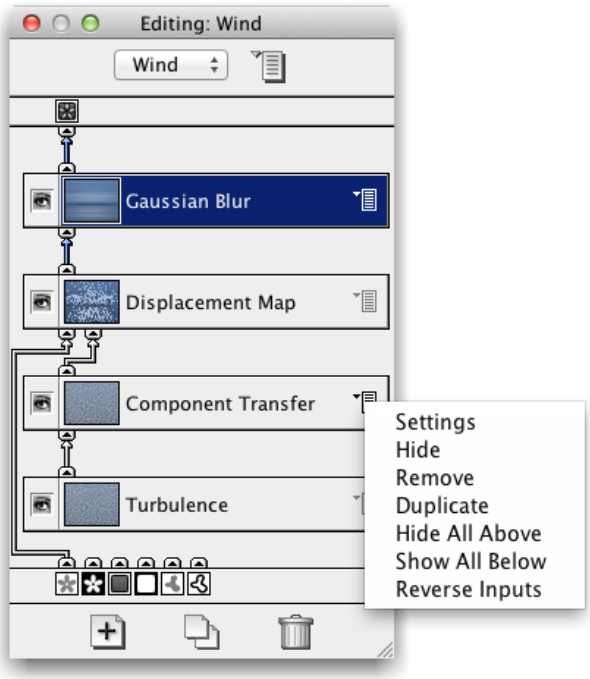


Figure 5.1: Layer graph window after usability improvements.

In addition, following major improvements were done – as well as numerous other smaller improvements:

- Support for undo and redo was added. Some confirmation dialogs were removed, as accidental modifications can now be undone. Each filter has a separate undo buffer, and undo/redo affect to primitives of that filter. Operations for creating and destroying whole filters are not handled by the undo mechanism, and they are still protected by confirmation dialogs.
- Preview capability when editing settings of filter primitives was added. The preview needs to be initiated by clicking a preview-button; this is for performance reasons. Immediate preview when modifying any filter primitive attribute would result in slow display refresh with complex filters.

- Capability to connect filters to graphic elements in SVG files was added. Functionality for importing filters from other files was also implemented, as well as functionality for cleaning up unused filters from the file being edited. Together these features allowed filtered to be used as a tool for adding filters to existing vector images, instead of serving just as a tool for creating the filters, and then joining them to the image in the main editing tool.

Figure 5.2 shows the GUI window for attaching filters to graphic elements in the SVG document tree.

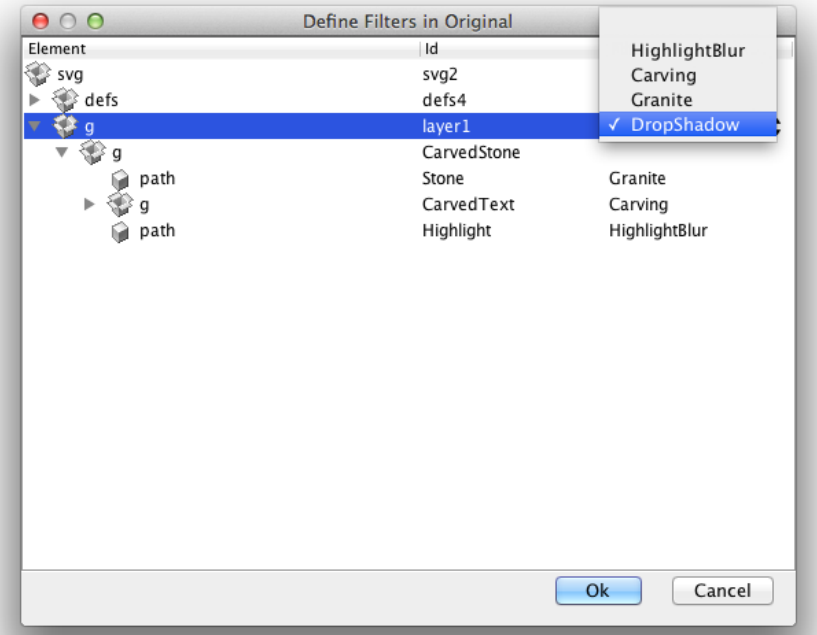


Figure 5.2: Interface for attaching filters to graphic elements in the SVG document tree.

Interfaces of filter primitives were not improved for the second evaluation round, even if usability problems with those were already recognized. The rationale was to gather insight of the implemented usability improvements in a tight loop, and to keep the implementation phase between the evaluation rounds as short as possible.

Even if the second evaluation round was implemented with a different set of evaluators, also the evaluators from the first round were informed about the improvements and they were asked if they would like to give further feedback. This feedback was not considered as part of the usability evaluation anymore.

5.4.3 Second Evaluation Round

The second evaluation round was performed with three evaluators again. There were some issues with the evaluation process on the second round though. One evaluator had hard time following the evaluation instructions, and another was somewhat struggling with them. Both provided valuable feedback about the evaluated interface still, but their performance with the evaluation also suggested, that the evaluation procedure needs to be defined more robustly.

Nevertheless, following set of issues was collected from the second testing round:

Violated Heuristic	#	Problem Description
Consistency and standards	1	Installer shortcut to "Applications" is not working correctly on MacOS X "Mountain Lion".
Consistency and standards	1	File dialogs are non-standard. (They reveal that this is a Java application, not native MacOS app.)
Consistency and standards	1	Term "primitive" used in the interface feels misleading.
Recognition rather than recall	1	Primitive connections easily become a mess.
Recognition rather than recall	1	Adding primitives to a filter is not an obvious first step.
Recognition rather than recall	1	Primitive inputs could have tooltips.
Recognition rather than recall	2	Some filter primitives (such as Gaussian Blur) have text fields with no labels.
Flexibility and efficiency of use	1	Keyboard shortcuts would be useful, e.g. delete or backspace should remove the active filter primitive.
Flexibility and efficiency of use	1	Input and output icons of primitives are too small.
Flexibility and efficiency of use	1	Constant clicking of preview-button when editing filter parameters is annoying.
Flexibility and efficiency of use	1	Double-clicking the thumbnail of a filter primitive doesn't open the settings; you have to hit the primitive name.
Flexibility and efficiency of use	1	Swapping input order using the drop down menu is not immediately obvious.
Flexibility and efficiency of use	1	Primitives should have presets.
Flexibility and efficiency of use	1	Modal dialogs for filter primitive parameters are annoying.
Help and documentation	1	Filters would benefit from tooltips or a separate help-screen.
Help and documentation	1	Basics and Principles chapters in the manual are unclear.

Help and documentation	1	A tutorial is needed.
Help and documentation	1	Workflow with Adobe Illustrator is not explained.

Results of the second round

The difference from the first round was interesting: the evaluation results pointed to far lower amount of direct violations of heuristic, and contained more opinions and improvement ideas from the evaluators, rather than reports of direct problems. Some of this discussion in the evaluation reports is not captured in the table above, as it wasn't following the format of heuristic evaluation.

Only one of the problems was identified by two evaluators, and even that one was already a known issue. In general, only a small minority of reported issues were new real usability issues. Several of the issues were already identified, and some reported issues were not really violating usability heuristics, but rather they were opinions about the user interface.

In general the second usability evaluation round gave an impression that even if one more design and implementation round is needed in order to fix the known usability issues, doing one more round of heuristic evaluation would not reveal anything critical anymore, and it is possible to move forward to the next steps.

5.4.4 Next Steps

The scope of this thesis ends at the analysis of the results of the second usability evaluation round. The results are encouraging regarding the maturity of the software, and after the remaining usability issues have been fixed, the next step with filtered is to go fully public and advertise the software for a larger audience.

This may lead to a larger user base with new possibilities for gathering usability information. As expanding the user base takes time, and getting feedback from a large group of users can be a major task on its own, it is practical to leave these steps outside of this thesis.

5.5 Conclusions of the Usability Evaluation

The chosen lightweight usability evaluation method worked surprisingly well. It revealed a set of usability issues during the first round. More than half of the issues were fixed for the second round, including all major issues. On the second round, only a small minority of listed issues were reports of new violated usability heuristics. This implies that the method captures usability problems relatively well.

However, it has to be recognized that all of the evaluators were experts in the field; they were senior-level graphic designers or game developers. Doing usability evaluation sessions without moderation probably would not work with random group of evaluators.

For a project like this, testing without moderation was very cost-effective approach. The evaluators were geographically separated, some living abroad. They all were busy professionals, and fitting a moderated evaluation session to their schedules would have been a

challenge on its own – now it was possible for them to freely choose the time and place for the evaluation session.

There were nevertheless some issues with the evaluation process. Some of the evaluators were not following the evaluation instructions well enough – although this didn't necessarily result in lower value of the results of the evaluation but rather in more post-processing work of the answers. Probably the instructions for the evaluation should have been more compact, and preferably more pictorial.

The replies of some evaluators were also very brief, recognizing just a few issues. A moderated session with these evaluators could have resulted in larger collection of issues, now they probably just didn't bother to write down issues they considered too minor.

5.5.1 Usability in Open Source Context

Usability of open source software (OSS) has been considered problematic already for a long time, although there is relatively little research data about the subject [34]. Somewhat simplifying the scope of the problem, usability issues in OSS are due to the following main reasons [35]:

- Developers are not typical end-users.
- Usability experts do not get involved in OSS projects.
- Open source projects have limited resources.
- Interface design and functionality design may require different approaches in development.

The other three issues can be solved by changes in the development process and by recruiting people with required skill sets to the development team, but limitations in available resources can be a hard limit. Only some large open source projects are financially backed up by corporations or governments. Vast majority of open source projects have no funding and rely solely on contribution of volunteers. Lack of resources often prevents applying traditional usability methods to OSS projects.

The usability evaluation method used with filtered is an encouraging example that it is still possible to perform basic level of usability evaluation with very scarce financial resources. It would be possible to develop this method even further, by e.g. offering a software tool for assisting with the evaluation. The tool could offer more structured way for recording the evaluation sessions and for reporting the results. Currently OSS community doesn't have tools like that, and existing solutions (such as bug reporting systems) are not the optimal method for handling usability issues [35].

OSS encourages collaboration and participation, but this concerns mainly developers, not regular users. Regular users may find it difficult to participate in OSS development because they lack the required software development skill set. Usability evaluation can serve as one development area where the average users are able to give their contribution to a project they appreciate.

6 Results

This thesis has been describing the development process of filtered, focusing on the user interface design. The current result is not finished software – "finished" being hard to define for an open source project in the first place – but it should be good enough already for real production use.

User's Guide of filtered is included as Appendix A – it gives a rough view of the functionality of the program. The best way to understand what filtered does, is to download the software from http://filtered.sourceforge.net and try it out. The software and some usage videos are also on the CD-ROM packaged with the printed thesis.

The software at its current stage is aimed to work as a companion for another vector graphics package, such as Inkscape or Adobe Illustrator. The resulting SVG images would be used on web pages, viewed by web browsers such as Apple Safari, Mozilla Firefox, Google Chrome, Opera or Microsoft Internet Explorer.

The software should enable artists to enhance vector graphic images with effects usually associated with bitmap graphics. It should also allow using these effects for manipulating bitmap images.

In this chapter, the interoperability with above products is evaluated, and the artistic possibilities available through the tool are explored.

6.1 Interoperability with Tools and Browsers

During the development of filtered it became obvious that SVG support in tools and browsers hadn't improved from the state it was in 2002 as much as I had hoped.

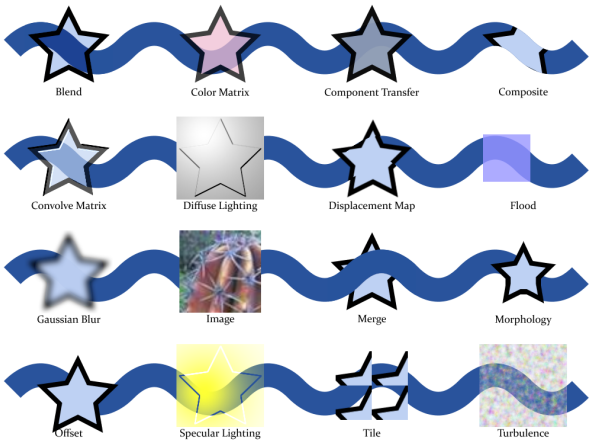
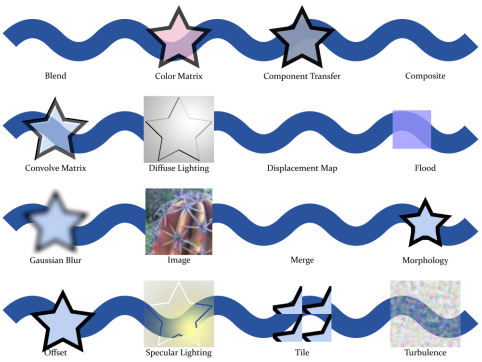
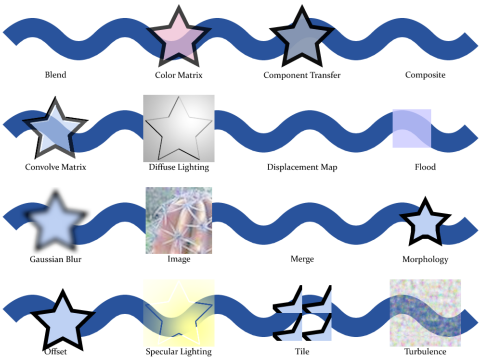
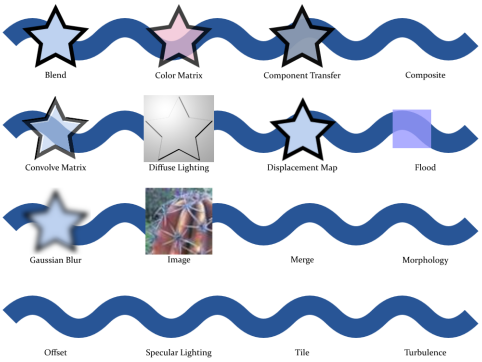


Figure 6.1: The test image as rendered by filtered.

In order to measure the level of filter support in tools and browsers, a test image containing

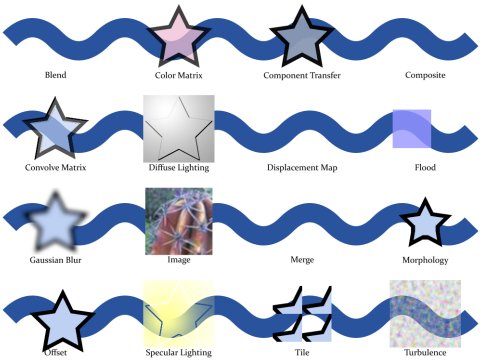
all filter primitives was prepared. This image is also included as one of the example images in filtered distribution. The image is shown as rendered by filtered in figure 6.1.

The table below contains images from various browsers and tools, along with description of defects.

Software & Description	Sample Image
Apple Safari Filters using the background graphic don't render anything. Coordinate system of lighting effects is flipped. Specular light effect is rendered too dark.	
Google Chrome Filters using the background graphic don't render anything. Coordinate system of lighting effects is flipped. Flood and Image effects are rendered too light.	
Microsoft Internet Explorer 10 Filters using the background graphic are not rendered correctly. Some filtered shapes are not rendered at all, possibly everything after an unsupported feature is discarded.	

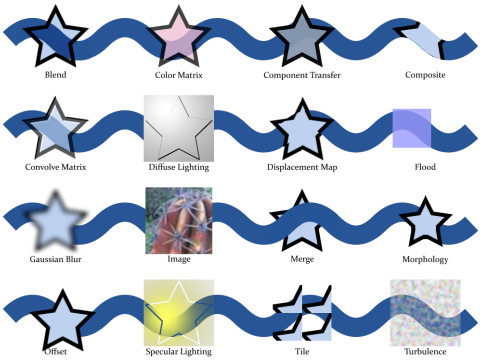
Mozilla Firefox

Filters using the background graphic don't render anything. Convolve-filter has clamping artifacts.



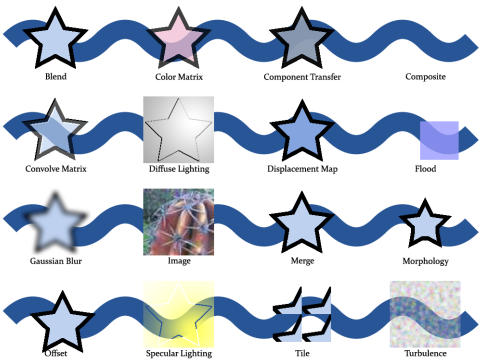
Opera

Specular lighting effect is too dark.



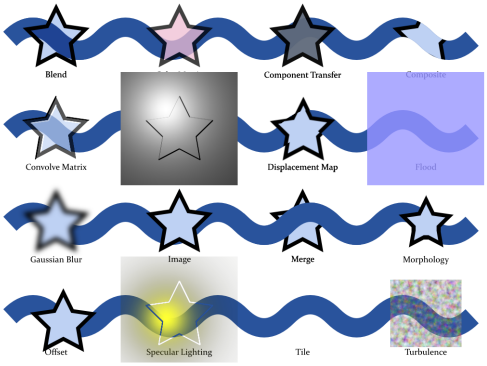
Adobe Illustrator

Filters using the background graphic are not rendered correctly. Coordinate system of lighting effects is flipped. Coordinate system of flood primitive is flipped. Antialiasing is turned off from shapes being filtered.



Inkscape

Area of filters is too large. Flood primitive coordinates don't have any effect. Image-primitive is not working correctly. Specular lighting effect is too dark. Turbulence effect is too opaque.



6.1.1 Results of the Comparison

Web browsers

The results of this comparison are rather staggering. It seems that all SVG renderers render the filter effects in a different way. This is unfortunate for a web developer – SVG content would need to be checked against all popular browsers for compatibility.

Features that are not working correctly across browsers are briefly:

- Background image. Only one out of five browsers respect the `enable-background="new"` attribute.
- Lighting effects. Browsers seem to be puzzled about the correct light direction.
- Color space conversions. Various graphic elements appear too dark or too light in some implementations. This is probably due to data being defined in a wrong color space.

Remaining features seem to work quite well, so by limiting the expressive power of the filters it is possible to stay compatible with the various browser implementations.

On the other hand, the situation indicates that SVG filters are not widely used in the web content today, and issues like this can go unnoticed in the browser implementations. The reason for the lack of content is naturally the lack of proper content creation tools. filtered has potential for serving as a catalyst for change in this respect.

Content Creation Tools

The sad situation with content creation tools is even worse than the result with the web browsers. In their current form, vector graphic content creation tools don't provide adequate environment for developing SVG filters that could be used reasonably well in the web pages.

Luckily filtered can be used as a companion for these tools rater easily. Both Inkscape and Adobe Illustrator have a "revert" mechanism for loading a previously saved file from the disk, and filtered has a "reload" command for the same purpose. By using save and "revert"/"reload" functionality, it is possible to edit a single SVG file simultaneously in two programs.

In this use case it's better to turn off filter rendering in the vector editing tool, and use it just for defining the vector shapes, while doing all filter-related work in filtered. This also ensures that the image can be edited in the vector graphic tool easily, as filters don't disturb the visual appearance of the vector shapes. Filter work in filtered on the other hand provides better match with the rendering available in the web browsers.

6.2 Result Images

Following pages represent some result images, as pairs of plain vector and with filters. The images are composed using a combination of Inkscape and filtered.

Figure 6.2: “Carved Stone” image as plain vector.

Figure 6.3: “Carved Stone” image with filters.

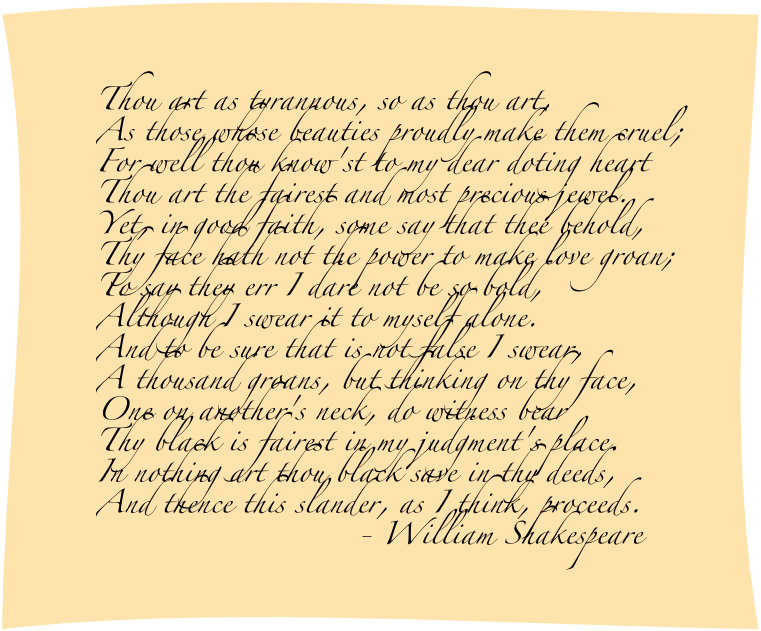


Figure 6.4: “Poem” image as plain vector.

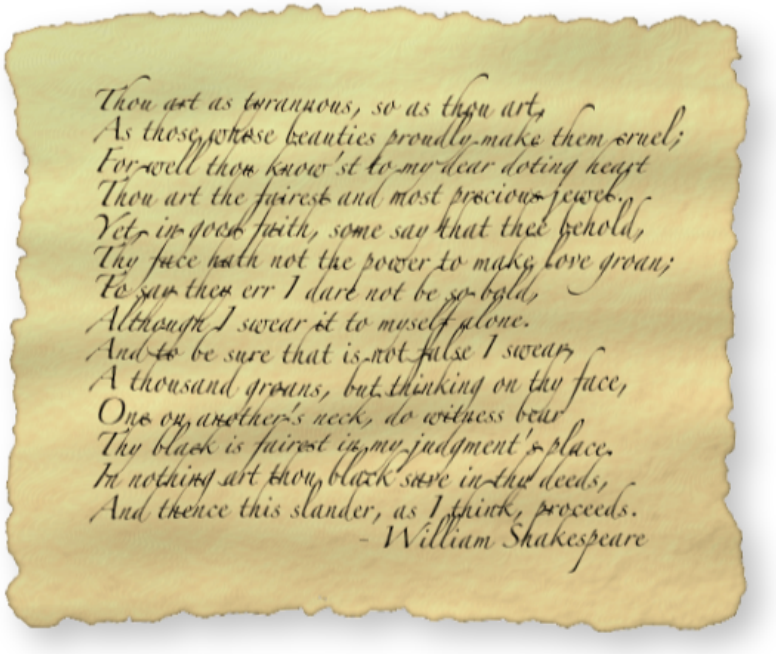


Figure 6.5: “Poem” image with filters.

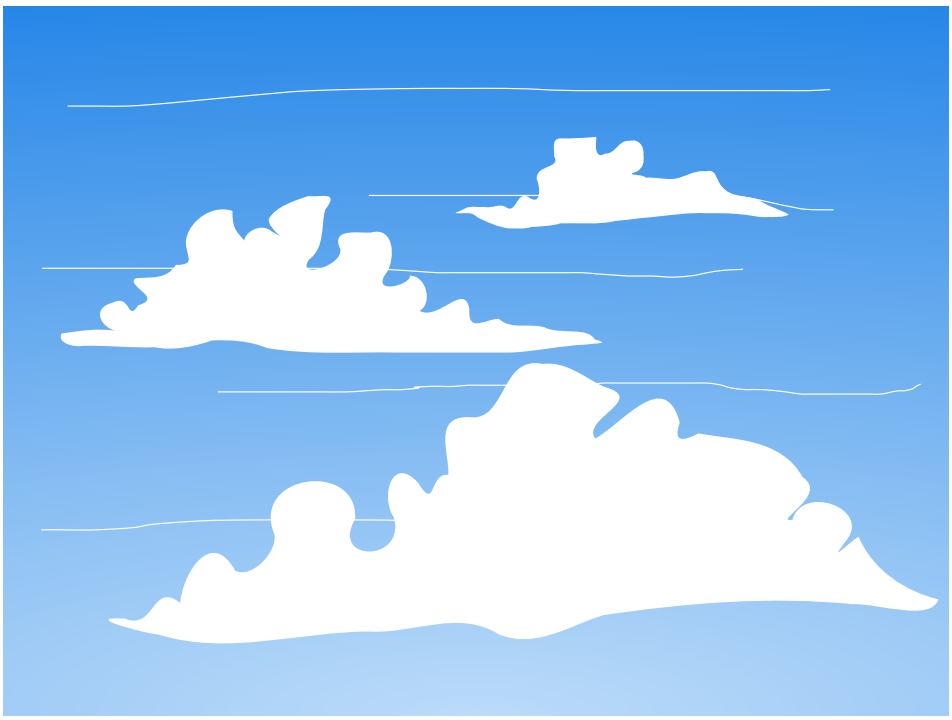


Figure 6.6: “Clouds” image as plain vector.

Figure 6.7: “Clouds” image with filters.

Figure 6.8: “Watercolor” image as plain vector.

Figure 6.9: “Watercolor” image with filters.

6.3 Conclusions

filtered started as a proprietary tool for a specific purpose of texture generation; during the journey it transformed into a more generic open source tool for SVG filter editing. This journey is not over yet, and further development of filtered will continue in SourceForge.

As the result images indicate, filtered can be used for enhancing vector graphic images in a variety of artistic styles. In theory, this capability has existed in the filters defined in SVG file format since 2001, from SVG revision 1.0. However, this far no content creation tool has been able to expose the SVG filtering capabilities for artists at the full extent.

Based on the usability evaluation, filtered can serve as a tool for larger graphic designer audience, especially when the remaining identified usability issues have been fixed.

With the current level of SVG support in web browsers, filtered may not be utilized to its full potential, as web designers need to be careful regarding the filter support in the web browsers. On the other hand, filtered could also serve as a vehicle of change by exposing these issues in browser implementations and by creating more demand for adequate filtering support.

Eventually the goal of filtered is not to become the definitive tool for editing SVG filters, but to be forgotten and abandoned. This will happen when the vector editing tools have reached a level where their filter support exceeds the possibilities of filtered. If filtered can drive this change forward, it has served its purpose.

Bibliography

[1] Adobe Systems Inc. *PostScript Language Reference Manual, 1st ed.* Addison-Wesley Longman Publishing Co., Inc., 1985.

[2] Adobe Systems Inc. *Adobe Type 1 Font Format, 2nd ed.* Addison-Wesley Longman Publishing Co., Inc., 1992.

[3] Adobe Systems Inc. Adobe Illustrator version 10 Reviewer's Guide, 2001. URL http://www.adobe.com/aboutadobe/pressroom/pressmaterials/pdfs/illustrator/Ai10RevGuidgFINAL09601.pdf.

[4] Adobe Systems Inc. SWF File Format Specification, version 19, 2012. URL http://www.adobe.com/go/swfspec.

[5] Adobe Systems Inc. Flash Player Release Notes, 2013. URL http://www.adobe.com/support/documentation/en/flashplayer/releasenotes.html. Retrieved 2013-02-21.

[6] Ola Andersson and ed. Scalable Vector Graphics (SVG) tiny 1.2 Specification, W3C Recommendation 22 December 2008, 2008. URL http://www.w3.org/TR/2008/REC-SVGTiny12-20081222/REC-SVGTiny12-20081222.pdf.

[7] Apple Inc. iTunes Connect Developer Guide, rev. 2013-01-10, jan 2013. URL http://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/iTunesConnect_Guide.pdf.

[8] Kent Beck and Cynthia Anders. *Extreme Programming Explained: Embrace Change, 2nd Edition*. Addison-Wesley, Boston, MA, USA, 2004. ISBN 0321278658.

[9] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile manifesto, 2001. URL http://agilemanifesto.org.

[10] Robin Berjon and ed. HTML5, a vocabulary and associated APIs for HTML and XHTML, W3C Candidate Recommendation 17 december 2012, dec 2012. URL http://www.w3.org/TR/2012/CR-html5-20121217/. Retrieved 2013-02-03.

[11] Benoit Bezaire and Lofton Henderson (editors). WebCGM 2.1, W3C Recommendation 01 March 2010, 2010. URL http://www.w3.org/TR/2010/REC-webcgm21-20100301/. Retrieved 2013-02-20.

[12] Stefan Blomkvist. Towards a model for bridging agile development and user-centered design. *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*, pages 219–244, 2005.

[13] Tim Bray and ed. Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation 26 November 2008, nov 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/. Retrieved 2013-02-20.

[14] Stephen Brooks. Concise texture editing. Technical report, University of Cambridge Computer Laboratory, 2004. URL www.cl.cam.ac.uk/techreports/UCAM-CL-TR-584.pdf.

[15] N. Cross. *Design Thinking: Understanding How Designers Think and Work*. Bloomsbury Academic, 2011. ISBN 9781847886378.

[16] Erik Dahlström and ed. Scalable Vector Graphics (SVG) 1.1 (Second Edition), W3C Recommendation 16 August 2011, 2011. URL http://www.w3.org/TR/SVG/REC-SVG11-20110816.pdf.

[17] Thomas DeWeese and Vincent Hardy. Introduction to the Batik project. In *SVG Open / Carto.net Developers Conference 2002 Conference Proceedings*, 2002. URL http://www.svgopen.org/2002/papers/deweese_hardy__batik_intro/.

[18] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002. ISBN 1558608486.

[19] ECMA International. Standard ECMA-262, ECMAScript Language Specification, 5.1 edition, jun 2011. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[20] Tolga Capin (editor). Mobile SVG Profiles: SVG Tiny and SVG Basic, W3C Recommendation 14 January 2003, edited in place 15 June 2009, 2003. URL http://www.w3.org/TR/2003/REC-SVGMobile-20030114/. Retrieved 2013-02-20.

[21] Jon Ferraiolo and ed. Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation 04 September 2001, 2001. URL http://www.w3.org/TR/2001/REC-SVG-20010904/REC-SVG-20010904.pdf.

[22] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural content generation for games: a survey, 2011. URL http://www.st.ewi.tudelft.nl/~iosup/pcg-g-surveyl1tomccap_rev_sub.pdf.

[23] Arnaud Le Hors and ed. Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation 07 April 2004, april 2004. URL http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/. Retrieved 2013-02-20.

[24] ISO. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. International Organization for Standardization, Geneva, Switzerland, 1994. URL http://www.iso.ch/cate/d18902.html.

[25] ISO. *ISO 13407:1999(E): Human-centred design processes for interactive systems*. International Organization for Standardization, Geneva, Switzerland, 1999.

[26] ISO. *ISO/IEC 8632-1:1999: Information technology — Computer graphics — Metafile for the storage and transfer of picture description information — Part 1: Functional specification*. International Organization for Standardization, Geneva, Switzerland, 1999. URL http://www.iso.ch/cate/d32378.html.

[27] ISO. *ISO/IEC 15948:2004: Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification*. International Organization for Standardization, Geneva, Switzerland, 2004. URL http://www.iso.ch/cate/d29581.html.

[28] ISO. *ISO 32000-1:2008. Document management — Portable document format — Part 1: PDF 1.7*. International Organization for Standardization, Geneva, Switzerland, 2008. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502.

[29] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 119–124, New York, NY, USA, 1991. ACM. ISBN 0-89791-383-3. doi: 10.1145/108844.108862. URL http://doi.acm.org/10.1145/108844.108862.

[30] Kiia Kallio. Scanline Edge-flag Algorithm for Antialiasing. In Ik Soo Lim and David Duce, editors, *Theory and Practice of Computer Graphics*, pages 81–88, Bangor, United Kingdom, 2007. Eurographics Association. ISBN 978-3-905673-63-0. doi: 10.2312/LocalChapterEvents/TPCG/TPCG07/081-088. URL http://mlab.uiah.fi/~kkallio/antialiasing/EdgeFlagAA.pdf.

[31] Clayton Lewis and John Rieman. Task-centered user interface design, a practical introduction, 1993. URL http://hcibib.org/tcuid/.

[32] David J. Malan and Henry H. Leitner. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 223–227, New York, NY, USA, 2007. ACM. ISBN 1-59593-361-1. doi: 10.1145/1227310.1227388. URL http://doi.acm.org/10.1145/1227310.1227388.

[33] Bruce Martin and Bashar Jano (editors). WAP Binary XML Content Format, W3C NOTE 24 June 1999, 1999. URL http://www.w3.org/1999/06/NOTE-wbxml-19990624/. Retrieved 2013-02-20.

[34] David M. Nichols and Michael B. Twidale. The usability of open source software. *First Monday*, 8(1), January 2003. URL http://http://firstmonday.org/article/view/1018/939. Retrieved 2013-03-20.

[35] David M. Nichols and Michael B. Twidale. Usability processes in open source projects. In *Software Process: Improvement and Practice*, volume 11, pages 149–162, March/April 2006.

[36] Jacob Nielsen. Jakob nielsen's alertbox: 10 usability heuristics, January 1995. URL http://www.nngroup.com/articles/ten-usability-heuristics/. Retrieved 2003-02-28.

[37] Jakob Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 373–380, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.142834. URL http://doi.acm.org/10.1145/142750.142834.

[38] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050.

[39] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 206–213, New York, NY, USA, 1993. ACM. ISBN 0-89791-575-5. doi: 10.1145/169059.169166. URL http://doi.acm.org/10.1145/169059.169166.

[40] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM. ISBN 0-201-50932-6. doi: 10.1145/97243.97281. URL http://doi.acm.org/10.1145/97243.97281.

[41] Donald. A. Norman. *The Design of Everyday Things*. Doubleday, 1988. ISBN 0-385-26774-6.

[42] Jef Raskin. Viewpoint: Intuitive equals familiar. *Commun. ACM*, 37(9):17–18, September 1994. ISSN 0001-0782. doi: 10.1145/182987.584629. URL http://doi.acm.org/10.1145/182987.584629.

[43] Daniel Rice and Robert J. Simpson (editors). OpenVG Specification, version 1.1, 2008. URL http://www.khronos.org/registry/vg/specs/openvg-1.1.pdf.

[44] Sony Computer Entertainment Inc. Business Development/Japan (1994-2004), 2013. URL http://www.scei.co.jp/corporate/data/bizdatajpn2004_e.html. Retrieved 2013-02-19.

[45] Sony Electronic Publishing Ltd. *Playstation Hardware, OS Hardware Guide, version 2.0*, 1994.

[46] K. Thulasiraman and N.S. Swamy. *Graphs: Theory and Algorithms*. Wiley, 1992. ISBN 9780471513568.

[47] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990. ISBN 978-0-9613921-1-6.

[48] John Warnock and Douglas K. Wyatt. A device independent graphics imaging model for use with raster devices. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '82, pages 313–319, New York, NY, USA, 1982. ACM. ISBN 0-89791-076-1. doi: 10.1145/800064.801297. URL http://doi.acm.org/10.1145/800064.801297.

[49] Wireless Application Protocol Forum, Ltd. WAP Architecture, Version 30-Apr-1998, 1998. URL www.wapforum.org/what/technical/SPEC-WAPArch-19980430.pdf.

Glossary

AI Adobe Illustrator. 29

algorithm is a step-by-step procedure for calculations. 11, 12, 23–25, 32

alpha is used as a name for the fourth color channel of pixels, representing pixel opacity rather than color value. 7

antialiasing is the process of reducing the jagged distortions in curves and diagonal edges so that they appear smoother. 10, 12, 28, 63

API application programming interface. 33

bitmap is a picture defined by a two-dimensional array of pixel values in computer memory, where each pixel consists of one or more bits. 7, 8, 10, 14–16, 23–26, 28, 33, 34, 50, 52, 61

C++ is a statically typed compiled programming language that adds object-oriented features to its predecessor, C. 14, 21, 33

CAD computer-aided design. 30

CD-ROM compact disk read-only memory. 22, 61

CGM Computer Graphics Metafile. 30

convolution filter calculates the output pixel value based on the weighted sum of values, defined by a convolution matrix, in the neighborhood of the input pixel. 25, 26

CSS Cascading Style Sheets. 14

DAG directed acyclic graph. 38

DOM document object model. 12, 14

DXF Drawing Exchange Format. 30

ECMAScript ECMAScript is a scripting language standardized in the ECMA-262 and ISO/IEC 16262 specifications. JavaScript is one commonly used dialect of ECMAScript. 12

EPS Encapsulated PostScript. 29

GB gigabyte, 10^9 or 1000000000 bytes. 14

GDI graphics device interface. 30

GPU graphics processing unit. 25

GUI graphical user interface. 12, 32, 33, 38, 57

gzip is an open source software application used for file compression and decompression. 12, 29

HTML HyperText Markup Language. 14, 44

HTML5 HTML Revision 5. 11

image filtering is a process that changes the appearance of a bitmap image or part of an image by altering the shades and colors of the pixels in some algorithmic manner. 7, 10–13

Java is an object-oriented programming language developed by Sun Microsystems. 7, 12, 33, 47, 58

JavaScript JavaScript is a scripting language commonly used in web pages. It is not to be confused with Java programming language. 12, 14

JPEG Joint Photographic Experts Group. 8, 23

MB megabyte, 10^6 or 1000000 bytes. 14, 22

open source refers to software in which the source code is available to the general public for use and modification free of charge. 7, 14, 32, 33, 60, 61, 69

OSS open source software. 60

PC personal computer. 21, 22

PDA personal digital assistant. 21, 22

PDF Portable Document Format. 10, 29

PNG Portable Network Graphics. 8

procedural texture is a texture described in an algorithmic way, instead of using a stored bitmap for the texture values. 13, 23–26

programming library is a collection of standard programs and subroutines that are stored and can be reused as components of other programs. 7, 28

PS PostScript. 29

rasterization is a process of creating a bitmap-based representation of data defined in vector format. 10, 14

SMIL Synchronized Multimedia Integration Language. 14

stack-based programming language is a type of a data-structured programming language that is based on the stack data structure. 30

surface normal is a vector perpendicular to the surface of a 3D model. 25

SVG Scalable Vector Graphic. 7, 10–16, 28–38, 40, 42–46, 54, 57, 61, 64, 69

SVGB SVG Basic. 12, 29

SVGT SVG Tiny. 12, 29

SVGZ compressed SVG. 12, 29

SWF Shockwave Flash. 28–30

texture synthesis is the process of algorithmically constructing a digital image from a smaller sample image. 13, 23–25

tokenization is the process of breaking a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. 29, 32

UI user interface. 23, 24, 33, 35, 46, 47, 54, 55

URL universal resource locator. 46

vector graphic is the use of geometrical primitives such as points, lines and curves, all based on mathematical expressions, to represent images in computer graphics. 7–15, 24–26, 28–30, 32, 35, 61, 64, 69

W3C World Wide Web Consortium. 11, 29, 30

WAP Wireless Application Protocol. 29

WBXML WAP Binary XML. 29

WMF Windows Metafile. 30

WYSIWYG “what you see is what you get”. 31

X-Forge™ was a mobile 3D game engine developed by Fathammer, Ltd. 21, 23, 27

XML Extensible Markup Language. 11, 12, 14, 29, 30, 32, 36, 44

Appendix A: User's Guide for filtered

A.1 Introduction

filtered is a program aimed at editing filters in SVG files. SVG (Scalable Vector Graphics) is a XML-based vector graphics format. SVG format is defined by World Wide Web Consortium, and the full technical specification of the format is available at http://www.w3.org/Graphics/SVG/.

SVG Files can be edited with several vector graphics packages, for instance Adobe Illustrator or Corel Draw. There are also packages for editing SVG natively, such as Inkscape.

However, the problem with all the available software is poor support for SVG filter effects. Filter effects are an important feature in SVG that set the format apart from most other vector formats. Cleverly used, they allow bitmap graphics like effects, such as glows, drop shadows etc. and much more, such as procedural texture generation. Since SVG is a vector format, filter effects are resolution-independent, and unlike bitmap files that can grow to be several hundred megabytes, SVG files are relatively small even with high resolution output.

In most of the tools, editing SVG filters is possible only in the SVG source code format. This makes the task very cumbersome and prone to errors, especially for graphic artists. Creating SVG filters has been closer to programming than designing, and in most of the tools artists are left with a basic set of pre-defined filters to use. The aim of filtered is to allow artists and graphic designers unleash the full power of SVG filter effects in an usable and effective manner by allowing visual editing of the filter effects.

A typical mistake with visual editing – usual in the first visual HTML editors for instance – is to place restrictions for the format in use. filtered however allows access to all parameters defined by the SVG 1.0 W3C Recommendation.

A.2 Principles of SVG filters

SVG is an XML format and thus can be edited as source code. Another representation is the DOM (Document Object Model) tree, that results when a file has been succesfully parsed. There are lots of web resources available for understanding SVG format, SVG web site at http://www.w3.org/Graphics/SVG/ is a good starting place. However, using filtered doesn't require thorough knowledge of the format, understanding basic principles is enough.

SVG files consist of various elements, such as groups, paths, shapes etc. Filter is defined as one such element. To make the format more effective, filters are not defined along with the graphical elements to which they are applied, but earlier in a document as a separate definition. This makes it possible to use the same filter for several different graphical elements. One SVG file can of course contain several filter definitions, referenced by various elements within the file. So unlike for instance a path element, which gets drawn when it is encountered in the file, filter element itself doesn't have a visual representation, but is

applied to a graphical element that references it.

Filters can be applied to just the basic graphical elements such as paths, or groups of elements. Using filters is easy, just set the "filter" property of the element to reference to the specific filter by its name. The way to perform this depends of course on the editor in use, in source code the notation for a filtered group is for instance:

```
<g filter="url(#MyFilter)" >
 ...elements in group...
</g>
```

Each SVG filter contains a selection of filter primitives that actually define what the filter does. Filter primitives are arranged into a sequence that defines the order of processing. Each primitive has one output, and possibly a number of inputs that are connected to other primitives of the filter. Each filter primitive can take the output of any previous primitive of the filter as its input. The primitive then performs some image processing function on its input(s), and generates a processed image to be used as an input for the successive filter primitives.

The filter itself has six "global" inputs, and one output. The inputs are:

- Source graphic – the graphical element to which the filter is connected.
- Background image – the background image over which the element is drawn.
- Source alpha – the transparency of the source graphic.
- Background alpha – the transparency of the background graphic.
- Fill paint – the paint used for filling the source graphic element.
- Stroke paint – the paint used for stroking the source graphic element.

A note about background image: background image and alpha are not necessarily available for a filter. To enable it, the "enable-background" property of some parent container element of the filtered graphic element has to have value of "new". See Scalable Vector Graphics (SVG) 1.0 Specification for more information.

The final output of the filter is the output from the last filter primitive. In SVG file, the filter primitives are represented as successive elements in the code, and in filtered, they are represented as elements stacked on top of each other, where the final output is the result of the topmost primitive.

A.3 filtered basics

filtered allows editing of only one SVG file at the time. Often this file is the file containing filter definitions for the vector drawing package, but it can also be an SVG file where filters are actually in use and whose filter definitions require adusting. File operations, such as loading and saving, can be accessed from the "File" menu.

As the file being edited doesn't necessarily use the defined filters at all, filters are displayed using a pre-defined preset graphic. There is a number of presets in the program to choose from. It is also possible to preview the filters using the original graphics contained in the file being edited. There can be a number of previews displaying various filters with various presets visible at the same time – considering this it is only a good thing that it is possible to edit only one file at the time! New previews can be opened from the "New Preview" item in the "Window" menu.

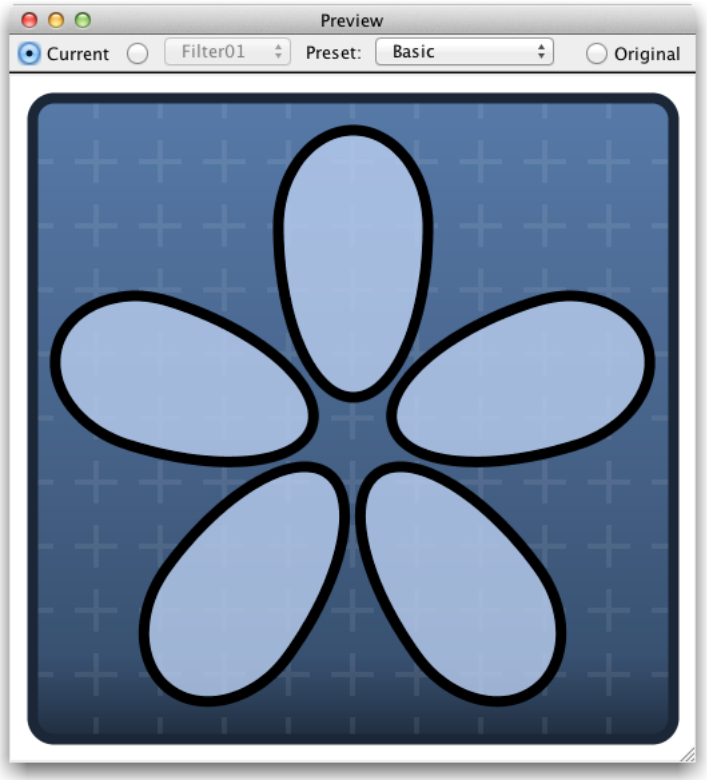


Figure A.1: Preview window interface.

The filters are displayed using a layer graph window. The layer graph shows the stack of filter primitives and connections between them. Only one filter at a time is shown, and it is possible to choose the filter to be edited from a drop-down menu in the layer graph. Layer graph is visible when the program starts, but if it is closed, it can be re-displayed by selecting "Layer Graph" from the "Window" menu.

Layer graph window has functions for adding, duplicating and removing filters. Also filter settings can be edited. There is similar functionality for filter primitives as well. Adding, removing and duplicating primitives is done from the buttons in the bottom of the layer graph. Filter primitive settings are displayed by double-clicking the filter primitive. It is possible to access the functions also from a drop-down menu.

The order of filter primitives can be changed by dragging them around in the layer graph. The connections can be edited by dragging from the input of a primitive to an output in a primitive below in the graph. As there can be only one output connected to each input, the existing connection will be disconnected and a new connection is formed. When drag-

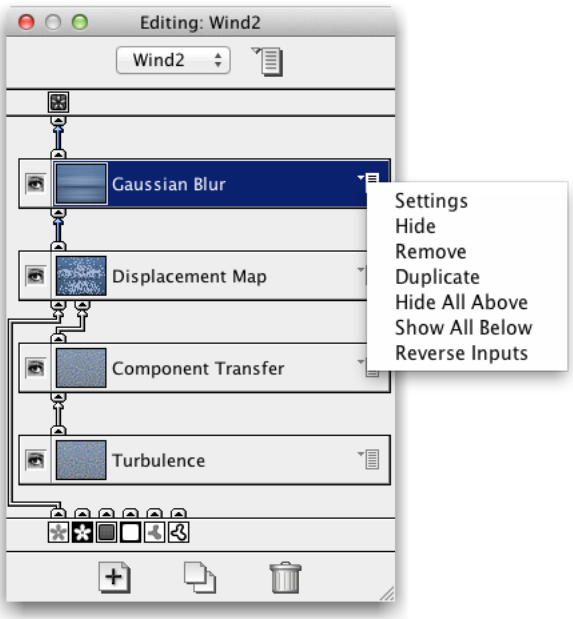


Figure A.2: Layer graph window interface.

ging the connections, the connection is displayed with red if it doesn't form an acceptable connection, and in green when it does.

The operations related to filters and filter primitives can also be accessed from the "Filter" menu.

A.4 Filter Settings

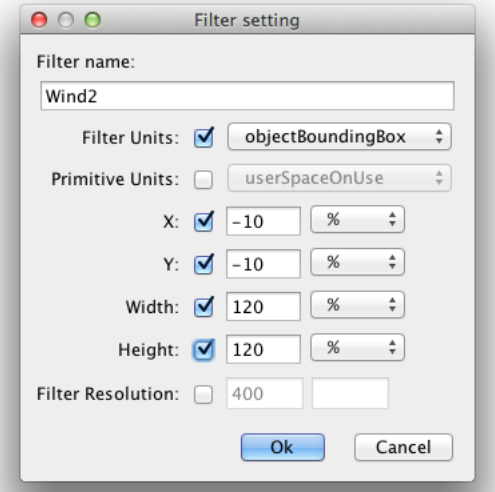


Figure A.3: Filter settings dialog interface.

Filter settings can be accessed from a drop-down menu at the top of the layer graph window, or from the window menubar.

Filter settings include the filter name and the size of the filter area. The filter area is defined either relative to the filtered object (objectBoundingBox) or as absolute coordinates in user space (userSpaceOnUse). Some filters primitives, such as Gaussian blur, displacement map or offset may require larger area than the bounding box of the object. This can be achieved by setting x and y values to negative values and width and height values to larger than 100%. The default values used by filtered are -10% for x and y, and 120% for width and height.

A.5 Defining Filter Usage in the Original Image

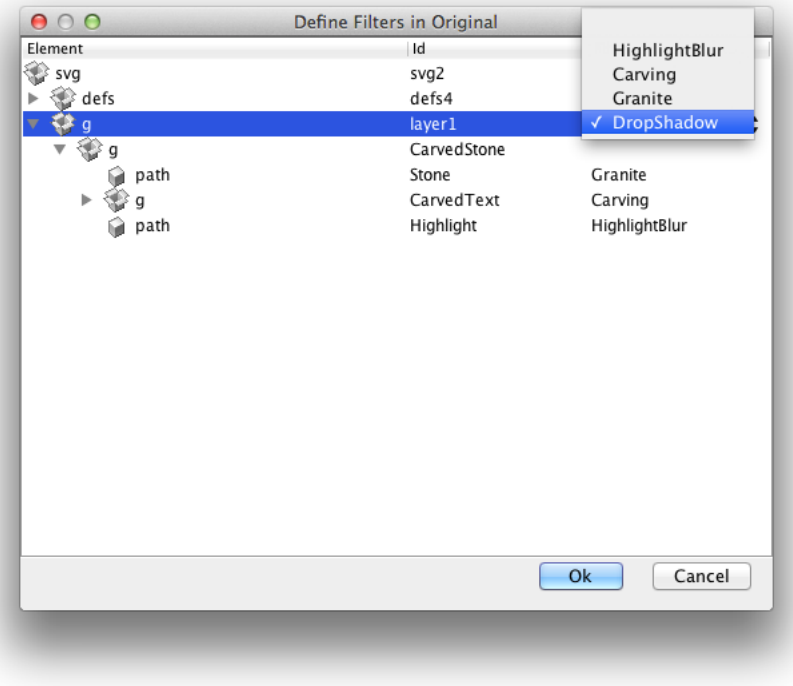


Figure A.4: Dialog interface for defining filters in the original image.

filtered is targeted to be used for editing SVG filters, not complete SVG images. It doesn't include tools for editing SVG paths or other graphics objects.

However, it provides functionality for assigning filter primitives to elements in the SVG document. For this, choose "Define Filters in Original..." from the "Edit"-menu. A dialog will pop up, showing the XML tree of the image on the left column (double-clicking container elements will expose their children). In the middle column the id attribute of the element is displayed, and the right column contains the name of the filter attached to the element. It is possible to modify this by clicking the cell; a drop-down menu containing all filters defined in the file will appear.

A.6 Using Filter Libraries

Filters can be collected to "libraries", i.e. SVG files containing a large collection of filters, but no graphic using them. Editing these files is similar to those files containing also some image content, but it's possible to use only the presets for previewing the filters, as the file contains no image, just the filter definitions.

When making new images – e.g. with Inkscape – and adding filters to them in filtered such filter libraries can be used in filtered for getting the filter definitions. "Import..." function in the file menu exposes a mechanism, where all filters from the imported file will be added to the file being currently edited. If the files contain filters with identical names, a running number is added to the end of the name.

After assigning the desired filters from the library file to the image elements, the result of this may be that the document now contain lots of unnecessary filters. It is possible to initiate a cleanup sweep on these by choosing "Remove Unused Filters" from the "Filter" menu in filtered.

A.7 Filter Primitives

The selection of filter primitives is defined by SVG standard, and for some filter primitives the effects are not obvious. The interface for filter primitive settings is modelled to support all the features the SVG standard imposes, but rarely used options are isolated behind "Additional Attributes" setting. Adjusting the settings of filter primitives requires some basic knowledge of image processing with other programs, such as Adobe Photoshop. Also going through the example files provided with filtered may help in learning the effects and correct use of filter primitives. More thorough explanation about filter primitives is available at Scalable Vector Graphics (SVG) 1.0 Specification.

Most of the attributes for filter primitives can be undefined, which means that the default value will be used. A checkbox on the right of the attribute defines whether the attribute is defined or not. Some attributes are used for defining two-dimensional values, such as units in x and y direction. In such cases, it is often possible to leave the later value blank, which means that the first value is used for both dimensions.

Common additional attributes for all filter effects are X, Y, Width and Height. These define the filter primitive subregion, which is the area used for this effect. If filter primitive subregion is not defined, the area defined in the previous filter primitive is used. Filter Color Interpolation defines the color space used for color interpolation.

Each primitive dialog has a "Preview" button. Clicking this button will temporarily apply the changes to the primitive, and they will be displayed in the preview window(s).

A.7.1 Blend

SVG filter primitive name: feBlend

Blends together two input images using commonly used imaging software blending modes. The available modes are normal, multiply, screen, darken and lighten. Default value is

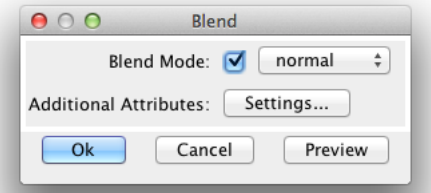


Figure A.5: Blend dialog interface.

normal.

A.7.2 Color Matrix

SVG filter primitive name: feColorMatrix

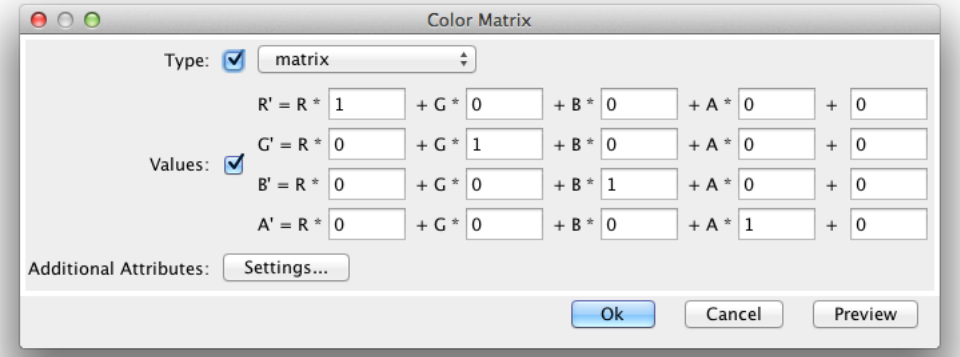


Figure A.6: Color matrix dialog interface.

Applies a matrix transformation on the RGBA values of a pixel. This can be used for various effects such as desaturating or adjusting the hue of an image. The type of the effect can be matrix, saturate, hueRotate or luminanceToAlpha. For matrix, the values define a 5x4 matrix that is used for calculating the new color value based on matrix arithmetic. Saturate provides a desaturating effect using a single value between 0 and 1. HueRotate rotates the hue of the color using a value defined in degrees. LuminanceToAlpha transfers the luminance of a RGB image to the alpha channel.

A.7.3 Component Transfer

SVG filter primitive name: feComponentTransfer

Applies a transfer function for each channel of an image. The function can be defined separately for each channel, for channels to which it is not defined, the values won't be changed. Type identity provides a function that has no effect. For type table, a table of

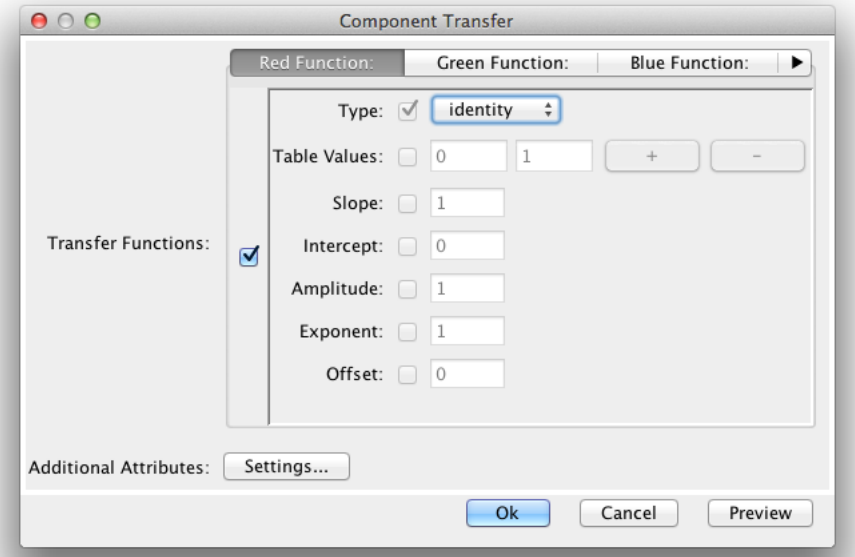


Figure A.7: Component transfer dialog interface.

values is used for linear interpolation of values. For type discrecte, the table is used for defining a step function consisting of n values. The table size can be changed from the two buttons "+" and "-" after the table values. For type linear, two values, slope and intercept are provided. Slope defines the slope of the function, and intercept the value the function provides for o color value. For type gamma, the function is an exponential function, whose amplitude, exponent and offset can be defined.

A.7.4 Composite

SVG filter primitive name: feComposite

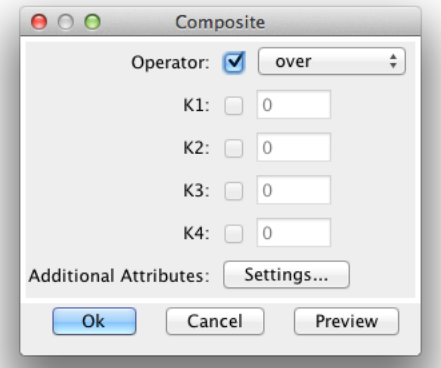


Figure A.8: Composite dialog interface.

Performs a pixel-wise composition operation of two images using Porter-Duff composition operations. The operations are over, in, atop, out and xor. The operators basically define how to use the alpha channels of two images when combining them.

Another operation, artihmetic, determines the colour value by using arithmetic function $result = k1 \times i1 \times i2 + k2 \times i1 + k3 \times i2 + k4$, where $i1$ is the pixel value from first input and $i2$ is the pixel value from the second input. This can be useful for instance when combining images produced with lighting operations with texture data. Good equation for such cases is e.g. $k1 = 1$, $k2 = 0$, $k3 = 0$, $k4 = 0$, but it is possible to also use some small values for $k2$ and $k3$ as long as $k4$ is adjusted with the similar negative value.

A.7.5 Convolve Matrix

SVG filter primitive name: feConvolveMatrix

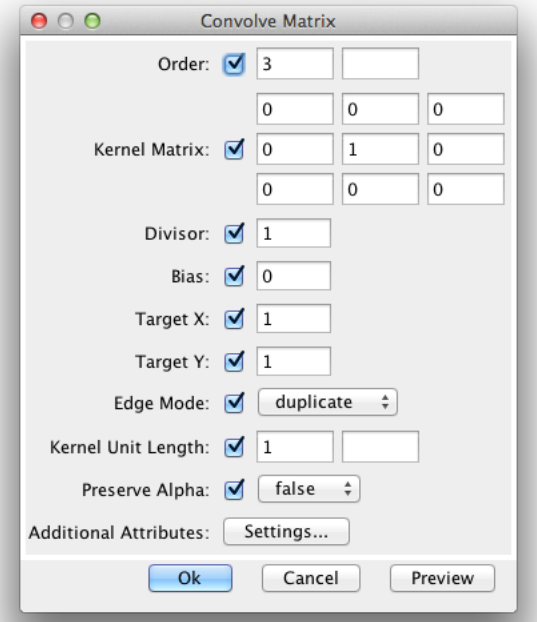


Figure A.9: Convolve matrix dialog interface.

Defines a convolution matrix used for filtering the image. It can be used for such effects as blurring, sharpening and embossing the image. Order defines the size of the convolution kernel. Kernel Matrix defines the matrix itself. Divisor is the divisor for the matrix values, default value is the sum of all values in the matrix. Bias is the bias value added to the result of kernel operation (currently unsupported by Batik). Target X and Y values define the position of the center cell of the matrix, default is the center of the matrix. Edge Mode defines the wrapping of the pixels at the edges of the area. Kernel Unit Length is used for defining the pixel size used for the operation. This is required in order to keep the results of the operation resolution independent. Preserve alpha defines whether the convolution is done also for the alpha channel or just for RGB channels.

A.7.6 Diffuse Lighting

SVG filter primitive name: feDiffuseLighting

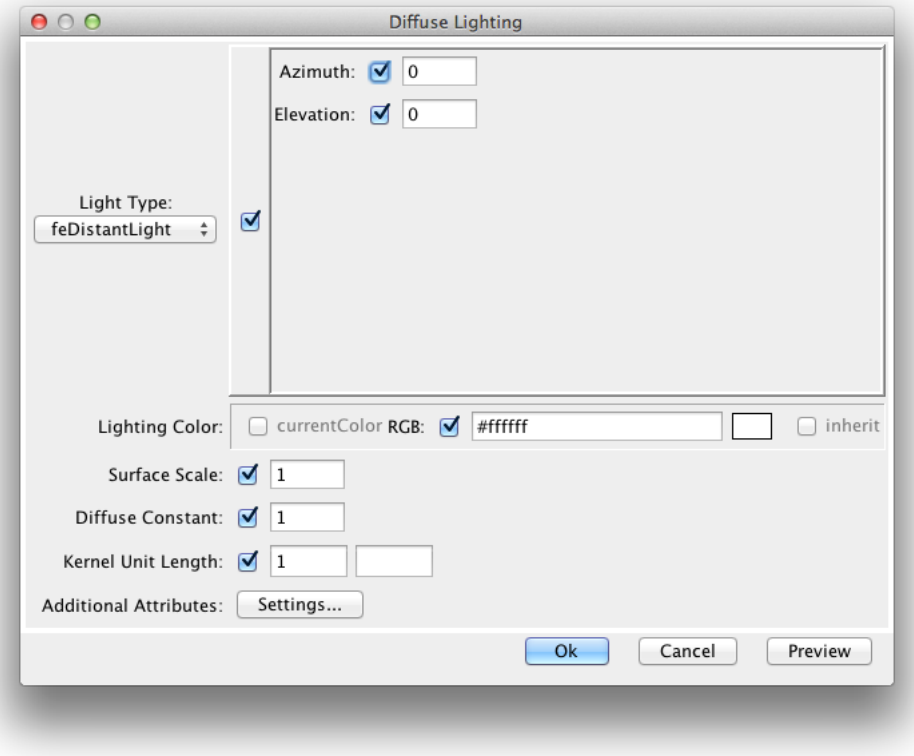


Figure A.10: Diffuse Lighting dialog interface.

Performs a diffuse lighting operation using the alpha channel of the input image as height data. Light can be one of the three possible light types: feDistantLight, fePointLight or feSpotLight. Lighting Color defines the color of the ligh. Surface Scale defines the height of the surface data. Diffuse Consant defines the brightness of the light. Kernel Unit Length define the size of the pixel used for the operation.

Light type feDistantLight is defined with two angles that define the direction of light. Light type fePointLight is defined by giving a 3D position for the light. Light type feSpotLight is defined by the 3D position of the light, 3D position of the light target, Specular Exponent that defines the fall-off of the light, and Limiting Cone Angle that defines the size of the spot light cone.

A.7.7 Displacement Map

SVG filter primitive name: feDisplacementMap

Displacement map displaces the first input image with a value read from the second inut image. The size of the displacement is defined with Scale value. The channels used for displacement in X and Y direction can be selected with X Direction and Y Direction settings.

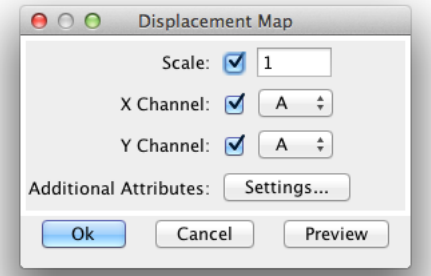


Figure A.11: Displacement Map dialog interface.

A.7.8 Flood

SVG filter primitive name: feFlood

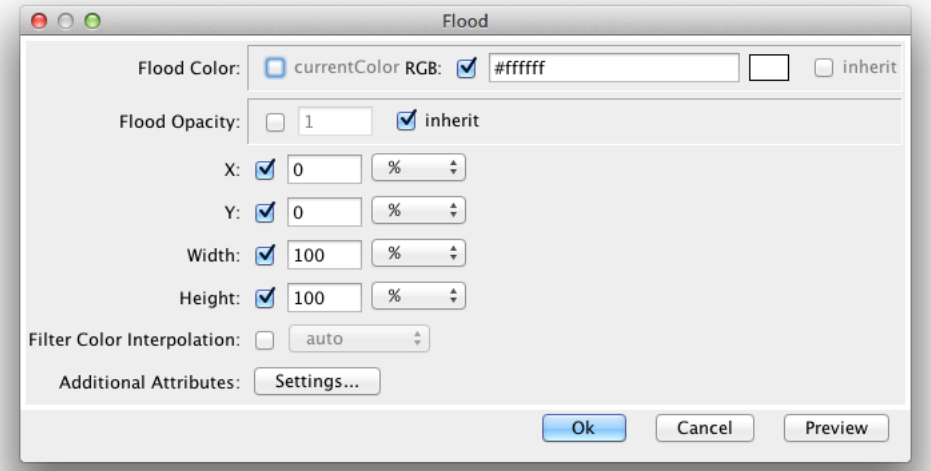


Figure A.12: Flood dialog interface.

Fills an area defined by filter primitive subregion with color and opacity defined by Filter Color and Filter Opacity. Filter primitive subregion is defined by X, Y, Width and Height. The flood operation takes one input image by the SVG specification, but the contents of the input image are ignored, and are flooded inside the area defied by the filter primiive subregion, and left to transparent black outside the area.

A.7.9 Gaussian Blur

SVG filter primitive name: feGaussianBlur

Performs a gaussian blur operation on the input image. Blur Size defines the amount of blur. Two values define the amount of blur in x and y directions, and if only one value is

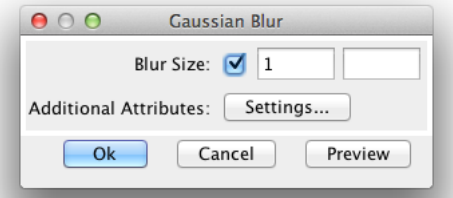


Figure A.13: Gaussian Blur dialog interface.

provided, it is used for both directions.

A.7.10 Image

SVG filter primitive name: feImage

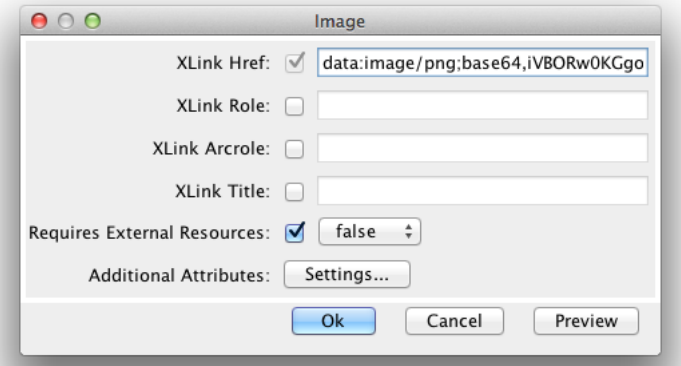


Figure A.14: Image dialog interface.

Produces an image source similar to source graphic using an external image. The image can be either reference inside the SVG document, in which case the Xlink Href is in the format "#Reference", or external document, in which case the Xlink Href is an URL of the image, such as "file:///C:/SVG/MyImage.jpg". It's also possible to set the XLink Href value to contain base64-encoded image data directly. For external images, Requires External Resources must be set to true. Xlink Role, Xlink Arcrole and Xlink Title are defined in the Scalable Vector Graphics (SVG) 1.0 Specification.

A.7.11 Merge

SVG filter primitive name: feMerge

Merges together a number of inputs. The inputs on the left are placed on top of the ones on the right. Number of Inputs defines the number of inputs.

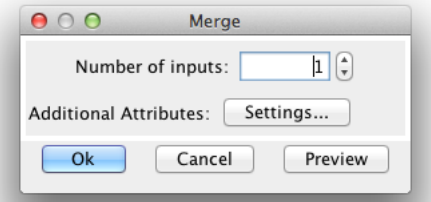


Figure A.15: Merge dialog interface.

A.7.12 Morphology

SVG filter primitive name: feMorphology

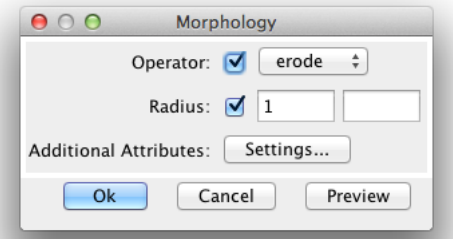


Figure A.16: Morphology dialog interface.

Performs “fattening” or “thinning” operation on artwork. Operation defines whether the question is about “fattening” (dilate) or “thinning” (erode). Radius defines the size of the operation.

A.7.13 Offset

SVG filter primitive name: feOffset

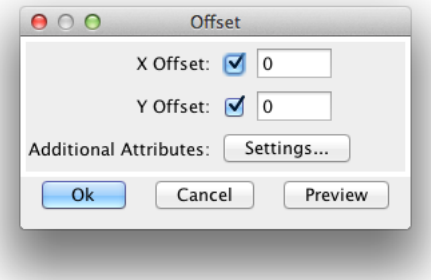


Figure A.17: Offset dialog interface.

Offsets the image by the amount defined by X Offset and Y Offset.

A.7.14 Specular Lighting

SVG filter primitive name: feSpecularLighting

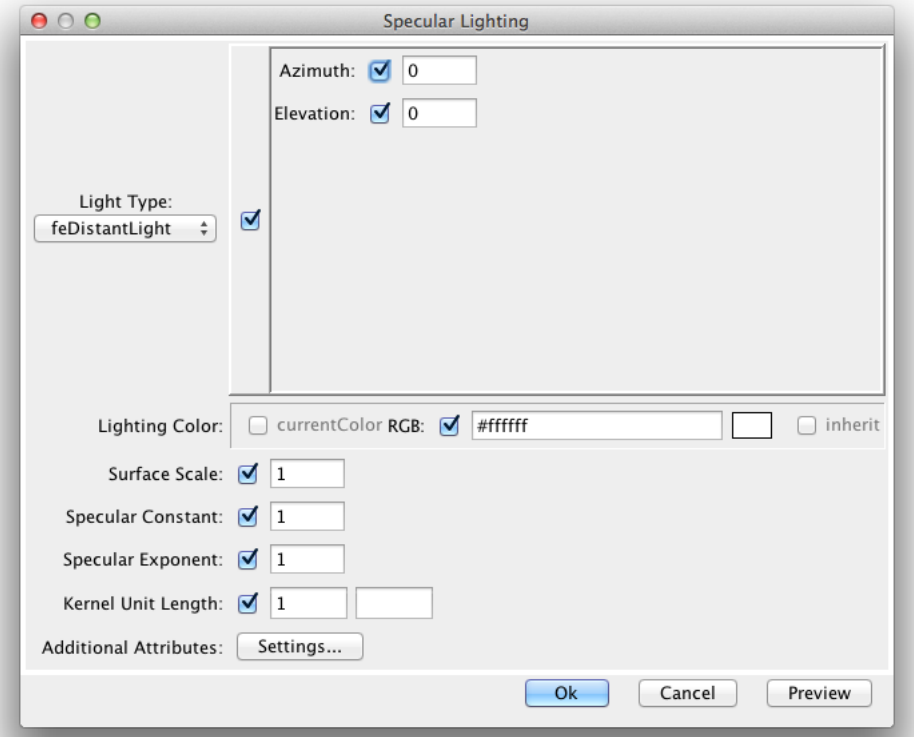


Figure A.18: Specular Lighting dialog interface.

Performs a diffuse lighting operation using the alpha channel of the input image as height data. Light can be one of the three possible light types: feDistantLight, fePointLight or feSpotLight. Lighting Color defines the color of the ligh. Surface Scale defines the height of the surface data. Specular Consant defines the brightness of the light and Specular Exponent defines the size of the specular hotspot. Kernel Unit Length define the size of the pixel used for the operation.

See Diffuse Lighting for more information about light types.

A.7.15 Tile

SVG filter primitive name: feTile

Tile takes the input image and tiles it over the area defined by X, Y, Width and Height. The size of the tile depends on the filter primitive area defined in the input primitive, so in order to get a tiling effect it may be necessary to adjust the X, Y, Width and Height values of the input primitive.

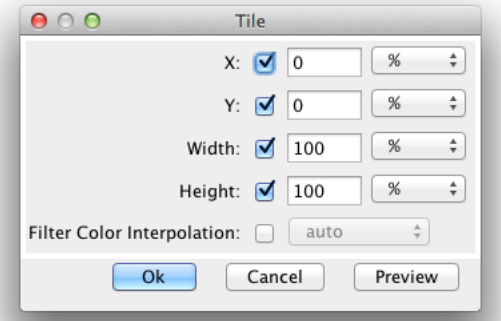


Figure A.19: Tile dialog interface.

A.7.16 Turbulence

SVG filter primitive name: feTurbulence

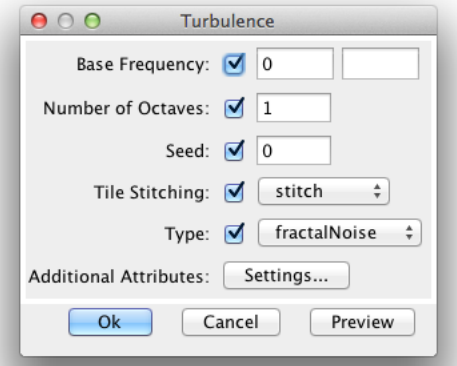


Figure A.20: Turbulence dialog interface.

Produces a turbulence image using Perlin turbulence function. This can be used for creating texture effects such as clouds or marble. The Base Frequency defines the base frequencey of the noise. Values close to 1 produde higher frequency noise and values closer to 0 produce lower frequency noise. Number of Octaves defines how many octaves will be used in the function. Seed is the starting number for the pseudo random generator. Tile Stitching defines if the turbulence generator tries to stitch the turbulence tiles at their edges. Type defines the type of the generated turbulence.

A.8 Using filtered with Inkscape

Inkscape is one of the best SVG editing tools – and it's a also open source. However, the filter support in Inkscape is not yet mature enough, and some of the filters don't get rendered

correctly. It is still possible to use filtered as a companion for Inkscape when editing filters. Here are some tips how to proceed.

In general, the editing flow between filtered and Inkscape can be done by loading a file to both editors simultaneously. After editing the file in Inkscape, choose “Save”, then switch to filtered and choose “Reload”. This will load a fresh copy of the file. When done with the filters in filtered, choose “Save”, then switch to Inkscape and choose “Revert”. This will now open the file including the changes done in filtered.

Inkscape doesn’t render all filters correctly. When editing with filtered, it’s best to use the display mode “No Filters” in Inkscape. This also means that editing the geometry is easier when the actual geometry is shown, not the potentially distorted result the filtering creates.

It is a good idea to give illustrative names to elements so it’s easier to recognize them in filtered. The name visible in filtered is the XML id of the element, but by default the name is just a running number in Inkscape and hard to find for filter assignment in filtered. This can be changed from “Object”->“Object Properties...” in Inkscape.

In order to use backgroud graphic for the filters, `enable-background="new"` attribute should be defined by some ancestor of the element using the filter. Inkscape doesn’t have a direct GUI item for setting this, but it can be done from Inkscape’s built-in XML editor. Just choose some ancestor group of the element – this can be a layer as well – write *enable-background* as the attribute name, *true* as the value and click “Set”.

Appendix B: Comparison of Existing Texture Generators

This comparison of texture generation software was done in 2002. It does not include newer products.

Most texture generator programs in 2002 were based on traditional procedural texture generation methods, and they either served as simple applications for producing web backgrounds or as front ends to the procedural texture programming language of some 3D package, such as Renderman (http://renderman.pixar.com/) or POV-Ray (http://www.povray.org/). The list of texture generators presented here does not include such texture generator packages, but only those that incorporate the ideas of image processing into the texture generation or otherwise use ideas that are relevant to the thesis.

Most modern image processing and paint programs – such as Adobe Photoshop (http://www.adobe.com/products/photoshop.html) or The Gimp (http://www.gimp.org/) – can be also used for creating textures in this manner by using scripts and filtering techniques. Such image processing packages are also left out of this comparison.

Common to all texture generators listed below is that they are 2D only programs; no 3D model feature extraction or other 3D features are available, excluding of course the 3D nature of volumetric procedural textures. If the program is based on procedural textures, it is listed because either it incorporates some image processing ideas, or it has a user interface that is interesting from the point of view of interface design.

B.1 DarkTree 2.0

Homepage: http://www.darksim.com/ Platform: PC Windows

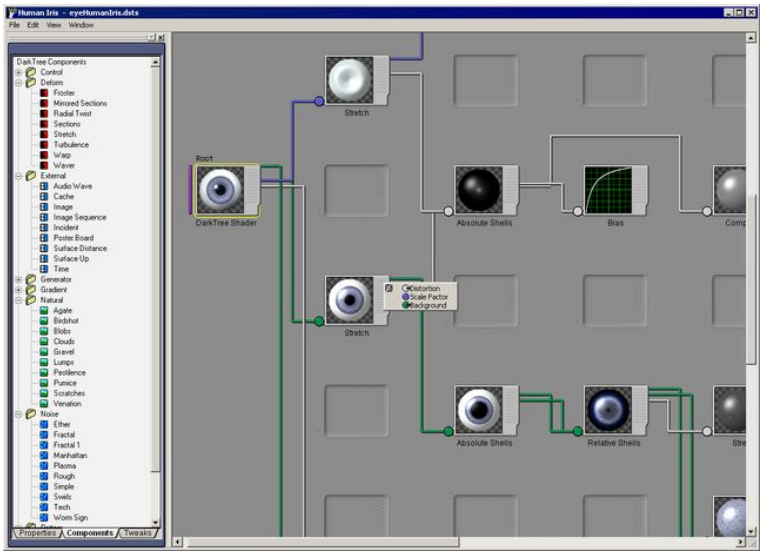


Figure B.1: DarkTree 2.0 user interface.

DarkTree 2.0 is a program for creating procedural textures using a graph – or tree – metaphor. In addition to the stand-alone program that is used for generating the texture description files, there are rendering plugins for several rendering packages, allowing the use of procedural textures in those programs. DarkTree can also render the textures into bitmaps for use in games or packages with no plugin support. There are plenty of operations for generating the textures, and great variety of results can be achieved. Because there was no demo available of the product, actual usability and speed of the product can't be evaluated. However, reading the downloadable manual gave the impression of intuitive and usable UI, proving that graph-concept can indeed be used in the interface. The program is clearly aimed at professionals; it is very thorough and supports major professional 3D packages. It has also been used in movie and computer game production. Also advanced hobbyists may find the program attractive; although it has many features, the interface seems to be elegant making it usable also for less-experienced users.

B.2 Impact Texture Studio

Homepage: http://www.scene.org/its/ Platform: PC DOS

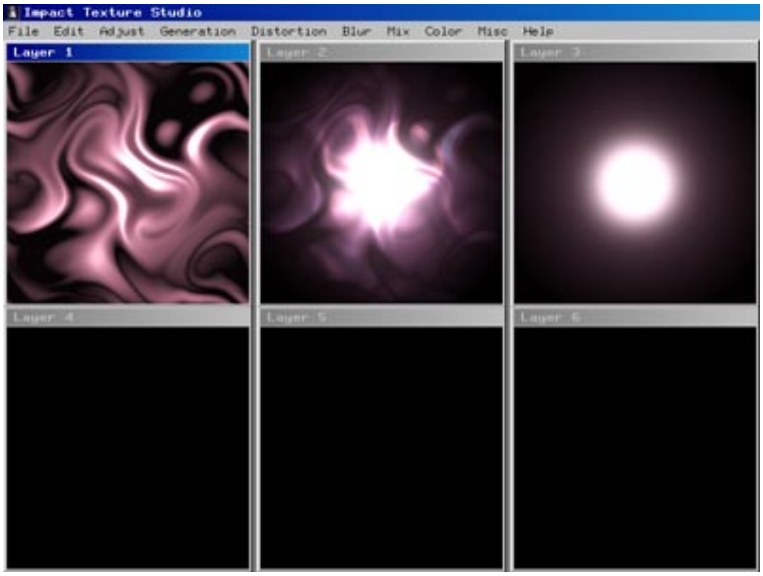


Figure B.2: Impact Texture Studio user interface.

Impact Texture Studio is based on the idea of creating textures with image processing commands. It has various tools for generating, filtering and adjusting the image. When the artist is creating the texture, the program makes a list of commands that has been used. This script-like list is then saved as the texture file – although it is also possible to save the texture as a bitmap. The script is not editable, however, so it is just a recording of the texture creation process, and editing the textures is difficult. There is unlimited undo, so the only way to edit the script is to use the undo for removing the last function from the script. The program has six buffers in use at once, and this makes it possible to have more complex texture constructions than with programs with layering approach. Although the set of filters is relatively limited, the program can be used for producing a wide variety of textures, but

the texture resolution is limited to 256*256. Lack of editability and use of buffers require good knowledge and understanding of the texture creation process, as it is not visualized in the program. All this means that Impact Texture Studio is not a good tool for beginners. The program is rather useless for professionals as well because of the lack of editability and small texture size. The program seems to be aimed especially at hobbyists on the "demo scene", as they benefit greatly from the small file size of the textures – if they get to use the same texture engine for creating the textures in their own applications.

B.3 Infinity Textures 2.02

Homepage: http://www.i-tex.de Platform: PC Windows

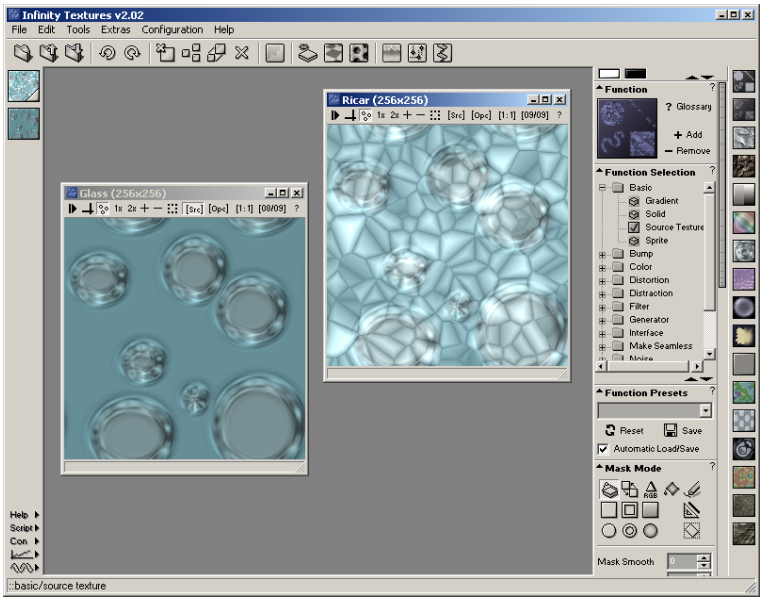


Figure B.3: Infinity Textures 2.02 user interface.

Infinity Textures is a polished texture generation program with lots of features. The user interface is rather complex and un-intuitive, and obtaining good results is difficult. The program has a limited set of drawing commands in addition to lots of filters. Most of the filters are special purpose filters, generating exotic effects, and some of the most basic filters are missing. Filters are applied one after another to an image, and although there is multiple undo and redo, it is not possible to edit commands already executed. Textures are saved as standard bitmaps; there is no internal texture format. The program features scripts for generating textures, but scripts can't be recorded, only typed in. Scripting language is simple, and does not include program control structures such as loops or jumps. The program seems to be aimed at advanced hobbyists, as professional artists would obtain better results with traditional image processing packages, but for the beginners the program is too difficult to use.

B.4 SynTex

Homepage: http://www.syntheticrealms.com/ Platform: PC Windows

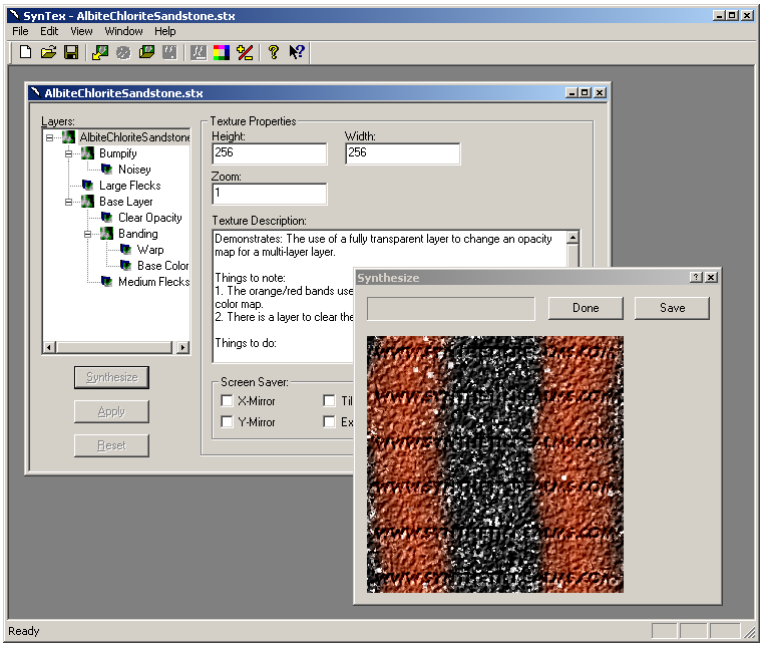


Figure B.4: SynTex user interface.

SynTex is a texture creation program based on the layer metaphor. It doesn't use filter approach, but the textures on the layers are created using traditional procedural texture methods. Layers, however, can be combined with various blending methods. Alpha masks are also supported. Syntex stores the textures as a freely editable layer structure. Textures can also be synthesized to a bitmap of any size. Synthesizing process is relatively slow. The program is relatively easy to grasp, but the results are limited and synthetic looking because of the procedural texture creation process. Program seems to be aimed at hobbyists, as the results are not necessarily satisfying advanced needs.

B.5 Texture Creator

Homepage: http://www.threedgraphics.com/ Platform: PC Windows

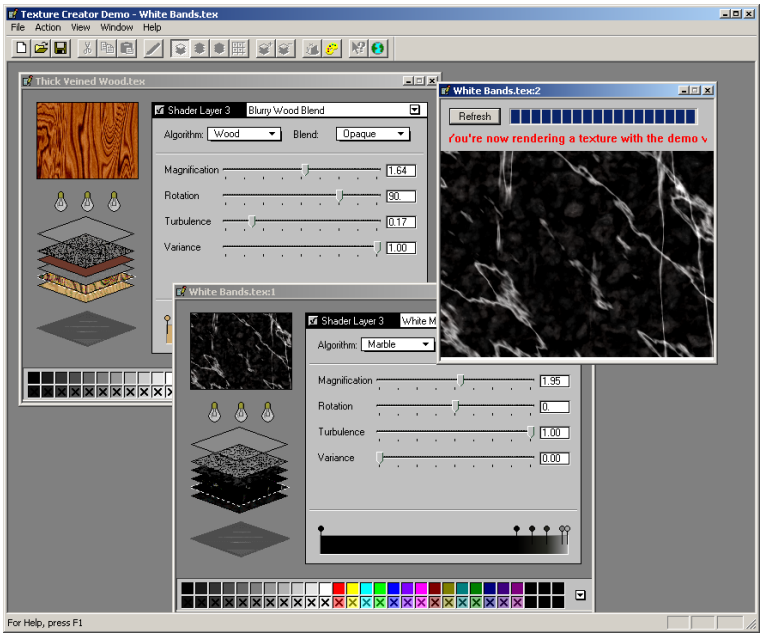


Figure B.5: Texture Creator user interface.

Texture Creator is generating the textures with traditional procedural texture approach. It uses layers for composing the textures, but there is no way of grouping layers, they are just piled on top of each other. This however makes the user interface very straightforward and easy to use. It is still not perfect and there are some annoying features. For instance scaling the texture layer has to be done by entering numbers instead of visible manipulation. As Texture Creator is based on traditional procedural approach, it has its own file format, and textures can be rendered to any size. Apparently the texture creation engine is capable of producing 3D textures as well, because according to the company's website there are rendering plugins for Lightwave and 3D Studio MAX under construction. The same engine has been used also in other products, such as Corel Texture and Adobe Texture Maker. Engine is faster than in SynTex for instance, but as the texture creation process is rather simple, the results are limited. There are some tasks where the program performs nicely, it is for instance possible to create rather convincing stone or wood textures as algorithms for those seem to be good. The program suits beginners because of its simplicity, but would suit also advanced hobbyists and maybe even professionals in situations where it is necessary to get textures for surfaces that the program handles well.

Appendix C: SVG Open 2003 Article

Using SVG for graphically rich 2D content in mobile 3D games

Keywords: 3D, mobile gaming, texture generation, SVG

Kiia Kallio
Software Engineer
Fathammer Oy
Helsinki
Finland
kkkallio@fathammer.com
http://www.fathammer.com

Biography

Kiia Kallio works as a software engineer in Fathammer, a company creating 3D game technology for mobile devices. He has long experience in the computer game industry, both as a programmer and a graphic designer. Prior to his career at Fathammer, he worked at Remedy Entertainment in a team that made "Max Payne", an award-winning 3D action game.

Abstract

Traditionally, textures and other 2D content in 3D games have been based on bitmaps. In a mobile 3D gaming environment the storage space limitations pose new challenges for the game content. Although processing power and runtime memory limitations allow visual quality comparable to Sony Playstation, the storage space is reduced to a fraction – instead of a CD-ROM the game should fit into a few megabyte memory cartridge.

The benefits of vector graphics, independency of resolution and small storage space requirements, have not been generally considered significant from the point of view of traditional 3D game developers. In mobile 3D gaming, these make vector graphics an attractive alternative.

The biggest problem with vector formats is the limited artistic expression. Vector graphics tend to produce images that are not suitable for rendering natural materials – things that are essential to rich 3D environments. However, SVG has a mechanism that can be used to get around these limitations: filter effects. SVG filters are powerful in the hands of a programmer who knows image processing and SVG specification inside out. The lack of good tools for artists is a problem though. SVG Filter Editor, included with the X-Forge™ 3D game engine, is a tool that enables artists to construct the filters visually without writing SVG code, but still allowing access to all the filtering features of SVG specification.

The SVG file format is not optimal for tight storage limitations in one aspect: as a text based format it takes more space than necessary. Also the XML parser adds up to the size of the executable, and DOM tree has quite large runtime memory requirements. This is solved in X-Forge™ by moving the parsing process from the load time to the content creation tool chain by using a pre-parsed binary format.

Table of Contents

Problem background

Mobile 3D gaming environment

Today's high-end mobile devices, notably PDA's running Pocket PC, Pocket Linux or Palm 5 operating systems, and cell phones running Microsoft Smartphone or Symbian operating system (for example Nokia Series 60 phones such as N-Gage or Sony Ericsson UIQ phones such as P800), are becoming a new platform for 3D games.

These devices typically run on some variant of ARM processor, with clock speed of 100-400 MHz. The amount of memory usually varies between 4 MB and 32 MB, but is restricted by the fact that the same memory is used also for storage, not just as run-time memory. The devices usually have colour displays, with resolutions ranging from 176×208 to 480×320. The display hardware is typically very straightforward, without any display acceleration components.

Within these limitations it is still possible to create 3D games that compete with the quality of console and PC games of the 1990's. Development in tight mobile environment requires however different approach from the methods used in desktop or console systems.

Mobile vs. console and desktop games

The form factors of mobile devices along with the varying usage situations give some new challenges to the game design. On devices primarily meant to be used as phones, the controls for playing games may be far from optimal. Also the size of the display and its poor resolution can be limitations. From the technical point of view however, the lack of memory and storage space is the biggest problem.

In the beginning of 1990's, CD-ROM entered the gaming market and changed a lot in console and desktop gaming. Prior to CD-ROM era, games were distributed on diskettes or game cartridges, with sizes ranging from few hundred kilobytes to few megabytes. Although some games were distributed as a pile of diskettes, arrival of CD-ROM media allowed much more massive games when the size of the content was no more a limitation: one CD could store a massive amount of 700 megabytes of data. One of the most successful systems having a CD-ROM was Sony Playstation, released in 1994. On desktop PC's, CD-ROM quickly became the standard media for distributing games in mid-1990's as well.

In respect to 3D rendering quality, today's mobile devices can rival with Playstation. In the amount of content however, the situation is even worse than it was in the times before CD-ROM. Most mobile devices for instance don't have any changeable media whatsoever. In the best case there are some changeable cartridges of a few megabytes, but for anything as massive as CD-ROM we still have to wait a few years. Typically the games are crammed to the device memory along with operating system, all the other applications and data, and there should be some memory left for running the applications as well. For devices with as little as 4 megabytes of memory this is clearly a problem. For comparison: Playstation has total of 3.5 MB of runtime memory (2 MB main memory, 1 MB video memory, 0.5 MB audio memory) devoted for one single game at a time, and the whole CD-ROM for storage. [PSX] Clearly mobile games require some clever strategies to overcome the situation.

Experiences from mobile 3D games

The experience gathered from 3D mobile games this far suggests that biggest portion of the content in a 3D game is still 2D content. This consists of textures for the 3D scenes and bitmap graphics for the user interfaces.

The following figures are from "Stunt Run", a game that was made at Fathammer and bundled with Sony Ericsson P800 phone. They demonstrate the content sizes in a 3D mobile game:

Content sizes of "Stunt Run", a 3D car racing game:

	uncompressed	z-lib compressed	compression ratio
Total size of content	1 140 137 bytes	577 999 bytes	50.70%
3D content	340 318 bytes	161 116 bytes	47.34%
2D content	573 820 bytes	291 653 bytes	50.83%
Audio content	188 358 bytes	115 693 bytes	61.42%
Misc content	38 916 bytes	10 819 bytes	27.80%

Table 1

In the final game, all content was packed to a single zlib-compressed package file. For testing purposes, each content category was packed to similar file. Table 1 shows the compression results.

Solution

The natural solution for the problem with storage space was to seek methods for defining 2D graphics more efficiently.

Bitmap formats have several benefits for this kind of use, as they are the standard way of doing things. Artist are familiar with content creation tools and techniques, and with enough resolution bitmaps allow unlimited artistic expression, thus being suitable for use both as textures and user interface components. Bitmap handling is fast and straightforward also on the programming side, as bitmaps are thoroughly supported by the graphics engine in any case. Bitmaps can be used for UI, textures, fonts etc. with no extra code bloat.

However, to fit the tight storage requirements, bitmap graphics should be scaled to low resolution and compressed heavily. As visual quality is one of the main concerns of a gaming application, this is unacceptable.

The mobile space is also populated by various devices with different screen sizes and resolutions. Bitmap graphics have to be prepared in suitable resolution for each individual device.

Requirements

A better solution than bitmap graphics was clearly needed. The requirements for the solution can be summarized as following:

- Small storage space.
- Good visual quality.
- Lightweight technology: fast image construction and rendering, small amount of code and small memory footprint.
- No limitations for artistic expression: images should be suitable for textures and user interface components.
- Support for fonts.
- Good tools available for content creation.

Evaluation of alternative technologies

After evaluating various solutions, SVG was chosen to be the technology to use. The competing technologies were other vector formats, such as Macromedia Flash, and various algorithmic texture generation methods. However, texture generation methods usually provide rather monotonic textures with no possibility to add features defined by the artist: for instance natural phenomena such as stone or wood grain is easy to model, but for instance a stone wall with a door and windows is not. Vector formats on the other hand don't have the functionality required for texture creation.

When used as textures, vector images still have to be rendered into bitmaps, but for UI components it is possible to render the vector image directly to the screen, thus saving the memory required for a bitmap.

SVG and the requirements

Although SVG doesn't completely fit all the requirements, it gets so close that it is possible to fill the missing gaps.

As a text based format, SVG consumes more storage space than necessary. Although compression can solve this partially, the results are not as good as they could be. SVG is not really a lightweight technology either: the SVG specification is huge [SVG] , and an XML parser and DOM add to the size of the implementation. Although lighter profiles such as SVG Tiny [SVGMobile] do exist, they don't provide enough functionality when considering the visual quality. Rendering vector graphics directly to the display surface takes typically only a little amount of memory, but some operations, such as filtering or complex nesting, require more. Also DOM implementations are typically hungry for memory. The tools for SVG content creation also have some space for improvement, especially on the filter creation side.

However, there are also extra benefits of choosing SVG: content is scalable, which allows using the same content on a variety of devices with different screen resolutions. SVG can be used also for animation, which provides great size optimisations when compared to video formats.

Components of the solution

The X-Forge™ game engine is a C++ based multi-platform mobile 3D game engine. The engine isolates the developers from the peculiarities of the operating system of the device, and allows them to concentrate on the most important: the game itself. X-Forge™ is not just 3D rendering engine, but also provides other components such as audio and networking, and high-level functionality such as collision detection and physics [X-Forge].

Considering an external toolkit

On some platforms, the engine is distributed as a part of the operating system, but in most cases the engine is linked to the game application. The binary size of the engine, depending on which features the developer has chosen to include, ranges typically from 200 to 500 kilobytes. The smallest readily available mobile SVG implementations are in the class of 200-700 kilobytes. Adding one of those would have doubled the binary size. They are not necessarily multiplatform products, and usually lack some features – such as filters – that are vital for good visual quality.

Integration of an external SVG library to the engine would also have meant duplicate code: the engine already has a solid application framework, a math library, code for bitmap image manipulation, an efficient antialiased polygon rendering algorithm etc. In order to avoid code bloat, external SVG library would have required heavy modifications to be able to use as much code from the X-Forge™ framework as possible.

Therefore, a decision to develop an internal SVG rendering engine was made.

XML parser and DOM handling

As an XML based format SVG also requires an XML parser and DOM tree. XML has several benefits as a file format, but for defining content of a mass-market product, a 3D game, these are not exactly required. Game developers typically place more effort in order to protect the game content from tampering than allow open access to it. XML files tend to grow rather large as well – an effect we have tried to avoid in the first place.

To reduce the size of the implementation and run time memory requirements, and also to boost the file parsing performance, we decided to move the XML parsing and DOM handling offline, and use a binary format for the graphics in the engine. This has been done for XML before, for instance WAP uses WBXML (WAP Binary XML) [WBXML], and Plazmic uses a binary SVG format in their media engine [Plazmic]. Another benefit from using a custom binary format is increased protection of intellectual property, as unauthorized use of the game content is not as easy as with plain SVG.

However, to also allow direct SVG use in the future applications, the binary format was designed in a way that it follows SVG specification closely. It can be described as a compiled format of a SVG DOM that can be executed directly by a virtual machine. Although the compiler is currently an offline tool written in Java, integrating a C++ version along with an XML parser to X-Forge™ game engine is all that is required for direct support of SVG in the future.

Feature subset

SVG – when using the full profile – provides all the functionality for the artists to allow unlimited artistic expression. The biggest improvement in this respect when comparing SVG to other vector formats is the filter support. In addition to the usual drop shadow effects, filters can be used for creating a wide variety of effects for texture images. As figures 2-6 demonstrate, SVG images can have the full visual quality of bitmap images.

SVG profiles for mobile use – SVG Tiny and SVG Basic [SVGMobile] – are not designed according to the requirements for visual quality. Features such as filtering are considered secondary, and more attention is placed on multimedia and interactivity features. Therefore, the subset of SVG supported by X-Forge™ 3D Game Engine is not based on current mobile profiles, although features required for SVG Tiny are included in the subset definition.

Because SVG is used as a replacement for bitmap graphics, features like scripting, interactivity, text editing capabilities etc. can be provided through the game application in a more efficient manner. Inclusion of these features was not seen as a necessity. X-Forge™ is focused on 3D games, and thus doesn't require the functionality required by web or cartography applications.

Content creation

A variety of tools already exist for SVG creation. However, all the tools we have evaluated have neglected the power of SVG filters, and support for filters is mediocre at best. Existing tools either require writing filter descriptions by hand, or allow only slight modifications of existing filter descriptions. Artists need tools where filters can be edited visually, without thorough understanding of SVG source code.

To overcome this limitation, the toolset for creating SVG content to be used in mobile 3D games required an addition, a SVG filter editor. As X-Forge™ is targeted for mobile platforms, not desktop environment, it was not possible to use X-Forge™ as the platform for tool development. Instead, the tool was written in Java using Batik SVG toolkit.

SVG filter editor is a stand-alone application that can be used along with a vector graphics package that has SVG support. At least Adobe Illustrator is suitable for this, as importing SVG filters from external files is easy and artists are usually familiar with the software.

The emphasis in the tool development was placed on two main principles: the tool should allow unlimited access to SVG filter functionality, while being a tool for artists, not programmers. As the typical user of the software is a professional game artist, the idea of the user interface was not to hide functionality from incompetent users, but to provide a visual representation of the SVG filter that is can be understood and manipulated without writing a single line of SVG code.

As one SVG document can contain several filter definitions, the tool allows previews of these either one by one or all combined in the finished image. There can be several previews of different filters visible at the same time, but only one filter is the current active filter, the one that is being edited. A filter is represented in a layer graph, where the filter effects are stacked on top of each other and connected with pipes. The pipes can be dragged around in order to rearrange the connections between the filter effects. Also the order of the filter effects can be changed by dragging them.

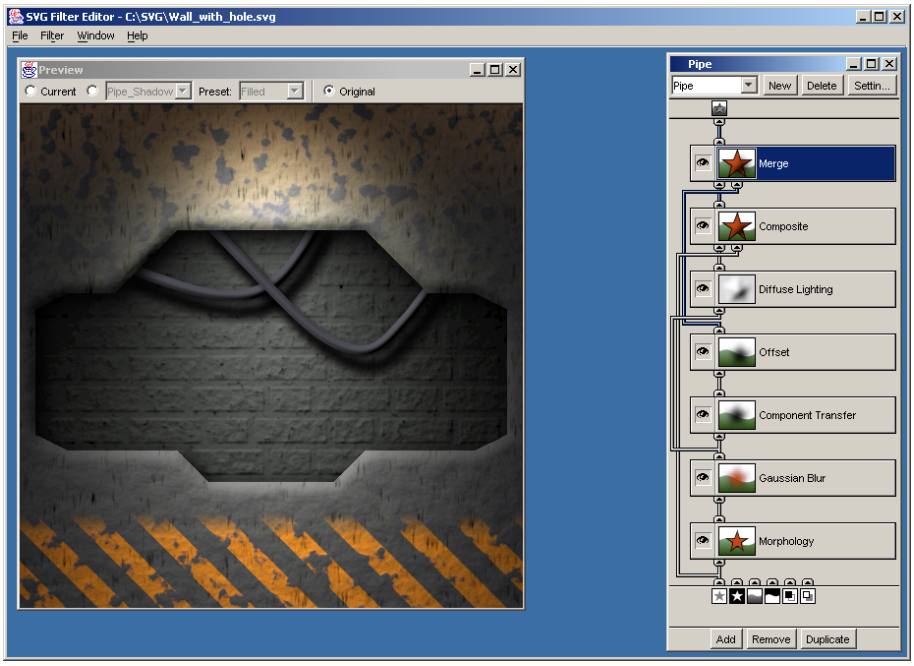


Figure 1: Screenshot of SVG Filter Editor

Summary

In conclusion, the full SVG solution of X-Forge™ consists of following components:

Components of SVG solution of X-Forge™

<table>
<tr><td rowspan="4">Offline tool chain</td><td colspan="2">Vector editing package
• Any vector editing package with SVG support</td></tr>
<tr><td colspan="2">SVG Filter Editor
• Tool for editing SVG filters</td></tr>
<tr><td>SVG compiler
• Two versions: a command line compiler and an exporter in SVG Filter Editor
• The implementation can be modified so that this is hosted on the mobile device</td><td>XML Parser
• Only a minimal parser could be hosted on the mobile device.</td></tr>
<tr><td>DOM Translator
• Translates the SVG DOM to binarized format.</td></tr>
<tr><td rowspan="3">Mobile implementation</td><td colspan="2">Vector Graphics Processor
• SVG byte code virtual machine</td></tr>
<tr><td colspan="2">Primitive Translator
• Translation of SVG drawing commands to vector primitives</td></tr>
<tr><td>Primitive Draw
• Vector drawing</td><td>Surface Toolkit
• Bitmap handling</td></tr>
</table>

Table 2

Conclusions

The SVG solution of X-Forge™ is approaching completion, and some test data is already available. Following images demonstrate the results that can be achieved by using SVG in mobile games.

Images

Following images were created using SVG Filter Editor and a vector-editing package. Although SVG support of X-Forge™ game engine is not yet in production use, most of the images are based on real production material.

Figure 2: Marble

Marble is a procedural texture created by using only SVG filters. It contains no drawn geometry, except the rectangle to which the filter is applied. The texture is looping in both x and y direction. This is achieved by using the stitchTiles-attribute of feTurbulence filter primitive, and feTile filter primitive.

Figure 3: Wall

A wall from “Unfinished Business”, a 3D shooter game example. The texture is looping in x direction.

Figure 4: Wall with a hole

A more complex wall texture from “Unfinished Business”. The texture is looping in x direction, and can be aligned seamlessly to the wall texture of image 3.

Figure 5: Ceiling

A ceiling texture from “Unfinished Business”. The texture is looping in x and y directions.

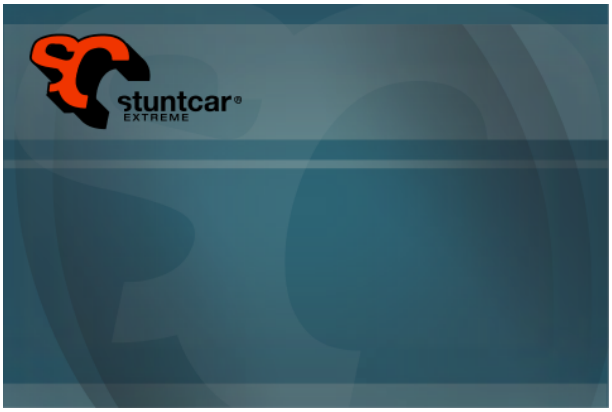


Figure 6: Menu background

Background of a menu from “Stuntcar Extreme” car racing game. The image is mostly based on vector art and uses only one filter for shading.

Visual quality

As the images above demonstrate, SVG can be used for creating images that are indistinguishable from bitmap images.

Figure 2 , “Marble” demonstrates results that are similar to procedural texture generation software. Figure 3 , Figure 4 and Figure 5 however demonstrate the real power of SVG in texture creation: combination of vector art and filter effects can produce results that are not possible by using only vector images or procedural texture generation.

Figure 6 demonstrates that SVG is suitable also for menu graphics and other 2D content in addition to textures.

File sizes

The savings in storage space are even more impressive. An example SVG file in 256×256 pixel size takes 10 kilobytes of disk space. The same image, converted to JPEG format takes 30 kilobytes when saved with quality where compression artefacts are not disturbingly visible. When the SVG file is converted to binary format used in X-Forge™, the file size drops even lower, to modest 3 kilobytes. Sizes this small can be achieved in JPEG format only by compressing the image beyond recognition with the highest compression settings. Even after this, the file can be compressed with zlib, typically bringing results of 50% compression.

Table 3 demonstrates the file sizes of example images:

Image:	Size as SVG	Size as binary	Size as compressed binary	Size as PNG	Size as JPEG
Marble	2 802 bytes	935 bytes	487 bytes	63 887 bytes	14 701 bytes
Wall	4 993 bytes	1 930 bytes	873 bytes	98 807 bytes	24 679 bytes
Wall with a hole	10 027 bytes	3 963 bytes	1 623 bytes	97 651 bytes	31 415 bytes
Ceiling	8 624 bytes	3 355 bytes	950 bytes	80 762 bytes	18 894 bytes
Menu background	19 480 bytes	11 310 bytes	6 507 bytes	63 601 bytes	23 304 bytes

Table 3

JPEG compression of the example images was made with lowest possible quality where visual artefacts were not disturbing. In the application used for compression, quality was selected in the range of 0 to 10; typically the values were between 6 and 8.

When using SVG images, another significant issue with file sizes has to be kept in mind: SVG files can reference other image files. For instance in image 6, Menu background, the single most complex vector component is the Stuntcar Extreme logo. This can be saved to a separate file, and be referenced from all the images that contain the logo, thus reducing the total amount of data considerably. In practice this may cause some problems though, as content creation tools don’t necessarily support this approach very well.

SVG Filter Editor, although still having room for improvement, has been proven to be a solid tool for creating SVG filters without going into the complexities of writing SVG code. As with all tools in computer graphics, efficient usage of the tool still requires good understanding of the underlying principles, SVG filter model and document structure.

Problem areas

The speed of image construction is a concern. Heavy use of SVG filters easily creates images that take long to render. Therefore, artists need to be aware of methods for reducing the execution times: some filters – such as lighting effects – are heavier than others, some effects can be performed in a lower resolution by adjusting the filter kernel size etc. Complex filter effects can also require surprising amounts of runtime

memory for intermediate bitmaps. Currently the tool chain doesn't provide means for artist to measure these properties, although rendering times in the SVG Filter Editor provide some hints. To remedy the situation, a tool that can load binary SVG files and measure the rendering speed and memory consumption using the actual X-Forge™ game engine is required.

Future developments

The problems with storage space of mobile gaming devices are likely to become less severe as the storage capacities grow. However, as the processing power, display resolutions and device memory sizes grow as well, the games require even more textures and other 2D content. Storage solutions that can handle that data in plain bitmap formats are not likely to appear on the smallest devices: optical disks or hard drives require mechanical parts and consume space and battery power, and thus will appear only on the higher-end devices.

Games played on mobile phones – or devices that are combinations of a game console and a phone – can also have connectivity features. These can be used for instance for downloading game content, but as the connection speeds are quite modest and the users pay for the amount of transferred data, it is good service to provide the data in as compact form as possible, even if the storage is not a problem. If the transfer times can be reduced to minimum with small content sizes, a solution where the data is stored only on a server and fed to the clients at request is possible to implement as well.

In wider scope, the textures created in SVG prove that SVG can be used to achieve effects that have been associated only with bitmap graphics. Since SVG is scalable and has small file size, there is a wealth of other applications in addition to mobile 3D gaming that benefit from switching from bitmaps to SVG.

Bibliography

[PSX]
Sony PlayStation/PSOne technical specifications, Available at http://www.playstation.com.

[SVG]
Scalable Vector Graphics (SVG) 1.0 Specification, J. Ferraiolo, editor, W3C Recommendation, 4 September 2001. Available at http://www.w3.org/TR/SVG/.

[SVGMobile]
Mobile SVG Profiles: SVG Tiny and SVG Basic, T. Capin, editor, W3C Recommendation, 14 January 2003. Available at http://www.w3.org/TR/SVGMobile/.

[X-Forge]
X-Forge product brochure, Available at http://www.fathammer.com.

[WBXML]
WAP Binary XML Content Format, B. Martin, B. Jano, editors, W3C Note, 24 June 1999. Available at http://www.w3.org/TR/wbxml/.

[Plazmic]
The Suitability of SVG for Deploying Wireless Applications, J. Hayman, Conference proceedings of SVG Open 2002. Available at http://www.svgopen.org/papers/2002/hayman__suitability_of_svg_for_wireless_applications/.