# Vectorization of Glyphs and Their Representation in SVG for XML-based Processing

*Stefan Pletschacher[1]; Marcel Eckert[2]; Arved C. Hübler[1]*

[1]Institute for Print and Media Technology, Chemnitz University of Technology
Reichenhainer Straße 70, D-09126 Chemnitz, Germany
e-mail: stefan.pletschacher@mb.tu-chemnitz.de; pmhuebler@mb.tu-chemnitz.de
[2]Department of Computer Science, Chemnitz University of Technology
D-09107 Chemnitz, Germany
e-mail: marcel.eckert@informatik.tu-chemnitz.de

## Abstract

This paper shows an approach for converting bitmap images of text glyphs into a vector format which is suitable for being embedded in XML representations of digitized documents. The focus is on a contour based vectorization method as the output can be easily transformed into SVG glyph descriptions. A concrete implementation is described and the results are discussed with special regard to the visual quality. The work is related to the development of a system for processing documents which are not suitable for current OCR methods. This is especially important in the field of retrospective digitization of historical works.

**Keywords:** SVG; XML; Vectorization; Document Image Analysis; Fonts

## 1 Introduction

Digitization of printed material has become more and more important due to several reasons. For business documents like invoices or contracts it is necessary to integrate them into electronically controlled workflows and to have the contained information accessible within appropriate information systems. Since most of these documents are a product of computer-aided work itself, they have mainly clear structures and easy to recognize typefaces. Hence, there are numerous tools and software applications supporting the preparation of digitized document images for further processing. OCR (Optical Character Recognition) is a method for getting the coded information back from its graphical representation within a digital facsimile of a printed document. A limitation of these techniques is the restriction of processible character sets and fonts by the used OCR engine. Characters are either matched against known prototypes or compared with predefined characteristics using feature-based methods. Critical is the underlying knowledge of the recognition engine. If there is no information about a certain language, its alphabet and the appearance of the glyphs, an OCR engine will hardly deliver acceptable results.

Therefore, other digitization projects like preservation of historical books and writings may need different approaches. Nevertheless, also these resources should be easy to handle and further processible on digital platforms after the preparation step. The initial bitmap images as results of the scan process are not very handy for network transfer or to serve different output formats and devices. Demands on alternative document storage formats and methods are among others:

- handling of arbitrary and potentially unknown characters
- avoidance of recognition or mapping errors during the transformation process
- reproduction as true to the original as possible
- memory saving representations
- encoding for standardized further processing, exchange and transfer via networks
- means for modifying the content
- support of various output formats and devices

To handle documents independent of the contained character set and typefaces there are several possibilities. We follow the approach of extracting templates for words or letters directly from the original document [1] [2] and do not rely on predefined alphabets. To avoid redundancy in this set of items obtained from the current document, prototypes for groups of sufficiently similar glyphs have to be ascertained by means of clustering methods. Whereas others make use of bitmap representations for the items to be utilized later on e.g. [3], we go a step further by extracting a document specific alphabet in combination with creating a corresponding font. This is achieved by vectorization of the candidate glyphs. The target format for such a document immanent alphabet and its typeface is SVG (Scalable Vector Graphics) [4]. It enables standardized XML-based processing

methods for the converted documents. XML (Extensible Markup Language) [5] offers best conditions for describing the content of documents and its structure. Furthermore, XML technologies provide efficient methods for transforming the content and for producing various output formats. SVG is an XML-based standard especially for describing graphical objects. Using these technologies it is possible to seamlessly integrate graphical descriptions of glyphs into any kind of XML document.
In order to obtain the outlined benefits the original document is to be converted based on the extracted alphabet. Therefore the corresponding XML instance must contain a definition part describing the used alphabet and its appearance as well as a content part referring to the defined characters. This is outlined in (Figure 1).

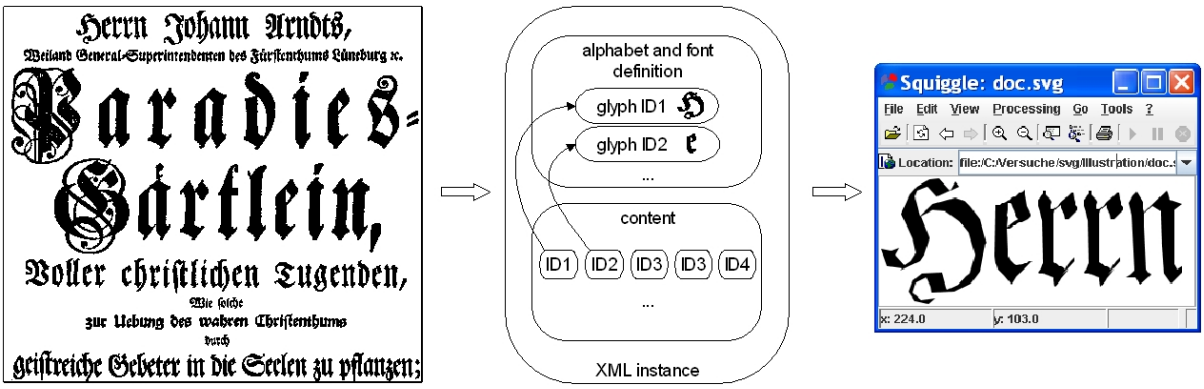


**Figure 1: Encoding of documents using a document specific alphabet and its glyphs (image on the right hand side rendered by Squiggle [6])**

The delineated approach focuses on the handling of graphical representations rather than the recognition of textual content. In order to access the meaning of the encoded text, further recognition methods have to be applied. But particularly for historical documents with specific typefaces and spellings this is a nontrivial problem. A benefit from choosing the non-recognition-based approach is the possibility of providing text-like processing features for documents which could otherwise only be handled as images. Therefore it seems possible to use this approach within systems for retrospective digitization. A benefit could be to gain better access to handed down resources and works of cultural heritage by means of digital information systems. Once an adequate format has been created, material can be output to various platforms and devices like mobile gadgets.
Of special importance for the visual quality of subsequent output formats is the vectorization task i.e. the raster to vector conversion of glyphs. Hence, especially the techniques for converting bitmap images into vector descriptions which have been implemented in our framework as well as involved quality issues will be discussed in the following.

## 2 Raster to vector conversion of glyphs

For the representation of images there are mainly two categories of formats: bitmap graphics and vector graphics. Both formats have specific properties which make them useful for different applications. An important difference is the behaviour of the particular formats when it comes to geometrical transformations like scaling. As vector-based formats describe the image by means of mathematical expressions, shapes and their mapping to certain output media will always be computed accordingly. Thus artefacts caused by scaling can be avoided whereas bitmap formats are prone to this kind of distortions.
Especially for glyphs it is desirable to obtain the same visual quality for different sizes and resolutions. When using bitmap fonts, the shape must be defined for each applicable size individually to guarantee optimal results. Since it is now aimed at the reverse process, i.e. automatically deriving a font from existing bitmap glyph images, there is no access to different magnification levels of the original shape. Moreover, bitmap images are consuming much more resources for storage and transmission. Vector-based formats therefore seem the better choice for this kind of use cases.
Starting point after the digitization is a bitmap facsimile of the original document delivered by the scanner as the raw format. For obtaining vector descriptions of the glyphs contained in the digitized document, firstly the image must be segmented to retrieve the single glyphs. Then the raster-to-vector conversion can be applied to these elementary images. For simplification, other processing steps involved in the over-all system are omitted at this point.
The main idea is to find characterizing vertexes that are suitable for describing the original shape by means of geometrical primitives like polygons or Bezier curves. The reduction of the number of characterizing vertexes

plays an important role for the image compression and is often done in a following post-processing step. In order to find points which can be utilized to describe shapes within an image, there are among others two main approaches. The first is the skeleton-based and the second is the contour-based approach (Figure 2).

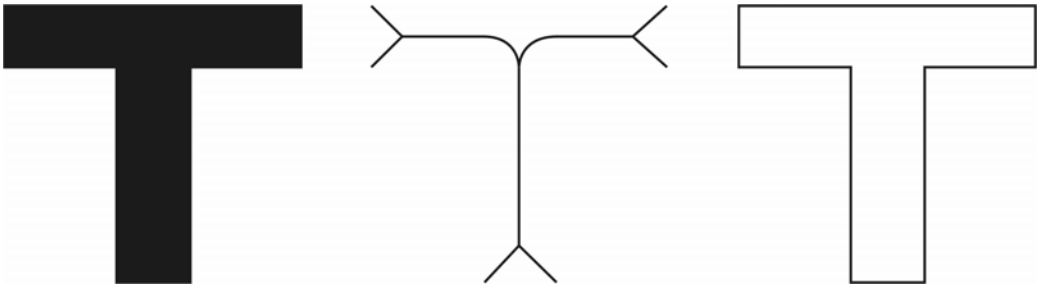

**Figure 2: Skeleton vs. contour based vectorization**

Methods based on the skeleton try to reduce an object to its basic form, whereas contour-based methods detect edges that form the outline of an object. Once the necessary vertexes are found, the object can be described using these support points together with the geometrical primitives connecting the points. Depending on the requirements this may involve only linear primitives or can be extended to non-linear shapes e.g. for describing arcs. Since a contour-based description can be used directly to reconstruct the original shape it is very practical for expressing glyphs. Furthermore, the target format SVG allows the definition of fonts and its glyphs by means of path elements, which can be produced straight forward out of extracted contours.
A more detailed overview about different vectorization techniques can be found in [7]. In the following we will describe an adopted contour-based method using polygonal approximation which has been implemented and tested for the specific task of vectorizing glyphs.

# 3 Implemented vectorization methods

The basis is a single black-and-white image for each glyph. The actual vectorization process then consists mainly of three tasks. The first is finding all disjunctive components of the glyph image by a region growing algorithm. This has to be done to handle characters which comprise more than one part e.g. “i” or “%”. Also characters with outer and inner borders such as “O” result in two or more parts. In this case there is an outer component of black pixels and an enclosed component of white pixels. The second step is extracting the contour for each component. That means finding the points being situated on the border of the particular pixels sets. Finally the contours have to be approximated to get a simplified shape description. Otherwise the result would be a large number of vectors describing the polygons whereas their length would be 1 for horizontally or vertically connected and sqrt(2) for diagonally connected pixels, assuming a scale of one pixel for the coordinate system.

## 3.1 Finding connected components

The initial step is ascertaining closed components within the glyph bitmap. This can be done using widely known region growing algorithms. Before such an algorithm can be applied, it must be ensured that there is a single outermost white pixel component. Therefore, the original image is expanded by one white pixel in each direction.
For deciding whether two pixels belong to the same component, it is necessary to define the neighbourhood of pixels. In order to avoid overlaying components, which could occur mutually on the steps of one pixel wide diagonal lines, this must be done for white and black components separately. If a pixel P is black, all eight surrounding pixels are defined to be neighbours of P (8-neighbourhood). In case P is white, only the four pixels straight to the left, right, up and down are considered as neighbours of P (4-neighbourhood).
Region growing works in a “breadth-first” manner. This means, the procedure starts by constructing an initial region which consists only of one single pixel. This pixel can be freely chosen. Now the region grows by examining the pixels in the neighbourhood of the whole region. If a neighbour has the same colour as the pixels of the region, this neighbouring pixel is added to the region. Thus the region grows and new neighbours have to be examined. (Figure 3) shows an example for the region growing method. After a few growing steps, the region can not grow any more, because all neighbour pixels have a different colour than the region. This region now forms one component of the picture. If there are remaining pixels which are not yet assigned to a component, the

next pixel can be chosen from this set as the new starting point for a second region. In this manner all black as well as white components of the picture are found.

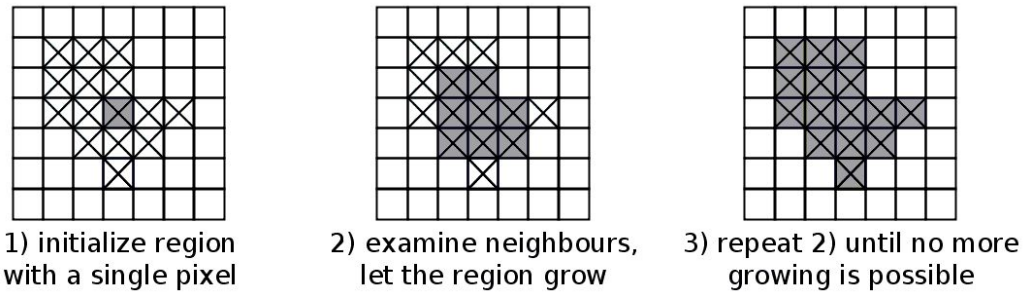


**Figure 3: Finding components by region growing (Square with cross represents a black, without cross a white pixel. The region is grey.)**

## 3.2 Contour detection

After finding the components of the image in form of pixel sets, now their corresponding contours have to be discovered. The outermost white component, guaranteed by expanding the initial image, forms some sort of margin and surrounds the real glyph components. Therefore this one can be omitted and only inner components have to be processed.

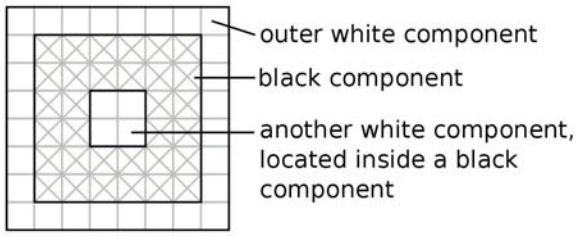


**Figure 4: Nested components**

Since black glyph components alternate with white hollows inside for forming punched characters, only the outer contours of both component types are needed for further steps. This is illustrated in (Figure 4). The contour is defined as the set of pixels belonging to the component and having neighbours regarding the 8-neighbourhood which do not belong to this component. At this point there is no difference between black and white pixels with regard to the neighbourhood definition.
An often used algorithm for finding the outer contour of a component is the so-called contour following [8]. Firstly, the algorithm has to detect one arbitrary pixel of the examined component that belongs to the contour of this component. This pixel is then taken as starting point. Now the algorithm follows clockwise (or counter-clockwise) the neighbouring pixels which also meet the contour criterion until the starting point is reached again. The process is outlined in (Figure 5). However, the procedure still needs to be refined since difficult situations might occur in which the starting point is reached again, but the whole contour has not yet been found. An illustrative example is the character “8” - if the algorithm starts at the crossing point in the middle it could stop after the lower cycle whereas the upper one is still missing. Therefore the stop condition must be extended to reaching the sequence of the starting point followed by the second pixel of the contour. In the example of the character “8” this would not yet be satisfied after finishing the first arc and the contour following principle would walk further to the upper part.

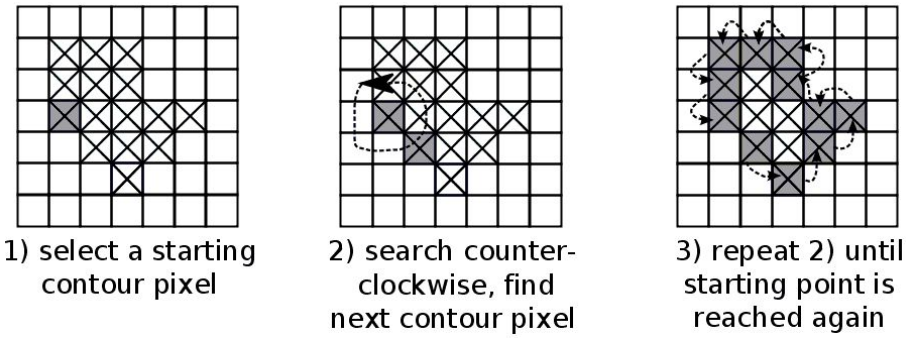


**Figure 5: An example for contour following (Square with cross represents a black, without cross a white pixel. The contour is grey, dashed arrows represent search/find steps.)**

A main advantage of the contour following algorithm is that pixels of the contour are examined in an ordered sequence. Thus, this sequence can already be used for drawing a polygonal shape.

## 3.3 Polygonal approximation

The contour ascertained in the previous step still suffers from a potentially high degree of redundancy if the transitions between the single points are treated as the vector description of the final shape. Many vectors between the true polygonal points can be possibly united without losing any detail information. This is especially true if a sequence of short vectors is pointing in the same direction. In this case the sequence should be replaced by just one longer vector. Furthermore, minor deviations from an otherwise linear shape should be smoothed since this often originates from image distortions due to the digitization process. Therefore, the algorithm described in this section reduces the number of points used for the polygon description, without loosing too much detail information by setting a maximum deviation range. The method is based on the algorithm named polygonal approximation based on relaxation presented in [9].
The polygonal approximation algorithm works with the contours obtained before and is divided into two steps. At first, the algorithm finds sets of pixels called clusters. This is the vector compression step. Thereby, a set of following points from the input polygon is summarized in one cluster, if and only if the distance between each point of the set and the straight line formed by all pixels of this set is less than a given deviation value $\varepsilon>0$. This can be seen in the left part of (Figure 6). The straight line can be calculated e.g. with the minimum square error method. The algorithmic implementation for constructing these clusters starts with two pixels next to each other. Afterwards, consecutive points are added step by step to the cluster while the above mentioned condition based on $\varepsilon$ remains true. If the condition is violated, a new cluster is started. The formation of clusters is done for the whole polygon. After this process each polygonal point is member of exactly one cluster.
The calculated clusters form the vectors of the approximated polygon. In fact, there is one problem left when using the vector compression algorithm: The points of the approximated polygon might be placed incorrectly if pixels out of the original contour are included in the wrong cluster. This is especially of concern at corner points. For example, if the original shape contains a sharp corner, then a few points round the corner are added to the current cluster, because the aborting criterion for the cluster formation (remember $\varepsilon$) is still fulfilled even if the new cluster should start exactly at the edge (Figure 6).
This misplacement of corner points in the approximated polygon is the reason for the second step of the polygonal approximation algorithm, called error relaxation. In this step, each pair of adjacent clusters is examined in a so called primitive relaxation step. The end points of the first cluster of a pair and the starting points of the corresponding second cluster are taken into account. Thereby it is calculated if shifting of some of theses points from one to the other cluster will result in a better approximation of the polygon. If a better formation can be found then the corresponding points will be reallocated. This relaxation step can be repeated several times until no better approximation is found.
The result is a simplified vector description of the actual contour consisting of vertexes for the corresponding polygon. Finally, this polygon representation should be expressed by means of SVG.

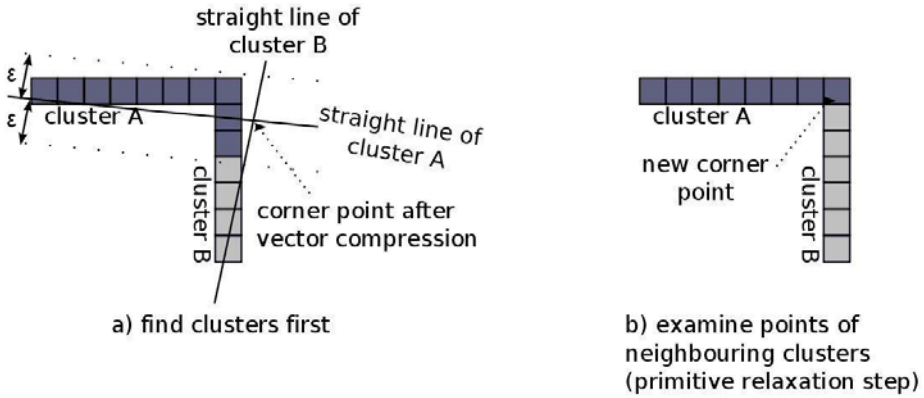


**Figure 6: Polygonal approximation**

# 4 SVG and glyphs

The results of the preceding activities may be easily transformed into the SVG format. The basis for describing arbitrary forms is the <svg:path> element. The implementation of paths can be compared to an imaginary pen which is moved to a starting position and then draws a line to the next reference point. Therefore, the path element must always be used in combination with the *d* (description) attribute defining the vertexes of the line segments. To move the pen to an absolute position, the *M* command together with the *x,y*-coordinates is used within the *d* attribute. From there a line can be drawn using the command *l* with the associated target coordinates. These two commands are already sufficient for drawing the polygons derived during the vectorization process. A special case is the composition of glyphs out of several components. To implement this, the pen can be moved to the second starting position without drawing after finishing the first contour. Then the next contour can be placed and so on. To cut inner parts out of an object, as it is necessary for characters such as "O" where the inner white component must be punched out, the fill-rule attribute of the glyph element must be set to *evenodd*. There is still one problem left since the coordinate system for glyphs within SVG is mirror-inverted regarding the *y*-axis compared to the one used before. However, SVG offers powerful features for transforming graphics. This is done by applying a *transform* attribute to the path element whereas the appropriate transformation matrix takes care of flipping the image vertically and translating it back to the origin.
The complete document specific alphabet and the corresponding glyph graphics are to be defined within a <svg:font> element. A font consists of one <svg:font-face> child and any number of <svg:glyph> elements describing the particular characters. Herein the concrete shapes are given by the already mentioned path expressions. To address the glyph entities later on, their *unicode* attribute must be set to an individual code point for each. These code points should originate from a private use area as defined in the Unicode [10] standard to distinguish between real text and representations using the document specific alphabet. Another important value is given by the attribute *horiz-adv-x*, namely horizontal advance. It determines the distance to the next glyph in the text flow. At this point additional information from the segmentation process is needed to ascertain the appropriate value. Special issues like kerning are not regarded at the moment.
The final XML instance is produced based on structural and layout information collected during the preceding document image analysis steps as described in [2] and must be combined with the new SVG font definition part. For reproducing the original document, the content is then encoded using references to the extracted alphabet within common <svg:text> elements. A simplified example can be found in (Figure 7).

```
<?xml version="1.0" standalone="yes"?>
<document xmlns="http://pmtuc.tu-chemnitz.de/2005/diadoc" xmlns:svg="http://www.w3.org/2000/svg">
        <alphabet>
                <svg:defs>
                        <svg:font id="font1" horiz-adv-x="300">
                                <svg:font-face font-family="DIA" units-per-em="128"/>
                                <svg:glyph unicode="&#x100000;" horiz-adv-x="109">
                                        <svg:path
                                                d="M 0,70 l 6,17 15,2 -2,2 19,-9 4,-6 5,-16 0,-1 -8,-
                                                style="fill:#000000; stroke:none" fill-rule="evenodd"
                                                transform="matrix(1 0 0 -1 0 100)"/>
                                </svg:glyph>
                                <svg:glyph unicode="&#x100001;" horiz-adv-x="42">
                                        <svg:path
                                                d="M 2,17 l -1,48 1,7 1,2 1,4 6,6 5,6 2,3 2,-3 3,-5 1
                                                M 14,20 l -1,5 1,2 -1,10 3,-3 4,-3 4,-2 -2,-2 -1,-2 0
                                                style="fill:#000000; stroke:none" fill-rule="evenodd"
                                                transform="matrix(1 0 0 -1 0 100)"/>
                                </svg:glyph>
                                ...
                        </svg:font>
                </svg:defs>
        </alphabet>
        <page>
                <textblock>
                        <line>
                                <word>
                                        <svg:text x="100" y="200" font-family="DIA" font-size="20">
                                                &#x100000;&#x100001;&#x100002;&#x100002;&#x100003;
                                        </svg:text>
                                </word>
                                <word>...</word>
                        </line>
                        <line>...</line>
                </textblock>
                <textblock>...</textblock>
        </page>
        <page>...</page>
</document>
```

**Figure 7: A simplified XML instance describing the original document structure and using the extracted alphabet**

Once the document is encoded using the presented method, additional transformations for producing various output formats and layouts may be applied. E.g. the generated font can be used in italic style if a *skewX* transformation is performed on glyph level. Thicker stroke widths can lead to bold text and also the text flow can be rearranged because the document structure consisting of blocks, lines and words is preserved in the XML instance. Thus it is possible either to obtain an almost exact reproduction of the original or to manipulate the appearance according to specific needs.

# 5 Results and discussion

The vectorization results were evaluated both with formal methods and based on the subjective impression. Ground truth methods deliver error values for an objective comparison of images by means of matching algorithms. Exact matching serves as a rough estimate for the similarity of a reconstructed image and the original. Thereby, the absolute values for the differences of grey levels for each pair of corresponding pixels is calculated and summed up. However, the quality perceived by a human being depends not only on the absolute number of errors but also on the position and spatial accumulation of distortions (see Figure 8). We tried to approach this problem by using a weighted matching algorithm which takes into account the distance of misplaced pixels to the closest true component.

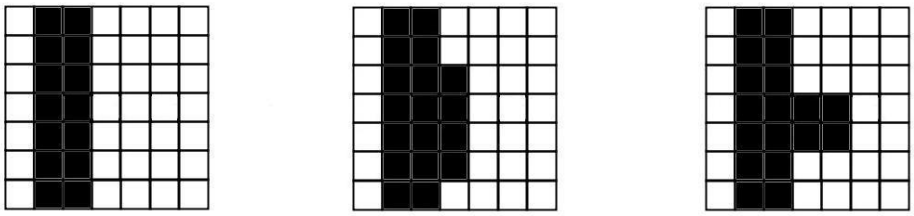

**Figure 8: The images in the middle and on the right hand side contain the same number of errors compared to the original on the left. Yet the right one seems to be more corrupted.**

The accuracy is then calculated based on the error rate, ranging form 0 to 1 whereas a value of 1 means identical images. The visual impression can not be expressed entirely by these characteristic values, however, they serve as objective evidence. The decisive factor is still the assessment of test persons. We ascertained an accuracy of

0.8 being the critical lower limit for sufficient visual quality. Typically around this point accuracy starts falling rapidly. (Figure 9) shows the accuracy for three different glyphs exemplarily. It can be further seen that the optimal vectorization parameter depends on the complexity of the actual glyph shape. The form of the "d" from (Figure 9) is less complex than the other two capital letters and can therefore be better approximated. This results in higher compression rates as well (Figure 10). Hence, the optimal vectorization parameters must be calculated individually for each glyph, based on the ascertained ground truth results. Most glyphs of the evaluated alphabets showed a local accuracy maximum especially for $\varepsilon$ values between 0.2 and 0.3.

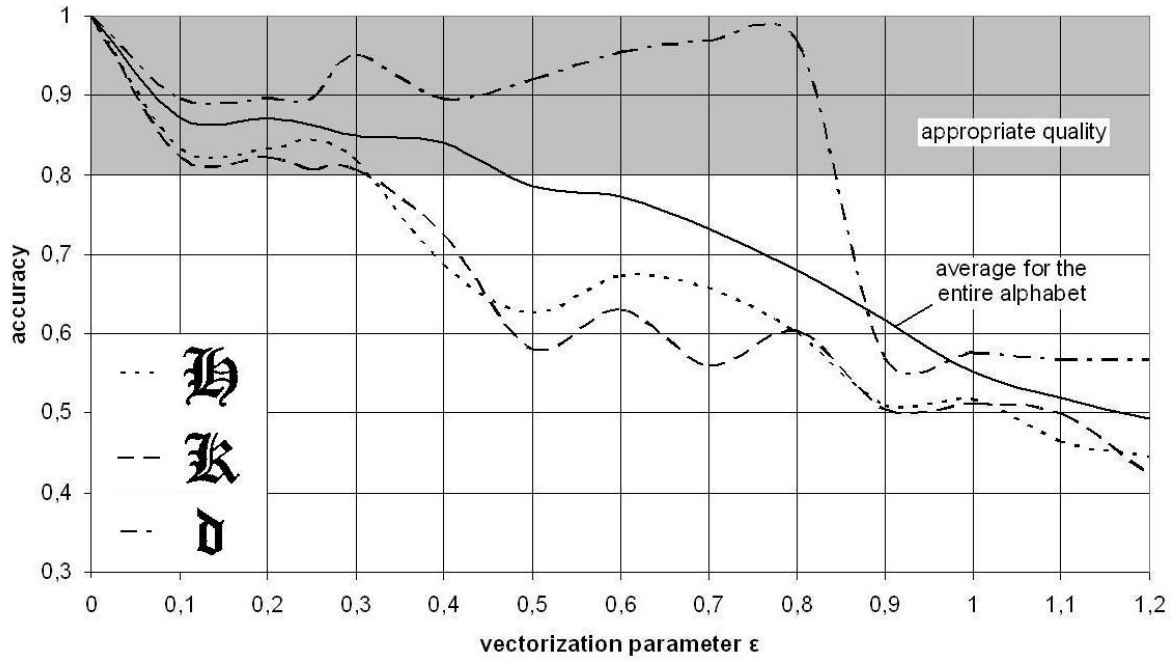


**Figure 9: Accuracy of reproduced glyphs for different $\varepsilon$ values**

There is an interdependency between the compression level and the quality to be reached. Compression means having fewer vertexes describing the contours and therefore fewer details. For the preparation of document images the visual quality is the primary criterion. With the used test images we could largely achieve compression rates between 4 and 5 at good quality. It should be mentioned that this value does only refer to the memory saving when producing contour descriptions of glyphs. In the over-all process there emerges an additional symbolic compression (see e.g. [11]) by assigning groups of similar glyph images to one prototype.

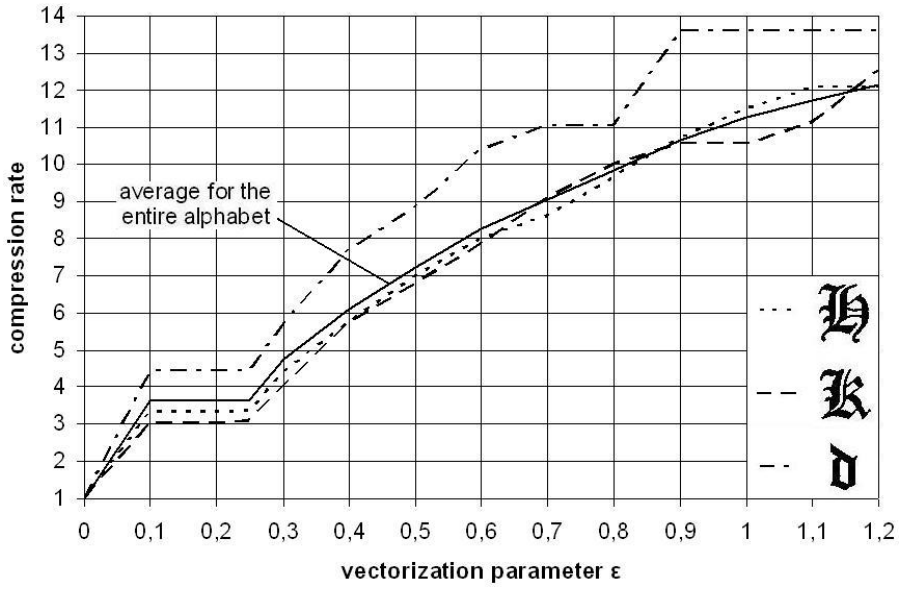


**Figure 10: Compression rates for the contour description of glyphs**

(Figure 11) shows the screened images of SVG glyphs obtained by using different $\varepsilon$ values for the aforementioned examples. The graphics were rendered using the Batik Rasterizer [12]. The original images as basis for the vectorization were scanned at 300 dpi using 256 grey levels.

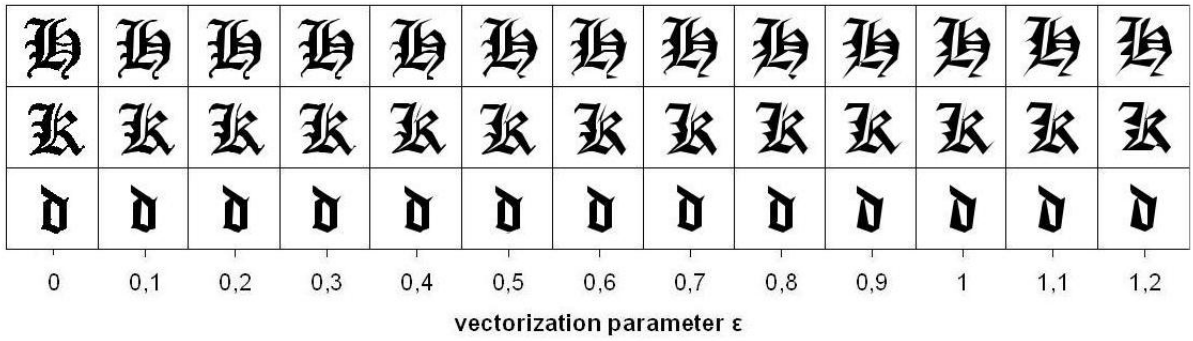


**Figure 11: Rendering of SVG glyphs**

The visual quality achieved with the appropriate vectorization parameters is quite convincing. Especially if the output format is to be enlarged, the vector description ensures smooth forms and leads to an even better reproduction quality compared to bitmap glyphs.

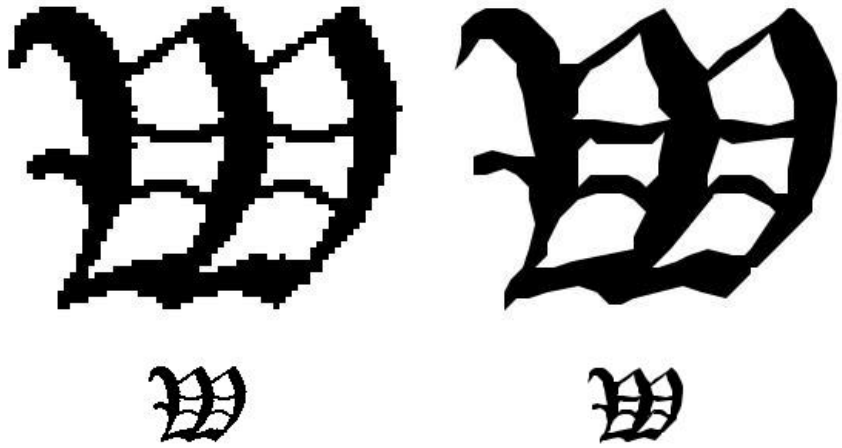

**Figure 12: Bitmap vs. vector glyph (original size and scaled to 400%)**

## 6 Conclusion and future work

The presented methods for vectorizing glyphs as well as SVG for defining whole alphabets are promising tools for the preparation of digitized documents. It could be shown that a good visual quality can be achieved despite the shape descriptions being additionally compressed. So far, we used only methods which lead to linear primitives for the representation of object contours. Advantages of this approach are the easy and fast computation and the straight forward transformation into SVG paths. These simple descriptions were found already sufficient for the given task. Considering primitives of higher complexity, like Bezier curves, could lead to a higher accuracy and a reduced count of vertexes. It is not yet clear if the slightly better results justify the higher computational effort especially for the task of vectorizing glyphs. This will be subject to investigation in future works.

Besides the refinement of the applied methods we are also working towards a framework for seamlessly supporting retrospective digitization and publishing of historical works. The results were quite promising for successfully integrating the delineated vectorization methods into the document image analysis system mentioned at the beginning.

## References


[1] Kopec, G. & Lomelin, M. (1996). Document-Specific Character Template Estimation. In Proc. SPIE Vol. 2660, Document Recognition III, Luc M. Vincent; Jonathan J. Hull; Eds.

[2] Pletschacher, S. (2005). OCR Alternatives for Electronic Publishing of Digitised Documents. In Proceedings 9th ICCC International Conference on Electronic Publishing, Peeters Publishing Leuven

[3] Breuel, T., Janssen, W., Popat, K., & Baird, H. (2002). Paper to PDA. In ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 1, IEEE Computer Society, Washington, DC

[4] SVG - Scalable Vector Graphics, W3C, http://www.w3.org/Graphics/SVG/, last visited February 2006

[5] XML - Extensible Markup Language, W3C, http://www.w3.org/XML/, last visited February 2006

[6] Apache Software Foundation, *Squiggle - the SVG Browser*, http://xml.apache.org/batik/svgviewer.html, last visited February 2006

[7] Tombre, K., Ah-Soon, C., Dosch, P., Masini, G. & Tabbone, S. (2000). Stable and Robust Vectorization: How to Make the Right Choices. In A. K. Chhabra and D. Dori, (Eds.), Graphics Recognition - Recent Advances, pp. 3-18. Springer Verlag, Lecture Notes in Computer Science, vol. 1941

[8] Peng, Jingliang (2000). An efficient algorithm of thinning scanned pencil drawings. In Journal of Image and Graphics, 5(5), 434-439, 2000

[9] Mikhef, Artem and Vincent, Luc and Faber, Vance (2001). High-Quality Polygonal Contour Approximation Based on Relaxation. In Proceedings 6th International Conference on Document Analysis and Recognition (ICDAR 2001), IEEE Computer Society

[10] Unicode Inc., Unicode Standard, http://www.unicode.org/standard/standard.html, last visited February 2006

[11] Howard, P. G. (1997). Text image compression using soft pattern matching. The Computer Journal, 40, pp. 146-156

[12] Apache Software Foundation, Batik Rasterizer, http://xmlgraphics.apache.org/batik/svgrasterizer.html, last visited February 2006