Optical Character Recognition with the Gamera Framework*

Christoph Dalitz, René Baston
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Reinarzstr. 49, 47805 Krefeld, Germany

Abstract

Due to its flexibility, the Gamera framework for document analysis and recognition has been used in the past primarily for very specific document types like ancient scripts and music notations. This paper describes how Gamera can be used for the recognition of ordinary bread-and-butter text documents. We present an abstraction layer for separating the page segmentation and character recognition steps, and make our source code freely available as a ready-to-run Gamera toolkit. The usefulness of the new toolkit is demonstrated in experiments on the synthetic images from the UW-I data set.

1 Introduction

The Gamera framework for document analysis and recognition [1] is not itself a recognition system, but, rather, a software library in the Python programming language for building document recognition systems. In combination with its generous licensing terms and its multi platform support, this makes Gamera perfectly suited for a number of document recognition problems not amenable to shrink-wrapped software systems. Gamera has indeed been deployed successfully for the recognition of texts in ancient scripts and languages [2] [3], or for historical music notations [4] [5] [6].

While the flexibility of a software library combined with an easy to use scripting language allows for non-standard application areas, it also has the effect that building a concrete recognition system requires some time and effort. Consequently, Gamera has not yet been used often for recognizing ordinary text documents, for which shrink-wrapped systems might be sufficient. It is the goal

*Published in C. Dalitz (Ed.): "Document Image Analysis with the Gamera Framework." Schriftenreihe des Fachbereichs Elektrotechnik und Informatik, Hochschule Niederrhein, vol. 8, pp. 53-65, Shaker Verlag (2009) of the present work, to ease the deployment of Gamera for ordinary text document recognition, while at the same time allowing for optional fine control of the individual stages of the recognition process. We make the resulting software freely available under the GNU general public license[1].

As text recognition is the most common document recognition problem, Gamera already provides for this use case a built in module *roman_text*, written between 2001 and 2005 by Karl MacMillan. This module includes a bottom-up page segmentation algorithm and a function for converting a set of classified glyphs to an ASCII text string. Up to now, neither its page segmentation algorithm, nor its usage from a user's point of view has been documented. A particular problem with this module is that its page segmentation algorithm is hard coded into the module and cannot be easily replaced by a user without completely rewriting the text recognition. The present work addresses this drawback by introducing an object-oriented abstraction layer that allows for optional custom implementations of specific segmentation steps.

This paper is organized as follows: section 2 describes the Gamera built in module *roman_text* and its page segmentation algorithm, and section 3 describes the new, more flexible architecture as provided by our OCR toolkit. The final section 4 describes some experiences made with our new toolkit on the synthetic images from the UW-I image database [7].

2 Text Recognition Support built into Gamera

Text recognition can be done in Gamera with the aid of the module *roman_text*. This module assumes that the entire image only consists of text, which means that non-text zones should have been removed beforehand. Here we first document how the module *roman_text* is applied from a user's perspective, and then describe its page segmentation algorithm in detail. The fact that this segmentation cannot be easily replaced by the user with a custom segmentation algorithm was our main motivation to develop a now toolkit for OCR.

[1]See the section "Addons" on the Gamera home page `http://gamera.informatik.hsnr.de/`

```
from gamera.core import *
init_gamera()
from gamera import knn, roman_text

img = load_image("textimage.png")
ccs = img.cc_analysis()

cknn = knn.kNNInteractive([], \
           ["aspect_ratio", "moments", "volume64regions"], 0)
cknn.from_xml_filename("trainingdata.xml")
ccs = cknn.group_and_update_list_automatic(ccs, \
           max_parts_per_group=3)

page = roman_text.ocr(img, classifier=None, glyphs=ccs)
for i,sec in enumerate(page.sections):
  print "section", i, "contains", len(sec.lines), "lines:"
  print roman_text.make_string(sec.lines)
```

Listing 1: Python code demonstrating the use of *roman_text* for text recognition.

2.1 Using roman_text

The module *roman_text* provides a function *ocr()*, which does the page segmentation and returns a *Page* object. Apart from the image to be segmented, it also needs a list of glyphs (data type *Cc*) as input. As these are considered to be characters that are grouped to lines and sections in a bottom-up way, they should have been classified beforehand with Gamera's grouping algorithm [8] so that diacritical signs are already attached to their main character. Alternatively, the classification is done in the *ocr()* function when a classifier is passed as second argument, but this does not allow for passing custom parameters to the grouping algorithm.

Listing 1 shows a typical usage of the *ocr()* function. The returned type *Page* contains a list *Page.sections* representing paragraphs, each of which in turn has a property *lines* representing the text lines within the paragraphs. Each line stores

Character	*Unicode Name*	*Class Name*
!	`EXCLAMATION MARK`	`exclamation.mark`
2	`DIGIT TWO`	`digit.two`
A	`LATIN CAPITAL LETTER A`	`latin.capital.letter.a`
a	`LATIN SMALL LETTER A`	`latin.small.letter.a`

Table 1: Examples for class names derived from the unicode character names.

the glyphs in its property *glyphs*. The text lines can be passed to the function *make_string()* to obtain an ASCII or unicode string representation with the lines separated by line breaks. The signature of *make_string* is:

```
make_string(lines, name_lookup_func=name_lookup_unicode)
```

where the second argument is a translation function that takes the class name as input and returns its string representation. When no custom function is provided, the python core module *unicodedata* is used. This assumes that class names have been chosen according to the unicode naming convention [9], with the spaces replaced by periods, as shown in Tbl. 1.

2.2 The page segmentation algorithm in roman_text

The *ocr()* function first segments the page into paragraphs, which are subsequently split into lines. This two step approach has the advantage that the line splitting step only needs to split a single column, even in case of multicolumn layouts, because text lines from different columns will generally be in different paragraphs. The page segmentation algorithm is a typical "bottom-up" algorithm with uses the connected components (CCs) as starting point, even though it is different from all CC based algorithms described in the review [10].

The segmentation of the *page into paragraphs* is done with a simple bounding box extending and merging procedure. The bounding boxes of selected CCs are extended in all four directions by the average size of all CCs, which is computed for n CCs $c_1, \ldots, c_n$ as

$$\mathit{avgsize} = \frac{1}{2n} \sum_{i=1}^{n} \Big(\mathit{width}(c_i) + \mathit{height}(c_i) \Big) \tag{1}$$

The selected CCs for this procedure are those that are not too small (*black_area* greater than *avgsize*) and not too large (both *ncols* and *nrows* less then $20 \cdot \mathit{avgsize}$). When two extended bounding boxes overlap, both boxes are considered to belong to the same paragraph. Building the transitive closure of overlapping bounding boxes results in a partitioning of all bounding boxes into disjoint paragraphs, each of which is represented by the smallest rectangle containing all of its bounding boxes. Eventually every CC from the original image is assigned to a paragraph with which it overlaps.

The segmentation of *paragraphs into lines* is done by building equivalence classes of vertically overlapping CCs. During this partitioning, very large CCs (height deviates more than 40 from the average glyph height) are first omitted. These are afterwards assigned to one of the lines with which they overlap.

The segmentation of *lines into words* is based on the horizontal white space between adjacent CC bounding boxes. When it is greater than twice the average within the line, a word break is assumed.

It should be noted that there are a number of thresholds and parameter values used in Karl MacMillan's algorithm which are chosen by a "rule of thumb" rather than by a theoretical or experimental justification. Moreover, the bounding box extension algorithm can also be tried for directly finding the textlines, by extending different amounts in the vertical and horizontal direction. A more detailed investigation of the actual effects of these parameter values seems to be an interesting subject for future investigations.

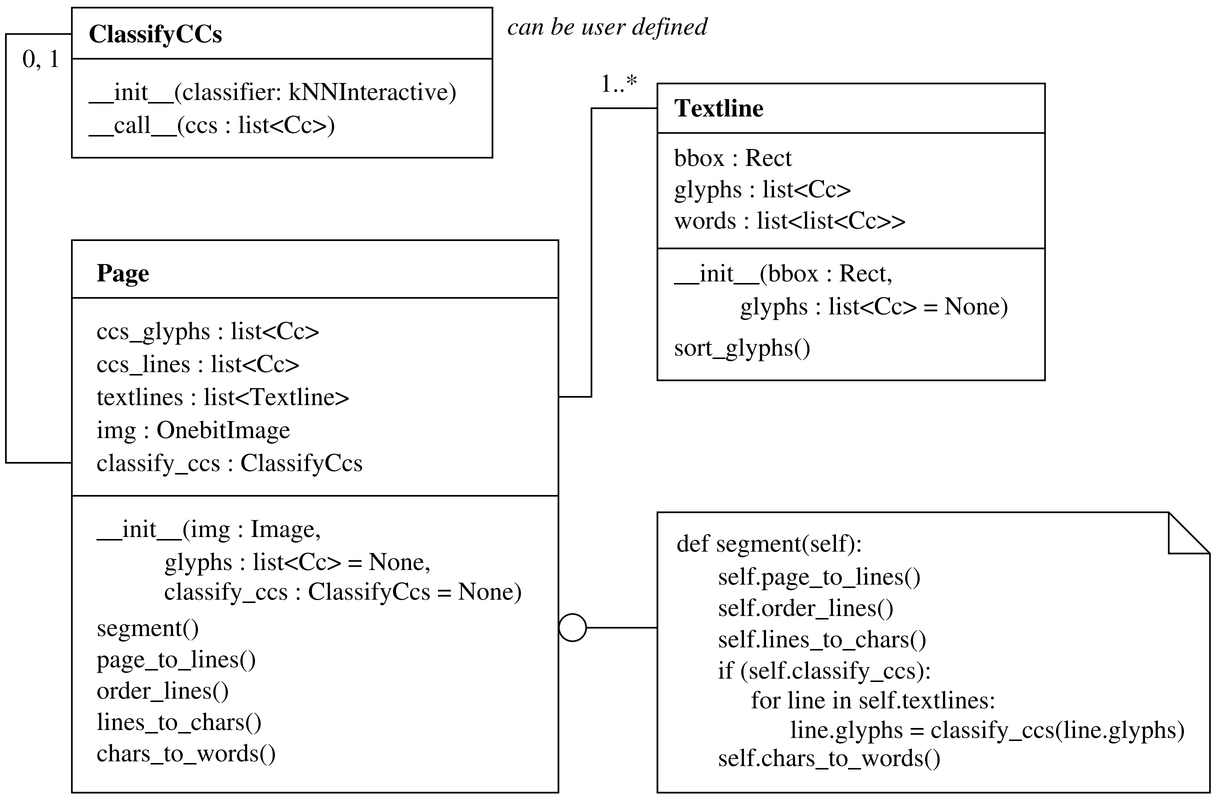
3 The new Architecture for Text Recognition

To make the recognition process more flexible, we must allow user defined functions to replace individual steps of the page segmentation process. Moreover the user must have a way to control whether the attachment of diacritical signs (accents, dots, etc.) to characters is left to Gamera's classifier or whether it is done during the page segmentation step.

We therefore define a class *Page* that provides a public method *segment* for doing the segmentation. Like *roman_text*, we assume that the image only contains text zones. In that case, the segmentation should not eventually yield paragraphs, but text lines[2]. The segmentation result is stored in the public member variable *textlines*. The user can then simply replace a specific segmentation step by deriving a custom class from *Page*, which overwrites the segmentation step in question with a custom method. Fig. 1 gives an overview over the classes defined in our OCR toolkit.

While it is possible to overwrite directly the *segment* method, it is typically more desirable to overwrite one of the functions called in *segment*, because in most application cases only a specific segmentation step might needed to be replaced. The substeps are implemented in the following functions (see Fig. 1):

[2]Even when paragraph zones are searched, like in *roman_text*, this is only an intermediate step for eventually finding the text lines.



```
def segment(self):
    self.page_to_lines()
    self.order_lines()
    self.lines_to_chars()
    if (self.classify_ccs):
        for line in self.textlines:
            line.glyphs = classify_ccs(line.glyphs)
    self.chars_to_words()
```

Figure 1: Class diagram of the classes involved in page segmentation. Custom algorithms can be used by deriving a class from *Page* that overwrites some of the functions called in *segment()*.

- *page_to_lines* splits the page into segments representing lines. The segments are of the Gamera image type *Cc* and are stored in the variable *ccs_lines*. The algorithm used in the base class *Page* is based on the bounding box merging as described in section 2.2.
- *order_lines* sorts the line segments in reading order. The base class *Page* uses the Gamera plugin *textline_reading_order*, which follows [11].
- *lines_to_chars* splits each line into its characters. The results are stored in the variable *textlines*, which is a list of *Textline* objects; the characters are stored in *Textline.glyphs*. The character segmentation algorithm in the base class *Page* uses a connected component segmentation followed by a rule based merging of diacritical signs to main characters, that is described in detail in [12].
- *chars_to_words* groups the characters in each line to words and stores the words in the variable *textlines* in *Textline.words*. The word grouping in the base class *Page* uses the method described in section 2.2.

Listing 2 shows a typical use of the OCR toolkit and how easy it is to replace

```
from gamera.core import *
init_gamera()
from gamera import knn
from gamera.toolkits.ocr.ocr_toolkit import *
from gamera.toolkits.ocr.classes import Textline, Page

# use runlength smearing instead of bounding box merging
class MyPage(Page):
  def page_to_lines(self):
    self.ccs_lines = self.img.runlength_smearing()

# load training data into classifier
cknn = knn.kNNInteractive([], \
          ["aspect_ratio", "moments", "volume64regions"], 0)
cknn.from_xml_filename("trainingdata.xml")

# segment page
img = load_image("textimage.png")
page = MyPage(img)
page.segment()

# classify characters and create output text
for line in page.textlines:
  line.glyphs = \
       cknn.classify_and_update_list_automatic(line.glyphs)
  line.sort_glyphs()
  print "Text␣of␣line", textline_to_string(line)
```

Listing 2: Python code demonstrating the use of the OCR toolkit. In this example, the *Page* class is derived to use a different page segmentation algorithm.

the page segmentation into lines by some other algorithm like the runlength smearing provided as the plugin *runlength_smearing* by Gamera. The function *textline_to_string* is provided by the OCR toolkit and assumes that the naming conventions from Tbl. 1 have been used in the training data.

In general, the classification can be done independently from the page segmentation, e.g. by first calling *Page.segment()* and then calling *kNNInteractive.classify_list_automatic()* on all *Textline.glyphs* in *Page.textlines*. In that case, the optional argument *classify_ccs* in the *Page* constructor can simply be omitted. It might be however, that a classification based character segmentation is preferable, e.g. by Gamera's grouping algorithm that joins broken characters and can also automatically attach (trained) diacritical signs. In that case, the user can pass a callable class [13] as the parameter *classify_ccs*. The signature for calling this class must be of the form as shown in the "def segment" box in

```
from gamera.core import *
init_gamera()
from gamera import knn
from gamera.toolkits.ocr.ocr_toolkit import *
from gamera.toolkits.ocr.classes \
         import Textline, Page, ClassifyCCs

# segment lines into chars only by CC analysis
class MyPage(Page):
  def lines_to_chars(self):
    subbccs = self.img.sub_cc_analysis(self.ccs_lines)
    for i,segment in enumerate(self.ccs_lines):
      self.textlines.append(Textline(segment, subccs[i]))

# define classifying function
cknn = knn.kNNInteractive([], \
          ["aspect_ratio", "moments", "volume64regions"], 0)
cknn.from_xml_filename("trainingdata.xml")
classify = ClassifyCCs(cknn)
classify.parts_to_group = 4 # to use grouping algorithm

# segment page
img = load_image("textimage.png")
page = MyPage(img, classify_ccs=classify)
page.segment()  # will call classify

# create output text
for line in page.textlines:
  line.sort_glyphs()
  print "Text␣of␣line", textline_to_string(line)
```

Listing 3: Python code demonstrating how the task of joining broken characters and diacritical signs can be handed to the classifier rather than to *Page.lines_to_chars()*.

Fig. 1. Moreover, in that case the *lines_to_chars* method should also be overwritten to only do a plain CC analysis without the diacritics merging step. Listing 3 shows an example how to let the grouping algorithm attach the diacritical signs.

4 Experiments on the UW-I Image Dataset

To demonstrate how the new toolkit works, we have measured its performance on images from the *UW English Document Image Database I*, which was released by Haralick et al. in 1993 at the University of Washington [7]. This dataset contains both scans from machine printed journal articles and synthetic images generated from LaTeX sources. For all images, zone descriptions and groundtruth texts are provided. There are, however, no character training data matching the images[3]. We have only used the synthetically generated images to circumvent the problem of having to deal with noise and copy margins that are present in almost all of the real scans. So the results provide insight into what can be achieved with the new OCR toolkit out-of-the-box on noise free images.

4.1 Test method

As the synthetic images in the UW-I dataset stem from technical journal articles, they contain a considerable amount of mathematical formulae, which are represented in the groundtruth data by their corresponding LaTeX escape sequences [14]. As this posed particular problems for our edit distance based performance measure (see below), we simply omitted lines containing any LaTeX escape sequences in the groundtruth data from our performance measurement[4]. Moreover, we have only evaluated those parts from the images that were marked as "text-body" in the zoning data.

To measure the accuracy of the recognition output in comparison to the ground truth text, we used the *edit distance*, also known as *Levenshtein distance* [15]. It measures the minimum number of insertions, deletions or substitutions necessary to transform one string into the other, and can be computed in $O(mn)$ time, where m and n are the lengths of the two strings. This leads to the following measure for the character recognition accuracy [16]:

$$\textit{accuracy} = \frac{|T| - \textit{distance}(S,T)}{|T|} \tag{2}$$

where T is the groundtruth string, S the OCR output string, and $|T|$ is the length of T, i.e., the total number of characters to be recognized.

For segmenting the page into text lines, we have used the default method implemented in the OCR toolkit, which is based on the Gamera plugin *bbox_merging*. This plugin is insofar a generalization of Karl MacMillan's bounding box merging algorithm described in section 2.2, as it allows for choosable extensions Ex

[3]The character images in the UW-I Image Dataset are synthetically degraded from a different source and have no relation to the document images.

[4]LaTeX commands can easily be identified because they all start with a backslash character.

and Ey in the x and y direction, respectively. We have set $Ey = 0$, and Ex to twice the average size of all CCs, as defined in eq. (1).

For character classification, we used Gamera's kNN classifier with the feature combination *aspect_ratio*, *volume64regions*, *moments* and *nholes*. This is almost the same feature combination that turned out to be optimal in two different studies on different kinds of printed symbols, both with respect to the hold out error rate [4] and n-fold cross-correlation [5]. The only difference is that we have replaced the feature *nrows* (the absolute character height) with *nholes*, because *nrows* is not scale invariant and would have required to provide training characters in all possible sizes. Due to the small size of our training set, we had to set $k = 1$ in the kNN classifier.

As training data, we have used a complete character set in lower and upper case extracted from the document images, which we have complemented with artificial training data generated by ourselves with LaTeX. The latter was a complete character set in upper and lower case letters both in Roman and italic face at 10pt character size with the LaTeX package "times". We have rasterized the LaTeX generated postscript files with 300dpi.

4.2 Results

Our test data contained in total 208,029 characters. The summed up edit distance to the OCR output string was 37,223, which leads according to (2) to an accuracy of only 82.1%. A closer look at the individual recognition errors showed that almost all of these errors were due to text line segmentation errors, mostly adjacent text lines merged into one. From a total of 4,099 text lines, 158 (about 4%) were not correctly segmented by our simple variant of the bounding box merging algorithm.

To evaluate word and character segmentation, and the classification independently from the text line segmentation, we have also counted the OCR error rate for only those text lines that were correctly identified. These contained 185,221 characters and the accuracy (2) turned out to be 99.2%. This shows that both the segmentation of lines into characters and the kNN classifier work very well.

A detailed analysis of the recognition errors showed that the kNN classifier tended to confuse some characters that only differ in size, like upper and lower case "s" or "w", and some very similar characters, like apostrophe and comma. This is easily understandable because we have only used scale invariant features. We therefore added heuristic rules[5] based on the text line dimensions as a post-correction step after kNN classification, which further increased the recognition rate to 99.8%.

To see the effect of different page segmentation algorithms on the total accuracy, we have in a different test run replaced the bounding box merging segmentation with runlength smearing with the default parameters provided by the corresponding Gamera plugin *runlength_smearing*[6]. In this case, the total accuracy was 87.7%, which was slightly better than *bbox_merging*. It should be noted however that a goal directed evaluation of page segmentation algorithms based on the edit distance of the OCR output is not very meaningful, because a single error can lead to a high edit distance. For evaluating page segmentation algorithms, direct methods based on pixel sets are preferable [17].

5 Conclusions and Future Work

The new OCR toolkit allows for an easy substitution of individual steps in the text recognition process. Our experiments on the UW-I dataset indicate that the most sensitive step is the segmentation of the page into text lines, which again emphasizes the importance of making this step substitutable by custom algorithms.

To save the user from the burden of implementing own page segmentation algorithms, it would be desirable to have more page segmentation algorithms in the Gamera core. While Gamera already provides three simple methods with the plugins *bbox_merging*, *runlength_smearing*, and *projection_cutting*, it would be nice to have some of the better performing page segmentation algorithms [18].

The OCR toolkit assumes that the image contains only text zones. In practice however, there might be additional zones containing figures or images, which need to be removed before applying the toolkit. For this task it would be very useful to have text/graphics separation algorithms implemented in the Gamera core, like those cited in [19].

Another point that can be cumbersome from a user's perspective is the need to train the classifier. For ordinary text documents based on the Latin alphabet, this could be made easier by providing an already prepared database, or by directly providing a special purpose trained classifier. For the latter option, it might be

[5]see [12] for details

[6]see the Gamera plugin documentation for details

worthwhile to consider including or wrapping third party Latin character classifiers like Tesseract [20] or gocr [21].

References

[1] M. Droettboom, K. MacMillan, I. Fujinaga: *The Gamera framework for building custom recognition systems.* Symposium on Document Image Understanding Technologies, pp. 275-286 (2003) (see also `http://gamera.informatik.hsnr.de/`)

[2] K. Canfield: *A Pilot Study for a Navajo Textbase.* Proceedings of The 17th International Conference on Humanities Computing and Digital Scholarship (ACH/ALLC), pp. 28-30 (2005)

[3] S. Reddy, G. Crane: *A Document Recognition System for Early Modern Latin.* Chicago Colloquium on Digital Humanities and Computer Science (2006)

[4] C. Dalitz, T. Karsten: *Using the Gamera framework for building a lute tablature recognition system.* 6th International Conference on Music Information Retrieval (ISMIR), pp. 478-481 (2005)

[5] C. Dalitz, G.K. Michalakis, C. Pranzas: *Optical Recognition of Psaltic Byzantine Chant Notation.* International Journal of Document Analysis and Recognition 11, pp. 143-158 (2008)

[6] C. Dalitz, C. Pranzas: *German Lute Tablature Recognition.* 10th International Conference on Document Analysis and Recognition (ICDAR), pp. 371-375 (2009)

[7] I. Guyon, R.M. Haralick, J.J. Hull, I.T. Phillips: *Data Sets for OCR and Document Image Understanding Research.* Handbook of Character Recognition and Document Image Analysis, pp. 779-799, Eds. H. Bunke and P.S.P. Wang, World Scientific (1997)

[8] M. Droettboom: *Correcting broken characters in the recognition of historical printed documents.* Joint Conference on Digital Libraries, pp. 364-366 (2003)

[9] Unicode, Inc.: *The Unicode Character Code Charts By Script.* `http://www.unicode.org/charts/` (1991-2009)

[10] R. Cattoni, T. Coianiz, S. Messelodi, C.M. Modena: *Geometric Layout Analysis Techniques for Document Image Understanding: a Review.* ITC-irst Technical Report TR#9703-09 (1998)

[11] T.M. Breuel: *High Performance Document Analysis.* Symposium on Document Image Understanding Technology, USA, pp. 209-218 (2003)

[12] R. Baston: *Ein OCR-Toolkit für das Gamera-Framework.* Bachelor Thesis, Hochschule Niederrhein, Fachbereich Elektrotechnik und Informatik, Krefeld, Germany (August 2009)

[13] D.M. Beazley: *Python Essential Reference.* 4th edition, Addison-Wesley (2009)

[14] L. Lamport: *LaTeX: A Document Preparation System.* 2nd edition, Addison-Wesley (1994)

[15] C.D. Manning, P. Raghavan, H. Schütze: *Introduction to Information Retrieval.* Cambridge University Press (2008)

[16] T.A. Nartker: *Benchmarking DIA Systems.* Handbook of Character Recognition and Document Image Analysis, pp. 801-820, Eds. H. Bunke and P.S.P. Wang, World Scientific (1997)

[17] F. Shafait, D. Keysers, T. Breuel: *Pixel-Accurate Representation and Evaluation of Page Segmentation in Document Images.* 18th International Conference on Pattern Recognition (ICPR), pp. 872-875 (2006)

[18] F. Shafait, D. Keysers, T. Breuel: *Performance Comparison of Six Algorithms for Page Segmentation.* 7th IAPR Workshop on Document Analysis Systems (DAS), pp. 368-379, LNCS 3872, Springer (2006)

[19] S.P. Chowdhury, S. Mandal, A.K. Das, B. Chanda: *Segmentation of Text and Graphics from Document Images.* 9th International Conference on Document Analysis and Recognition (ICDAR), pp. 619-623 (2007)

[20] R. Smith: *tesseract-ocr.* `http://code.google.com/p/tesseract-ocr/` (2005-2009)

[21] J. Schulenburg: *gocr.* `http://jocr.sourceforge.net/` (2000-2009)