A complete Document Analysis and Recognition system for GNU/Linux

Master's Thesis in Computer Science Engineering

Joaquim Rocha

Supervisor: Luís Arriaga, PhD

December 2008

Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

UE
168
661

A complete Document Analysis and Recognition system for GNU/Linux

Master's Thesis in Computer Science Engineering

Joaquim Rocha

Supervisor: Luís Arriaga, PhD

December 2008

168 661

Esta dissertação não inclui as críticas e sugestões feitas pelo júri.

Resumo

Um sistema completo de Análise e Reconhecimento de Documentos para GNU/Linux

Os motores de Reconhecimento Óptico de Caracteres (OCR) comuns simplesmente "lêm" uma imagem não considerando a sua estrutura ou formatação. A formatação de um documento é um assunto muito importante na compreensão de um documento. Assim, o uso de motores de OCR não é suficiente para converter fielmente uma imagem de um documento para um formato electrónico.

A Análise e Reconhecimento de Documentos (DAR) engloba a tarefa de reconhecer a estrutura de um documento o que, combinado com um motor de OCR, pode resultar numa conversão fiel de um documento para um formato editável. Estes sistemas existem como aplicações comerciais sem uma verdadeira equivalência em Software Livre actualmente e não estão disponíveis para o sistema operativo GNU/Linux.

O trabalho descrito neste relatório tenta responder a este problema ao oferecer uma solução que combina componentes de Software Livre e sendo comparável, mesmo na sua fase inicial, a soluções comerciais disponíveis.

Abstract

A complete Document Analysis and Recognition system for GNU/Linux

Regular OCR engines simply "read" an image not considering its structure or layout. A document's layout is a very important matter in the understanding of a document. Hence, using OCR engines is not enough to fairly convert an image of a document to an editable format.

Document Analysis and Recognition (DAR) encompasses the task of recognizing a document's structure which combined with an OCR engine can result in a fair conversion of a document to an editable format. Such systems exist as commercial applications with no real equivalence in Free Software nowadays and are not available for the GNU/Linux operating system.

The work described in this report attempts to answer this problem by offering a solution combining only Free Software components and being comparable, even in its early stage, to available commercial solutions.

Aknowledgements

This project wouldn't be possible without the support from many people that contributed for its final result.

First, I would like to express my appreciation and deep gratitude to Professor Luís Arriaga for being my supervisor in this project and for all his understanding, attention and enthusiastic support.

I would also like to thank my colleagues, and friends who I couldn't name all here, that always supported me along the development of this work and other projects.

I want also to express my gratitude to my girlfriend who was always there for me with all her support and understanding.

Last but not least, I would like to thank my parents for their unconditional support and belief, of whom I am truly proud of, and for educating me to become the person I am today.

Contents

Resumo i

Abstract ii

Aknowledgements iii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 General Concepts 3
- 1.3 Document Analysis and Recognition 3
- 1.4 Objectives 4
- 1.5 Structure 5

2 State of the art 7
- 2.1 Optical Character Recognition 8
  - 2.1.1 OCR History 8
  - 2.1.2 Recent Solutions 8
- 2.2 Document Analysis and Recognition 10
- 2.3 DAR History 10
- 2.4 Modern systems 11
  - 2.4.1 Mobile devices 11
  - 2.4.2 Recent solutions 12
- 2.5 Conferences about DAR and OCR 14

3 The System 15
- 3.1 Overview 16

3.2 Technology and Development Tools 16
3.2.1 Python 16
3.2.2 PyGTK 18
3.2.3 PIL 18
3.2.4 PyGoocanvas 19
3.2.5 XML 19
3.2.6 ODT 20
3.2.7 ODFPy 21
3.2.8 Ghostscript 22
3.2.9 Unpaper 22
3.3 Architecture 23

4 Implementation **25**
4.1 Layout Analysis 28
4.1.1 The sliding window algorithm 31
4.1.2 Retrieving blocks 37
4.2 Recognition 45
4.2.1 OCR Engines 45
4.2.2 Classification 47
4.2.3 Text properties recognition 48
4.3 Content representation 53
4.3.1 Data boxes 53
4.3.2 Page data 54
4.4 Exportation to editable formats 55
4.4.1 Exportation to ODT 56
4.4.2 Exportation to HTML 56
4.4.3 Adding support for more formats 57

5 OCRFeeder **58**
5.1 Design and usability 59
5.2 Interface overview 61
5.2.1 Document images area 61
5.2.2 Selectable boxes area 61

5.2.3 Box editor area 63
5.3 Features 64
5.3.1 Adding document images 64
5.3.2 PDF importation 64
5.3.3 Exportation 65
5.3.4 Project loading and saving 65
5.3.5 Preferences 67
5.3.6 Edit page 68
5.3.7 Delete images 68
5.3.8 Zoom 68
5.3.9 OCR engines 69
5.3.10 Unpaper 69
5.3.11 Layout analysis and OCR 71

6 Testing **72**
6.1 Features Comparison 73
6.2 Tests 74
6.2.1 Lyrics document 76
6.2.2 Make: magazine 80
6.2.3 Beautiful Code book page 84
6.2.4 The Search book cover 90
6.2.5 Linux Magazine page 94
6.3 General Appreciation 99

7 Conclusions and Future Work **100**
7.1 Problems 101
7.2 Future Work 102

References **105**

A Example of a project XML file **108**

B Installation and usage **111**
B.1 System Requirements 111

B.2 Installation on Ubuntu 112
B.2.1 Installing the packages 112
B.3 Command line usage 113

List of Figures

3.1 The system global architecture 23
3.2 A more detailed architecture diagram 24

4.1 Lyrics document image. 26
4.2 Recognized text for lyrics 27
4.3 Lyrics document with outlined format structure 28
4.4 Illustration of part of the detection algorithm for 1-column simple documents 30
4.5 A not so simple document example 31
4.6 Binary representation 34
4.7 Example of contrasting colors 35
4.8 Optimization of the function to find contrast within a window . . . 36
4.9 A block in a BRL 39
4.10 Distances and lines in typography 40
4.11 An example of extra charge 41
4.12 A simple block with a legend 42
4.13 Font size and letter spacing detection 50
4.14 Text angle detection 52

5.1 OCRFeeder Studio final paper prototype 60
5.2 OCRFeeder Studio main areas 62
5.3 Preferences dialog (appearance tab) 67
5.4 Paper sizes dialog 68
5.5 Example of the OCR engines dialogs 69
5.6 Unpaper dialog 70

6.1 Test: Lyrics document with OCRFeeder 77
6.2 Test: Lyrics document with Omnipage 78
6.3 Test: Lyrics document with FineReader 79
6.4 Lyrics document exported to ODT by OCRFeeder 80
6.5 Lyrics document exported to Doc by FineReader 81
6.6 Lyrics document exported to Doc by OmniPage 82
6.7 Test: Make: magazine with OCRFeeder 83
6.8 Test: Make: magazine with OmniPage 84
6.9 Test: Make: magazine with FineReader 85
6.10 Test: Beautiful Code book page with OCRFeeder 87
6.11 Test: Beautiful Code book page with OmniPage 88
6.12 Test: Beautiful Code book page with FineReader 89
6.13 Test: The Search book cover with OCRFeeder 91
6.14 Test: The Search book cover with OmniPage 92
6.15 Test: The Search book cover with FineReader 93
6.16 Test: Linux Magazine with OCRFeeder (automatic window size) . . 95
6.17 Test: Linux Magazine with OCRFeeder (manual window size) . . . 96
6.18 Test: Linux Magazine with OmniPage 97
6.19 Test: Linux Magazine with FineReader 98

List of Tables

6.1 Features comparison for several DAR and OCR solutions 75

Chapter 1

Introduction

This report describes the work done in a very specific area that relates to computer vision, artificial intelligence and image processing – Document Analysis and Recognition. The main purpose of this project is the creation of a system for GNU/Linux capable of performing Document Layout Analysis (DAR) and Optical Character Recognition (OCR) on document images.

1.1 Motivation

Over the centuries, humanity have used paper to record information starting with the Egyptians, Greeks and Romans who used a paper-like material called *papyrus* [1]. While paper as we know it was first developed in China around the 2nd [2], the use of papyrus dates back to 3500 B.C. and it was the first writing material previous to the usage of paper [3].

Therefore, we owe the transfer of information, education and knowledge along the times to the production and wide use of paper [1].

The generalized use of electronic documents to store and preserve information is a big advance from paper and have been used since a while with all its advantages (like searching, converting between formats, etc.) towards paper documents. Therefore, it is of extremely importance to convert paper documents to electronic formats.

Although there are good solutions available, mainly commercial ones, there is no solution that fairly converts a document image into an editable document for the GNU/Linux operating system. Even when considering other operating systems, there is not a Free Software solution available for any system that can compare itself to the top commercial solutions.
However, instead of "cloning" a commercial solution, this project tries to create a complete and original system. This system was designed to be used by anyone without having to spend much effort learning how to install or use it.

Outlining the contents of an image seems an easy task to do – even a child can do it – but to develop such a system represents an interesting challenge. It is

specially interesting because it's an attempt to make a computer perform basic functions of a human-being – to see and to write.

1.2 General Concepts

Optical Character Recognition (OCR) is a field of research of artificial intelligence, pattern recognition and machine vision. By definition, OCR is the conversion of an image of text by mechanical or electronic means to digital text (machine-editable).

Although OCR is closely related and plays an important role in the development of this project (even being present on its very name), this project is not about OCR or the recognition of characters. It is about Document Analysis and Recognition.

1.3 Document Analysis and Recognition

The main purpose of Document Analysis and Recognition (DAR) is to automatically segment the text and graphical contents (also called zoning) of an image which represents a document and then process them to recognize their logical role in the document. Therefore, DAR encompasses two steps: geometric and logical analysis [4]. While the former refers to the extraction of the regions of interest (the homogeneous regions that contain a picture, a text paragraph, a logotype, etc.) within a document image, the latter relates to the classification of each region according to its role in the document – that is, if a region represents a title, a footnote, a column of text, an image's caption, etc.

This project focus on the geometric analysis as it attempts to retrieve only the regions not trying to find their role in the document.

DAR systems are related to several fields of computer science like image processing, artificial intelligence, pattern recognition and even databases.

The output from an extraction performed by DAR techniques is preferably in a format that may be processed by a machine. [5].

The name and acronym DAR is not as recognized as OCR. Often publications or products use the acronym OCR when referring to DAR systems. The name used for DAR also varies, beyond DAR, usually *image analysis and recognition*, *layout analysis*, *document analysis*, *document segmentation*, *document image understanding* and other combinations can be read on references to DAR systems.

In this document however, a restriction to the use of the names *DAR* and *layout analysis* is attempted in order to simplify the reading and understanding of this report.

1.4 Objectives

The main objective of this project was to develop a DAR and OCR system for GNU/Linux. The system should analyze a document image, retrieving the location and properties of its contents; after that, the system should perform OCR over the contents and identify which of them are graphics and text; in the end, an editable document format should be generated.

Since the number and variations of document layout formats can be countless, the system should not be restricted to accept only a single type of document layout or structure nor should it know the type previously – the objective was that the system could analyze and retrieve the contents of a document with any layout.

Part of the challenge consisted in the fact that there were free OCR engines available that offer good recognition rates but there was no direct way of getting the contents of a document image – that is, most OCR solutions available only return the text but little or no information about the layout. Hence, a technique to retrieve this contents was developed from scratch (like it's explained in further chapters).

Another objective was that the development of this project would hopefully result in a solution comparable to the commercial ones. By opting to develop this project as Free and Open Source Software, it will make it possible for every researcher or enthusiastic of DAR and OCR to extend and improve the ideas

explored in the creation of it and hence continue towards the creation of a great tool and make the conversion of document images on the GNU/Linux operating system a solved problem.

1.5 Structure

This section describes this document's structure and organization.

Chapter 1 gives an introduction to the work done, the motivation to do it and the objectives behind it. Some general concepts are presented as well in order to better identify the areas where this project belongs.

On Chapter 2 some of the main research done in the fields of OCR and DAR is introduced since its early days until recent times. Modern systems are also covered with the efforts dating back until around ten years ago and a brief presentation of the most famous solutions is given. The most important conferences about OCR and DAR are also mentioned.

Chapter 3 gives an overview about the project in what comes to choices about the technology used and intentions for the development of the system. It also presents two very simple diagrams that illustrate the system's architecture.

The details about the implementation of this project are covered in Chapter 4. The main algorithms created and used are explained using images and diagrams for an easy understanding of them.

Chapter 5 explains major concepts and concerns about the design and usability of the system. An overview about the graphical user interface and the main features is also given.

On Chapter 6 an evaluation of the system is done by comparing its features to other solutions either free and commercial ones. The same documents are tested on OCRFeeder and on two of the a main solutions available currently and the results are commented. The chapter finishes with a general appreciation of the system considering the tests results the comparisons with other systems.

Chapter 7 presents the conclusions about the system and analyzes in a general way its strengths and weaknesses as well as some problems and possible solutions

for them. Future work and improvements on the system are also mentioned in this chapter.

Chapter 2

State of the art

This section gives an overview of the research in the fields of Document Analysis and Recognition since its early times until nowadays where several free and commercial solutions are available.

Since Optical Character Recognition also plays a very important role in this project, an overview of it is also given.

2.1 Optical Character Recognition

2.1.1 OCR History

Tauschek registered a patent on OCR in Germany in 1929, a U.S. patent on OCR was registered later 1933 by Handel [6]. Although these represented great efforts, the beginnings of Optical Character Recognition (OCR) in what comes to computers date back to the 1950s where images of text and characters were attempted to be captured by mechanical and optical means [7]. In the 1960s and 1970s, OCR was used in post offices, banks, hospitals, aircraft manufacturers, etc. By then, the results given by OCR techniques presented too many flaws due to the state of the printed paper being analyzed – the conditions of the surface where the text was printed, the type fonts and the residue left by typewriters difficulted the process. Hence, OCR manufacturers had a big interest in the creation of standards in what comes to type fonts, ink and paper quality. Due to this, important institutions like *ANSI*[1], *ECMA*[2] and *ISO*[3] came up with the development of new fonts that could help accomplish high recognition rates.

2.1.2 Recent Solutions

Nowadays, OCR is much more developed and divulged than in its early years achieving really high accuracy rates. Nonetheless, as [7] states:
"*[...] 99% accuracy rates translates into 30 errors on a typical page containing 3,000 characters.*"

[1] *American National Standards Institution*
[2] *European Computer Manufacturers Association*
[3] *International Standards Organization*

There are many applications that provide Optical Character Recognition for images. Some of these applications also feature Layout Analysis. It is even normal nowadays to find an OCR application bundled with a cheap scanner device.

Commercial and free solutions are widely available. Considering only OCR engines and not complete OCR and DAR, among the existing Free Software solutions the most known ones are:

Ocrad

License: GNU General Public License[4]

Ocrad[5] is an OCR engine developed within of the GNU Project.

Gocr

License: GNU General Public License[4]

The *GOCR*[6] engine was developed by Joerg Schulenburg. Beyond reading text, it can also translate bar codes.

Tesseract

License: Apache 2.0 License[7]

Tesseract[8] was first developed by Hewlett-Packard[9] (HP) from 1985 until 1995. It was one of the top 3 OCR engines in the 1995 *University of Nevada, Las Vegas* (UNLV) Accuracy test and after that year not much work was done on it until it was released in 2006 by HP and UNLV. Since then it has been under active development by Google.

[4] `http://www.gnu.org/licenses/gpl-3.0.html`
[5] `http://www.gnu.org/software/ocrad/`
[6] `http://jocr.sourceforge.net`
[7] `http://www.apache.org/licenses/LICENSE-2.0`
[8] `http://code.google.com/p/tesseract-ocr`
[9] `http://www.hp.com`

2.2 Document Analysis and Recognition

According to [5], 85% of the new information stored on paper in the world is office documents. This surely contributes to the fact that DAR is largely used to process business related documents like obtaining the information from forms and checks, organization of documents, etc.
Nevertheless, recently the uses of DAR have been moved to other types of documents like ancient documents, digital documents like PDF as well as other types of images, for example images from surveillance and traffic cameras.

2.3 DAR History

In the early years of DAR, one of the first projects dedicated to the subject received a very direct name – *Document Analysis System* [8]. This publication from 1982 gave an overview of a system to convert printed documents to a way that can be processed by a computer. The idea was to subdivide an image in regions of a data type like text, graphics, etc. An approach to recognize font styles and types was also studied that involved a pattern-matching method.

The *Scientist's Assistant* (SA) was also an interesting project that appeared in 1991. The SA was a system designed to scan, perform OCR and tag a document by using the *Standard Generalized Markup Language* (SGML) [9]. The conclusions were that it would take more time to review and correct each scanned document than by typing it from the beginning [10]. Another early effort in this field was the *Licensing Support System* (LSS). This system would capture and track documents that belong to the Nuclear Regulatory Commission and a prototype was built featuring both OCR and manual key entries. After the construction of the prototype, it was concluded that the "*costs of conversion of hard copy documents to electronic form dominate the life cycle of the Licensing Support System.*" After that, LSS responsibles claimed that this system would only be able to give minimum document format (no information about certain styling formats like italics or bold type fonts) information [11].

2.4 Modern systems

The *Department of Computer Science* of *Cornell University* presented a way in 1995 that consists in two main steps: *segmentation* and *classification* [12]. In the segmentation step, the type font shape and layout information is used. In the classification step, the segmented content is compared with structure prototypes. The information contained in the predefined prototypes is about the present or not present symbols. If there is any previous information about the document's styles, it can be also used as additional help in the processing steps [13].

In 1998, in the *University of Nevada*, USA, the *Information Science Research Institute* developed a project named Autotag. Its purpose is to automate the conversion of general technical documents by performing a physical analysis, by means of OCR, followed by a logical analysis of the document.
According to [14]:
"*Autotag accepts a physical document representation as input. It analyzes and combines the information contained in this form and maps it to a logical representation.*"
Autotag takes the physical representation information (given by the used OCR software) about the document being analyzed and converts that information into another representation in SGML. For example, [14] refers that *ScanWorX* OCR device from *Xerox* would produce output in a format called XDOC which Autotag then would convert to SGML. This conversion step exists so Autotag remains device independent [14].

After the physical information retrieval, Autotag interprets the useful information contained in it. To do this, and since the are many types of documents and the logical representation changes from type to type, they focused on the relevant logical components for scientific journal articles. This way, Autotag can retrieve abstracts, authors names, tables, figures, etc. from this class of documents [14].

2.4.1 Mobile devices

The growing importance, power and variety of portable devices such as PDAs and mobile phones have also resulted in the adaptation of DAR solutions for these

kind of devices to process images captured by the devices' cameras.

An example of this is the recognition of business cards. The smart phone *Sony Ericsson P990i* is an example of this. A business card photo is taken using a cellphone's camera and the business card's owner information like his or her name, telephone, company name, etc, is stored in a database on the device.

2.4.2 Recent solutions

Nowadays there are recent solutions available that can be used with satisfactory results depending on what's needed. Bellow, some of the most known solutions are presented.

ABBYY FineReader

License: Comercial

FineReader is a DAR and OCR software developed by the Russian company *ABBYY*[10]. It's first version was released in 1993 [15] and it can perform automatic DAR and OCR as well as manual edition/correction of the results. To do this, it features a graphical user interface (GUI) and its latest version (version 9.0 Professional Edition) is available only for Microsoft Windows.

Nuance OmniPage

License: Commercial

OmniPage is also a DAR and OCR software. It is developed by *Nuance Communications*[11]. Like FineReader, OmniPage has a GUI where it's possible to perform automatic DAR and OCR as well as manual corrections and edition. Its latest version (Professional 16) is available only for Microsoft Windows as well.

SimpleOCR

License: Freeware

[10] `http://www.abbyy.com`
[11] `http://www.nuance.com`

This program is developed by *Cyril Cambien* as an OCR solution but also a royalty-free OCR SDK[12] that can be used in other applications. It does not perform DAR and its GUI allows one only to manually select the parts of the image that the engine is supposed to process. It is available only for Microsoft Windows.

Vividata OCR Shop XTR

License: Commercial

The *OCR Shop XTR* DAR and OCR solution was released in 2003 by *Vividata*[13]. It is available only for Linux and UNIX and can be only used from the command line.

OCRopus

License: Apache 2.0 License[14]

OCRopus is different than the previously commercial solutions. It is supported by Google and leaded by Thomas Breuel from *German Research Centre for Artificial Intelligence*[15]. It performs DAR and OCR, the latter is accomplished by using Tesseract. The project is highly modular to be pluggable with more OCR engines and DAR systems.

It does not have a GUI, instead it can be used from the command line. Another particularity is that instead of exporting the recognized data to a widely used document format like PDF or Doc, it generates *hOCR*[16] files which are HTML files with embedded OCR information.

It is officially developed for Linux although there are efforts to make it usable on Microsoft Windows and Mac OS.

---

[12] *Software Development Kit*
[13] `http://www.vividata.com`
[14] `http://www.apache.org/licenses/LICENSE-2.0`
[15] `http://www.dfki.de`
[16] `http://code.google.com/p/hocr-tools/`

2.5 Conferences about DAR and OCR

Along with the years of research in the field of DAR and OCR, several important workshops and conferences were created. Probably two of the most famous conferences are the *International Workshop on Frontiers in Handwriting Recognition* (IWFHR) and *International Conference on Document Analysis and Recognition* (ICDAR).

Chapter 3

The System

3.1 Overview

OCRFeeder was designed to be used in two ways: as a command line tool and as a full application with a complete graphical user interface.

Since this project is Free and Open Source Software, there was the concern of using open source technology and open standards. All the used technology is Free Software. Although the system was designed and implemented thinking on its usage on the GNU/Linux operating system, most of the technology used is fully supported on other operating systems. Also, the implementation, that is, the code, was written in an independent way that will not require major changes if it happens to be ported to other operating systems in the future.

3.2 Technology and Development Tools

This section presents a list of the technology and tools chosen for the development of this project. It gives an overview of each of the technologies/tools without going into very technical details, there will be also an explanation on why the technologies were chosen.

3.2.1 Python

Python[1] is a general-purpose object-orientated programming language. It uses garbage-collection to manage the memory and is both dynamically type checked and strongly typed. Python focus on the readability and clearness of the code as well as in the programmer's productivity. Its strong introspection capabilities make it very fast for a programmer to automate tasks that usually would require a larger sum of code lines. It also features an extensive and useful standard library for which the Python community often uses the phrase "*batteries included*".

Python was created in 1991 by Guido van Rossum at the *Stichting Mathematisch Centrum* in Amsterdam and its name comes from *Monty Python's Flying Circus* -

[1] `http://www.python.org`

the BBC comedy series of which Guido is a big fan [16]. It was originally designed as a scripting language for the Amoeba system in which Guido was involved.

Its first version was released in January 1994 and in 1995, Guido continued his work at *Corporation for National Research Initiatives* in Verginia, USA, releasing several versions of Python. In 2001, the non-profit organization *Python Software Foundation*[2] was founded and have been managing the open source licensing of Python since version 2.1 [17].

Modularity and reuse should always be key concepts when it comes to write code that may be then adapted by others. The below excerpt from [16] represents how Python suits these needs:
"*Besides being well designed, Python is also well tooled for modern software methodologies such as structured, modular, and object-oriented design, which allow code to be written once and reused many times. In fact, due to the inherent power and flexibility of the language, writing high-quality Python components that may be applied in multiple contexts is almost automatic.*"

Python also has excellent portability and the Python interpreter is available for a large set of platforms, from major operating systems like GNU/Linux, Mac and Windows to cellphones or even the .NET platform and the JAVA Virtual Machine.

Documentation also plays a very important role in every project for purposes of understanding what the code does and how it does it. Python features what's called *docstrings* which standardize the way to write documentation in Python. Docstrings are simply strings with the actual documentation text and go under the declaration of classes, methods, etc.

The properties of Python make it very suitable for several areas of programming like system programming, rapid prototyping, text processing, graphical user interfaces (GUI) programming or web programming.

In this project, Python was chosen because it filled every requirement. It suited every mandatory need like processing text, creating the GUI, abstracting certain

[2] *Python Software Foundation*: `http://www.python.org/psf/about/`

tasks and others.

3.2.2 PyGTK

GTK+[3] is a toolkit to create multiplatform graphical user interfaces and was created by Peter Mattis, Spencer Kimball and Josh MacDonald in 1997. GTK stands for *"The GIMP Toolkit"* since it was originally developed for the *GNU Image Manipulation Program*[4] (GIMP).

Although it was originally created for *X Windows* it is available for the most common operating systems like GNU/Linux, Mac and Windows and was adopted as the default graphical toolkit of *GNOME*[5] and *XFCE*[6]. Beyond all the common graphical interface components it supports, the main features of GTK+ include theme support, thread safe, localization and internationalization among many others [18].

PyGTK[7] is a set of Python wrappers for GTK+ that make it possible to create GUI applications with all the GTK+ advantages.

Even though Python already includes a toolkit (*Tk*[8]) in its standard library, PyGTK was the chosen toolkit for OCRFeeder because of all its advantages comparing to Tk, mainly accessibility, localization, internationalization and the object oriented approach.

3.2.3 PIL

The *Python Image Library*[9] (PIL) is a powerful framework created by Fredrik Lundh that allows operations on images like creation, manipulation or conversion [19].

[3] *GTK+*: `http://www.gtk.org`
[4] *GIMP*: `http://www.gimp.org`
[5] *GNOME*: `http://www.gnome.org`
[6] *XFCE*: `http://www.xfce.org`
[7] *PyGTK*: `http://www.pygtk.org`
[8] *Tk*: `http://www.tcl.tk/`
[9] *Python Image Library*: `http://www.pythonware.com/products/pil`

OCRFeeder extensively uses PIL for all the advanced operations on images mentioned in Chapter 4. It was chosen because it is the most complete and powerful imaging library available for Python.

3.2.4 PyGoocanvas

GooCanvas[10] is an advanced canvas widget for GTK+. It uses the *cairo*[11] 2D library that offers a powerful API for 2D drawing and vector graphics operations. GooCanvas make it easy to control the vector elements that cairo provides for example it allows to create geometrical forms, control their properties or check which other forms are contained within given bounds.

PyGoocanvas[12] is a Python wrapper for Goocanvas. It allows to use Goocanvas powerful capabilities but using Python instead of C.

PyGoocanvas was chosen as it provides the needed tools to be able to do a very important part of the graphical user interface, like Section 5.2.2 describes.

3.2.5 XML

XML stands for eXtended Markup Language and is an open standard recommended by the *W3C*[13] for document markup.

According to [20]:
"*It defines a generic syntax used to mark up data with simple human-readable tags. It provides a standard format for computer documents that is flexible enough to be customized for domains as diverse as web sites, electronic data interchange, vector graphics, genealogy, real estate listings, object serialization, remote procedure calls, voice mail systems, and more.*"

[10] *GooCanvas*: `http://live.gnome.org/GooCanvas`
[11] *Cairo*: `http://cairographics.org`
[12] *PyGoocanvas*: `http://developer.berlios.de/projects/pygoocanvas`
[13] *World Wide Web Consortium*: `http://www.w3.org`

XML documents that should follow a certain schema can be validated according to an XML schema rules. For example, an XML schema may define that an element "person" must contain one (and only one) attribute called "name" but must not contain an attribute "manufacturer" which belongs to an element "car". If an XML document must obey an certain schema, then it must include information on where to find the schema. Documents that validate their schema are said to be well-formed.

In this project, XML is used for:

- Storing the general preferences configurations;
- Storing the settings when saving a project;
- Specifying OCR engines' settings;

The XML usage in OCRFeeder is covered in Sections 4.2.1 and 5.3.4. It was chosen because of being an organized and standard way of representing and storing data.

3.2.6 ODT

OpenDocument Text (ODT) is the format referring to text in the *OpenDocument Format* (ODF).

The OpenDocument Format is a file format based in XML for office documents like text, presentations, spreadsheets and graphics.

ODF was first started in 1999 by *StarDivision* for its office suite *StarOffice* and *Sun Microsystems*[14] then acquired the company the in the same year. In 2000, the open specification of ODF started as Sun Microsystems released *OpenOffice.org* [21] – a complete and cross-platform office suite derived from StarOffice and compatible with *Microsoft Office* formats.

[14]`http://www.sun.com`

ODF got approved as an *OASIS*[15] standard in May 2005 and in 2006 in the same month, *ISO*[16] and *IEC*[17] unanimously approved it as ISO/IEC 26300 [22].

Being an open standard by OASIS and vendor independent, ODF allows the creation of new solutions independent from office applications. It is free of licensing, royalty payments or other restrictions.

Many countries and institutions have been supporting and adopting ODF.

After OpenOffice.org, many other programs began supporting ODF. Some widely used programs that are an example of supporting ODF, either fully or partially, are:

- *Abiword*[18];
- *Google Docs*[19];
- *IBM Lotus Symphony*[20];
- *KOffice*[21].

For ODF openness and recognition as an ISO and OASIS standard together with its portability and support by many applications, ODF was chosen as the primary exportation format of OCRFeeder.

Section 4.4.1 describes how the ODT exportation is accomplished.

3.2.7 ODFPy

From the *OpenDocument Fellowship*[22] [23]:
"*Odfpy aims to be a complete API for OpenDocument in Python.*"

[15] *Organization for the Advancement of Structured Information Standards*: `http://www.oasis-open.org/`
[16] *International Organization for Standardization*: `http://www.iso.org/`
[17] *International Electrotechnical Committee* `http://www.iec.ch/`
[18] `http://www.abisource.com`
[19] `http://docs.google.com`
[20] `http://symphony.lotus.com`
[21] `http://www.koffice.org`
[22] *OpenDocument Fellowship*: `http://opendocumentfellowship.com`

Hence, ODFPy offers a way to generate an ODF document using Python. It is a complete API as it lies just above XML in what comes to abstraction, this way it makes possible to control all ODF constructions.

ODFPy produces valid documents as it raises exceptions when an invalid action (according to the XML schema) in the document generation occurs.

ODFPy was chosen as the way to generate ODT documents for this project due to the easiness of generating valid the documents over the more "manual" way of doing it with a plain XML parser.

3.2.8 Ghostscript

Ghostscript[23] is an interpreter for *PostScript* and *Portable Document Format* (PDF) created by L. Peter Deutsch [24]. It provides a set of tools that allows for operations such as viewing or converting the mentioned file formats and have been ported to the many operating systems like GNU/Linux, Mac and Windows.

In this project, Ghostscript was chosen to convert PDF documents to images in the PDF importation functionality (see Section 5.3.2).

3.2.9 Unpaper

Unpaper[24] is a tool developed by Jens Gulden to perform corrections on images originated from scanned paper sheets. It is extremely useful as a pre-processing tool for an OCR engine since it clears the image by removing the "dust" and wipes out other marks like, for example, dark edges usually created from a photocopy machine. It can also rotate a text image to the correct angle.

This tool is used in this project as an optional plugin to clean images. The use of Unpaper is presented on Section 3.2.9.

[23] *Ghostscript*: `http://ghostscript.com/awki`
[24] *Unpaper*: `http://unpaper.berlios.de`

3.3 Architecture

Modularity was a concept always present when designing the system due to all its known advantages. Figure 3.1 shows the system architecture in a global way for an easy understanding of it.

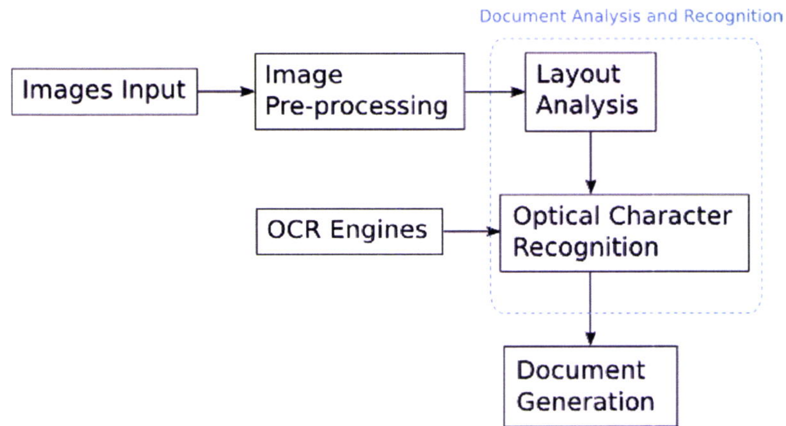


Figure 3.1: The system global architecture

A more detailed yet simple diagram is shown in Figure 3.2. It outlines the main modules and actions that occur on them, the next chapter will cover each of the modules and actions in a more detailed and technical way.

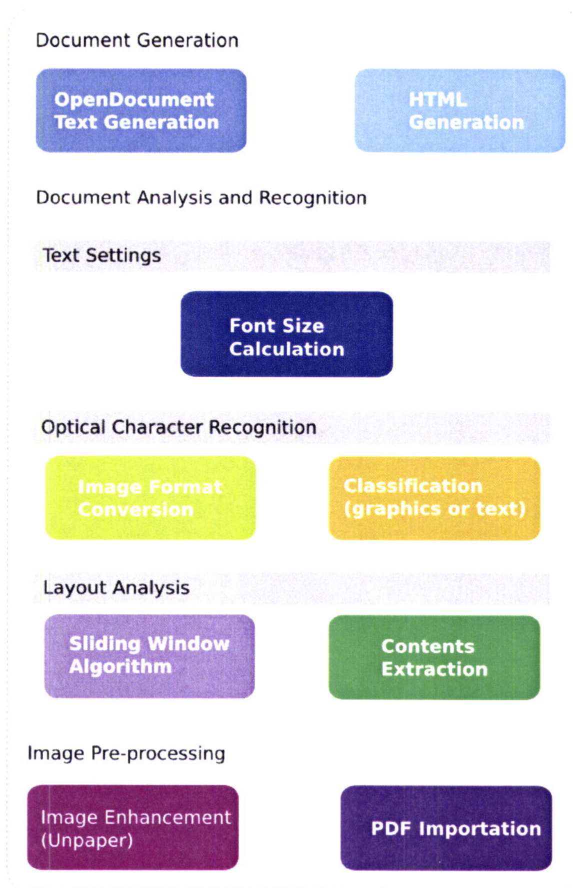


Figure 3.2: A more detailed architecture diagram

Chapter 4

Implementation

OCRFeeder is composed by two parts or particularly, two interfaces. These are the command line interface (CLI) and the graphical user interface (GUI). Although there are two different interfaces, the basis is the same.

For a printed document to be converted to an electronic format, simple OCR tools are not enough. Most OCR engines available only perform OCR over the text, returning the text it recognizes from the given document image. The output does not contain any information about the layout, it doesn't include any reference whether the document followed a two-column format, three-column, had additional information boxes, etc. So, there is the need to include layout analysis when using such OCR engines in order to produce a fair electronic version of the input document image.

Nine Inch Nails - Discipline

Am I still tough enough?
Feels like i'm wearing down
Is my visciousness
Losing ground?
Am I taking too much?
Did I cross a line?
I need my role in this
Very clearly defined

I need your discipline
I need your help
I need your discipline
You know once I start I cannot help myself

And now it's starting up
Feels like I'm losing touch
Nothing matters to me
Nothing matters as much
I see you left a mark
Up and down my skin
I don't know where I end
And where you begin

I need your discipline
I need your help
I need your discipline
You know once I start I cannot help myself

Once I start I cannot stop myself

Figure 4.1: Lyrics document image (preview of contents area)

To give a visual example, consider the clean document image shown in Figure 4.1 with the lyrics of a song[1]. If processed by the OCR engine Tesseract, for example:

```
$ tesseract nin_discipline.tiff lyrics_text
```

The result is a file[2] named *lyrics_text.txt* with the text that Tesseract. The generated text is illustrated in Figure 4.2.

```
Nine Inch Nails - Discipline
Am I still tough enough?                      And now it's starting up
Feels like iam Wearing down                    Feels like I'm losing touch
Is my visciousness                             Nothing matters to me
Losing ground'?                               Nothing matters as much
Am I taking too much'?                         I see you left a mark
Did I cross a line?                          Up and down my skin
I need my role in this                           I don't know where I end
Very clearly defined                             And Where you begin
I need your discipline                            I need your discipline
I need your help                                  I need your help
I need your discipline                           I need your discipline
You know once I start I cannot help myself      You know once I start I cannot help myself
Once I start I cannot stop myself
```

Figure 4.2: Recognized text for lyrics

As it shows, the text is recognized with a great success rate but all the format structure is lost. Tesseract and most engines alike recognize the space between the columns jumps as simple spaces between one word and another. Paragraph information is also lost, the song's chorus does not preserve the space which divides it from the rest of the verses. The title doesn't show up aligned or separated from the song's lyrics as well.

[1]The song is *Discipline* by *Nine Inch Nails*, licensed under Creative Commons Attribution-Noncommercial-Share Alike 3.0

[2]Actually it generates two more files with the extensions *.map* and *.raw* but have no interest for this example.

From the above example, it is easy to see that using just an OCR engine is not enough to produce an electronic version of a document image. It would work for example if the objective was just extracting the words from the image for indexing purposes.

4.1 Layout Analysis

To preserve the document's original structure, layout analysis must be employed in the process of converting the document to an electronic version. Using the previous example of the lyrics document, if a person was asked to outline the format structure in the document, the result would be something like the Figure 4.3.

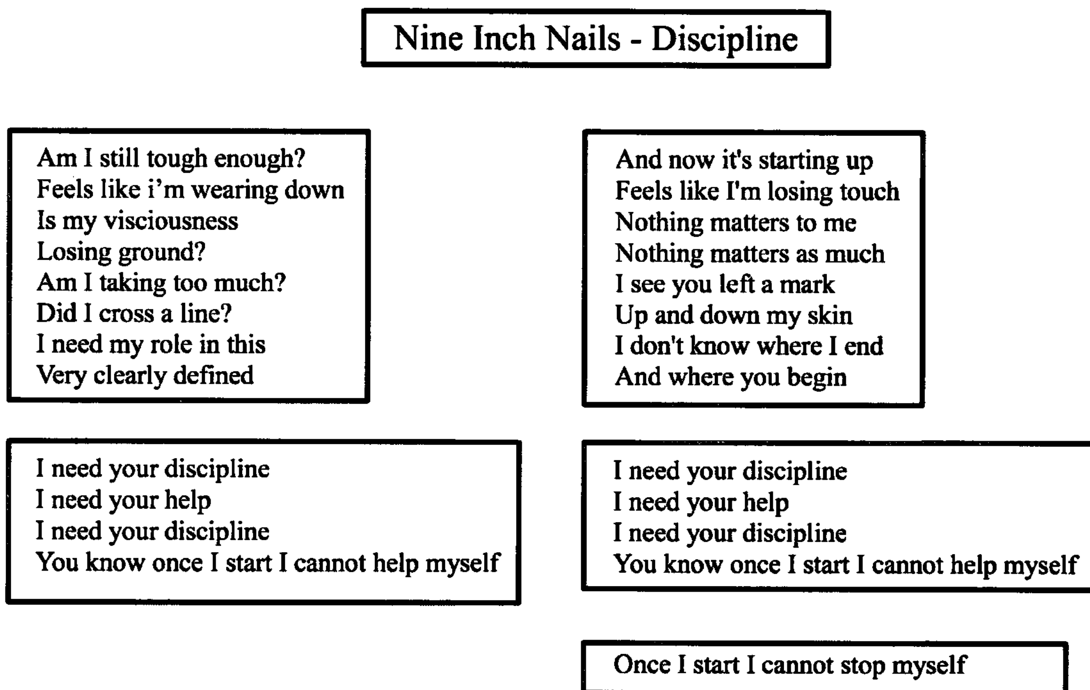


Figure 4.3: Lyrics document with outlined format structure

To automatically find the regions of interest in a document is a challenging problem. Raw images like the JPEG, PNG, PNM, or TIFF do not hold any useful information abouts its contents when it comes to document images. They don't keep information whether the graphics they hold represent a flower, a dog, a house, a taxes form or a restaurant menu. For example, there is nothing present in the JPEG format that tells whether it starts with a paragraph of text or has a picture starting from the middle of the page until the end.

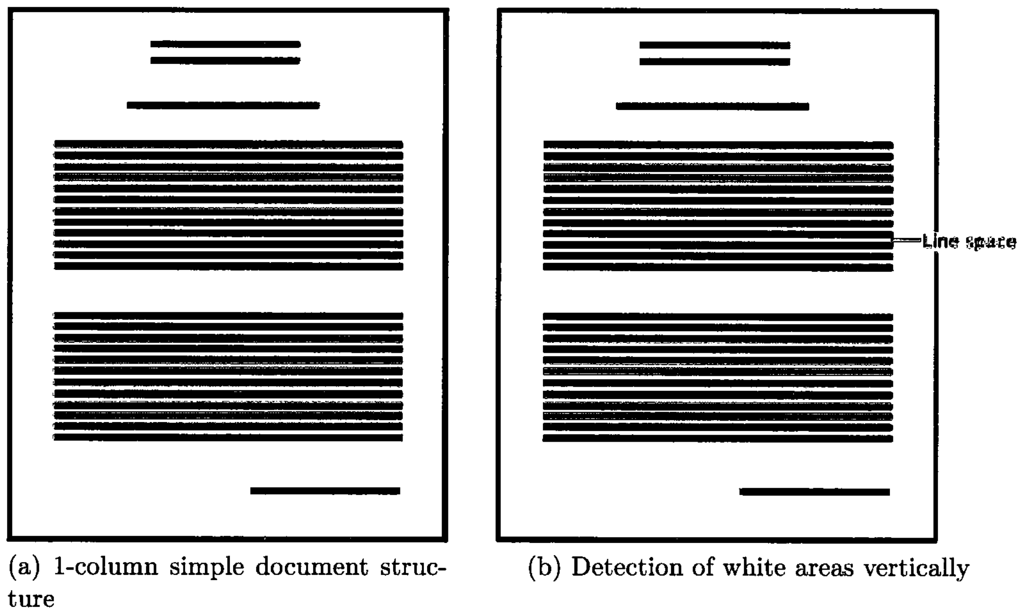
With a way to know the document's structure it would be easier to perform OCR on each retrieved "piece" of the document and finally generate a version of it in an editable electronic format.

For the purpose of simplifying, this section presents some figures that show examples of possible document structures by presenting text as black lines instead of real text, such as Figure 4.4.

If it was just dealing with simple documents with one column of text, the solution was easier. For an example of this, consider Figure 4.4a. The layout format is pretty simple and to retrieve the contents is rather easy. It may be accomplished by going from top to bottom and tracking the white areas of width equal to the image width and of height slightly greater than a pre-defined text line spacing. The result of this step is illustrated in Figure 4.4b where the light blue rectangles mark the white areas found. After the white areas are tracked, it is simple to get the actual text areas. To find the horizontal beginning of the text in the retrieved text areas, the white areas of it are also tracked in a similar way as before but now horizontally.

The previously suggested algorithm is somewhat trivial and not enough for documents that don't follow such simple structures because this algorithm only considers the documents to be structured vertically. Figure 4.5 gives an example of a "not so simple" structure. In this case, the document starts exactly as the previous mentioned ones to be then split in two columns with the left one representing itself two paragraphs.

Obviously, the simple algorithm used before doesn't work for documents like



(a) 1-column simple document structure

(b) Detection of white areas vertically

Figure 4.4: Illustration of part of the detection algorithm for 1-column simple documents

this. For the document illustrated in Figure 4.5, when the algorithm reaches the white space in the middle of the left column separating the paragraphs, it will still detect the right column's text and so, it assumes there is still text. A more flexible approach is needed.

Instead of targeting a specific type of document, the approach taken in this project was that it must be usable for virtually every document with any structure. If the goal was converting scientific papers for example, then an approach similar to those already mentioned in Chapter 2 involving structure patterns would be the right way to go.

By targeting any time of document it means that the document can start for example with a logotype, then the title, then text; or first the text in a 2-column fashion and then a picture followed by its legend; or maybe even just a picture

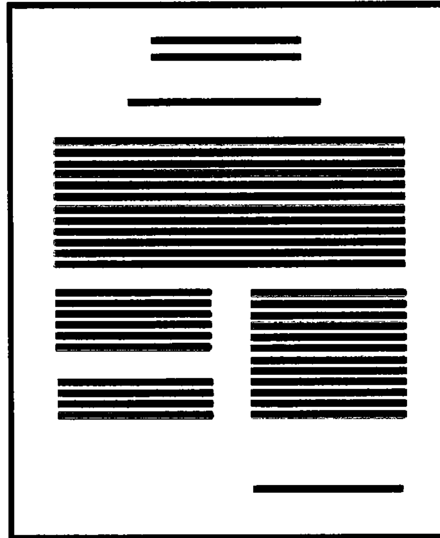


Figure 4.5: A not so simple document example

occupying the whole document area and nothing else. Combinations are endless. So, this project accomplishes the task of retrieving the contents from a document image by following a simple principle:

It doesn't matter what structure is being analyzed, it only matters to retrieve the contents.

Contents are any picture, any column, any paragraph which is part of a column, any image caption, etc. However, creating a way to retrieve the contents that must be suitable for many document structures isn't a trivial task.

4.1.1 The sliding window algorithm

This section presents the actual algorithm created for getting the contents from an image. For consistency, the parts that need to be retrieved will be mentioned mainly as contents.

If one looks randomly at a small part of a document image, that small part will be either *content* or not; foreground or background; a region of interest or not. This means that there is a binary condition – there is *one thing* or *nothing* – and so, it means that for any part of the document, it can be categorized as *1* or *0*.

This way, the theory behind the algorithm is a pretty simple concept:

If a document can be divided in several small pieces and each piece categorized as 1 (if it is foreground) or 0 (if it is background), then it is possible to group blocks of 1s and hence, outline the image contents.

The algorithm operates as follows:

1. A *NxN* pixel window runs trough the document from left to right, top to bottom;

2. For every iteration, if there is at least one pixel within the window whose color contrasts with the background, then it is assigned the value 1 otherwise it is assigned the value 0;

3. After all windows are assigned a value, the ones who have been assigned the value 1 are grouped;

4. Every time a group of 1s is gathered, every window in the group is reassigned the value 0;

5. When the all windows have the value 0, the algorithm is finished.

Because the basis of this algorithm is a window "sliding" through the image, the algorithm was named the *Sliding Window Algorithm.*

The window size

The size of the window is *NxN* pixels. Hence, *N* needs to be set before the algorithm starts. The problem is that document images may vary in size and so, choosing a small size for the window might result in the window fitting within the line spacing. On the other hand, a big window size might result in a window

bigger than the space separating two paragraphs and so, considering them to be part of the same content. For this, the window size needs to be carefully chosen and ideally bigger than the line spacing, and smaller than the paragraph spacing.

OCRFeeder calculates the window size automatically. This calculation is as follows:

$$N = H/60, N \in \mathbb{R}, H \in \mathbb{N}$$

Where N is the window size and H is the document image's height. The value 60 was considered to be the best one after testing several values.

Obviously, this is not flawless. It depends if the line and paragraph spacings used in the document are "standard" and consistent. For example, a paragraph spacing slightly greater than the line spacing might be enough for a human to detect the structural separation of contents but small enough to fit within the window automatically calculated.

Due to this problem, OCRFeeder lets the user decide whether to automatically calculate the window size or to define it manually.

Binary Representation

Figure 4.6 represents a document divided in windows of size automatically calculated. The figure is 450 pixel high and so, dividing its height by 60 will result in 7.5. The resulting window size is correct because it doesn't fit inside the line spacing but fits inside the paragraph spacing. Although the number of pixels must be an integer value, the window size is not rounded at this time because further arithmetic operations will be performed with this value. This way it is more accurate to do the operations with the value not rounded. Figure 4.6 also shows the windows with the respective values already assigned.

The way to see if a pixel belongs to the images' contents or to the background is to check if its color contrasts with the background color.

Color contrast needs to exist between the contents and the background of a document. If the document's background color is white and the text written on it is light yellow it is difficult for anyone to read it. According to [25]:

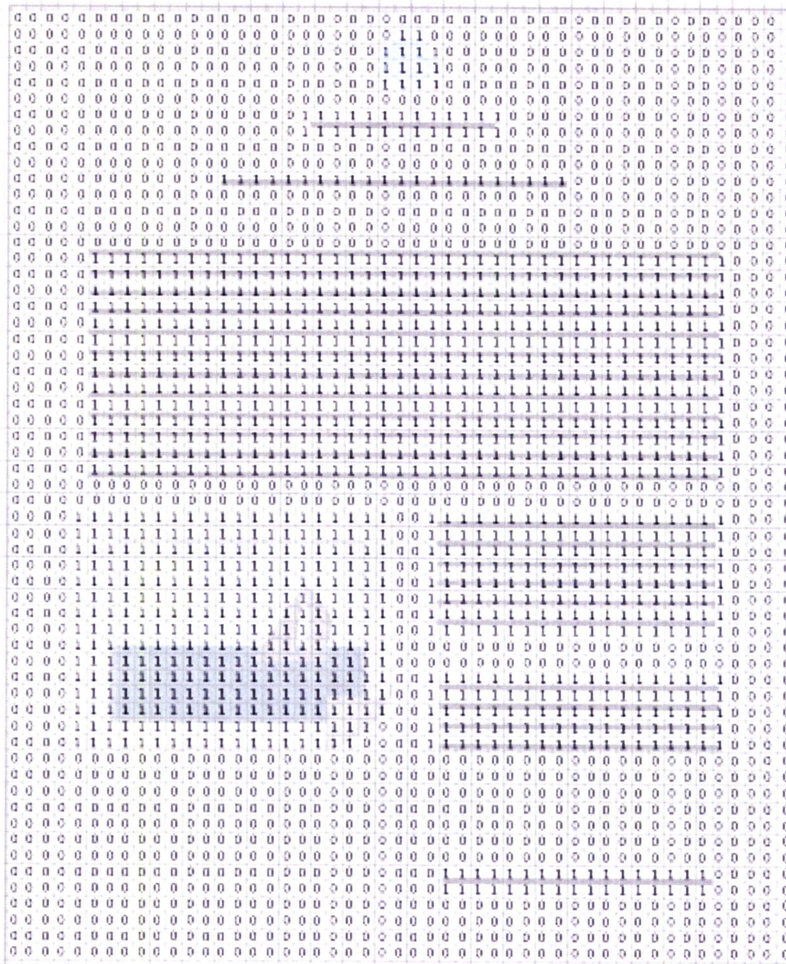


Figure 4.6: Binary representation

"[...] object detection depends in general on the color difference of the object from the background."

The document images are converted to grayscale before being analyzed. This is done in order to simplify the calculation of the contrast between the different elements in the document image. Grayscale images are simpler for this purpose because they have only one channel (gray) per pixel instead of the three channels of RGB images. By observation, the default value that OCRFeeder considers to be the minimum distance the colors of two pixels is 120. Figure 4.7 shows an example of two contrasting color with the difference of 120, the background color has the value 240 and the inner square has the value 120.

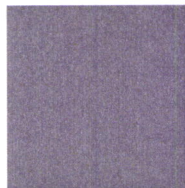


Figure 4.7: Example of contrasting colors

Thus, color contrast can be calculated as the absolute value of the difference between two colors as follows:

$$D = |A - B|, A, B \in \mathbb{N}$$

Where A and B are the two colors to be evaluated and D is the resulting difference. If D is greater than or equal to 120, the window gets assigned the value

1, otherwise it gets a 0.

Optimization

At first, from left to right, top to bottom, all pixels inside a window were being checked for contrast. Since the algorithm only needs to find one pixel that contrasts with the background, the polynomial time for this, considering a 16x16 pixel window is, at most, O(256). This approach turned out to be considerably slow, even when ran on a modern machine.

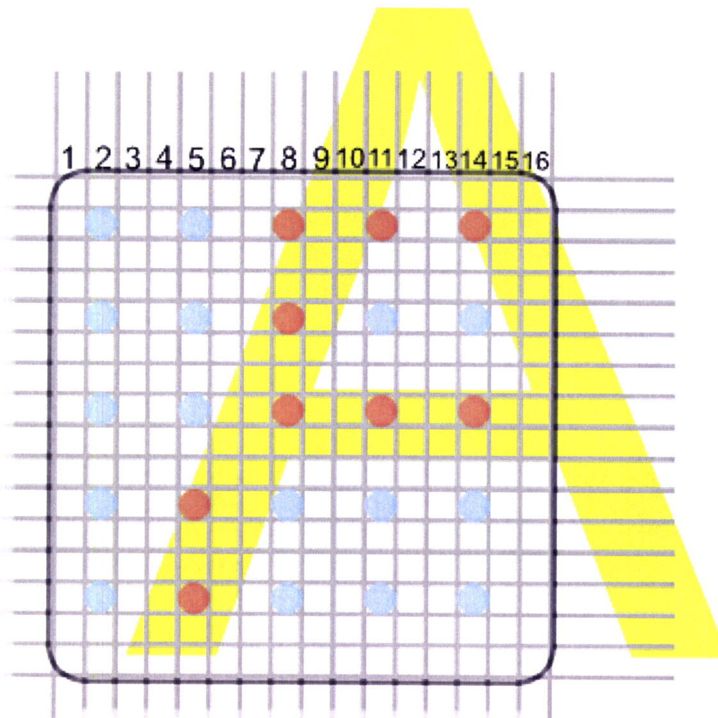


Figure 4.8: Optimization of the function to find contrast within a window

An optimization was needed so the performance was better. Since even a small picture or a small font size are not likely to occupy only a single pixel, there is no need to run through all the pixels in a window. Instead, the algorithm checks the

pixels within an interval of two other pixels. Figure 4.8 depicts this optimization, it shows a window of 16x16 pixels that partially overlays the character A. Each square is a pixel, the blue dots represent the pixels being checked, the orange dots represent the ones whose color contrasts with the background.

The resulting polynomial time for the same example is now at most O(25) (the algorithm stops at the topmost pixel in column 8) because only 25 pixels out of the 256 are checked. This is clearly a better approach than the previous one.

In the end, a string list with "0" and "1" characters is returned which was named as the *Binary Representation List* and will make it easier to process and interpret the contents in the image.

This concludes the explanation of the Sliding Window Algorithm. The next subsection will explain how the groups of 1s in the Binary Representation are joint.

4.1.2 Retrieving blocks

After having the Binary Representation List (BRL), its information must be processed in order for it to represent the actual contents in the original document image. The 1s present in the list must be gathered in groups, this groups were named blocks. Blocks act as bounding boxes that can be mapped in the image and will contain its contents.

A block was defined as having five properties. These properties are:

- *Start line*: the line in the BRL where the block starts (the topmost 1). The top edge of the block.
- *Finish line*: the line in the BRL where the block ends (the bottommost 1). The bottom edge of the block.
- *First one*: the column in the BRL where the block starts (the leftmost 1). The left edge of the block.
- *Last one*: the column in the BRL where the block starts (the rightmost 1). The right edge of the block.

- *Extra charge*: extra adjustment to include half a window beyond its start or finish line.

Figure 4.9 shows a block in a BRL. The procedure to retrieve the block information is as follows:

1. Find the first not blank line (that has at least a 1), this will be the block's *start line*;
2. In that line, find the index of the first 1, this becomes the *first one*;
3. Go to the next line and check the index of the first 1:
    - If the absolute value of the difference between this index and the *first one* is greater than *tolerance*, then the previous line becomes the *finish line* and enters step 4;
    - Otherwise, if the new index is less than the *first one*, it becomes the *first one* and repeats this step;
4. Store the block's information and replace the block's belonging 1s by 0s in the BRL and restart from step 1 until the BRL has only 0s.

The property *extra charge* is explained ahead.

The *tolerance* is a predefined value that OCRFeeder defaults to 3. It means, that a line only belongs to the block being currently created if the first 1 is not more than two characters away from the block's current *first one*.

After these three properties are found, the *last one* still needs to be found. This is done by checking the first columns of zeros within the already found *start line* and *finish line* and going right from the *first one*. The value of the *last one* will be the index of the last 1 found in the block.

For a better understanding, here's the algorithm applied to the example shown in Figure 4.9 (all indexes considered start from 0).
The first line is blank and so, the next one is tested:
0000111111111111111111111000
the index of the first 1 found is 4, so the *first one* is set as 4. Next line's first 1

0 00000000000000000000000000000
1 00001111111111111111111111000
2 00011111111111111111111111000
3 00000111111111111111111100000
4 00011111111111111111111111100
5 00011111111111111111111110000
6 00111111111111111111111111000
7 00011111111111111111111111000
8 00000000001000000010001000000
9 00000000000000000000000000000

Figure 4.9: A block in a BRL

index is 3 so, it's less than 4 and $|3-4| < 3$, the *first one* becomes 3:
00011111111111111111111111000
then, the first 1 has the index 5 which doesn't modify the current properties:
00000111111111111111111100000
Lines 4 and 5 have their first 1 at the same index of 3, which is equal to the current *first one* and so, everything stays the same. Moving on to line number 6, the index of the first 1 is 2 which, according to the rules, becomes the new *first one*:
00111111111111111111111111000
Line number 7 also leaves the properties as they are but the index of the first 1 in line number 8 is 10. Since $|2-10| = 8$, is not interpreted as being part of the block and so, the previous line (number 7) becomes the *finish line*.

At last, all columns from the value of the *first one* and delimited by *start line* and *finish line* are checked until the first blank one is found. This occurs at column 26, so the *last one* is 25.

At this point, the block already has the main four properties and so, it can already define a bounding box. Every block built is appended to a list containing

all blocks retrieved so far. All "1" characters within the block's area in the BRL are then replaced by "0". This is obviously done because the algorithm for retrieving the blocks goes from top to bottom, left to right (starting from the upper left corner) and restarts every time a block is retrieved. If that block's belonging 1s were not replaced by 0s, it would find the same block again.

The extraction of blocks from the BRL is just one part of creating the blocks. After having the list of all blocks, operations will be performed over it so all blocks get the right interpretation.

Extra charge

Characters aren't all the same height and this reflects in the BRL. In typography, characters have identified distances and lines such as the baseline, ascent, cap-height or x-height [26] like shown in Figure 4.10.

Figure 4.10: Distances and lines in typography

This will result in something like the line 8 in Figure 4.9 because a line of windows may have its lower edge ending at the baseline of a paragraph's last line that may have characters like "p", "j", "ç" and thus causing the next line to have a smaller number of 1s. It is also truth for characters in the paragraph's first line that may pass the x-height like, for example, "t", "l", uppercase characters and characters with diacritics like "á", "ü", etc.

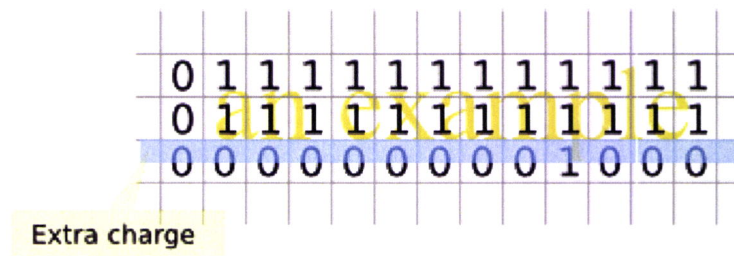


Figure 4.11: An example of extra charge

Figure 4.11 shows an example of the need of *extra charge*. The character "p" in the word "example" passes the bottom line and stays in a line that is not considered as part of the block – the first 1 is too far from the block's *first one*. To solve this, *extra charge* is set so this block will include half of the line under the *finish line* – like shown by the light blue rectangle. In this example the window is too small which makes the part of the character "p" occupy the whole *extra charge* height, normally, characters' parts do not occupy all that height.

So, the need of *extra charge* depends on the window size, font size, space from the top where the paragraph starts, on the font family (fonts descents and diatrics dimensions vary), etc.

Extra charge can be set with constant values that are interpreted as follows:

- *TOP*: Takes the half of the line after the block's *finish line*;
- *BOTTOM*: Takes the half of the line before the block's *first line*;
- *BOTH*: Takes both the half of the line before the block's *first line* and the line after the block's *finish line*;
- *NONE*: Does not take anything.

With the definition of this special property, the block structure gets an extra flexibility. Figure 4.12 shows the block presented previously in Figure 4.9 this time with a legend to help understand what was told about the block's structure. This concludes the presentation of the block.

Figure 4.12: A simple block with a legend

Operations over the blocks list

Although already presented, *extra charge* is not set for the respective block right after it is created. It is set in the first operation performed over the blocks list. This operation is performed by the function `extendBlocksByBelongingSingles`. What it does is that it runs through the list and whenever a *single block* is found, and depending on its position relatively to another block, the latter will be expanded to include the single one. A single block is a block that occupies only one line in the BRL. This can occur because in fact there are contents that occupy only one line (for example one line of text, an horizontal line separator or an image), or because of something like the case mentioned as an example for the extra charge – a line in the BRL originated by parts of characters. This function deals with the latter case.

To expand the blocks, the block class has four methods that increase the finish line, decrease the start line, extra charge the top and extra charge the bottom.

Whenever it finds a single block it gets its surrounding blocks – blocks that start one line after or finish one line before and horizontally contain the single block. For example, the ones present on the already mentioned Figures 4.12 and 4.11. The surrounding blocks are returned as the preceding and the succeeding blocks in relation to the current single block. The following list describes the interpretation of the results:

- If there is a preceding and a succeeding block then extra charge the bottom of the first and extra charge the top of the latter. After this, check if the preceding and the succeeding blocks can be joint and if so, join them.
- If there is a preceding block but no succeeding block, increase the preceding block's finish line by one and delete the single block;
- If there is a succeeding block but no preceding block, decrease the succeeding block's start line by one and delete the single block;

In the first case present in the list, two blocks can be joint if either the *first one* or the *last one* properties on both blocks have the same value and one block

vertically ends where the other starts. The function to join the two blocks sets the preceding block's *finish line* and *first one* with the value of the same properties of the succeeding block. After the blocks are joint, the succeeding block must be obviously removed from the blocks list.

Because several blocks may be joint, the function first performs a cleaning action by removing any blocks from the blocks list that end up contained in other blocks.

Whenever an extension or unification occurs, the function's main loop starts from the beginning because these actions originate new blocks that might need to be extended themselves.

Once the extension of blocks is done. The function `unifyBlocks` will unify any blocks that need to be joint. So, the functions first checks each block getting its surrounding blocks and joining them with the block if they can be joint. Like the previous function, this needs to start all over again whenever an unification occurs.

When there are no blocks left to be joint, for each block in the list, the function gets the blocks overlapped by it and joins them, deleting the block that was overlapped. Again, when two of these blocks are joint, the blocks list is checked again from the beginning.

Every content in the document image is now represented by a block. The block class has a method called `translateToUnits` that given the window size, returns the upper left and lower right corners, representing the block's bounds, in pixels. This is done by multiplying the block's main properties – start and finish lines, first and last ones – by the window size. Like mentioned before, the extra charge takes half of the window size.

With the actual size of each block bounds in pixels, it is possible to clip each block's area from the original document image and then perform other operations to retrieve more information about the image clip. This finishes the Layout Analysis section.

4.2 Recognition

This project does not try to classify each retrieved part of the document with its logical role on it, that is, it does not try to classify contents as being the abstract, left column, logotype, etc., instead, it only classifies the retrieved parts as containing either *text* or *graphics*. OCR engines play an important role in this task and so, they're usage/configuration together with the classification of contents is explained in this section.

4.2.1 OCR Engines

Like mentioned before, this project does not supply any OCR engine. The idea is to use any OCR engine that is installed in the system and can be used from the command line.

Configuration

The engines configuration should be independent from the source code, that is, the configuration should involve no changes in the source code. For this, the properties that are present in most OCR engines were enumerated and can be configured using XML files. These XML files must reside in the **engines** folder under the configurations folder of OCRFeeder – **.ocrfeeder** – within the user's home in the system. Considering the Tesseract OCR engine, the XML files should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<engine>
    <name>tesseract</name>
    <image_format>TIFF</image_format>
    <engine_path>/usr/bin/tesseract</engine_path>
    <arguments>$IMAGE $FILE; cat $FILE.txt;</arguments>
    <failure_string/>
</engine>
```

The elements under the document element *engine* are the most general properties that can be found in the engines based on what was observed and the

Open Source engines tested. The *name* element contains the engines descriptive name. All the engines tested had a defined input image format so, the element *image_format* defines this format that must be the image format's common extension name, for example *PBM*, *TIFF*, *JPEG*, etc.

The path to the engine's executable in the system must be also provided by using the element *engine_path*.

In most cases, the engines usage need arguments and this can be specified by the *arguments* element. Configuration arguments can be used as well as shell script code but usually, only one argument is needed – the path to the image to be processed. Since the purpose of XML here is to abstract the engines configuration and the image path that the engines must process is something that varies, two special variable names can be used inside the arguments element, those are *$IMAGE* and *$FILE*. The *$IMAGE* variable will be replaced by the path to the image that needs to be processed; *$FILE* will be replaced by a temporary file name.

In this project the text recognized by the engines must be returned to the standard output like most engines do and so, usually only the *$IMAGE* variable is needed. Nevertheless, the engine used as an example above needs the *$FILE* because it does not return the text to the standard output, instead it will generate a text file with the recognized text which needs to be then returned to the standard input, hence the need the use of the command *cat*.

To end, some engines replace unrecognized characters by a character or set of characters, for example, the engines Ocrad and GOCR replace unrecognized characters with "_". This property was called failure string and can be configured using *failure_string*.

OCRFeeder sets default values to some elements if they are not used. Those are *image_format* (defaults to *PPM*) and *failure_string* (defaults to an empty string, thus it is only present in the example to show it can be used), the rest need to be included in the XML.

Recognition

The engines are represented by the class *Engine* that provides methods to use the engine. The way to use an engine is:

- Set the target image;
- Perform operation over that image.

The `read` method is the one that performs the actual optical character recognition over the given image. This method replaces the *$IMAGE* and *$FILE* – in case they are present in the arguments – with the path to the image and a generated temporary and unique file name, respectively. The generated file name will have the temporary folder path in the system has its prefix so it is created under it. The temporary folder is configured in the project preferences, explained in Section 5.3.5.

After the arguments replacement, it will run the command (the path to the engine's executable concatenated with the arguments) like if it was in a terminal and get its output text. The file and the image are then deleted from the system and the output text is encoded in the *UTF-8* character set and returned.

Using XML and using all the mentioned properties, especially the arguments allowing the engines' configurations and shell script code, make it a flexible way of defining an engine.

4.2.2 Classification

Getting the recognized text is the first step to perform the classification. The classification of the document contents as either text or graphics is based on the analysis of the output text from the used OCR engine. It was observed by analyzing an image containing graphics and no text that the resulting text from the used OCR engine was none or appeared to be jammed in the way that it had more spaces, punctuation characters and failure characters (if the engine has it).

The classification algorithm first removes the leading and trailing white spaces (new lines characters, spaces, tabulators) – this is called the stripped text in opposite to original text. The contents are classified as graphics if any of the cases in the following list apply, and as text otherwise:

1. The stripped text is an empty string – contains no characters at all;

2. The engine contains failure characters and the number of failure characters in the stripped text is more than half of its length;

3. The stripped text's length after replacing any existing spaces, punctuation characters and failure characters (in case the engine has such) is less than half the length of the original stripped text;

At this point, having the recognized text, the type of content and the dimensions and location of each of the contents, it is already possible to generate a document that would could look like the original document image. However, several other important properties – like the font face, size, alignment, etc. – would be left behind when they constitute a vital information to produce a fair conversion of the document.

4.2.3 Text properties recognition

In this project, the considered text properties were:

- Font face;
- Font size;
- Font style;
- Font weight;
- Text justification;
- Line spacing;
- Letter spacing;
- Text angle.

The font face, style, weight, letter and line spacing and the text justification are not automatically set. To find these properties is not on the objectives of this project and none of the OCR programs used was able to recognize such properties.

The rest of the properties – font size and text angle – are automatically detected and set by analyzing the image clip for the current block. The next paragraphs explain this analysis.

Font size detection

For the font size, at the beginning a simple approach was thought: the font size was approximately the size of the image clip height divided by the number of lines in the recognized text.

This turned out to be less than efficient because of two problems, 1) the spaces from the edges of the image clip to its actual text content, 2) the line spacing was not considered and thus, would influence the text size. A new approach was needed.

Thus for the new approach, the letters present in the image clip are actually measured. This is accomplished by checking the colors of each line (from top to bottom) of 1 pixel height and with the same width as the image. If the line contains any pixel whose color contrasts with the background's one, then it increases the font size; if the font size is being increased (if the previous line increased the font size) but the current line has no pixels contrasting with the background color, then the current font size is stored in a list and the next time the font size is increased it will start from 0. The background color is considered to be the most common color in the image clip – the one that most pixels have.

In the end, there will be a list with the font sizes. The arithmetic average is then calculated the values of the list; the value chosen for the font size is the one present in the list which is greater than or equal to the calculated average value.

Figure 4.13 depicts this algorithm for a better understanding. The detection starts at the vertical beginning of the text, thus the 1 pixel line starts at the top of the character "T" (in the word "This") and the font size gets the value 1, it goes down until the vertical end of the character "p" (in the word "example"); after this, the next lines have no pixels that contrast with the background color until it reaches the top of the letters of the second sentence where a new font size starts being calculated.

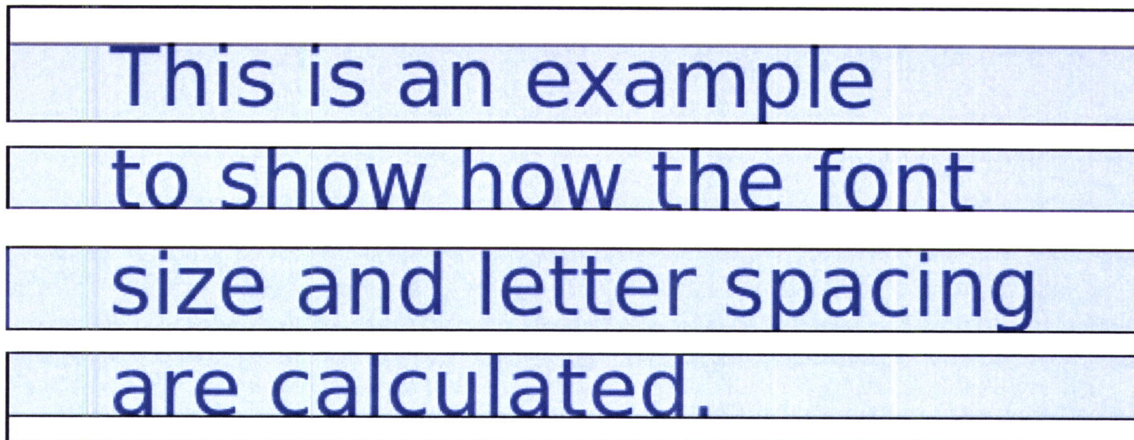


Figure 4.13: Font size and letter spacing detection

It's easy to see from the blue (font sizes) rectangles that the sizes of each text line differ and a first approach to get a balanced result was to calculate arithmetic average of the sizes in the list. However, characters like "i" or characters with diacritics can originate very small sizes in the list – that is, if there is no character higher in size than, for example, the letter "á", its diacritic will be recognized as if it was a line due to the space between it and the actual character.

As concluded in the tests (see Chapter 6), these sizes, although slightly different from paragraphs of the same original font size, are very close to the original ones and correct in most cases.

At this point, the sizes are still measured in pixels and must be converted to points. This is done by converting the pixels to inches (see Section 4.3.2 further in this chapter) and then dividing the value by 72 – since this is the value used in the *PostScript* conventions, also called *DTP point* (*DeskTop Publishing point*) [27].

Text angle detection

The used OCR engines can recognize text even if it isn't 100% horizontal. However, if the angle is not just slightly greater or less than 0, the engines cannot recognize it. So, the purpose of detecting the text angle is to rotate the image until the text appears to be horizontal, use the engine to read it and when converting it to an editable format, rotate the text to the original angle.

Consider the Figure 4.14a with the black text being the the original one. The fastest way to rotate it until it gets horizontal is to rotate it negatively (clockwise).

What's needed to be known is when to stop rotating it. The gray text was rotated a number of degrees clockwise and as the orange line shows, its upper point is lower than the original text's upper point (shown by the blue line). Continuing rotating the text will end up as shown in the Figure 4.14b where the upper point of the text is higher (shown by the red line) than the previous upper point when the text was in the horizontal. As a conclusion, the text must be rotated until the maximum distance between its upper point and the image top.

The angle chosen to rotate the text in each iteration was 5 degrees because the used OCR engines can read the text if it has a positive or negative angle of 5 and the calculation would take too much time performing unnecessary checks if the angle was 1.

However, if the text in the mentioned example was in the same angle but written from top to bottom, rotating it with this algorithm would put it upside down. Similar situations would occur if the text was written in other directions. Maybe by checking the output text of each rotated image could be a good way to see which of the rotations was correct but the problem is that performing OCR on upside down text results in some valid characters but recognized incorrectly, that is, a character "M" may be recognized as a "W" if it's upside down, or an "E" might be recognized as a "3", an "i" as a "!", etc.

So, although the rotation could be automatically detected, it would need the user to manually choose which original text direction should be assumed before the rotation starts. On top of it, rotating a text frame and placing it in the desired point of the sheet using the PyODT API turned out to be too complex and with no

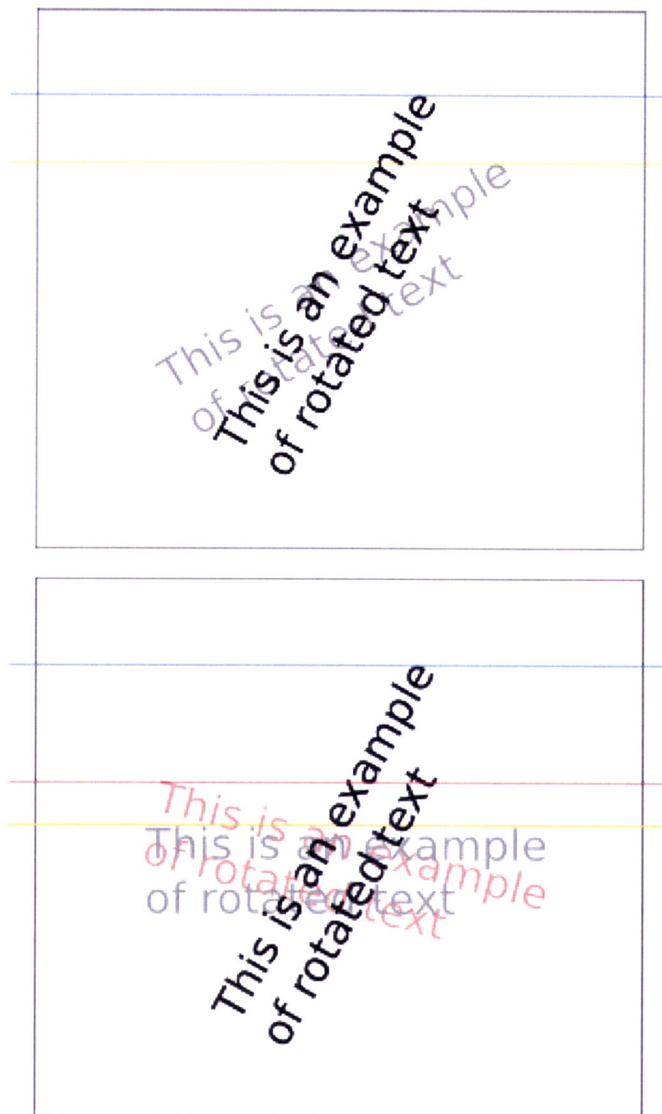


Figure 4.14: Text angle detection

satisfatory results. Plus, in what comes to other formats like HTML, text rotation is not supported. Due to all these facts, the rotation is able to be automatically detected in OCRFeeder Studio but just to demonstrate the rotation algorithm, rotation as a fully implemented feature was abandoned.

4.3 Content representation

4.3.1 Data boxes

To keep all the properties mentioned in the previous section, a more advanced structure than the blocks previously presented was needed. This new structure was called *data box*. A data box acts as an enhanced version of a block because it keeps the information about the bounds of an area like a block does but includes advanced information about the type and properties of the content it represents.

A data box contains eight properties presented in the list bellow:

- x: the horizontal distance of the box's upper left corner from the left edge of the document image;
- y: the vertical distance of the box's upper left corner from the upper edge of the document image;
- *width*: the width of the bounding box it represents;
- *height*: the height of the bounding box it represents;
- *image*: the image clip of the original document image defined by the box's area;
- *type*: whether the box represents text or graphics in the document image, by having the values `TEXT_TYPE` or `IMAGE_TYPE`, respectively;
- *text data*: contains some information about the font style, size and text angle;
- *text*: the recognized text from the image clip;

Of course, the text and text data properties are only needed in case the box represents text.

The text data property is represented by a class that contains many information useful to represent the text as closer as possible to the original one present in the document image. The following list shows a list of the properties the text data holds:

- *face*: the type font face;
- *size*: the font size;
- *line space*: the line spacing;
- *letter space*: the letter spacing;
- *justification*: the justification or text alignment;
- *style*: the font style like italic or normal;
- *weight*: the font weight like bold or oblique;
- *angle*: the angle of the text.

With the data boxes, all the outlined contents present in a document image are represented but there is one last set of properties that also define the document – the page dimensions.

4.3.2 Page data

Until this point, all that was mentioned was the contents of a document image or page but the page itself, particularly its dimensions, also represent an important property of the document. A magazine page may have different dimensions than a book page, a poster or a newspaper and so, to fairly convert the document image it is necessary to also keep the original page dimensions.

This way the properties present in the page data are:

- *pixel width*: the width of the image in pixels;
- *pixel height*: the height of the image in pixels;
- *image path*: the path to the original document image in the system;

- *width*: the width of the image in inches;
- *height*: the height of the image in inches;
- *resolution*: the image resolution in *dpi* (*dots per inch*);
- *data boxes*: a list with the data boxes that belong to the page data.

Resolution

The resolution is an important property as it will be used to calculate the picture's print dimensions (the real dimensions). In certain image formats, the resolution is embedded on them and can be easily retrieved (for example in the PNG format) whereas others do not include this information (like the PNM format). For the latter, a resolution value of 300 dpi is assumed.

Hence, to find the image's real size in inches, the image's width and height in pixels are divided by the image's resolution. For example, an image with a height of 2000 pixels and a resolution of 300 dpi is approximately 6.67 inches high. The inch measurement unit was chosen instead of, for example, the centimeter, because the resolution is given in dots per inch and so, by using the inch, no conversions are needed at this point.

With the definition of the page data and the data boxes, the document can be fairly represented as they store all the needed information to finally generate an editable document format. This concludes the explanation of the main structures created for this project.

4.4 Exportation to editable formats

In this section, the actual conversion of the the document image to an editable document format is explained. For the first version of this project the exportation/conversion can be done to two formats – ODT and HTML.

4.4.1 Exportation to ODT

ODT is the primary exportation format of OCRFeeder. The idea is to produce an exact ODT version of the document image from the information that the page data (and its data boxes) supplies. Two main kinds of objects are created in document, text frames and images, depending whether a data box has the type text or image, respectively.

The text frames will obviously have the text present in the data box as well as all the text properties mentioned before (font face, size, style, etc.). The images will be simply the image clip that the data box outlined.

For any data box, the way to place them is the same and very straightforward. An image or text frame in the ODT document are placed according to the corresponding data box's variables x and y after being converted to the print size using the image's resolution. For example, if a data box's x and y variables are respectively 200 and 100 pixels and the image's resolution is 300 dpi, then the text frame or image that the data box represents will be placed at 0.67 inches from the document's left edge and 0.33 inches from the document's upper edge. The data box's width and height are analogously calculated from the same property of the text frame or image.

4.4.2 Exportation to HTML

The exportation to HTML is a little different than the ODT one. To begin, instead of generating only one file, this exportation will create a folder with the name given by the user and inside the folder there will be the HTML files that represent the exported pages, a style sheet file and a folder with images if the document has such. The generated HTML files are named like *index.html*, *page2.html*, *page3.html*, and so on. Every HTML file gets the elements' styles from a generated *CSS* file calles *style.css.* If the document contains any images, those will be inside a folder called *images.*

The text contents originate paragraphs (the tag *<p>*) and the graphic contents originate images (the tag *<img>*). Both are placed in each HTML page in an *absolute position* way, this will make the paragraphs and images behave similar

to the frames in the ODT files (placed independently from each other). However, since the contents are placed using real sizes and the HTML pages do not relate to resolution, the results are not as good as in the ODT exportation.

Both the ODT and the HTML exportation deal with the styles in order not to repeat unnecessary information. For example, if two paragraphs are added and their font face, size, line spacing and other properties are equal, than this information is stored only once. In the HTML this is done by creating a CSS class and in the ODT this is done in the paragraph class.

By not repeating information, not only space is saved but also if the user edits a paragraph style, the changes are also honored by all elements that implement the changed style.

4.4.3 Adding support for more formats

A class called `DocumentGeneration` was created and has all the main methods (some of them are abstract methods) necessary to perform export the page data to a document format. This class was created to make it easy to implement other exportations, this way, all classes that perform exportation should be subclasses of this main class.

In further versions of this project, main document formats used widely should be supported.

Chapter 5

OCRFeeder

This project can actually be used in two ways, it can be used from the command line for automation and quick conversion of document images and it can be used with a graphical user interface. OCRFeeder Studio is the main part of this project featuring a graphical user interface. The GUI not only lets the user choose the document images in a graphical way but also allows him or her to review and edit what the document layout analysis algorithms have done before the actual document conversion. Hence, the word "studio" was added to the name in order to distinguish the two ways of using the project. All the features present in OCRFeeder Studio give the user extra freedom to control what's being done on the conversion and hence, produce a better conversion.

This section presents all that a user can do with OCRFeeder Studio. Usability was also a big concern while projecting the graphical user interface and so, some concepts about it are also presented.

5.1 Design and usability

Before actually start building the GUI, it was designed using paper prototyping techniques. Paper prototyping have been used since the 1980s to design and test user interfaces [28]. It consists in drawing the GUI in paper dividing the components. This way, GUI changes and replacements can be reviewed easily. Figure 5.1 shows the final paper prototype for the user interface of this project.

The GUI design and behavior of OCRFeeder Studio was built based on the *GNOME Human Interface Guidelines* [29] (GNOME HIG). Like the name suggests, GNOME HIG is a document that describes how to create graphical user interfaces following the principles and philosophy of the GNOME interface. By following the GNOME HIG the interface will look and behave in a familiar way making users adapt to it faster. The interface will adapt to the user's custom properties like desktop themes, fonts, colors and be accessible even to users with special needs.

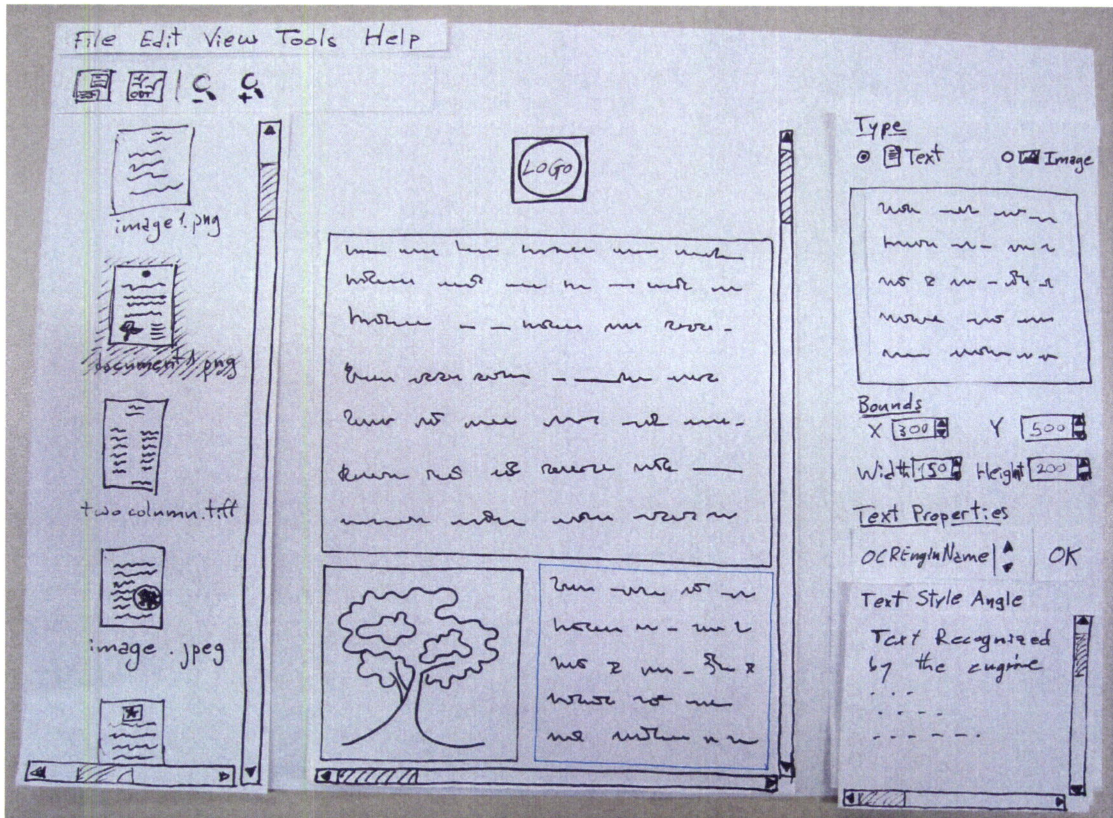


Figure 5.1: OCRFeeder Studio final paper prototype

5.2 Interface overview

The user interface was thought to allow the user to add the document images, perform the layout analysis automatically and manually edit the layout analysis results. To perform this, three main areas were projected:

- The document images area;
- The selectable boxes area;
- The box editor area.

5.2.1 Document images area

Like it is shown in Figure 5.2, the left pane of OCRFeeder Studio is the *document images area*. That's where all added document images will take place and are represented by a thumbnail version of the original images. The user can drag the images within the area to reorder them, that will be the order of the pages when the images are converted to an editable document format.

When the same image is added more than once, OCRFeeder Studio acts as if different images were added but the added image's name will have a suffix in order to distinguish the images. For example if the image *picture.jpeg* is added, the name will be *picture.jpeg* but the second time the user adds the same image, the name will be *picture.jpeg (2)*.

Pressing the right mouse button when an image is selected on the document images area pops up a menu to remove that image. The program will ask for the user's confirmation and if the answer is affirmative, the image as well, as all the work done on it, will be removed.

5.2.2 Selectable boxes area

When the user presses one of the images thumbnails, the original document image will appear in the area in the center. This area is called the *selectable boxes area* because that's where the user can see the data boxes mentioned in Section 4.3.1. The data boxes are represented as rectangles that outline the contents.

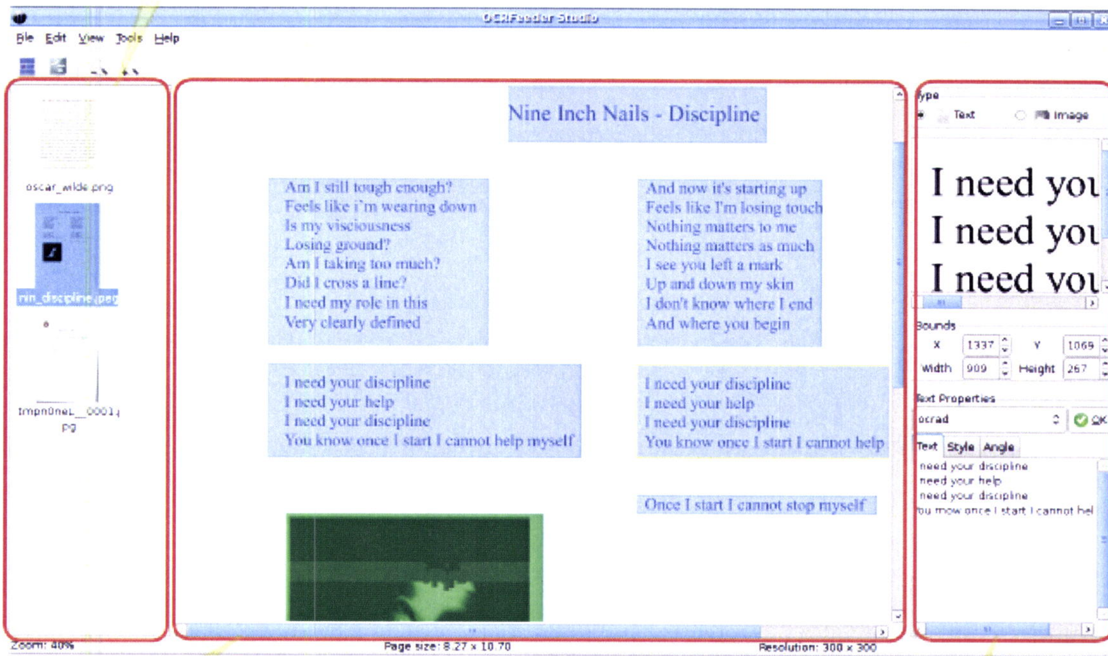


Figure 5.2: OCRFeeder Studio main areas

Since there are no widgets in GTK that offer the selectable rectangles functionality, a first version of the selectable boxes area was implemented using *Cairo*[1] (particularly the Python bindings). Although Cairo provides a good way to create and manipulate vector graphics, that control was too low-level to what was trying to be accomplished. Hence, this first version didn't offer much of the functionalities that are now present in the selectable boxes but then a better way to develop it was found in Goocanvas. Goocanvas saved much work that would be needed using Cairo because it keeps track of any object present in the Canvas and offers many useful methods over them (see Section 3.2.4 for an overview of the Goocanvas' Python bindings). This final version of the selectable boxes area allows to set a background image and create selectable boxes – rectangles – that can be selected, dragged, colored and delete.

Like mentioned before, the selectable boxes represent the data boxes and hence, when the user clicks a selectable box, the box editor that corresponds to that data box is shown on the box editor area.

5.2.3 Box editor area

The box editor shows every property present in the data box. The whole area is divided in frames that can be identified by their title label. From top to bottom, the first frame contains the type of the window, pressing the *image* radio button will change the respective selectable box's color indicating that the data box is of type image and the *Text Properties* frame will be grayed out not allowing the user to use the text properties; the *text* radio button will change back the selectable box's color and the *Text Properties* becomes sensitive again. In Figure 5.2 the text boxes are colored blue and the box outlining the album cover is colored green.

After that comes the image frame, the only one without a label because the image contained in it suggests what it is – the image outlined by the data box. This image is shown in its original size and works as a preview of the image.

The frames that follows the image one is the box's bounds frame. In it, four spin buttons control the box's x, y, height and width properties. The Interactions with these buttons will be immediately reflected in the respective selectable box

[1] *Cairo*: a library that provides a vector graphics API – `http://www.cairographics.org`

in the selectable boxes area.

Finally the *Text Properties* frame controls the data box's text properties but also lets the user set them automatically by choosing the desired OCR engine from the combo box and pressing the OK button. Under the combo box and the OK button there are three tabs that show the data box's text content, the text style and the text angle. In the text content there is a text area where the text recognized by the OCR engine will be and where the user can edit it. The text style contains other three frames that allow to set the font face, text alignment and the letter and line spacings. On the angle part, the text angle is shown as well as a button to detect the angle automatically. Due to the problems mentioned in section 4.2.3, the angle can be detected only for text that is written from the lower left to the upper right of its bounding box.

5.3 Features

Although the functionalities above also represent features, they were explained outside of this section because they constitute the program's main action areas.

This section presents every other feature in this project.

5.3.1 Adding document images

Adding a document image is normally the first thing a user does when using this project. Images can be added in the most common image formats like JPEG, PNG, PNM, TIFF, etc.

One image can be added at a time selecting *Add Image* from the *File* menu, or all the images in a folder can be added selecting *Add Folder* from the same menu.

5.3.2 PDF importation

Sometimes scanned documents images are converted to PDF documents and a user might want to make an editable document from them. For this purpose, PDF importation was implemented. The importation uses the Ghostscript command line tool to convert the PDF documents into images and then adds the images

like they were ordinary images. PDF documents can be imported from the menu *Import PDF* of in the *File* menu.

5.3.3 Exportation

The exportation to the supported document formats explained in Section 4.4 can be done using the *Export...* menu (in the *File* menu) that pops up a dialog with the exportation formats in a combo box.

Since ODT is the primary exportation format, the user can quickly export the document images to an ODT file by using the corresponding button in the toolbar.

5.3.4 Project loading and saving

There might be a situation where the user is using this project to develop an editable document from many document images. This task may take some time and the user might want to close the program but be able to continue the work later. To accomplish this, a file format called *OCRF* was created that represents the project that the user was working on.

OCRF

The OCRF format is nothing more than a zip file that contains all the information about the document images, page data and data boxes that the user was working on. The actual document images are included in the file so the user can use a project file in a machine that does not have the images used in the project. For example, a user is using OCRFeeder at work, saves the project and wants to continue working on his or her computer at home which doesn't have the document images used in work.

All the data about the pages, data boxes and images' paths is stored in an XML file called *project.xml*.

Saving the project

When saving the project all properties of each page data and data box object are stored in a Python dictionary which is then converted to XML. For example,

the data box's method `convertToDict` returns a dictionary whose keys are the names of the variables x, y, width, height, type, text and text_data and the values are the values of the respective variables in the object instance. The text_data key in the dictionary will be itself a dictionary as a result of the same method `convertToDict` but from the `TextData` object instance.

The class responsible for saving the project is the `ProjectSaver` and after calling the methods to convert the mentioned object instances to dictionaries, it uses its method `serialize` which converts the dictionaries and the document images information to XML and finally creates the project file.

The generated *project.xml* files should look like the example present in Appendix A.

The *image* elements hold the original and embedded names for the images, the original name is the path to the image in the user's system, the embedded name is the image's name in the *images* folder that is included in the zip file.

Loading the project

The way to load the project from a project file is pretty much the opposite of saving it. The `ProjectLoader` unzips the project file in the configured temporary folder, runs through the XML file and instantiates the page data, data box and text data objects described in the file. To instantiate the page data object, it matches the *image_path* element with the respective *image* element. If the *original_name* element of the latter contains an existing path in the current system, that path is used as the page data's image; if the original path doesn't exist, the embedded image's name is added to the path to the unzipped project folder and used instead. This grants the possibility of using the project file in different machines like mentioned before.

Appending a project

It is also possible to apppend a project. When a project is appended, all the project information is loaded but instead of substituting what the user is doing, it will append the images and the information to the already existing ones in the current working project. A project can be append using the menu *Append project*

in the *File* menu.

5.3.5 Preferences

Like many other graphical programs, OCRFeeder Studio also allows to set the program's preferences. The preferences dialog is called from the edit menu in the menu bar. The dialog is divided in tabs, in the *General* tab, the temporary folder and the window size can be set. The temporary folder is the folder used for storing temporary files like image clips. The window size is related to the Sliding Window Algorithm mentioned in Section 4.1.1, it allows to set a custom window size in case the automatically calculated one doesn't fill the user's needs.

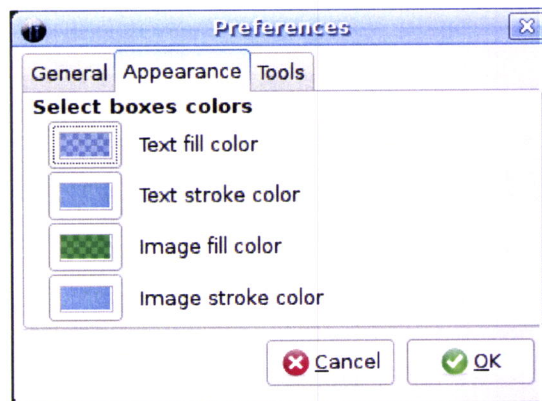


Figure 5.3: Preferences dialog (appearance tab)

In the *Appearance* tab (see Figure 5.3), the selectable boxes' fill and stroke colors can be set for the different types of boxes.

The *Tools* tab allows to set the path in the system for the Unpaper tool[2] as well as the favorite OCR engine. The favorite OCR engine is the engine that will be used to recognize the text when the layout analysis is performed.

[2]For an overview of Unpaper, consult the subsection 3.2.9 for an overview of Unpaper

5.3.6 Edit page

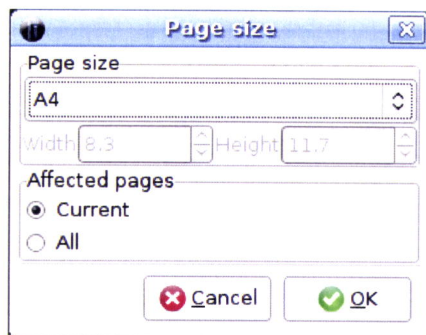


Figure 5.4: Paper sizes dialog

To join documents with different page sizes or simply convert the documents with a custom page size, the user may want to change the images' corresponding page size. This can be done by editing the page from the *Edit page* menu in the *Edit* menu. Selecting this menu will pop up a dialog (see Figure 5.4) that allows choosing a standard paper size or set a custom one and affect the currently selected image or all images.

5.3.7 Delete images

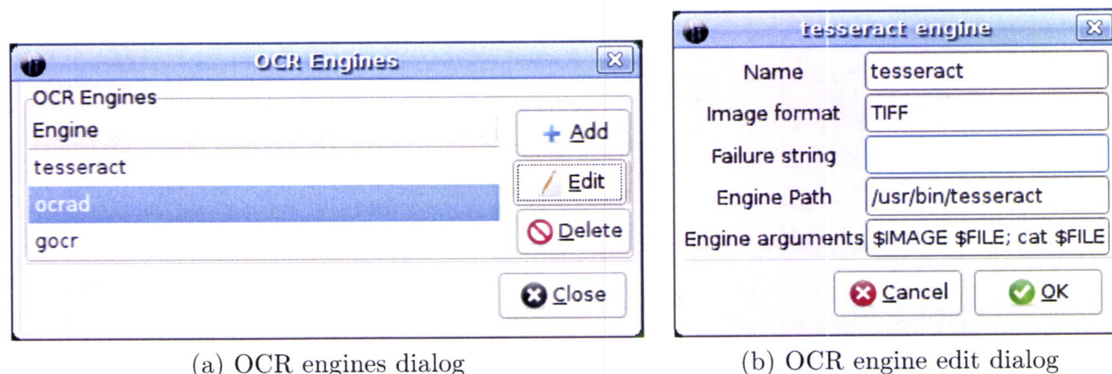
The user can delete the currently selected image by clicking it with the second mouse button or using the menu *Delete page* from the *Edit* menu. All images can be deleted at once by using the menu *Clear project* under the *Edit* menu.

5.3.8 Zoom

In the menu *View*, the user can decrease and increase the zoom or set the zoom to be the normal size.

The zoom can also be quickly increased and decreased using the corresponding toolbar buttons or by using the keys + and - when the selectable boxes area is focused.

5.3.9 OCR engines



(a) OCR engines dialog (b) OCR engine edit dialog

Figure 5.5: Example of the OCR engines dialogs

Apart from the XML files, the user can also create, edit or delete the OCR engines from the graphical interface. This can be done using the menu *OCR Engines* from the *Tools* menu. The dialog that pops up shows a list of the existing OCR engines (Figure 5.5a) with add, edit (see Figure 5.5b) and delete buttons. The creation and edition dialogs contains text entries that represent the XML elements explained in the 4.2.1.

5.3.10 Unpaper

The *Tools* menu also contains a menu called *Unpaper* that provides an easy way of using the command line tool with the same name. The main filter utilities of Unpaper can be easily set in the dialog but an extra text entry can be also used to apply extra options. Extra options should be used like they would the command

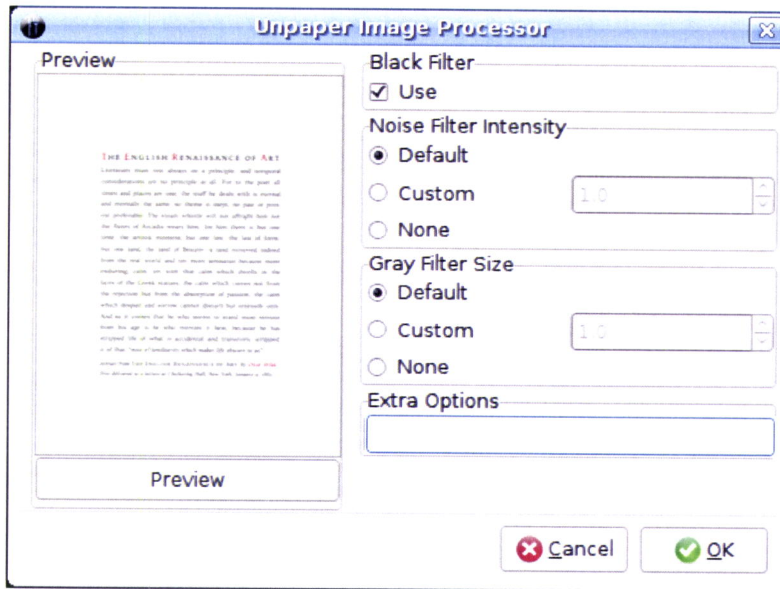


Figure 5.6: Unpaper dialog

line and in case any of the main filters is used as an option, it will override the ones set directly from the widgets. As shown in Figure 5.6, the user can preview the changes that Unpaper does to the image before applying them.

5.3.11 Layout analysis and OCR

The layout analysis and optical character recognition can be performed on the currently selected document image by using the toolbar's first button. When the user presses this button, a progress bar is shown during the time the layout analysis and optical text recognition occurs. This time depends of the complexity and size of the image.

Chapter 6

Testing

In this chapter, different types of documents are tested using OCRFeeder Studio and the results are commented. In what comes to Layout Analysis, the tests are not easily measured or numerable. For example, in a system to perform only OCR, the tests could be measured by dividing the number of characters or words recognized correctly by the total number of characters or words. In this project, rather than measuring the tests anyhow, the tests are commented according to the page segmentation accuracy, problems, font size detection, etc.

The following sections present the results obtained and comparisons with other existing solutions. These results refer to the automatic layout analysis and recognition not being present or considered, obviously, any manual correction.

The chosen solutions for the comparisons are:

- Nuance OmniPage Professional 16;
- SimpleOCR 3.1;
- ABBYY FineReader 9.0 Professional;
- Vividata OCR Shop XTR;
- OCRopus (SVN version from November 15th 2008).

6.1 Features Comparison

This section compares the features of the chosen solutions. The criteria considered for this comparison is:

- *License*: the license under which the software is published;
- *Operating Systems*: the operating systems on which the software is available;
- *Manual Zoning/Correction*: whether the user can manually edit the regions of interest;

- *Automatic Layout Analysis*: whether the software performs automatic layout analysis or not;
- *Graphical User Interface*: whether the software provides a graphical user interface;
- *Input Formats*: the file formats that the software can import;
- *Exportation Formats*: the formats to which the software exports the document;
- *Project Saving/Loading*: whether the software provides a way to save and load a working project letting the user continue a previously started work;
- *Image Enhancement*: whether the software provides a way to perform some kind of image enhancement (for example, removing the dust);

Table 6.1 presents the comparison between the mentioned solutions.

6.2 Tests

The tests will be performed with 5 different types of documents. The next sections will present each document and the tests results as well as a comparison between the results of OCRFeeder Studio and the results of OmniPage and FineReader. These solutions are perhaps two of the most advanced solutions available and are the ones more similar to OCRFeeder. Both of them were tested using their trial versions since they are not freely available.

The tests are shown with no manual intervention, that is, the window size used to find the images' contents is the automatically calculated one and the text is the one that the OCR engine recognized (with no manual corrections). The engine Ocrad was used in all the tests.

All the images were scanned from the same device, the multi-function printer *HP Deskjet F370*.

Table 6.1: Features comparison for several DAR and OCR solutions

	OmniPage	SimpleOCR	FineReader	OCR Shop XTR	OCRopus	OCRFeeder
License	Commercial	Commercial	Commercial	Commercial	BSD License	GPL License
Operating Systems	Windows, Mac OS	Windows	Windows	Linux, UNIX	Linux	Linux
Automatic Layout Analysis	Yes	No	Yes	Yes	Yes	Yes
Manual Zoning/Correction	Yes	Yes	Yes	No	No	Yes
Graphical User Interface	Yes	Yes	Yes	No	No	Yes
Input Formats	BMP, PCX, DCX, JPEG, PNG, TIFF, GIF, XPS, DjVu, PDF	TIFF, JPEG and BMP	Yes	GIF, JPEG, PBM, PDF, PNG, TIFF, PPM, PostScript	PNG, TIFF, PNM, JPEG, ...	PNG, GIF, TIFF, PNM, BMP, JPEG, PDF
Exportation Formats	PDF, HTML, Microsoft Word XML, DOC/-DOCX, RTF, XLS/XLSX, PPT, DBF, CSV, TXT and LIT	TXT and RTF	Yes	TXT, XDOC, PDF, HTML	HOCR[a]	ODT and HTML
Project Saving/Loading	Yes	No	Yes	No	No	Yes
Image Enhancement	Yes	No	Yes	No	Yes	Yes

[a]HTML with additional OCR information

6.2.1 Lyrics document

The lyrics example previously introduced in Chapter 4 was created for showing an example of a two-column document with several paragraphs and an image. This document was created using OpenOffice, printed and scanned.

OCRFeeder

The zoning shown is successful to what is expected – each text paragraph is contained in a box marked as text, and the image is contained in a box marked as graphics (see Figure 6.1). The text boxes have the respective recognized text in the *Text Properties* of their box editors. It originated 5 individual text boxes apart from the title because each paragraph is separated by a length greater than the window size. The calculated font sizes were 13 points for the title text and 10 points for the rest of the paragraphs.

One extra block is shown that contain a "dot", that's originated by the dust on the scanner. That dot contrasts with the background and hence gives the impression of being something like a period as recognized by the OCR engine.

Dots like these are removed if the Unpaper tool is used with its defaults settings and they can also be removed manually by clicking each of the boxes and pressing the *delete* key.

OmniPage

OmniPage reads the image properly but the boxes are set differently than in this project. Like seen on Figure 6.2, the paragraphs are not devided like in OCRFeeder, two paragraphs on the left and on the right are joint whereas the last paragraph has its own box. The calculated font sizes were 14 points for the title and 11 points for the rest of the text.

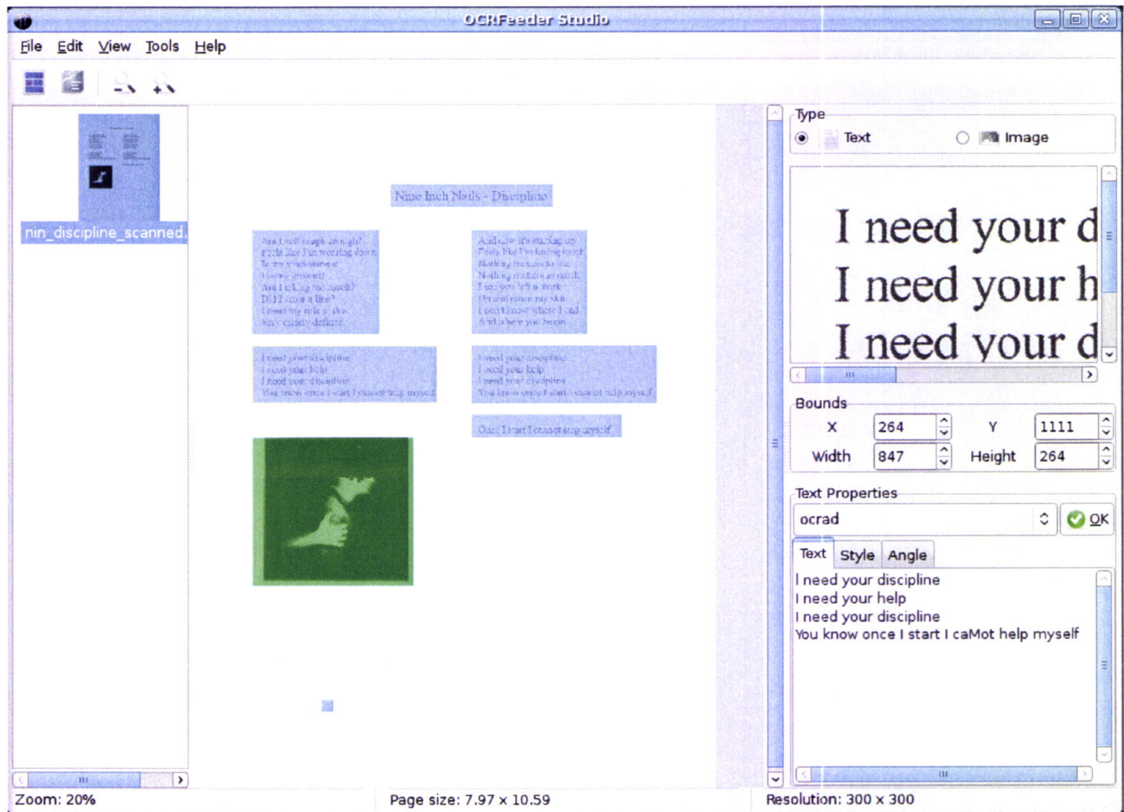


Figure 6.1: Test: Lyrics document with OCRFeeder

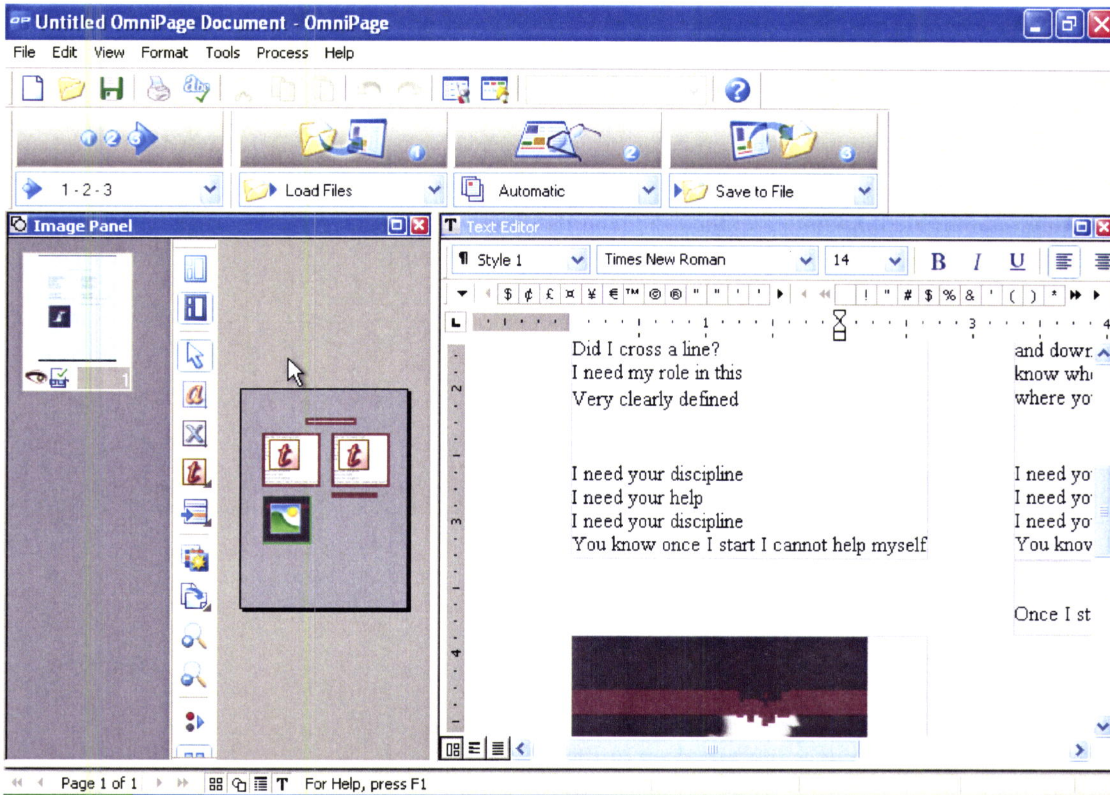


Figure 6.2: Test: Lyrics document with Omnipage

FineReader

FineReader did not considered the image (didn't put a box around it) but the text paragraphs were in this case outlined in a very similar way to what OCRFeeder did since it creates six text boxes like shown on Figure 6.3. The font sizes for this solution were calculated as 12 and 9 points for the title and the rest of the paragraphs, respectively.

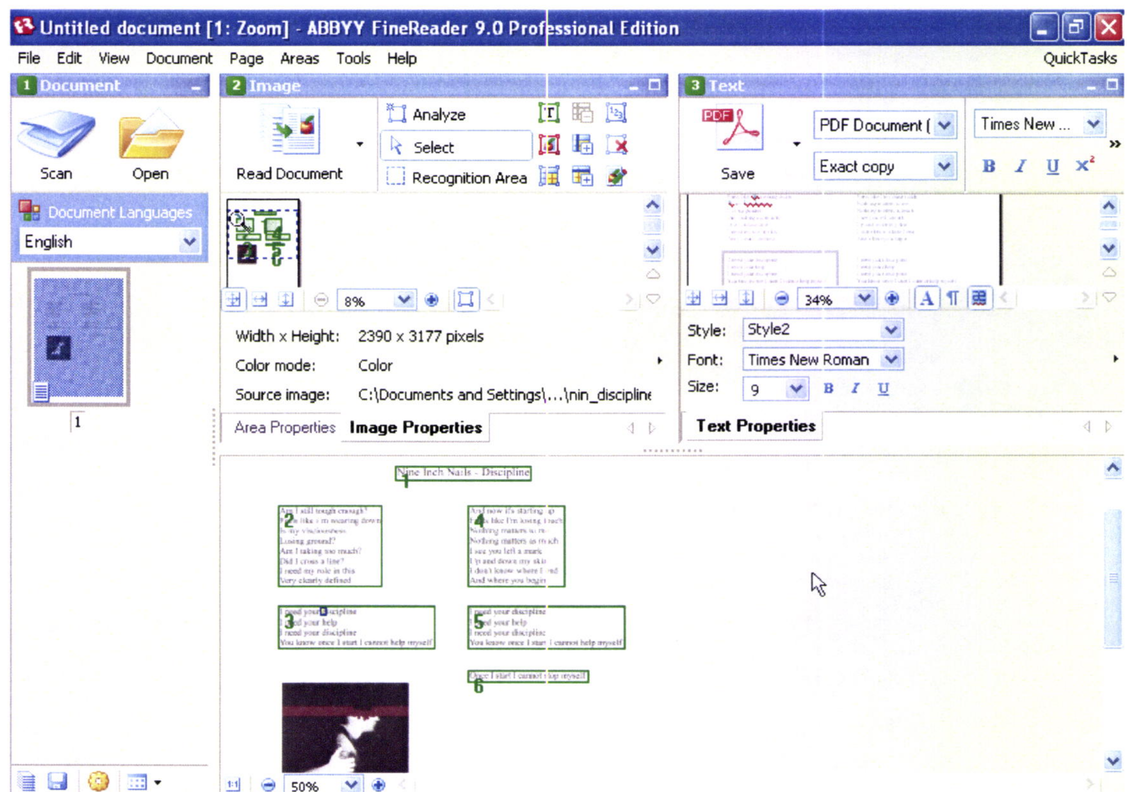


Figure 6.3: Test: Lyrics document with FineReader

Generated Files

In what comes to generating a text document, none of the commercial solutions used can generate an OpenDocument Text file so, the most similar format to

this (so the results can be compared) is Doc. Figures 6.4, 6.5 and 6.6 show the differences between the mentioned files opened with OpenOffice.

Figure 6.4: Lyrics document exported to ODT by OCRFeeder

Since it's the Layout Analysis that is being tested and because no other solution is able to export the documents to ODT, for the other tests, the document generation is not evaluated.

6.2.2 Make: magazine

Description: Page 86 of Volume 11 of the Make: magazine – [30].

OCRFeeder

Figure 6.7 shows the page segmentation for a scanned page of the Make magazine. This is a good example to test because it has different kinds of elements

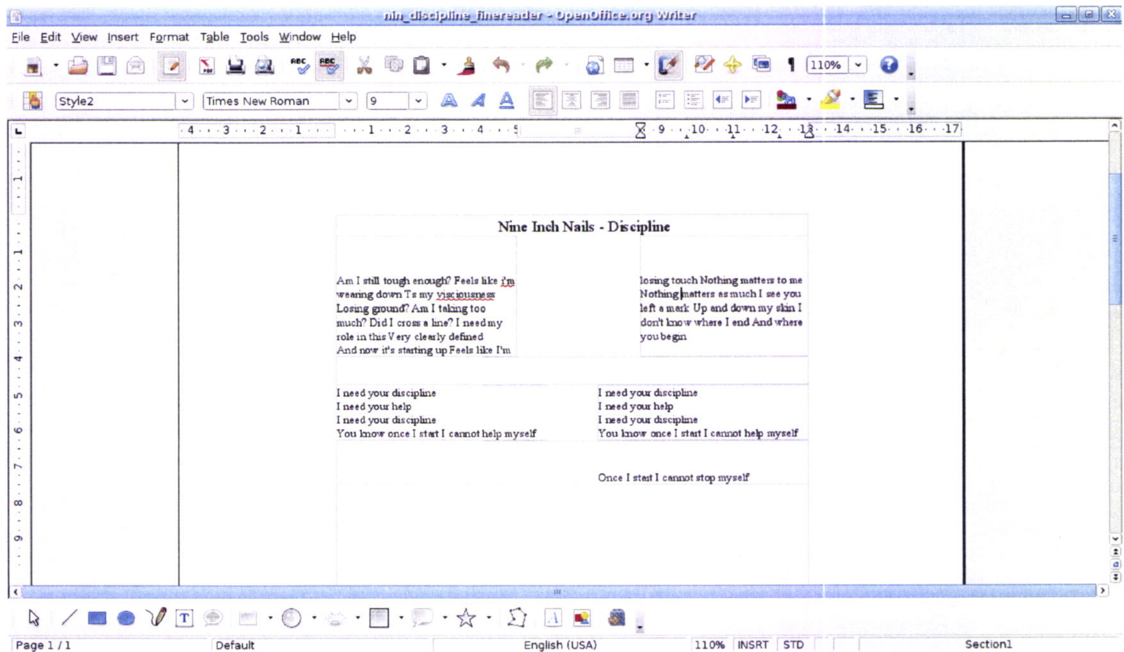


Figure 6.5: Lyrics document exported to Doc by FineReader

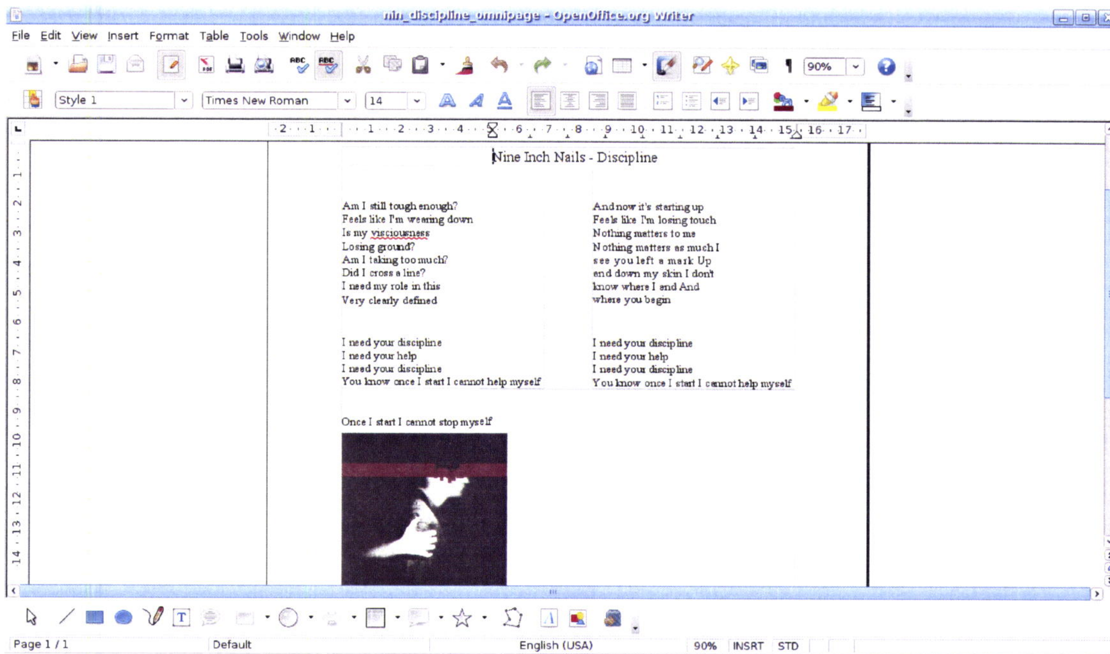


Figure 6.6: Lyrics document exported to Doc by OmniPage

like big and tiny font sizes, with colored text and different font faces. The only problem for OCRFeeder in this example is that the text *"Spirits Guy"* is to close to the next paragraph for the calculated window size. Since the window size does not fit in the space between those two paragraphs, then it joins both paragraphs in the same box. This will obviously result in a wrong font size because the way it is calculated (see the page 49 for an explanation of the font size calculation).

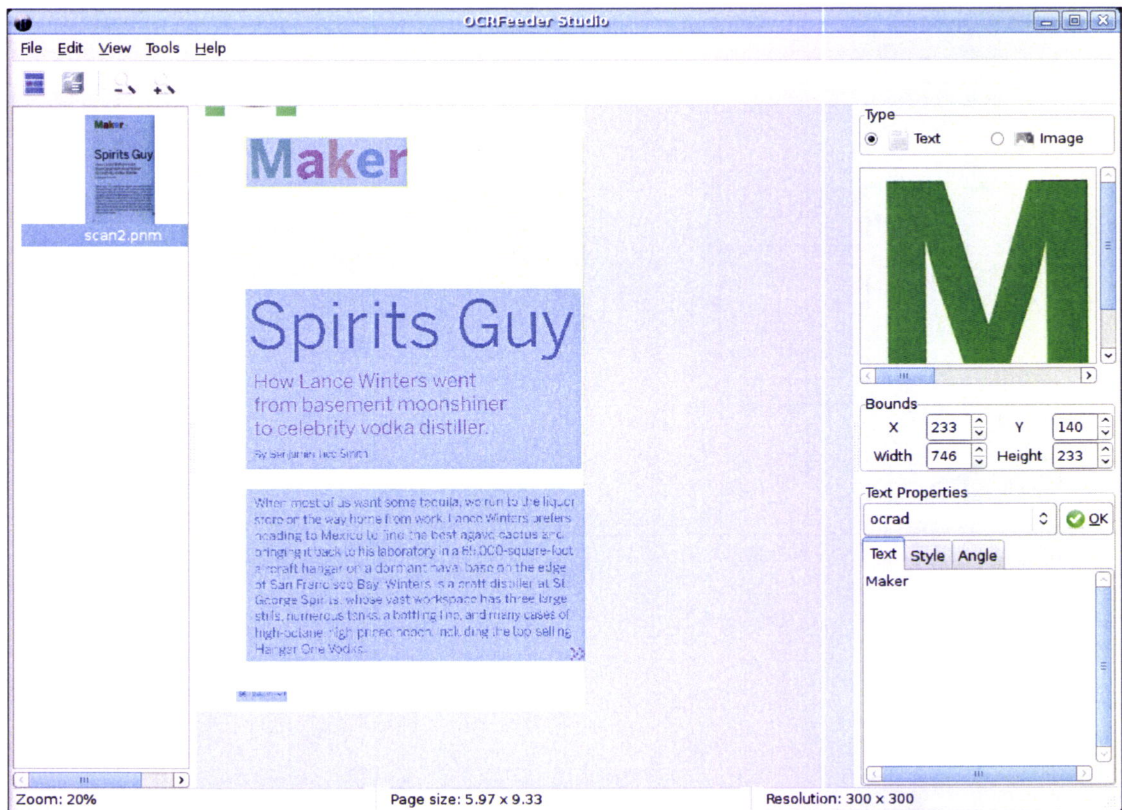


Figure 6.7: Test: Make: magazine with OCRFeeder

OmniPage

The layout analysis of OmniPage works better for the magazine page. Like shown on Figure 6.8, it separates the mentioned paragraphs and hence, giving a

better result after performing OCR over the different parts.

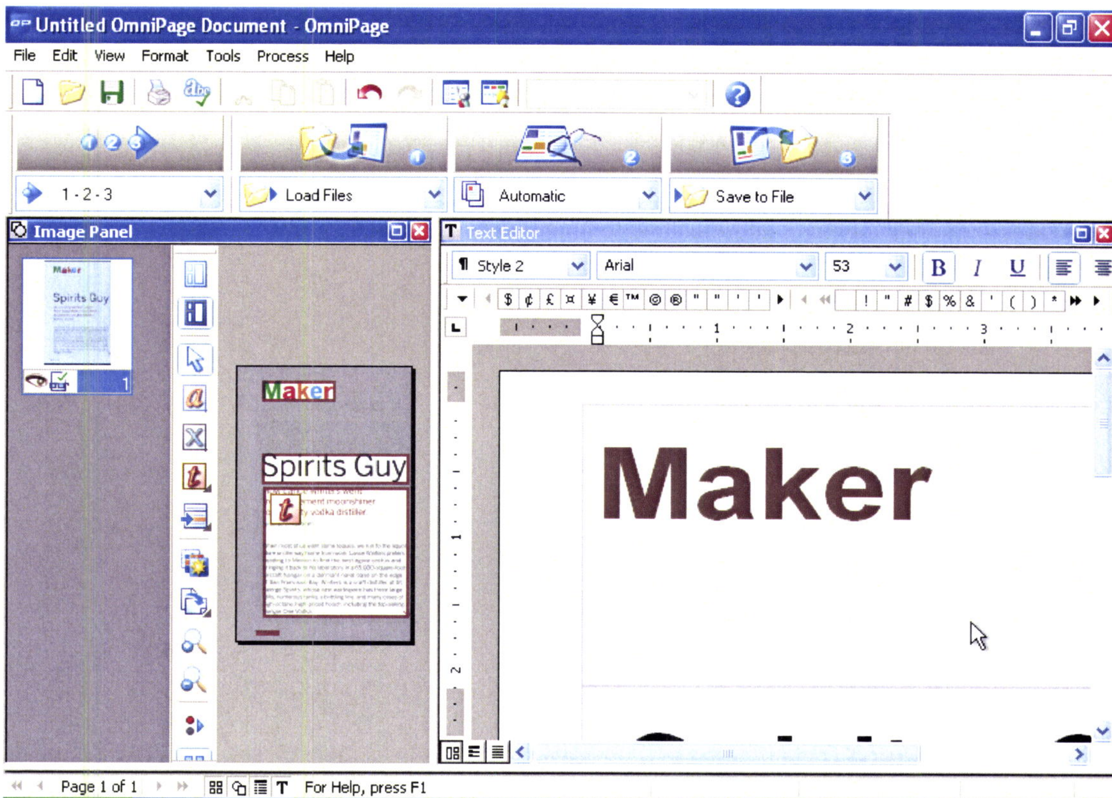


Figure 6.8: Test: Make: magazine with OmniPage

FineReader

FineReader also separates the big text paragraph from the one bellow it. However, it ignores the word "Maker" on the top and doesn't consider it when exporting the image to an editable format (see Figure 6.9).

6.2.3 Beautiful Code book page

Description: Page 323 of the Beautiful Code book – [31].

Figure 6.9: Test: Make: magazine with FineReader

This page was chosen because it contains a very complex picture from a layout analysis point of view. The picture is in fact a diagram which combines graphics, text, horizontal and verticals lines. The text paragraph that comes after the diagram also presents a very usual problem when scanning images from big books – the text near the book center is scanned while the page is slightly curled making the text blurred out.

OCRFeeder

Like Figure 6.10 shows, OCRFeeder divides the diagram in two and detects the text paragraph that follows as well as the footnote.

The reason why the two parts of the diagram are not classified was graphics is because the used engine detected some text from the diagram as well the text right on the side of it and, according to the classification rules previously explained, the two blocks get classified as text.

Performing OCR over the main text paragraph results as expected, the characters in the beginning of each text line (the blurred ones) are not well recognized comparing to the rest of the text. The font sizes calculated for the main paragraph and for the footnote were 9 and 5 points, respectively.

OmniPage

The font size calculated by OmniPage for the two paragraphs was the same as the ones calculated by OCRFeeder. On the other hand it puts text boxes on top of the diagram for some of its labels. Nonetheless, not all the labels are assigned a text box and several parts of the diagram are cut as shown in Figure 6.11.

FineReader

The results from FineReader are similar to OmniPage in what comes to the diagram (see Figure 6.12), it assigns text boxes to some labels but it cuts even more parts of the diagram than OmniPage. The font sizes calculated for the main paragraph and the footnote resulted in the same values as the ones calculated by OCRFeeder and OmniPage.

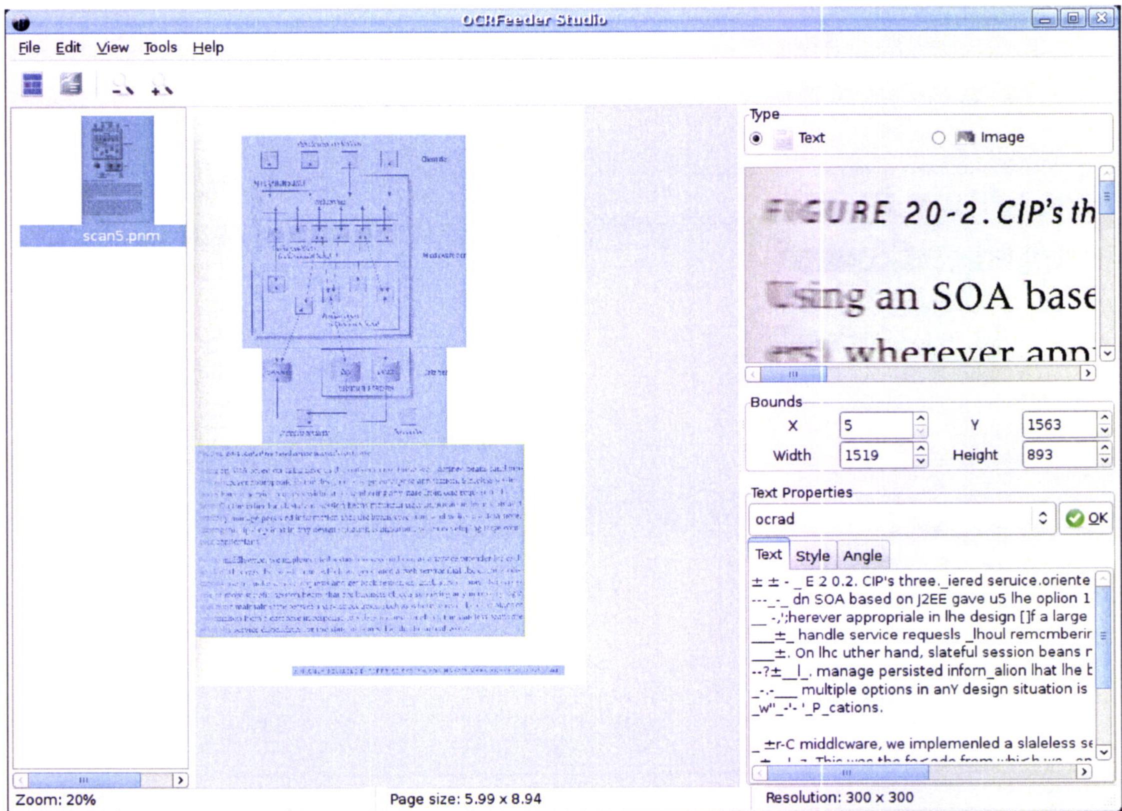


Figure 6.10: Test: Beautiful Code book page with OCRFeeder

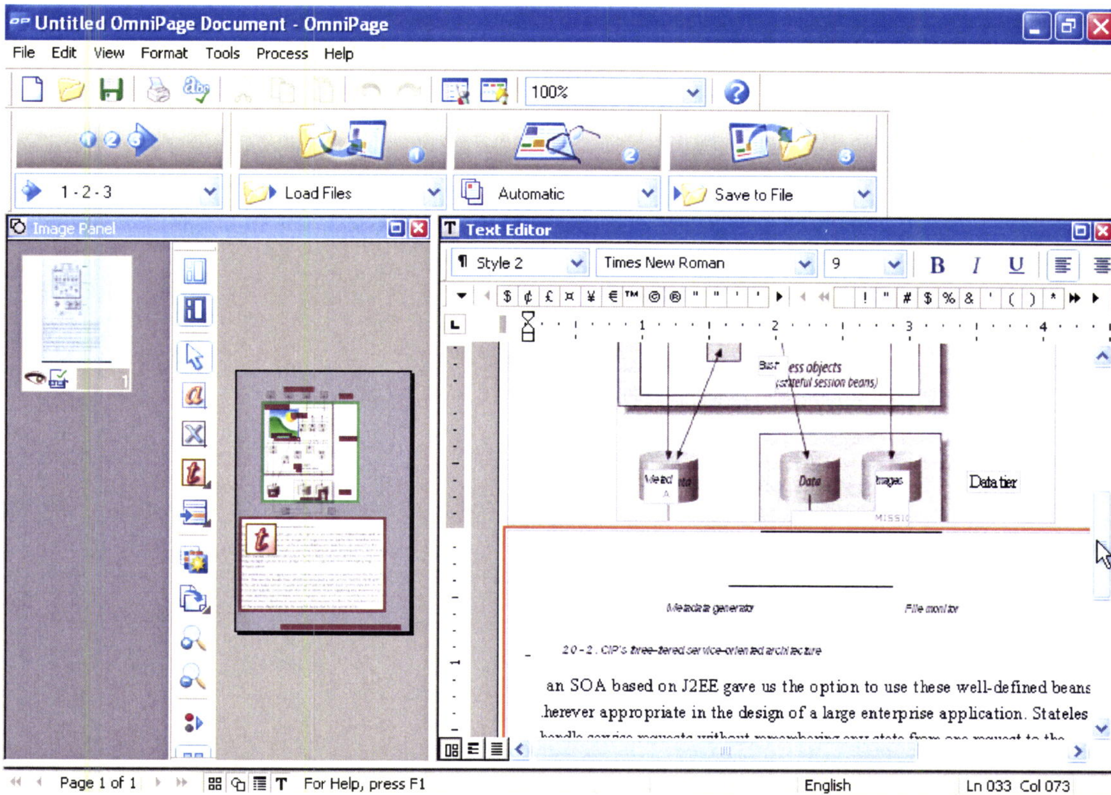


Figure 6.11: Test: Beautiful Code book page with OmniPage

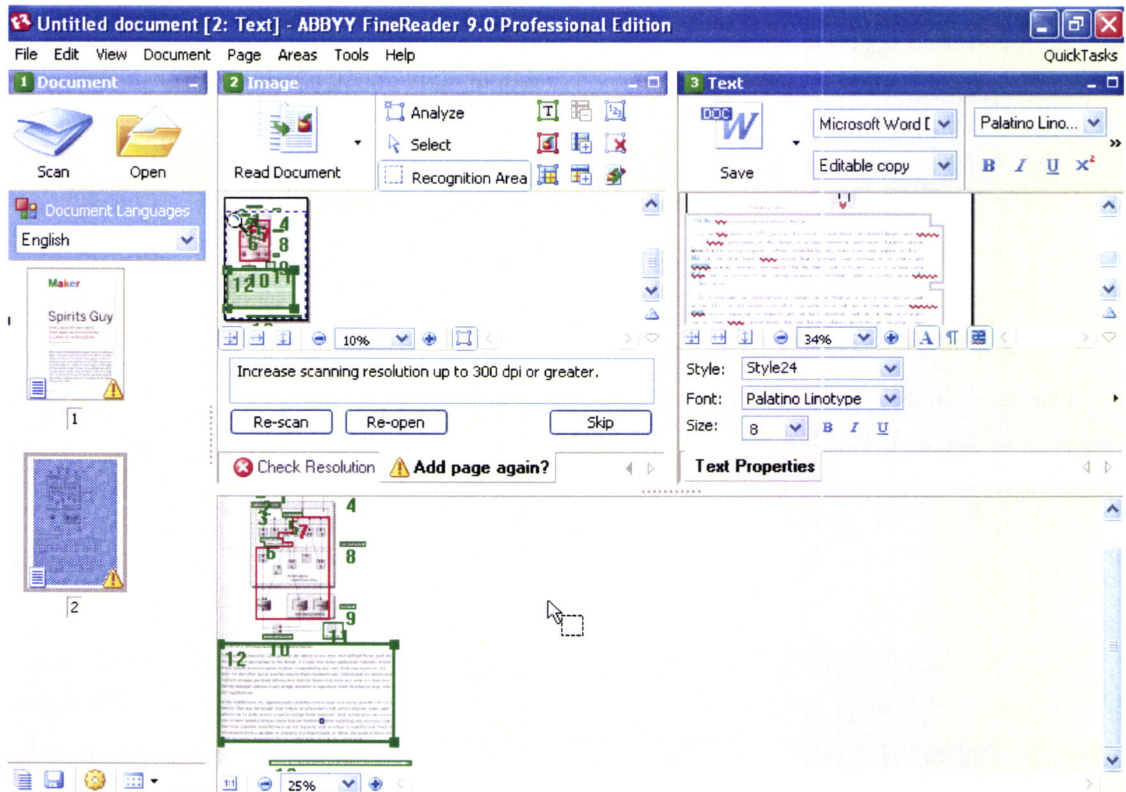


Figure 6.12: Test: Beautiful Code book page with FineReader

6.2.4 The Search book cover

Description: Cover of the Portuguese edition of The Search book – [32].

OCRFeeder

A book cover is another interesting example as it combines different kinds of elements. In this case the cover contains the a big and colorful title text. The word "The" in the title is placed within the "Search" text height and thus, very difficult to be read by an OCR engine.

Like shown in Figure 6.13, none of this page's contents is classified as graphics because every element contains text that gets recognized by the engine and so, making the section be classified as text. Even the three columns in the publisher's logo (on the left bottom) is recognized as text – "| | |". Logos like this are obviously very difficult, even for a human, to classify either as text or graphics.

The text inside the box on the bottom right is also accurately detected and so, the box is classified as text.

OmniPage

This program identifies the book's big title as being graphics. However, it classifies all the rest of the contents as text and for those that are contained or right next to images, the text boxes cut partially or totally the images – for example the publisher logo seems to be discarded, the graphics on the bottom right is completely replaced by a text box and the top image of the page, as well as the images right under the title, are partially replaced (see Figure 6.14).

FineReader

Like Figure 6.15 shows, the title, the top graphics and the bottom right box are ignored and not considered for exportation.

The rest of the text seems to be well recognized.

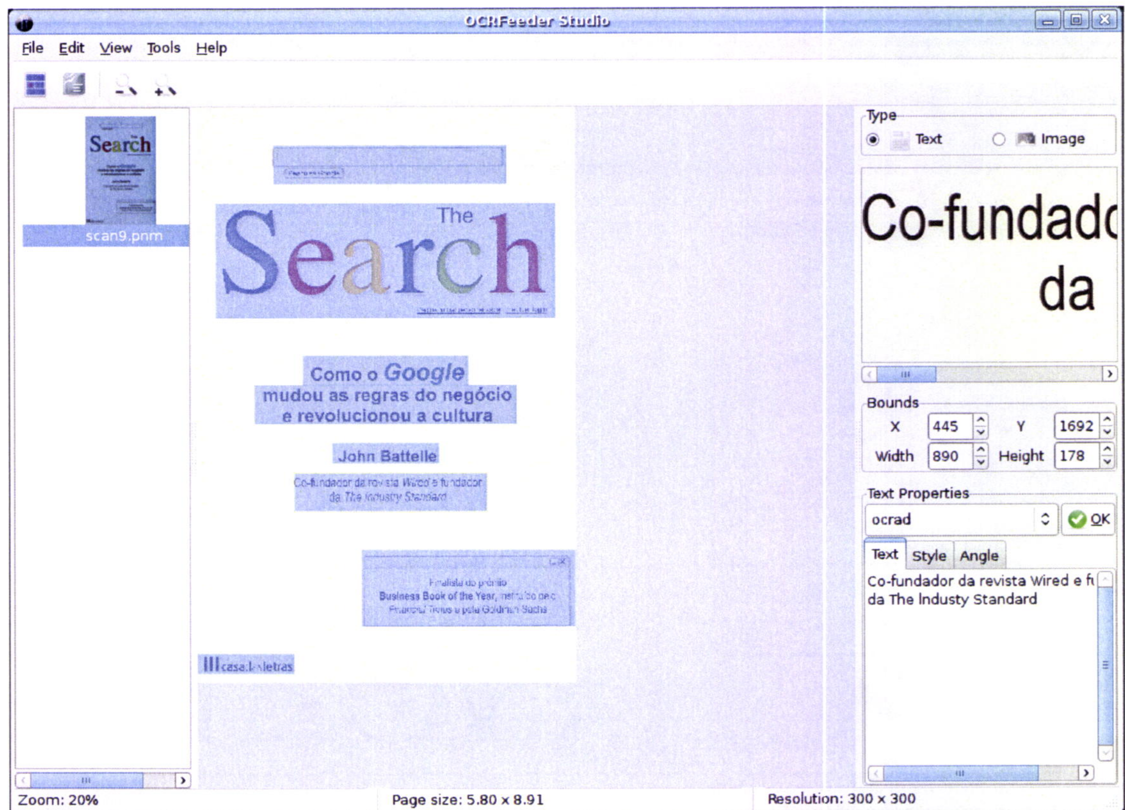


Figure 6.13: Test: The Search book cover with OCRFeeder

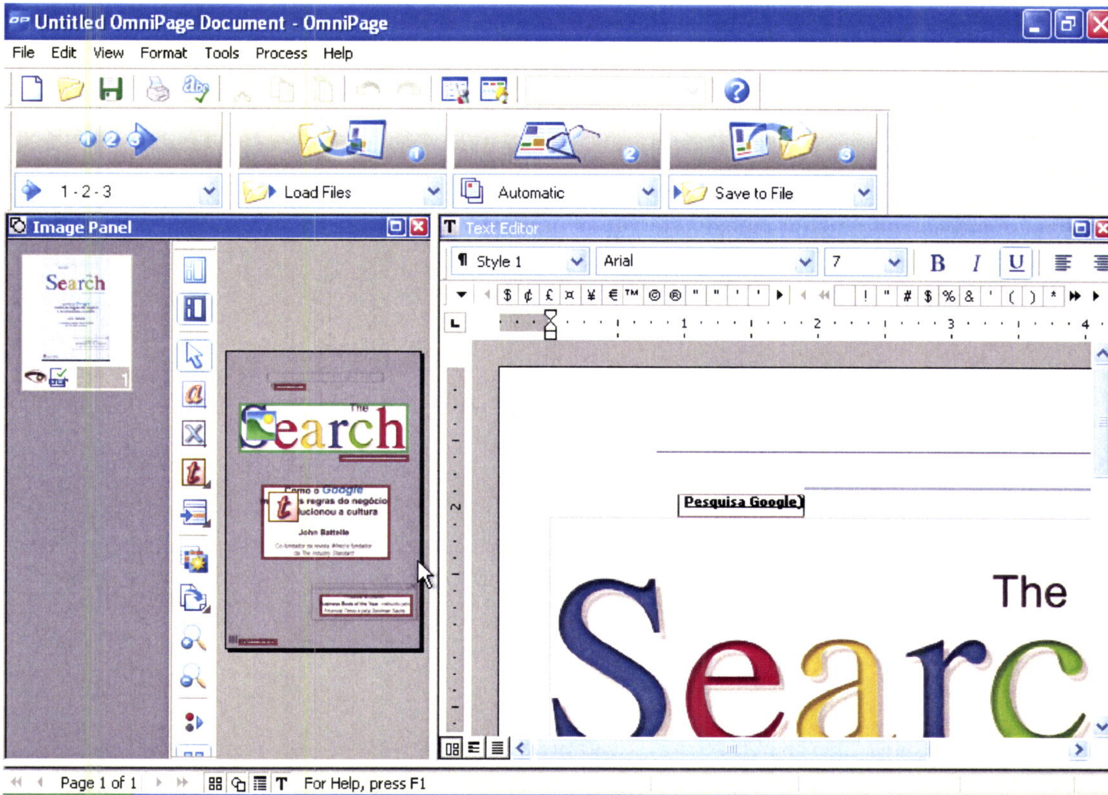


Figure 6.14: Test: The Search book cover with OmniPage

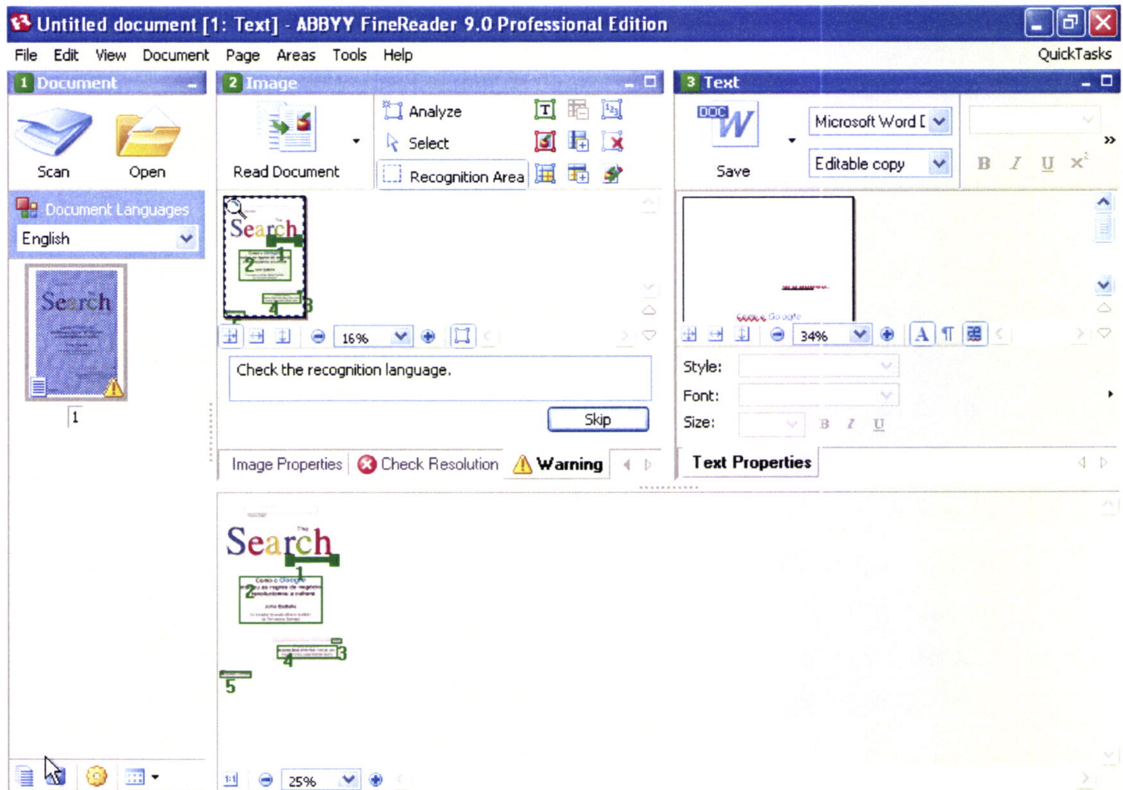


Figure 6.15: Test: The Search book cover with FineReader

6.2.5 Linux Magazine page

Description: Page 95 of the issue 25 of the Linux Magazine (Spanish edition) – [33].

OCRFeeder

Figure 6.16 shows another curious example of how magazines' layouts vary. This page has more than 50% of it occupied by an image which contains some text information itself.

Under the big image, there are three columns of text whose first paragraph has the common style effect of having its first letter bigger than the rest of the text – this effect is called a *drop cap* or *initial*. This makes the function to find the text size not to work properly since it considers every line that the initial occupies to be only a single line and hence results in a font size of 34 points.

Apart from this, OCRFeeder also considers the three paragraphs as being one. This happens due to the window size automatically calculated. Since the space between each paragraph is small and the window calculated doesn't fit within it, it simply doesn't find a space there. The issue number in the bottom – "Número 25" – was classified as being graphics because the used search engine doesn't give any output after processing it. If the engine GOCR is chosen instead of Ocrad, then the text is recognized and the font size results in 6 points.

If the window size is set manually to a size of, for example, 30 pixels instead of being automatically calculated, than the three columns are correctly outlined this time (see Figure 6.17). In this case, one can see how should be the correct font size for the paragraphs in the columns – the columns without the initial are assigned a font size of 9 points while the one with the initial has a font size of 32 points. Still with the 30 pixel window, the page number text is this time recognized accurately by the Ocrad engine.

The font size for the magazine's web page text (in the bottom) and the issue number is 6 points. The font size for the page number is 8 points.

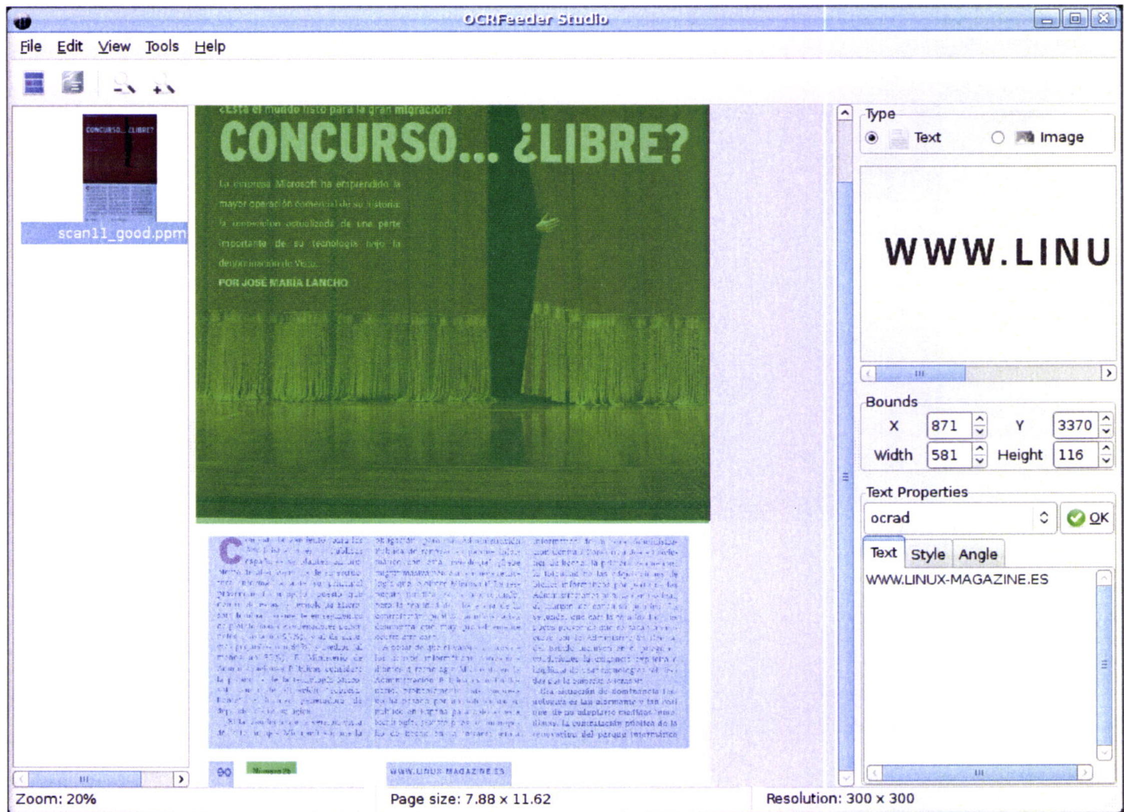


Figure 6.16: Test: Linux Magazine with OCRFeeder (automatic window size)

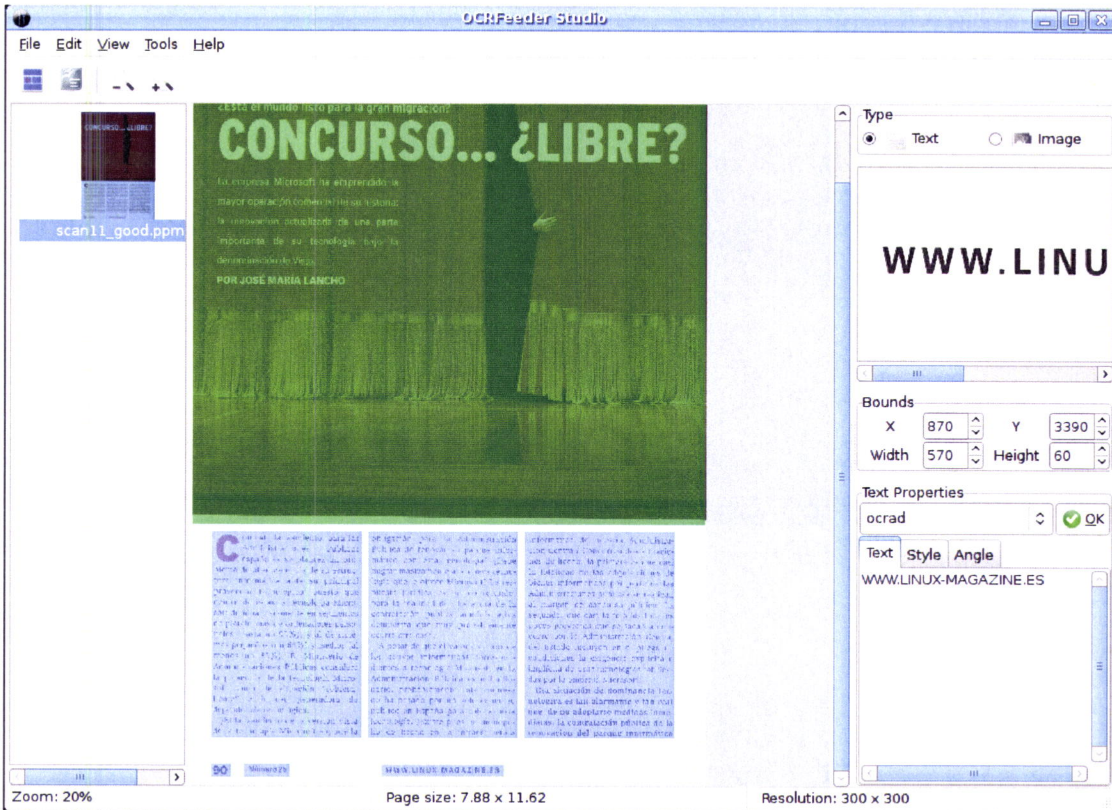


Figure 6.17: Test: Linux Magazine with OCRFeeder (manual window size)

OmniPage

This solution detects the three columns automatically (see 6.18) and also sets text boxes for the text placed on top of the image although it doesn't recognize it accurately. By putting text boxes on the text it attempts to recognize from the image, part of the image is cut and replaced with the color red (as the background of the text boxes).

The font size for the three columns text, the magazine's web page address and the issue number is 8 points. The page size is assigned with a font size of 12.

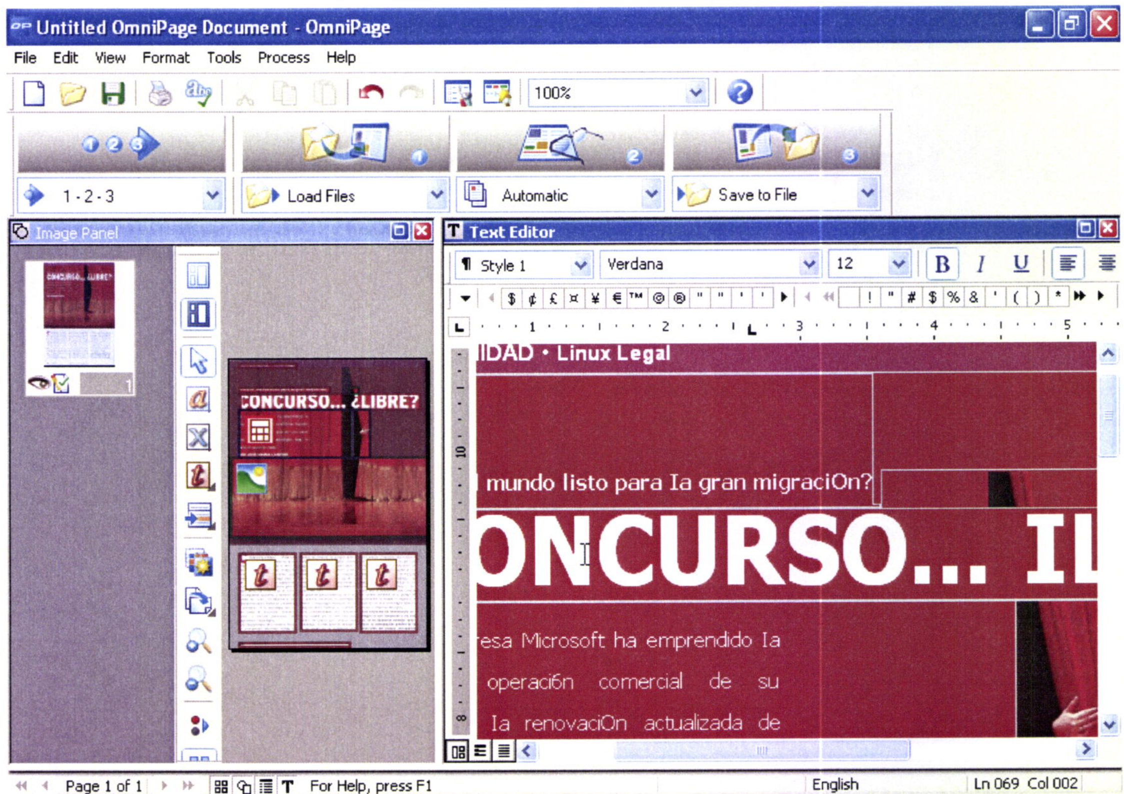


Figure 6.18: Test: Linux Magazine with OmniPage

FineReader

Like OmniPage, this program also detects the text present on the image but discards all of it only including small parts of it like Figure 6.19 shows. The text is also detected as three columns and with the same font size as OCRFeeder calculates – 9 points. However, it sets a font size of 10, 12 and 14 points to the issue number, magazine's web page and page size, respectively.

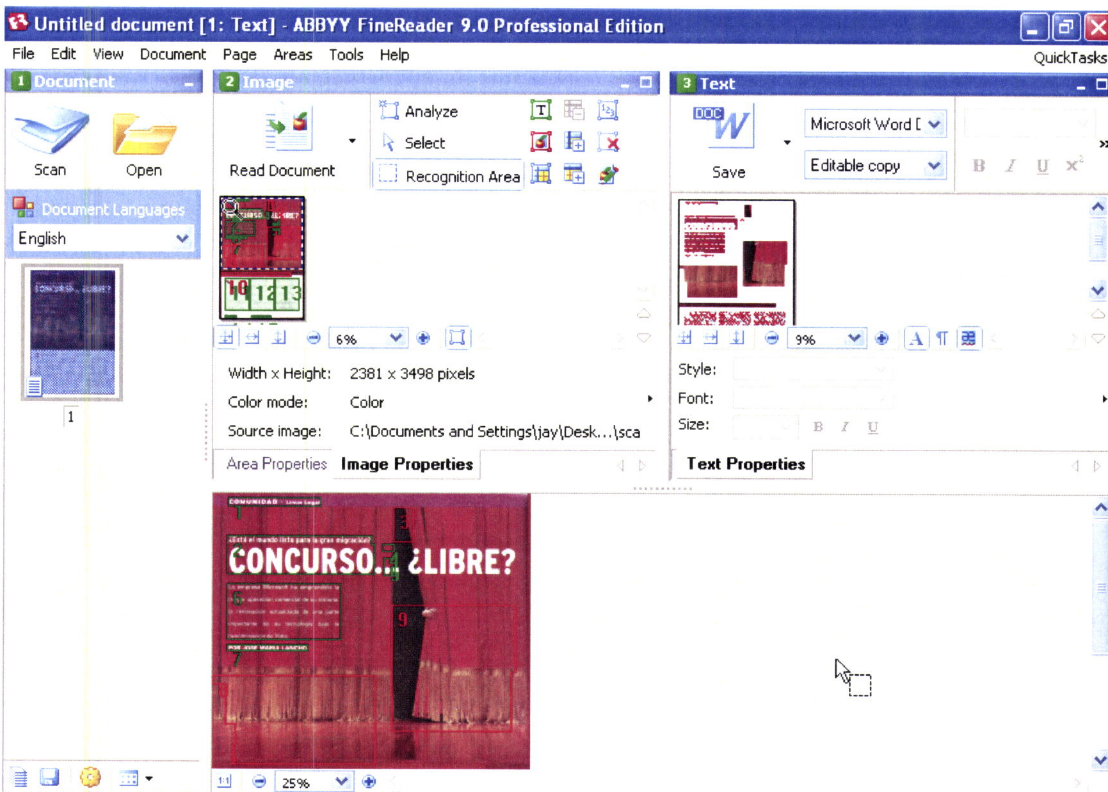


Figure 6.19: Test: Linux Magazine with FineReader

6.3 General Appreciation

All but one of the images tested and presented in this section do not have a plain white background. Because they were scanned with other pages on top of them and have contents on the other side of the sheet, those contents still appear in a translucent way in the resulting image.

Like the tests have shown, this is not a problem has only the real page contents are considered. This means that the contrast function works as intended.

Tables were not considered when developing this project as it's target was actual text documents. If a spreadsheet, for example, is attempted to be converted, the result would be something like what happened for the box on the bottom right on the "The Search" book page test.

As mentioned, these tests were supposed to have no previous configuration or manual intervention on the programs' settings. This means that whatever the results were, they could always be corrected manually by the user until the documents were as most similar to the original ones as they could.

The graphical user interface is an important feature to be considered in the tested solutions.

OCRFeeder's GUI was designed according to the GNOME Human Interface Guidelines like described in the Section 5.1. It is meant to be simple, intuitive and easy.

OmniPage's GUI, on the other hand, presents kinds of widgets and organizes those in a not very familiar way. For example, the big buttons under the tool bar contain a combo box under them which is not a very usable interface.

FineReader's interface is more clear than OmniPage's but still, all the information and widgets present in the main areas make it look less clear than OCRFeeder's. Of course one needs to consider that both these two commercial solutions have a more advanced text editor than OCRFeeder.

Chapter 7

Conclusions and Future Work

The purpose of this project was to create a Document Analysis and Recognition and Optical Character Recognition system for the GNU/Linux operating system that allows to perform automatic zoning, recognition of the type zones and text and exportation to an editable format.

This purpose was clearly accomplished and even surpassed as OCRFeeder can be compared with existing state-of-the-art solutions, either free and commercial ones with a number of years of continuous development, like the previous chapter proved with the tests.

The modular and extensible project architecture guarantee the project longevity by make future work on the system easier, cleaner and more organized.

Comparing to the most modern systems it seems to be the first one that offers OpenDocument Text exportation and uses this as the primary exportation format. It is safe to affirm that OCRFeeder brings most of the features of commercial solutions to GNU/Linux, hence being a pioneer in what comes to having a graphical user interface, automatic layout analysis, manual correction/zoning and optical character recognition on this operating system. Even on the Microsoft Windows and Apple Mac OS operating systems, the free solutions found do not have the functionalities this project offers.

7.1 Problems

Although very functional, OCRFeeder obviously presents some problems. This section presents some of the problems and ideas for solutions to solve them.

Problem: The font size sometimes is not the correct one because the image is skewed (which is normal when scanning images).

Possible solution: By being skewed, the vertical size of the text appears bigger than it really is because algorithm to find the font size goes line by line checking for the contrast (see page 49). The solution for this is to use the Unpaper tool to correct the image's skew before performing the layout analysis and font size detection.

Problem: Text in columns is recognized as being only one paragraph (single column).

Possible solution: This problem is related to the window size and hence, a smaller window size must be manually set. This can be done by using the *Preferences* menu under the *Edit* menu.

Problem: Images are recognized as parts of a paragraph.

Possible solution: This problem is usually because of the window size and the solution is the same as the one for the previous presented problem.

However, if the case is an image that is floating in the middle of the text (which is sometimes common in magazines) than the problem cannot be easily solved. A possible solution for this that can be implemented in future work is to let the selectable areas overlap in the graphical interface and when they do, the contents of one area are not included in another. That is, if an area contains a whole paragraph and another area contains an image that is floating in the middle of the paragraph, than the image is erased (not considered) when the text recognition is performed for the paragraph area.

Problem: Paragraphs with initials are set a wrong font size.

Possible solution: A possible solution for this problem that might be implemented in the future is to vertically clip the paragraph image by half and perform the font detection algorithm with the right part. Since in most cases the initial does not occupy half for the paragraph horizontally, by considering only the right half of the paragraph it should result in the correct font size.

7.2 Future Work

In future work, a solution to the problems mentioned above will be studied, the HTML exportation will be improved and several extensions will be developed like support for tables and spreadsheets.

Regarding the easiness of adding a document generator, exportation to other popular document formats – like PDF, LaTex or ReStructuredText – is likely to be one of the first extensions developed.

A useful feature would be to increase the input sources, that is, to make it able to get the input images directly from a scanner device or a web cam.

A feature that would come against the principle of automatically detecting the documents structure but could save time by improving the performance would be to allow the user to set a pre-defined document structure for a set of documents. For example, by knowing if a set of images have a two-column layout, a simple adjustment of the two-columns could be done for each of the images and hence saving time by not trying to detect every content in the image.

Porting the project to the Windows and Mac OS operating systems may also become a reality. Making OCRFeeder available on more operating systems will surely increase its usage and, since it is Free and Open Source Software, have more developers contributing to it.

Bibliography

[1] Herbert Holik. *Handbook of Paper and Board.* Wiley-VCH, 2006.

[2] Michael Loewe and Eva Wilson. *Everyday Life In Early Imperial China.* Hackett Publishing, 2005.

[3] Jan Seaman Kelly and Brian S. LindblomHerbert Holik. *Scientific Examination of Questioned Documents.* CRC Press, 2nd edition, 2006.

[4] Nikolaos G. Bourbakis. *Artificial Intelligence Methods and Applications.* World Scientific Pub Co Inc, 1992.

[5] Simone Marinai and Hiromichi Fujisawa. *Machine Learning in Document Analysis and Recognition.* Springer, 2008.

[6] Mori et al. Historical review of ocr research and development. In *Proceedings of the IEEE*, 1992.

[7] Mohammed Cheriet et al. *Character Recognition Systems: A Guide for Students and Practitioners.* Wiley-Interscience, 2007.

[8] K. Y. Wong, R. G. Casey, and F. M. Wahl. Document analysis system. In *IBM Journal Res. Dev.*, 1982.

[9] C. F. Goldfarb. *The SGML Handbook.* Oxford University Press, 1990.

[10] Robert P. Futrelle et al. Document analysis, understanding, and knowledge access. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, 1991.

[11] Science Applications Intl. Corp. Capture station simulation: Lessons learned, final report, for the licensing support system. Technical report, November 1990.

[12] Kristen Summers. Toward a taxonomy of logical document structures. In *In Electronic Publishing and the Information Superhighway: Proceedings of the Dartmouth Institute for Advanced Graduate Studies (DAGS)*, pages 124–133, 1995.

[13] Udo Kruschwitz. *Intelligent Document Retrieval: Exploiting Markup Structure.* Springer, 2005.

[14] K. Taghva, A. Condit, and J. Borsack. Autotag: a tool for creating structured document collections from printed materials. Technical report, 1998.

[15] ABBYY. Abbyy history. http://www.abbyy.com/company, accessed on November 20th, 2008.

[16] Mark Lutz. *Programming Python.* O'Reilly, 3rd edition, 2006.

[17] Guido van Rossum et al. Extending and embedding the python interpreter. http://docs.python.org/ext/ , accessed on October 1st, 2008.

[18] Andrew Krause. *Foundations of GTK+ Development.* Apress, 2007.

[19] Mark Lutz and David Ascher. *Learning Python.* O'Reilly, 2nd edition, 2003.

[20] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell.* O'Reilly, 3rd edition, 2004.

[21] Karen Cegalis. Opendocument overview. http://opendocument.xml.org/overview , accessed on September 30th, 2008.

[22] OpenDocument XML.org. History of opendocument. http://opendocument.xml.org/milestones , accessed on September 30th, 2008.

[23] OpenDocument Fellowship. Odfpy - python api and tools. http://opendocumentfellowship.com/projects/odfpy, accessed on September 30th, 2008.

[24] Stig HackVän. Interview with l. peter deutsch. *USENIX association's monthly ;login: Magazine.*, 24(5), October 1998.

[25] Werner Backhaus, Reinhold Kliegl, and John Simon Werner. *Color Vision: Perspectives from Different Disciplines.* Walter de Gruyter, 1998.

[26] Yannis Haralambous. *Fonts & encodings.* O'Reilly, 2007.

[27] Roger Hersch. *Visual and Technical Aspects of Type.* Cambridge University Press, 1993.

[28] Carolyn Snyder. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces.* Morgan Kaufmann, 2003.

[29] Bryan Clark et al. Gnome human interface guidelines. http://library.gnome.org/devel/hig-book/2.24/index.html.en, accessed on October 5th, 2008.

[30] Benjamin Tice Smith. Spirits guy. *Make: Magazine*, 11, August 2007.

[31] Andy Oram and Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think.* O'Reilly, 2007.

[32] John Battelle. *The Search: Como o Google Mudou as Regras do Negócio e Revolucionou a Cultura.* Casa das Letras, 2006.

[33] José María Lancho. Concurso... ¿libre? *Linux Magazine*, (25), March 2007.

Appendix A

Example of a project XML file


```
<ocrfeeder>
    <pages>
            <PageData>
            <data_boxes/>
                <pixel_height>1169</pixel_height>
                <resolution>(100, 100)</resolution>
                <image_path>/home/user/Desktop/test1.png</image_path>
                <pixel_width>826</pixel_width>
        </PageData>
        <PageData>
            <data_boxes>
                    <DataBox>
                    <text>This is the detected text</text>
                    <height>160</height>
                    <width>748</width>
                    <text_data>
                            <TextData>
                        <style>STYLE_NORMAL</style>
                        <line_space>0.0</line_space>
                        <angle>0.0</angle>
                        <weight>WEIGHT_NORMAL</weight>
                        <justification>0</justification>
                        <face>Sans</face>
                        <letter_space>0.0</letter_space>
                        <size>13</size>
                        </TextData>
```

```
        </text_data>
        <y>267</y>
        <x>962</x>
        <type>1</type>
    </DataBox>
    <DataBox>
        <text>
            Another example
            of detected
            text!
        </text>
        <height>481</height>
        <width>641</width>
        <text_data>
                <TextData>
                <style>STYLE_NORMAL</style>
                <line_space>3.12</line_space>
                <angle>0.0</angle>
                <weight>WEIGHT_NORMAL</weight>
                <justification>0</justification>
                <face>Sans</face>
                <letter_space>0.0</letter_space>
                <size>10</size>
            </TextData>
        </text_data>
        <y>534</y>
        <x>267</x>
        <type>1</type>
    </DataBox>
    <DataBox>
        <text/>
        <height>481</height>
        <width>534</width>
        <text_data>
                <TextData>
                <style>STYLE_NORMAL</style>
                <line_space>0.0</line_space>
                <angle>0.0</angle>
                <weight>WEIGHT_NORMAL</weight>
```

```
                        <justification>0</justification>
                        <face>Sans</face>
                        <letter_space>0.0</letter_space>
                        <size>1</size>
                    </TextData>
                </text_data>
                <y>534</y>
                <x>1337</x>
                <type>0</type>
            </DataBox>
        </data_boxes>
        <pixel_height>3209</pixel_height>
        <resolution>(300, 300)</resolution>
        <image_path>/home/user/Desktop/test2.jpeg</image_path>
        <pixel_width>2480</pixel_width>
    </PageData>
</pages>
<images>
        <image>
        <original_name>/home/user/Desktop/test1.png</original_name>
        <embedded_name>test1.png</embedded_name>
    </image>
        <image>
        <original_name>/home/user/Desktop/test2.jpeg</original_name>
        <embedded_name>test2.png</embedded_name>
    </image>
</images>
</ocrfeeder>
```

Appendix B

Installation and usage

This chapter covers the installation and usage of this project.

B.1 System Requirements

The following list presents the system requirements for this project to run with all its functionalities. Each dependence has the minimum version that should be used. While newer versions are likely to keep working as well, older versions may or may not work as they were not tested.

- Python (version 2.5);
- PyGTK (version 2.13);
- Python Image Library (version 1.1);
- PyGoocanvas (version 0.12);
- Ghostscript (version 8.63);
- Unpaper (version 0.3);

The module ODFPy (presented in Section 3.2.7) is included as part of this project in order to make the installation and execution of the project easier.

Since this project doesn't use a particular OCR engine, no engine was listed as a dependence above. Nevertheless, the project is usable without an OCR engine. The configuration XML file for the engine Ocrad is already included with the project so only what's needed to be installed for a first test is the Ocrad engine itself. In case the user doesn't want to use Ocrad, the configuration file that is placed in the OCRFeeder configuration folder (see Section 4.2.1) must be deleted.

Other engines that might also be considered are the ones presented in Section 2.1.2.

B.2 Installation on Ubuntu

Since Ubuntu[1] is nowadays one of the most used Linux distributions, a complete guide to install and run OCRFeeder on this operating system – its 8.10 version, to be precise – is covered in this section. Since Ubuntu is based on Debian[2] which, together with all its derivates, is on the top of the most used Linux distributions, this guide should be helpful for Debian and other distributions based on it.

B.2.1 Installing the packages

The only packages needed to be installed on Ubuntu 8.10 is PyGoocanvas and Unpaper, the rest of the dependences are already installed in a fresh install of this version of Ubuntu. The engine Ocrad is also installed for the reasons explained in the previous section. To install PyGoocanvas, Ocrad and Unpaper, the following command should be executed as superuser:

```
# apt-get install python-pygoocanvas ocrad unpaper
```

After all of the packages finish the installation, the project is ready to be executed and used.

[1] http://www.ubuntu.com
[2] http://www.debian.org

B.3 Command line usage

Section 5.2 gives an overview of the graphical user interface and its usage not being, for this reason, covered in this appendix.

This section explains the command line usage.

The command line interface part of OCRFeeder aims at users who want to perform quick and unattended conversions of document images to editable formats. It also makes this project usable from other applications.

Two parameters are mandatory: 1) the path to each document image to be processed is given after the parameter *−−images*; 2) the name of the document to be generated is given after the parameter *−−o*. For example:

```
$ ocrfeeder-cli --images ~/image1.png ~/image2.jpeg
    --o converted_document
```

The pages of the generated documents honor the order of the given paths.

It is also possible to specify the format of the document to be generated (HTML or ODT) with the option *−−format*. In case no format is specified, the images will be exported to ODT. Continuing with the example above:

```
$ ocrfeeder-cli --images ~/image1.png ~/image2.jpeg --format HTML
  --o converted_document
```

OCRFeeder Studio can also be launched from the command line. Two options can be used to load images right after the program initiates. Those are *−−images* which will add the images given as the option's arguments and *−−dir* that will add all the images under a given directory path. The options can be used individually or combined, for example:

```
$ ocrfeeder --images ~/image1.png ~/image2.jpeg
    --dir ~/Desktop
```

For any usage the options and parameters can be given in any order.