Implementing Optical Character Recognition on the Android Operating System for Business Cards

Sonia Bhaskar, Nicholas Lavassar, Scott Green
EE 368 Digital Image Processing

***Abstract*—This report presents an algorithm for accurate recognition of text on a business card, given an Android mobile phone camera image of the card in varying environmental conditions. First a MATLAB implementaton of the algorithm is described where the main objective is to optimize the image for input to the Tesseract OCR (optical character recognition) engine. Then a simplified reduced-complexity implementation on the DROID mobile phone is discussed. The MATLAB implementation is successful in a variety of adverse environmental conditions including variable illumination across the card, varied background surrounding the card, rotation, perspective, and variable horizontal text flow. Direct implemntation of the MATLAB algorithm on DROID proved time-intensive, therefore, a simplified version was implemented on the phone.**

I. Introduction

Our objective is to utilize the visual capabilities of the Android mobile phone to extract information from a business card. We use the camera features of the Android to capture data. Extracting information from the business card requires accurate recognition of the text of the business card. Any camera image of the business card would be subject to several environmental conditions, such as variable lighting, reflection, rotation, and scaling (we would desire the same data to be extracted from the business card regardless of the distance from the camera), among others. Our project involves working with the optical character recognition engine Tesseract, since it is the most accurate one currently available. We optimize the environmental conditions to make it conducive to using Tesseract.

This report presents an algorithm for accurate recognition of text on a business card, given an Android mobile phone camera image of the card in varying environmental conditions. First a MATLAB implementaton of the algorithm is described where the main objective is to optimize the image for input to the Tesseract OCR (optical character recognition) engine. Then a reduced-complexity implementation on the DROID mobile phone is discussed. The MATLAB implementation is successful in a variety of adverse environmental conditions including variable illumination across the card, varied background surrounding the card, rotation, perspective, and variable horizontal text flow across the card. Direct implementation of the MATLAB algorithm on DROID is too slow, therefore, a simplified version was implemented.

II. Choice of Optical Character Recognition Engine: Tesseract

Optical character recognition (OCR) is the mechanical or electronic translation of scanned images of handwritten, typewritten, or printed text, to machine encoded text. OCR has been in development for almost 80 years, as the first patent for an OCR machine was filed by a German named Gustav Tauschek in 1929, and an American patent was filed subsequently 1935. OCR has many applications, including use in the postal serivce, language translation, and digital libraries. OCR is currently even in the hands of the general public, in the form of mobile applications.

We are using open source OCR software called Tesseract as a basis for our Business Card Recognition project. Development on Tesseract began in 1985 by Hewlett Packard and continued until 1996. After this time, the project was changed hands and some development was done by the University of Northern Las Vegas, although little or no development was done between the years of 1996 and 2006. The code was eventually released under the Apache 2.0 license as open source. Google is currently developing the project and sponsors the open development project. Today, Tesseract is considered the most accurate free OCR engine in existence. Google has benefitted extensively from Tesseract in their Google books project, which has attempted to digitize the world's libraries.

A. Tesseract Algorithm

The Tesseract OCR algorithm has the following steps:

1) A Grayscale or color image is provided as input. The program takes .tiff and .bmp files natively but plug-ins can be installed that allow the processing of other image compression formats. The input data should ideally be a "flat" image from a flatbed scanner or a near parallel image capture. No rectification capability it provided to correct for perspective distortions
2) Adaptive thresholding (Otsu's method) - performs the reduction of a grayscale image to a binary image. The algorithm assumes that in an image there are foreground (black) pixels and background (white) pixels. It then calculates the optimal threshold that separates the two pixel classes such that the variance between the two is minimal.
3) Connected-component labeling - Tesseract searches through the image, identifies the foreground pixels, and marks them as "blobs" or potential characters.
4) Line finding algorithm - lines of text are found by analyze the image space adjacent to potential characters. This algorithm does a Y projection of the binary image and finds locations having a pixel count less than a

specific threshold. These areas are potential lines, and are further analyzed to confirm.
5) Baseline fitting algorithm - finds baselines for each of the lines. After each line of text is found, Tesseract examines the lines of text to find approximate text height across the line. This process is the first step in determining how to recognize characters.
6) Fixed pitch detection - the other half of setting up character detection is finding the approximate character width. This allows for the correct incremental extraction of characters as Tesseract walks down a line.
7) Non-fixed pitch spacing delimiting - characters that are not of uniform width or of a width that agrees with the surrounding neighborhood are reclassified to be processed in an alternate manner.
8) Word recognition - after finding all of the possible character "'blobs"' in the document, Tesseract does word recognition word by word, on a line by line basis. Words are then passed through a contextual and syntactical analyzer which ensures accurate recognition.

III. Implementation Overview

We developed our own application, called "OCReader", to capture and rectify images to feed to the Tesseract OCR engine. OCReader allows the user to select an image already stored on their Android device or use the device's camera to capture a new image; it then runs through an image rectification algorithm and passes the input image to the Tesseract service. When the OCR process is complete it produces a returns a string of text which is displayed on the main "OCReader" screen, where the user is allowed to edit the results and presented with several options. The user may choose to copy the text to the phone's clipboard, send it by email, or create a new contact. If the user selects to create a new contact, a series of contextual checks are preformed to identify contact information in the results.

Since Tesseract is written in the C++ programming language, it is no trivial task to use it on the Java-based Android OS. The C++ code needs to be wrapped in a Java class and run natively via the Java Native Interface (JNI). Though there is some effort involved, one great benefit to running Tesseract natively is that C++ is substantially faster than Java. After implementing our own rudimentary wrapper, we fortunately discovered that Google had already made a more complete wrapper available to the public as a part of its "Eyes-Free" project, so we began using their version. The Google implementation of Tesseract has a convenient architecture which allowed us to seemlessly interface our preprocessed images with the ocr engine.

While default Tesseract settings are configured in "OCReader", the user has limited control over the settings via a preference panel, where processing speed intervals can be set and training data and languages can be modified.

Before implementing our project on the Android mobile phone, we first developed the image rectification algorithms in MATLAB in order to take advantage of the image processing toolbox and faster processing times.

IV. MATLAB Implementation

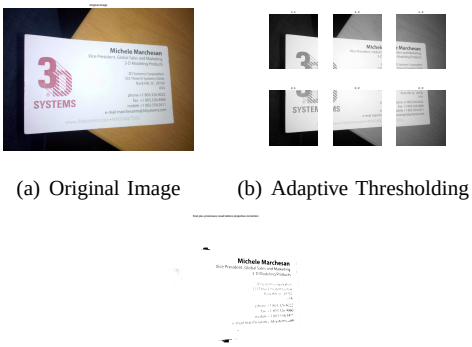
Prior to attempting to implement the algorithm on the Android mobile phone, an algorithm for optimal text recognition by the Tesseract OCR engine was developed. The algorithm consists of several processing steps and a detailed description is given Sections IV-A-IV-B. The algorithm is motivated by the need to optimize the conditions for text recognition before feeding the image to Tesseract. The business cards under study were subjected to a variety of adverse conditions, including variable illumination across the card, varied background surrounding the card, rotation, perspective, and variable horizontal text flow.

A. Preprocessing

Prior to establishing a correspondence map that would allow rectification of any indesirable geometric properties of the image (a business card that is rotated, or is in perspective), the image needed to be preprocessed in order to separate the business card from its background, as well as to establish an outline of the business card to use for the correspondence map. The following steps were used for preprocessing:
1) Adaptive thresholding and card mask.
2) Locating the four corners of the card.
3) Consistency check via text flow directions.
4) Perspective transformation.

Details of these steps follow.

1) Adaptive Thresholding and Card Mask: [David Chen provided a large amount of help for adaptive thresholding; thanks David!]. The image was subdivided into an array of 2×3 blocks, over which the MATLAB command **graythresh** was used to compute local thresholds, all of which were stored into another array. (In order to assure smooth transitions in the threshold array between blocks, the adjacent edge was not subjected to thresholding.) The MATLAB command **imresize** was then used to upsample the threshold array to match the size of the image. The original image was then thresholded using the resized threshold array. The results of the adaptive thresholding are shown in Fig. 1.



(a) Original Image (b) Adaptive Thresholding

(c) Preprocessed Image

Fig. 1. Original Image and Adaptively Thresholded Image

Following the adaptive thresholding, the morphological operations opening, closing, and dilation, using square structuring elements, were performed. This created regions of white

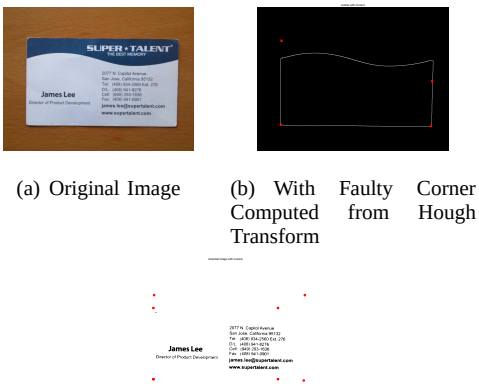
space, and the largest region found was chosen to be the mask of the card. The mask was then stored for use in determining the outline of the card.

The thresholded image was generally a white quadrilateral with black text, surrounded by a black background. This black background was set to white for processing. This image was also stored for use in the next processing step.

2) Outline and Corner Detection: The outline of the card was then obtained from the card mask using a disk structuring element, and the difference of the dilated and eroded images. This difference image was then thresholded. While the outline of the card could have been determined by common edge detectors such as Sobel and Canny, we found that these edge detectors gave edges that were weak and often were beyond repair from morphological operators. Since we desired to use the Hough transform in the next step, we desired to have highly defined lines. Additionally the edge detectors would potentially be computationally expensive to implement on the Android mobile phone.

The Hough transform for line detection was then run on the outline of the business card. The outline in all cases is a quadrilateral, which is defined by four lines. Therefore, the four highest "non-contiguous" peaks in the Hough matrix were taken. From the output of the MATLAB command **hough**, we were given parameters for the four lines in terms of ρ and θ. In order to determine the four corners of the business card, the four equations for the lines were solved for their intersections, and the relevant intersections within the image boundaries were taken. A series of comparisons were then made in order to determine which corner was the "top left" corner, the "top right" corner, and so on. Several approaches could have been used for corner detection, such as the Harris corner detector. Unfortunately, the Harris corner detector was found to be a too sensitive, computationally expensive, and the output of the related MATLAB commands was found unhelpful. If four corners could not be determined, for example if we had a white business card on a white background, the original image was accepted for input to the OCR engine and no perspective transformation was performed.

3) Text Flow Check: In considering several business cards that contained a "pattern", we discovered that the outline of the business card may not follow a straight line, and this could potentially throw off the calculations of the four corners. For this case we use an idea from [3] regarding text flow. First we find a rectangular bounding box for the thresholded image within the card mask by looking for the smallest rectangle that contains all the black pixels. We segment the card within the bounding box into 6 equal-height horizontal strips, and calculate the angle of the horizontal text flow within each strip using Radon transform.The topmost strip text flow angle should be "close" to the angle of the top-line of the card outline (the one connecting the top two corners); the same holds for the bottom-most strip and the bottom line angle of the card outline. If there is any inconsistency, we accept the text flow angle as the correct angle and redraw the top (or bottom) lines of the card outline to redefine the "right" corner(s). [We are unfortunately only able to make a comparison for the horizontal edges of the business card, since business cards typically lack substantial vertical text flow for us to calculate angles from (we would desire a good deal of vertically aligned text).] Once this is done we have four points corresponding to a corner on the card. An example where this is necessary is seen in Fig. 2.



(a) Original Image

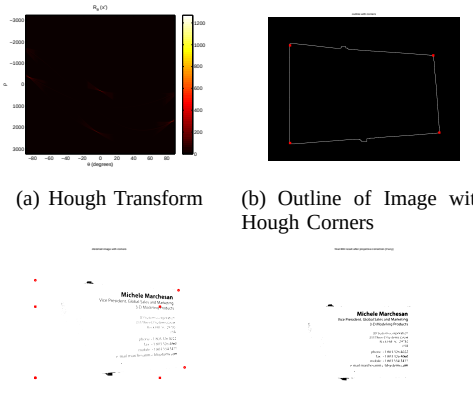
(b) With Faulty Corner Computed from Hough Transform

(c) With Corner Correction from Horizontal Text Flow

Fig. 2. Business Card with Horizontal Text Flow Corner Correction

4) Perspective Transformation: We desire to establish a correpondence map between the image of the business card we have and a standard US business card, whose dimensions follow the ratio $3.5/2$. We fit the lower left corner of this standard business card to the lower left corner from the image, and compute the other three corners of the standard business card, scaling it relative to the size of the image at hand and using the $3.5/2$ ratio.

Once these correspondences are established, we use the MATLAB command **maketform** to make a perspective transform based on the two sets of four corner points. The MATLAB command is then used to perform the perspective transform on the thresholded image as well as the original image (with the background intact). Both these images are input to the post processing step. The steps described can be seen below in Fig. 3.



(a) Hough Transform

(b) Outline of Image with Hough Corners

(c) Image with Corners after Text Flow Correction

(d) Rectified Image

Fig. 3. Geometric Rectification of the Business Card

B. Post Processing

1) Image Bounding, Resizing, and Thresholding: Initial adaptive thresholding was done on the entire image (card plus the background). After perspective correction, we again adaptively threshold the extracted card image (this time only within the rectangular bounding box; we also enlarge it by a scale factor prior to thresholding) to check if it provided a better binarized text enhancement. It turns out that it is not always so. We used the ratio of black pixels within the card boundaries in the binarized text images from the re-thresholded and the original thresholded images as a measure of improvement. If the ratio was > 2.3 (significant change), we accept the re-thresholded image and store it as output for the next step, else we use the initial thresholded image (not significant change: extra "black blobs" appear) and store it as output for the next step.

2) Vertical Segmentation: Tesseract often has trouble reading lines of text that are not horizontally aligned. For this reason, we include an additional post processing step that segements the business card vertically. We sweep a line from the top line at different slopes from different starting points, and search along a 10 pixel wide swath to the right of it. If we find that there are no more than 2 black pixels in this 10 pixel-wide swath along the entire card image, we split the card into into "halves:" left part and the right part. [We do so only if the two split cards are not entirely white: no black pixels]. We see the benefits below in Figs. 4 and 5.

(a) Card with Line Segment (b) Right Segment

(c) Left Segment

Fig. 4. Vertical Segmentation

C. Tesseract Outputs

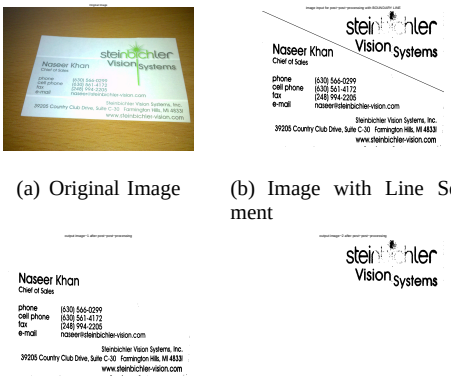
The images were then input to the Tesseract OCR and the output was obtained. The image shown in Fig. 6 provided a nearly ideal output.

1) Tesseract Output, Fig. 6:

```
GARY A. RUST, M.D.   A
EAR. NOSE & THROAT
BOARD CERTIFIED IN OTOLARYNGOLOGY/HEAD & NECK SURGERY
45 CASTRO SI. STE. 210
SAN FRANCISCO, CA 94114
(415) 626-4900
FAX (415) 626-4901
```

2) Tesseract Output, Original Image from Fig. 1:

```
Michele Marchesan
Vice President, Global Sales and Marketing
3-D Modeling Products
_.t 3D Systems Corporation
  4   333 Three D Systems Circle
```



(a) Original Image (b) Image with Line Segment

(c) Left Segment (d) Right Segment

Fig. 5. Another Case of Vertical Segmentation

(a) Original Image (b) Rectified and Post Processed Image

Fig. 6. Original and Rectified Image

```
  "   Rock Hill, SC 29730
?     USA
C S phone +1 803.326.4022
fax +1 803.326.4-060
mobile +1 803.554301
e-mail marchesanm@3dsystemsc0h
```

3) Tesseract Output, Original Image from Fig. 2:

```
James Lee
Director of Product Development
2077 N. Capitol Avenue
San Jose, Califomia 95132
Tel: (408) 934-2560 Ext. 276
D/L: (408)941?82‘/6
Cell: (949)293?1636
Fax: (406794%-0901
jamos.|ee@supenaIent.com
W\Wl.$Up0fl8|6I\i.C0m
```

4) Tesseract Output, Original Image from Fig. 5:

```
SCSI!
Visi0n SyS,em$
Naseer Khan
Chief oi Sales
lhone (630) 566-0299
cell phone (630) 56l -4l 72
tax _ (248) 994-2205 e
e-mail naseer@sleinbichler-vision.com e
Steinbichler Vision Systems, Inc. I
39205 Country Club Drive, Suite C-30 Farmington Hllls, Ml 4833l r
 www.steinbichIer-vision.com
```

V. Conclusion

Reliably interpreting text from real-world photos is a challenging problem due to variations in environmental factors. Even the best open source OCR engine available (Tesseract) is easily thwarted by uneven illumation, off-angle rotation and perspective, and misaligned text, among others. We have shown that by correcting for these factors it is possible to dramatically improve an OCR technology's accuracy. A simple adaptive thresholding technique was used to reduce the effect of illumination gradients. Rotation and perspective were determined by processing a Hough transform. Text alignment was improved by performing vertical segmentation.

While some of these described operations proved to be too intensive for inclusion in version 1.0 of the Android application, we discovered it was possible to secure a majority of the accuracy gains by correcting for only illumination and rotation in the constrained environment of an Android user. The result is a highly functional and fun to use mobile application!

VI. Future Work

Due to time and resource constraints we were unable implement all of the preprocessing steps designed in MATLAB on the mobile device. The MATLAB-based algorithm makes very heavy use of morphological and other computationally expensive operations which are not well suited to the phone's limited resources. As a result, we decided to implement an abbreviated version of the described algorithm which performs only the most important of the processing steps.

The processing algorithm used on the Android device performs the following steps:

1) Image resize to 800x600 to ease burden on mobile device.
2) 2x3 adaptive thresholding (using our own JJIL addition for Otsu thesholding).
3) Hough transform (specialized to horizontal edge pixels)
4) Flat rotation

Notably, we do not implement a perspective transformation, which saves considerable computational expense (especially since we do no need to identify the corners of the card). The lack of perspective transformation on the Android platform is arguably unimportant, since the user can simply take another photo if their first attempt is heavily distorted. A simple

flat rotation will correct for most of the user's error in a majority of situations. Also, trusting the user to determine the appropriate perspective prevents the app from making assumptions about the content (that it is in fact a business card) and allows it to be used as a more general OCR tool.

Probably the most important feature left out of the Android image rectification algorithm is vertical segmentation. This step considerably improves Tesseract's accuracy, but it is also extremely computationally expensive. We firmly believe, however, that by optimizing the current matlab implementation of this operation and implementing it in C++ (via the JNI as with the OCR service) this step could be brought to the Android phone.

ACKNOWLEDGMENTS

The authors would like to thank David Chen for his advice throughout this project as well as Professor Bernd Girod.

CODE RESOURCES

Eyes-Free Project (Speech Enabled Eyes-Free Android Applications, http://code.google.com/p/eyes-free/) JJIL (Jon's Java Imaging Library, http://code.google.com/p/jjil/) GMSE Imaging, JDeskew

REFERENCES

[1] H. Li and D. S. Doermann. *Text Enhancement in Digital Video Using Multiple Frame Intergration*. ACM - Multimedia 99, Orlando, Florida, pages 19-22, 1999.

[2] G. Zhu and D. Doermann. "Logo Matching for Document Image Retrieval," *International Conference on Document Analysis and Recognition (ICDAR 2009)*, pp. 606-610, 2009.

[3] J. Liang, et. al. "Geometric Rectification of Camera-captured Document Images," *IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 591-605, July 2006.*

[4] B. Girod. "EE 368 Digital Image Processing Notes," EE 368 Digital Image Processing Spring 2010.

[5] D. Chen. Email correspondence. May-June 2010.

APPENDIX

A. Sonia Bhaskar

Sonia Bhaskar's work is detailed in Section IV, MATLAB Implementation. She developed the algorithms (as described in Section IV) for the image processing and implemented the MATLAB side of the project.

B. Nicholas LaVassar

Nick LaVassar produced the Android application "OCReader". He researched a vast number of image processing techniques and Android resources to determine the best way to create an app which met the group's goal of making a business card reader. He learned it was necessary to modify the original algorithm to operate smoothly on the Android phone and created a substitute which suited the platform. While coding OCReader, he found it necessary to code many image processing operations, including one which he plans to contribute to the JJIL Project.

C. Scott Green

Scott Green researched the Tesseract source code. He also reproduced a rudimentary version of Tesseract in MATLAB which processed flat images and found characters similar to the Rice example in class. However, full reproduction was found infeasible in the time alotted. He coded the shell to enable bitmap transformations, added new contextual rules for contact creation in Nick's Android application (multiple line addresses, multiple telephone number distinctions (cell, fax, work, home), and contact job title extraction), and added a home screen and help window. He also made the poster presentation.