

Abstract

The Optical Character Recognition (OCR) technology allows for scanning and conversion of documents into machine editable text, and has seen a dramatic improvement in the past two decades, with regards to speed, accuracy and language support. Experimentation with OCR-Engines however, exposes the inability to perform text recognition on a scene image containing non-textual elements. This project looks to improve upon this existing technology, enabling it to read text from any scene image. The project is more of a Type I and less of Type II, in other words it is more about software development with some investigation required. It requires significant amounts of programming and experimentation on the device for better results.

Text localization algorithms are implemented on the iPhone mobile device for the development of an application. The user is to take a picture of a scene that contains text, and the application should extract any text regions available and translate them from one language to another. Technologies employed for the task include OpenCV - an image processing and computer vision library of programming functions, Google's Tesseract OCR-Engine and Translation Service.

Various algorithms have been implemented and evaluated towards text extraction. The approach of Maximally Stable Extremal Regions (MSER) detection proved to outperform one of the most widely used approaches towards text extraction, which is based on edge detection. This technique has been tested both on images with plain text on monochrome background, as well as in images taken from a natural environment. Both techniques produced many non-textual regions which needed to be removed with further processing. Region filters were employed to remove regions which don't satisfy text properties. Texture filtering assisted with the removal of regions resembling text. Finally, a perceptual text grouping method improved the results by checking for character-to-character and word-to-word relations between regions.

The final application is assessed in terms of improvement over basic-OCR on scene images, and compared to existing solutions already available for the iPhone mobile device.

- This project involved the development of a complete iPhone application with text extraction and translation capabilities.
- The edge detection-based approach has been implemented and compared against the Maximally Stable Extremal Regions detection, with regards to text extraction performance.
- The existing MSER algorithm available through the OpenCV Library has been modified for speed and efficiency.
- Special sets of region and texture filters have been created which help towards a more successful text extraction.
- A perceptual text grouping algorithm has been developed to further assist with this task and the results of the final application were compared to other work already available.

Acknowledgments

I would like to thank:

- My project supervisor Dr Majid Mirmehdi for his valuable comments and guidance throughout the project.
- The course director Dr Steve Gregory for his general assistance regarding this Masters Course.
- My family and friends for their support during the period of this project, and my studies in general.

Contents

1. Introduction.....	1
1.1 Objective	1
1.2 Image Segmentation	2
1.3 Text Extraction	3
1.4 Optical Character Recognition	3
1.5 Translation - Web Services.....	3
2. Theoretical Background.....	5
2.1 Thresholding.....	5
2.2 Connected Components Labeling	5
2.3 Edge Detection and Canny Algorithm	6
2.4 Maximally Stable Extremal Regions	7
2.5 LU Transform.....	9
2.6 Noise Reduction	11
2.7 The iPhone mobile device	12
2.8 Development on the iPhone	13
2.9 Related Work in the field of Text Extraction.....	15
2.10 Related Work – iPhone Applications.....	18
2.10.1 Creaceed Prizmo	18
2.10.2 Keystone Technology – CapnTrans	19
2.10.3 Quest Visual – Word Lens	20
3. Basic framework implementation and assessment.....	21
3.1 Implementation Considerations	21
3.2 Libraries and Algorithms	21
3.2.1 OpenCV.....	21
3.2.2 Noise Reduction	21
3.2.3 Maximally Stable Extremal Regions	22
3.2.4 Canny Edge Detection	22
3.2.5 LU Transform	22
3.2.6 Tesseract.....	22
3.2.7 Google Translate	23
3.2.8 Parameters and Initialization	23
3.3 Preliminary Results and Evaluation.....	24
3.4 Observations and Decisions	25
4. Basic model enhancement and text extraction	27
4.1 MSER	27
4.2 Noise Reduction	27
4.3 Filtering	27
4.3.1 Region Filters:.....	28
4.3.2 Texture Filter:	28
4.4 Text Grouping.....	29
4.5 Optimizations	30
4.5.1 MSER and Filtering	30
4.5.2 Analysis of the image	31
4.5.3 Google’s Spell Checker	34
5. Mobile Application	35
5.1 iPhone Device Issues	35
5.2 Application	35

5.2.1 Features	35
5.2.2 User Interface	36
5.2.3 Final Product – Release	37
6. Experimental Results	38
6.1 Noise Reduction	39
6.2 MSER	40
6.3 Final Product	47
6.3.1 Application vs. Basic OCR	47
6.3.2 Application vs. other iPhone Software	48
6.3.3 Application Benchmark on iPhone 3GS and 4	50
7. Discussion	51
7.1 Basic Framework	51
7.2 Framework enhancement	52
7.3 Mobile Application	52
7.4 Critical Evaluation	53
8. Future Work – Conclusion	54
8.1 Further Directions	54
8.1.1 Improve on MSER	54
8.1.2 Improve on Filtering	54
8.1.3 Improve on Text Grouping	54
8.1.4 OCR Guided Text Extraction	54
8.2 Conclusion	55
Bibliography	56
Image Sources	58
Appendix: Source Code	59

1. Introduction

The most informative regions of an image are perhaps those which contain text. For us humans, it is an easy task to focus and read text of any size, orientation, color and style. It is many times the case though that it might be in a language we are not familiar with, meaning we cannot process the information available to us. Consider a scenario where you visit a foreign country for vacation, and stumble upon a sign similar to the one shown in Figure 1, which seems fairly important. You may know which language is used in this country, but cannot speak it yourself. Hence you need a way to translate the text in front of you to take an important decision. Carrying a dictionary with us everywhere we go is not always a desirable solution. We would instead prefer to use something we already carry with us such as our mobile phone for other tasks as well. That is the reason why most people nowadays prefer to use their mobile device for tasks such as reading books, keeping memos, taking a picture and listening to music. In this situation it would be desirable to have an application on our mobile phone which could read the text from an image we take, and translate it for us.



Figure 1: Scene Image with Text

Current technology called optical character recognition allows for computers to read text from an image and reproduce digital text for editors. For this to happen, an image should contain regions of clearly visible text on a monochrome background, free of non-textual elements. But text is not found only in books, or white walls with signs, hence we need a way to take this technology a step further, and enable it to read text from any natural scene.

1.1 Objective

It is useful to indicate the reasons why text extraction is necessary prior to feeding an image to the OCR engine for text recognition. As stated earlier, modern OCR engines yield very good results on images with text only, such as a picture of a page of a book, a poster on the wall or a big street sign. Once other elements are introduced in the picture, the results begin to degrade until finally there is no readable text result returned to the user.

Assume Figure 1 is a picture taken from your mobile phone's camera on your vacation to the UK. The following table shows the results of the OCR engine Tesseract when fed with parts of the image: first with only the sign region, and then progressively adding the rest of the elements of the picture.

Input Image	Input Image	Input Image
Tesseract Result	Tesseract Result	Tesseract Result
NO W\DE VEHICLES Do Not Fo\low SAT NAV Very narrow road	wx :fs 179 T' φ~< >, \n S , \ X = xy- ,,	\2 \\.

Table 1: Tesseract Results with no processing

As it is not always desirable to take pictures containing only text and because in many cases the background of the text may not be monochrome but a texture instead, there is the need to extract the text only from the image and feed it to the OCR engine for results. Preliminary work with image processing yields the following results as indicated in Table 2.

Input Image	Result after process	Tesseract Result
		A NO WIDE VEHICLES Da Not Follow SAT NAV Vefy narrow road “if 5?

Table 2: Preliminary work and results with MSER

Compared to the results in Table 1 on the second image, we get back something we can work with and further improve as discussed in later chapters. Hence there is a need to create a system with text extracting capabilities from scene images.

1.2 Image Segmentation

Segmentation refers to the process of separating different image elements into segments (or regions), which would ease the process of analyzing. There are many



Figure 2: Original and Segmented Image

approaches to image segmentation. These include classifying regions according to color, border energy (edges), prior knowledge, etc. Figure 2 demonstrates the concept of segmentation: Sky is represented by the light gray region, power mills by dark gray and ground by black color. This particular segmentation is based on prior knowledge, meaning the program takes three input sample images for each region to be represented, in this case the sky, the power mills and the ground. The segmented regions of this image could then be used for further analyzing, editing or display purposes.

For this project, the segmentation step is what will classify text in a region we could extract and use for our purposes. We chose a particular technique called MSER detection which is described later on in this thesis.

1.3 Text Extraction

Following segmentation, text can then be extracted using its properties:

- Clearly defined Edges.
- Contrast with the Background.
- Horizontal or with slight slope.
- Nearly constant character size and distance.

By filtering out everything else that does not seem to represent text, we can obtain textual regions which can then be used further on.

1.4 Optical Character Recognition

Optical Character Recognition (OCR), is the process of translating text (handwritten or typewritten) from an image into machine-encoded text. It is now a standard feature of all home scanner devices. The option to scan a piece of paper “as document” performs character recognition and digitizes text to be saved on a computer for editing.

Tesseract by Google is known as one of the most accurate and commonly used of all open-source character recognition engines, and serves the needs of this project perfectly. It is characterized by speed, accuracy and results evaluation providing confidence values and word validation.

1.5 Translation - Web Services

Google is a leading technology company which offers a large amount of quality web services, mostly free of charge. These include search engines, world and road maps, text to speech, text translation, auto-completion and spell checkers. Google API is the programmers’ interface which provides for an easy way to use of these services in applications.

Google Translate offers many features such as spell check and correction, text to speech and language detection. Because of the continuous improvement of this service by online communities, the results are far more accurate than using a simple offline word translator.

Figure 3 demonstrates Google's online translation service, along with the spell correction feature.

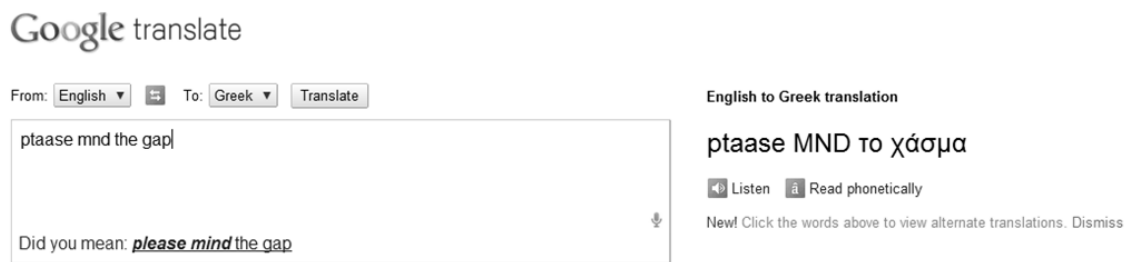


Figure 3: Google's Translation Service

2. Theoretical Background

This chapter reviews various image processing methods and concepts mentioned earlier that are used for this project, as well as the systems involved in the implementation part.

2.1 Thresholding

Thresholding in image processing is the simplest method to segment an image. It is many times useful to separate the regions of an image to objects of foreground, and the background. With an input grayscale or color image, usually the output of thresholding would be a binary (black and white) image. In the simplest

implementation, the intensity values of all pixels are compared to a threshold (intensity threshold). Pixels with intensity values larger or equal to the threshold are set as white, while those with values less than the threshold are set to black. In this case we determine all white pixels to belong to objects of the foreground, and all black pixels to be part of the background. Figure 4 is a simple thresholding of an image with threshold at value 125. More complex implementations could involve specifying multiple thresholds to identify different objects or colors on an image.

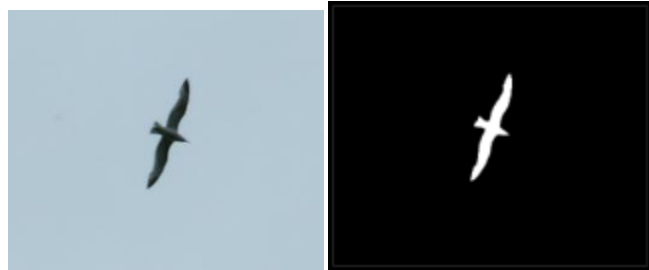


Figure 4: Original and Thresholded Image at 125

2.2 Connected Components Labeling

In image processing, the connected components algorithm finds regions of connected pixels which have the same color/intensity value. Think of this process like the flood-fill tool on a paint program, which fills a closed region with a selected color. This step helps to correctly segment text to help with tracking at a later step. Figure 5 demonstrates the result of connected components labeling on an image containing text. Each closed connected component region is marked with a different color denoting a different region. Connected Components (CoCos) can also be referred to as contours.



Figure 5: Connected Components Labeling

2.3 Edge Detection and Canny Algorithm

Edge Detection is fundamental in image processing and aims to identify parts in an image at which the brightness changes sharply [1] or more formally has discontinuities. There could be many applications for edge detection in image processing, but the most common reason why it is used is to find those curves in an image which indicate bounds between different objects.

The Sobel operator computes an approximation of the gradient of the image intensity at each point, returning the direction of the largest increase from light to dark, and the rate of change in that direction. The operator uses the convolution of two 3x3 kernels with the image, to calculate the approximation of the derivatives on horizontal and vertical changes. The convolution masks used in the Sobel operator are shown below:

$$\begin{array}{ccc} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{array} \quad \begin{array}{ccc} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{array}$$

$G_x \qquad \qquad \qquad G_y$

The magnitude (or edge strength) of the gradient are calculated using:

$$G = \sqrt{G_x^2 + G_y^2} \quad (1)$$

The Canny edge detection algorithm was developed by John F. Canny in 1986 [2], and performs noise removal, gradient detection and edge strength analysis. A Gaussian filter is applied to the image, and the algorithm tries to find the intensity gradient of the image.

An example Gaussian filter applied to reduce noise would be (with $\sigma=14$):

$$B = \frac{1}{159} \begin{array}{ccccc} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{array} * A$$

The next step to detecting edges with the Canny algorithm is using the Sobel Operator as described above to find edges strength. To find the edge direction, the following formula is used:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Then the edge orientation is resolved by relating the edge direction to one that can be traced in the image, depending on which direction it is closest to (e.g. 30 degrees is closest to 45, 120 closest to 135 and so on). The algorithm then applies non-maximum suppression which is used to suppress any pixel value along the edge direction that is not considered to be an edge. Finally,

hysteresis is used to eliminate streaking, that is, the breaking up of an edge contour caused by the operator output fluctuating above and below the threshold.

The following figure shows the results of the Canny edge detection algorithm applied on an image:



Figure 6: Canny Edge Detection

Marcelo G. Roque *et al* showed [3] that the Sobel edge detection algorithm was faster when implemented on the iPhone 3GS, and dramatically improved the performance of real-time applications. This is mainly because Canny involves more steps to detecting edges more precisely. In their work, Marcelo G. Roque *et al* optimized the Sobel edge detector algorithm by changing the 5x5 Gaussian mask filtering to a separable Gaussian filtering. This resulted in reducing the operations per pixel from 25 to 6, which in turn reduced the calculation time from 225 to 102 milliseconds for a 320x480 image.

To help understand how edge detection helps in better results when detecting and extracting text, the following image has been used to extract edges.



Figure 7: Edge Detection on a picture with threshold 45 and 100

The two resulting images are canny outputs with threshold 45 and 100. Because text has clearly defined edges, its edges are preserved even at high thresholds. This is the key to the extraction and segmentation of regions of interest that contain text.

2.4 Maximally Stable Extremal Regions

In the previous chapter we introduced the concept of image segmentation and discussed its importance especially when it comes to detecting text. MSER is a technique initially proposed for

finding correspondences of elements from two images. For our purposes we use it as a type of segmentation technique to classify MSER regions of interest, which include text.

By definition, MSER refers to Maximally Stable Extremal Regions on an image. To visualize the concept of MSER regions and its use in this project, imagine a picture converted to grayscale subjected to all possible values of thresholding (0-255) as described in section 2.1. Regions representing various elements on the image will keep growing until finally merging and becoming black, meaning they will be considered as part of the background. We are interested in those regions which maintain a nearly constant area size over different values of thresholding. Those regions are said to be Maximally Stable Extremal Regions.

In mathematical terms, MSER can be described as [4]:

Let I be a real function on image domain $D \subset \mathbb{Z}^2$.

An adjacency relation $A \subset D \times D$ is needed for MSER. Usually, 4 or 8 bit connectivity is used.

Consider a **Region** Q to be a contiguous subset of D , i.e. for each $p, q \in Q$ there is a sequence:

$p, a_1, a_2, \dots, a_n, q$ and $pAa_1, a_iAa_{i+1}, a_nAq$.

(Outer) Region Boundary $\partial Q = \{q \in D \setminus Q : \exists p \in Q : qAp\}$, i.e. the boundary ∂Q of

Q is the set of pixels being adjacent to at least one pixel of Q but not belonging to Q .

An **Extremal Region** $Q \subset D$ is a region that for every $p \in Q$ and every $q \in \partial Q$ one has

$I(p) > I(q)$ (maximum intensity region) or $I(p) < I(q)$ (minimum intensity region).

To define a **Maximally Stable Extremal Region (MSER)**, let $Q_1, \dots, Q_{i-1}, Q_i, \dots$ be a sequence of nested extremal regions, i.e. $Q_i \subset Q_{i+1}$.

Extremal region Q_{i^*} is **maximally stable** if and only if the area change rate $q(i) = \frac{|Q_{i+\Delta}| - |Q_{i-\Delta}|}{|\Delta|}$ has a local minimum at i^* ($|\cdot|$ denotes cardinality). Δ (variation) is a parameter of the method.

These regions are of particular interest because of their properties [4]:

- Invariance to affine transformation of image intensities.
- Covariance to adjacency preserving (continuous) transformation $T : D \rightarrow D$ on the image domain.
- Stability, since only extremal regions whose support is virtually unchanged over a range of thresholds is selected.

- Multi-scale detection. Since no smoothing is involved, both very fine and very large structure is detected.
- The set of all extremal regions can be enumerated in $O(n \log \log n)$, where n is the number of pixels in the image.

The MSER algorithm proceeds by sorting all the pixels by their intensity. Then it iteratively places pixels in the image and updates the list of connected components and their areas. By selecting intensity levels that are local minima of the area change rate as thresholds, MSERs are produced [5].

The MSER algorithm runs on the image twice and produces MSER+ and MSER- regions. MSER+ are light regions over dark background. Imagine white text on a black sign; the regions of text are declared as MSER+. To detect dark regions over light background, MSER- is obtained by applying the algorithm to the inverse of the original image (or negative). That would detect black text on a white sign. Because of the way MSER is performed (thresholding), both of these passes are needed to obtain regions of interest, especially textual.



Figure 8: Original Image and MSER

Figure 8 demonstrates the results of the MSER algorithm on an image. MSER+ regions are represented in white, while MSER- in green. All the letters (or otherwise MSERs) which have a light background were detected in the first pass (+), while the ones with a dark background were detected in the second pass (-). It is then up to further processing to determine what is text and what should be filtered out. This process will be discussed in the next section.

Our implementation uses a similar model to the one proposed by Donoser *et al* [6]. A connected component (CC) tree is constructed representing all CCs resulting when various thresholds to the grayscale image. The rate of change in the area of the connected components is also calculated for each node. Nodes towards the root of the tree are identified as local minima. We follow the work of Merino *et al* [7] with an algorithm which runs the image twice and produces MSER+ and MSER- regions.

2.5 LU Transform

Every object around us has some type of texture. Some characteristic textures are those of wood, grass, stone and so on. The LU Transform is a texture descriptor which yields results according to roughness of image regions. It was originally proposed by Targhi *et al* [8] and compared against similar



Figure 9: Gray Value Image Patch ($w=20$)

descriptors such as the SVD-transform and the Eigen-transform. It proved to be faster and produce similar results.

The first step to perform the LU Transform is to split the image in smaller patches. Patches are square windows of a preset size w . For every pixel on that $w \times w$ window, get the gray value (0-255) and construct the $w \times w$ matrix. A simple LU Decomposition is performed of the type: $A = L U P$, where A is the image window matrix, L and U the lower and upper triangular matrices, and P a permutation matrix. Consider the 3×3 ($w=3$) A matrix with gray values below.

$$\begin{array}{ccc|ccc|ccc|ccc} \left| \begin{array}{ccc} 255 & 125 & 66 \\ 205 & 128 & 68 \\ 185 & 68 & 40 \end{array} \right| & = & \left| \begin{array}{ccc} 1 & 0 & 0 \\ 0.803 & 1 & 0 \\ 0.725 & -0.824 & 1 \end{array} \right| & \left| \begin{array}{ccc} 255 & 125 & 66 \\ 0 & 27.509 & 14.941 \\ 0 & 0 & 4.439 \end{array} \right| & \left| \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right| \\ A & & L & U & P \end{array}$$

The decomposition yields L , U and P matrices. We are interested in the values of the **U diagonal**. The next step for the LU Transform is to sort the absolute values of the diagonal, in this case as $[4.439, 27.509, 255]$, and set l as the number of lower frequency values to skip. Setting $l = 1$ for example, will skip the first lower value of the array, 4.439. The algorithm then proceeds to sum the remaining values and find the mean value.

$$27.509 + 255 = 282.509 \rightarrow \text{Mean Value} = 282.509/2 : \Omega = 141.2545$$

This value, Ω (omega), describes the texture response of the image patch, and can be applied to all pixels as a threshold to filter out the ones with lower response. The mathematical form of the LU Transform can be expressed as:

$$\Omega(l, w) = \frac{1}{w - l + 1} \sum_{k=l}^w |U_{kk}|, 1 < l < w \quad (2)$$

Where w is the window size, l the number of lower frequencies to skip, and $|U_{kk}|$ the absolute value of the diagonal element of matrix U . The values w and l are set empirically after testing with various images.

The following figure demonstrates the LU transform applied on an image with $w=32$ and $l = 8$ as well as a threshold value of 2. The results indicate that the transform does well on regions where the rate of change of texture is high, such as the cheetah fur, and the dirt.

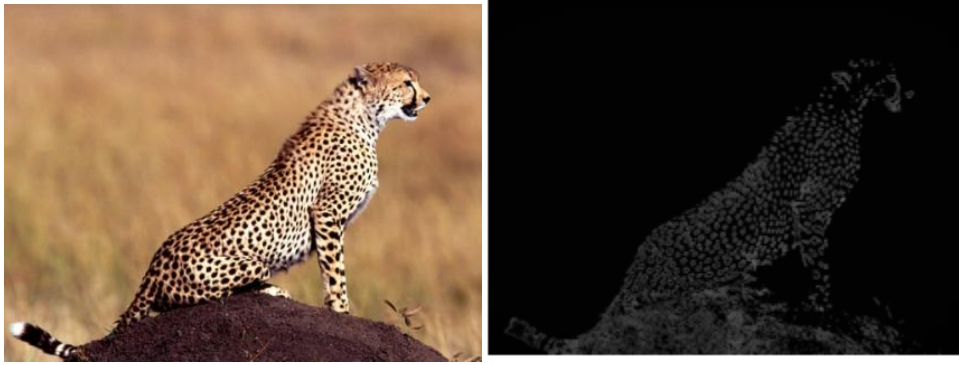


Figure 10: LU Transform $w=32, l=8$

The results of LU Transform are particularly useful for text as well, because of the rate of change of the texture. In spite of the fact that LU Transform is only a texture descriptor with results as shown in figure 10, it cannot be used alone for text detection. It is however a particularly useful filter for non-textual regions produced by initial steps of segmentation.

2.6 Noise Reduction

An important step for better results in image processing is noise reduction. The Canny Edge detection algorithm deals with this problem by using a filter based on the first derivative of a Gaussian, but the same does not apply for the other techniques (MSER, LU). A nice and simple technique to remove noise from an image is by the use of the Median Filter. The main idea behind the median filter is to remove impulsive noise in the image, by running through the signal entry by entry, and replacing it with the median of the neighbouring entries. The figure below demonstrates the results of an applied Median Filter.



Figure 11: Median Filter applied on an image

The algorithm on a grayscale image:

1. Assume window size $w = 5$, image: of dimensions 300×300 .
2. Find the center pixel of the 5×5 window: $(\text{windowWidth}/2, \text{windowHeight}/2 - \text{rounded})$. In this case (2,2).
3. For all windows 5×5 in image, find neighboring pixel values and store in an array.
4. Replace the center pixel value with the median value of the array.
 $(\text{windowWidth}/2, \text{windowHeight}, 2)$.

Another important technique for reducing noise is the Bilateral Filtering. This filter replaces the intensities of pixels by a weighted average of values from nearby pixels. This weight is based on Gaussian distribution and depends on Euclidian Distance and differences in intensities.

In their work, Tomasi *et al* [9] describe a simple and important case of bilateral filtering, the shift-invariant Gaussian filtering, in which both the closeness function c and the similarity function s are Gaussian functions of the Euclidean distance between their arguments.

Mathematically, they describe c as radially symmetric:

$$c(\xi - x) = e^{-\frac{1}{2}\left(d \frac{\xi - x}{\sigma_d}\right)^2} \quad (3)$$

where:

$d(\xi - x) = \|\xi - x\|$: the Euclidean distance.

The similarity function s is analogous to c :

$$s(\xi - x) = e^{-\frac{1}{2}\left(\frac{\delta(f(\xi) - f(x))}{\sigma_r}\right)^2} \quad (4)$$

where:

$\delta(f(\xi) - f(x)) = |f(\xi) - f(x)|$: A suitable measure of distance in intensity space.

The parameters sigma-r (color sigma) and sigma-d (spatial) will be used in the implementation part, as discussed in chapter 3.

2.7 The iPhone mobile device

The iPhone is a multimedia-enabled smart-phone created by Apple Inc. The first model known as the iPhone 2G was released in July, 2009. Since then, 3 new models have been designed and marketed which offer new features and capabilities.

The iPhone 2G and 3G feature an 89mm multi-touch display and a 2 Megapixel camera. The system comprises of a 620 MHz ARM processor with 128 MB DRAM internal memory. It runs on iOS, a mobile operating system developed initially for the iPhone, but has now been extended to support other Apple's devices such as the iPod touch, the iPad and Apple TV. As for connectivity, the phone has Wi-Fi support and has Bluetooth 2.0, but in addition to previous, the 3G model also has assisted GPS. The newer models 3GS and 4 have a faster CPU and larger memory capacity (833MHz/256MB and 1GHz/512MB respectively). In addition, the 4 model has a higher resolution screen, and a front camera. Software-wise, the new models have true multi-tasking capabilities, meaning the ability to switch between applications running at the same time, while older models could only have one foreground application with inability to switch.

For the development of the application in question, iOS version 4.2 was used. We require a device with autofocus since it greatly improves the results of both segmentation and filtering, as well as for the output of the OCR engine. The application targets the iPhone 3GS and 4 devices, and greatly benefits from the improved CPU and memory capabilities they offer. Although the 3G model runs on iOS 4 and could potentially run this software, it has not been tested and therefore cannot be supported, due to the lack of the autofocus feature, which significantly aids the results of this application. Nevertheless, since the application is not about focused pictures, we cannot rule out older devices considering good results can still be produced with a well taken image.

2.8 Development on the iPhone

For the development of an application for the iPhone, one must be using Apple's Mac OS, as well the SDK distributed for developing applications for it. Xcode is a suite of tools which allow developers to create applications for Mac OS X and the operating system of Apple's mobile devices. The programming language used is Objective C. The following information regarding Objective C and the iPhone SDK is part of a previous work of ours in the field of developing an application for the iPhone [10].

Objective C

Objective C was chosen to be the programming language to be used for the development of the application in question, mainly because it is the only option available for an iPhone developer. Objective-C was created primarily by Brad Cox and Tom Love in the early 1980s at their company Stepstone. Apple's Mac OS X and iPhone OS environments use Objective-C, a reflective, object-oriented programming language, which consists of C programming language, but derives its object syntax from SmallTalk. This means that the syntax of every non-object-oriented operations remains the same with C, but object-oriented syntax uses SmallTalk (an object-oriented, dynamically typed, reflective programming language) style messaging. On Mac OS X, Objective C is compiled using the GnuCompilerCollection (a compiler system produced by the GNU Project supporting various programming languages), which is an open source compiler collection.

Like most object-oriented development environments, Objective C consists of its own library of objects, a suite of development tools, and a runtime environment. Should anyone be interested in getting themselves familiar with Objective C, Apple offers a comprehensive tutorial for beginners, and gives them the chance to get to know the language and how it works, as well as the developing environment, known as Xcode.

The iPhone SDK

Xcode is what allows developers to create their own applications on Mac OS X. As far as developing applications for the iPhone goes, this is the only SDK by Apple that provides the tools and compiler to do so. Since the release of Xcode 3.1 and Steve Jobs' announcement of the iPhone SDK, developers are able to build applications for the iPhone and the iPod Touch. The following figure demonstrates the process followed from the stage of an application development, to the stage of distribution to end users.

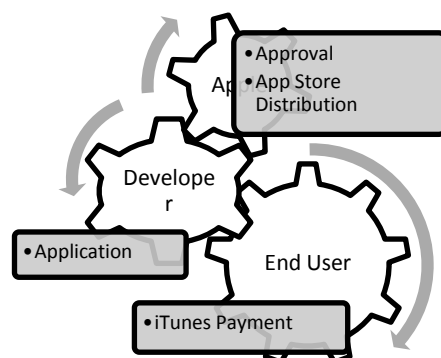


Figure 12: Apple's application distribution

Xcode IDE Overview

The Xcode IDE is the primary means to developing the application for the iPhone. It uses the Objective C programming language and files are split into .m (implementation) and .h (header). To design the interface of the application, the developers are provided with the Interface Builder, a tool which allows fast and simple ways to build the interface, by dragging objects such as buttons and text boxes onto the Views (or forms as known in Windows). Figure 13 shows the environment of Xcode and Interface Builder.

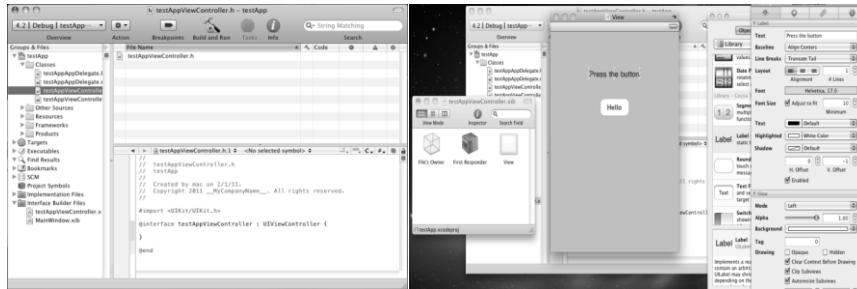


Figure 13: Xcode Environment

Libraries and OpenCV

Like in most modern programming languages, a developer can find many readymade libraries that perform a specific task. These can be imported and re-used for the application needs. Libraries that could benefit the development of this project involve text translation, OCR and Computer Vision/Image Processing packages. OpenCV is a cross-platform open source Computer Vision Library developed by Intel. It focuses mainly on real-time image processing and it is structured in five modules, four of which are shown in figure 14.

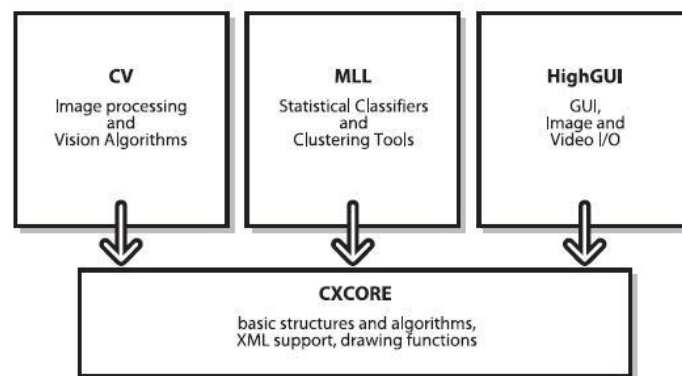


Figure 14: OpenCV Modules

Adapting OpenCV to the iPhone greatly benefits the results in terms of speed and efficiency. OpenCV has been cross-compiled to work for the iPhone, and is Apple Store safe, and used to aid with the development of our application. Techniques and algorithms such as MSER, Edge Detection and Thresholding discussed earlier, are by a large part already implemented by the OpenCV community, but several modifications and adjustments are necessary to optimize and calibrate for better results. More on the implementation part of these libraries is discussed in the following chapter.

2.9 Related Work in the field of Text Extraction

Computer Vision, a relatively new field of study, is as the words suggest the technology behind machines that can see. In other words, it aims to teach a machine, a computer, to extract visual information from images, and make sense of it. Related to many areas such as mathematics, physics, imaging, artificial intelligence and so on, computer vision is rapidly expanding, offering solutions to various problems. For this specific project, we were interested to find out how computers can be taught to spot text in an image and extract it.

Almost all approaches towards text localization we have reviewed use some sort of edge detection techniques along with connected components generation and grouping as a first step, which is the technique for finding regions of connected pixels which have the same color/intensity value. Fletcher and Kasturi [11] used an Area/Ratio Filter and Collinear Component Grouping to group characters together since text should lie along a given straight line, and taking into consideration the inter-word gaps they group a string to logical character groups, i.e. words. Their results were commendable for the time their experiments were carried out, taking into consideration hardware and CPU speed have dramatically improved over the last 20 years.

In their work for creating a framework towards real-time detection and tracking of text, Merino and Mirmehdi [12] used region filtering depending on size, border energy and texture to extract these text groups. The following figure shows the tree they used to detect dark text over bright backgrounds and vice versa, a pruning technique initially employed by Leon *et al* [13]. For their needs, they improved the tree region labeling method to detect text both ways (dark over bright and bright over dark background) with only one pass.

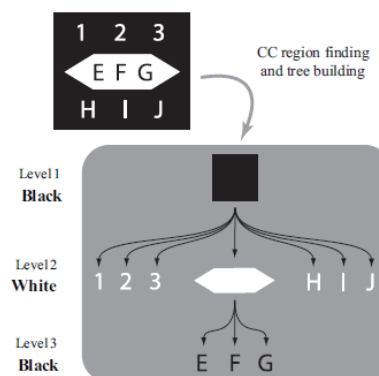


Figure 15: Tree of connected regions. This picture is copied from Ref. [12].

Their results show that their algorithm works fast enough to detect and track text in real time, maintaining a good FPS rate after the implementation of the full algorithm. The table below contains indicative results of their algorithm implementation.

	Text segmentation	Full algorithm
Scene 1	31.9 fps	13.2 fps
Scene 2	21.3 fps	4.9 fps
Scene 3	30.7 fps	9.6 fps
Scene 4	32.0 fps	10.6 fps

Table 3: Results. The data is copied from Ref. [12].

Srimaiyee *et al* [14] used non linear filters and found the morphological gradient of the image, and then applied a threshold to extract candidate text regions. Figure 16 shows an instance of their algorithm's results on an image.



Figure 16: Text Extraction. This picture is copied from Ref. [14].

They note that their technique gives noisy distorted binary images, and morphological clearing had to be used to improve the results.

Work has also been done using an active camera. Tanaka and Goto [15] presented a system for helping blind people by recognizing text from a wearable camera output using their revised DCT (discrete cosine transform) based method and using particle filtering to track text. Similarly Mirmehdi *et al* [16] extracted text using an active camera. Another noteworthy work was presented by Ezaki *et al* [17] which presents a method using Otsu's binarization and FDR (Fisher's Discriminant Rate) to distinguish text characters from other image elements with good results, but as stated, results were not accurate enough for practical use. The following figure demonstrates how they take higher FDR values to distinguish text characters in an image.

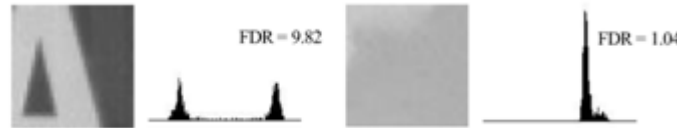


Figure 17: Character area and background area and their histograms. This picture is copied from Ref. [17].

Another related work is that of Sushma and Padjama [18]. They used two different approaches of text extraction in color images: Edge Based and Connected-Components Based. Their results indicate that the Edge Based method is more accurate and invariant to scale, lighting and orientation, as opposed to the Connected-Components Based algorithm.

Other interesting techniques use features such as the focus of a camera, like the work presented by Kim *et al* [19]. In their work they present an algorithm to extract text from an image using a focused area. They used a pixel sampling and a mean-shift algorithm as an initial step to select candidate regions of text, and then they form connected components. Using an iterative region search method they find neighboring components, and finally they verify the text using heuristic rules to determine what is actually text. Their work is related to this project in terms of using the focus of the camera, to restrict the area to be searched for text and therefore increasing the chances of correct text classifications.

Haritaoglu [20] used SNF (Symmetric Neighborhood Filter) and a hierarchical connected components algorithm to help segmenting characters from an image, and described how text can be verified using DWT (Discrete Wavelet Transform). In his work Haritaoglu explains that due to

limitations of a handheld device's processing power, data should be sent to a server where image processing methods can be computed a lot faster and at the time they were using GSM wireless networks. This meant that a complete communication between the device and a server could take up to 20 seconds. Contrary to this notion, our application runs on a very fast device which is able to complete processing, character recognition and translation in less than 20 seconds, so there is no need for sending any image data to a server.

Bhattacharya *et al* [21] proposed an algorithm to extract Devanagari and Bangla text from images. Again they use special properties of the language, but they also use generic properties as well such as text height, and use a technique that deals and picks neighborhood regions to further improve the results. Similar work by Rahman *et al* [22] recognizes signs using artificial neural networks for Bengali Textual information. In their work they use MLP (Multilayer Perception) Neural Networks to identify characters. What is interesting and relevant to our project in their work is their algorithm to detect and extract text. They read an input image in .jpg format, and convert it to grayscale. This step is usually performed by many researchers to ease and speed the process of text detection. Then they apply 3x3 median filter convolution masks and calculate edges using the Sobel Operator. They thicken the edges using dilation, and apply a vertical Sobel projection filter on dimmed image. They create the histograms by calculating those projection values and find the threshold value of the image. They then loop on possible positive identifications according to the histogram values and extract a plausible area which contains text. Figure 18 demonstrates the results of their algorithm on an image with text.



Figure 18: Extraction of region of interest. This picture is copied from Ref. [22].

This general algorithm could well be used for any situation for extracting text.

Clavelli *et al* [23] presented a framework evaluating ground truth performance labeling “atoms” (text parts) for segmenting and extracting text on complex color images at multiple levels. They also discuss a real-life scenario on how such an evaluation framework can be used. In their paper they conduct a thorough performance evaluation to assess the performance of text extraction methods on color images and concluded that the framework reports qualitative and quantitative results, which helps to identify where problems could lie in the text extraction process.

One less related but of equal importance piece of work is that of Roque *et al* [3] which adapts edge detection algorithms to the iPhone, such as the Canny Edge Detection algorithm, or the Sobel Operator. They compare the two algorithms efficiency and discuss the results and quality of each of them, which as discussed later on, helps us make decisions for the project. Since we consider Canny Edge Detection as one of the techniques we want to use for this project, we're interested in their findings regards efficiency. The following figure is an indication of their results showing how the Canny edge detection algorithm compares to Sobel Operator in terms of speed of execution on the iPhone.

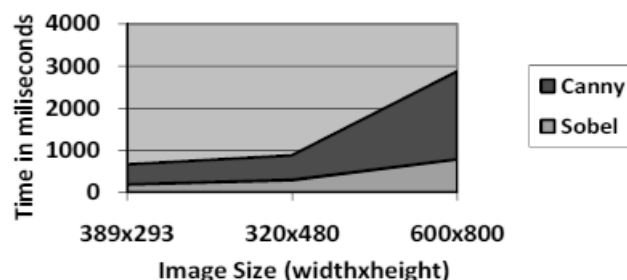


Figure 19: Canny vs Sobel Speed Comparison on the iPhone. This picture is copied from Ref [3]

While experimenting with edge detection, we have found that the Canny Edge Detection technique is more efficient than the Sobel Operator for text extraction. We used the OpenCV library algorithms which run very fast on both a PC and the mobile phone. The following table indicates our results performing Sobel and Canny algorithms on the iPhone 3GS using 1024x1365 resolution images.

The results indicate that Sobel Operator is indeed faster than the Canny Algorithm, but under the margin of 1 second a difference of 10% is negligible when quality is of the essence.

Input Image:		
Canny Algorithm Time (s)		
0.95	0.95	0.95
Sobel Operator Time (s)		
0.82	0.82	0.76

Table 4: Edge Detection execution time on iPhone 3GS

2.10 Related Work – iPhone Applications

There are a few applications already developed for the iPhone, which attempt to find and recognize text on images. In this section we review some of these applications by software other developers, and try to spot problems in their design and results to help us build an improved application for this project's needs.

2.10.1 Creaceed Prizmo

One of the applications we tried was *Prizmo* developed by *Creaceed*. This application offers various features such as cropping an image to the part of a text region, rotating and correcting perspective, as well as a white picker tool which whitens areas that are likely to be part of the background, to further help OCR with recognizing the characters correctly. Furthermore, it offers profiles specific to what the user is trying to capture such as a business card, the bill, or a whiteboard. Finally, it offers a Voice Reader feature, a text to speech tool that speaks back the words typed by the user, in various languages (available for the user to buy). This application has been downloaded and tested on the iPhone 2G. The results were satisfying in some cases, but overall it failed to provide 100% accurate recognition of characters even with a good image. The application depends on user input to specify the region correctly and is affected by other

background elements, which means the image is expected to contain only text on a monochrome background.

The following figure indicates the behavior of the application given two different images. In the first case the application gets the text almost fully correctly, but in the second case it fails to identify text because of the background.



Figure 20: Prizmo Screenshots - Results

2.10.2 Keystone Technology – CapnTrans

Similar to Prizmo, this application requires the user to spot the text and crop the image in a way that would make it easier for the OCR machine to recognize the characters. The difference is that this application translates the text using Google Translate (as discussed above) as well. A friendlier user interface but still the results are not always accurate.

The following figure shows how the application may fail to produce results given two similar images. In the first case we specified a smaller region containing the words “Scene Text” which were correctly recognized, but in the second where the region was larger the application failed.



Figure 21: CapnTrans Screenshots - Results

It should be noted that for this project we are not interested in having the user capture an image with only text on it on a white background. The idea is to be able to spot candidate regions that could contain text, and perform some processing on them to “clear” any non-text elements, and prepare an identified area to be send to the OCR engine for text recognition.

2.10.3 Quest Visual – Word Lens

A rather new and interesting application is out that claims to be able to recognize and translate text in real-time. The Spanish translation pack has been tested which replaces text in from English to Spanish in real-time. It does seem to work well in most cases, but it requires the user to hold the phone steady and point it directly to a text source. Furthermore, this application is only available for iPhone models with autofocus which means it depends on that feature to make out text. The problem many of the users have with this application is that it requires them to pay an amount of seven pounds for translating from one language to another and seven more for backwards translation, an amount which most users are not prepared to spend on such an application.

The idea of translating the text in real-time and replacing the original with the translation is quite interesting, and can only be run on the 3GS and 4 models, which have higher processing capabilities.



Figure 22: Word Lens English to Spanish Translation Screenshots

Figure 22 shows the results of the application in two instances. Word Lens seems to be working with small regions and did well in most of our tests. Because text translation is done offline though, the results could sometimes be wrong as it does word to word translation, as opposed to sentence translation performed when using Google Services.

These applications will be compared to our final product in the experimental results chapter.

3. Basic framework implementation and assessment

This chapter discusses the implementation part of the project, describing the algorithms, the user interface, and systems/technologies involved.

3.1 Implementation Considerations

Since every library available for image processing purposes is available both for a PC with Windows and the iOS on the iPhone, implementation was carried out in parallel, first on the PC and then on the mobile device. Because of the nature of the problem and the need for continuous testing and improving, it is wiser to work on a fast machine to aid development, before carrying on the implementation on the iPhone.

Two separate applications have been created, both with the same functionality. The application on the Windows machine is coded in C++ using Microsoft Visual Studio 2008, as opposed to Objective C which is used for iPhone development. The important functions are shared, meaning they are ported to the mobile device exactly as they are on the PC, but the desktop application offers some extra features helpful for debugging such as real-time text extraction using a camera, output of different processing results, enabling and disabling of processing methods etc.

3.2 Libraries and Algorithms

As mentioned earlier, most of the following algorithms were partially or fully implemented and available to use with the OpenCV library. We're interested in evaluating and improving the methods, with regards to efficiency and quality of results on the iPhone.

3.2.1 OpenCV

OpenCV can be obtained freely from the Internet and can be cross-compiled for use on the iPhone. It offers a large amount of fundamental image processing methods such as edge and contour detection, thresholding etc, as well image transformations and conversions. Most modern Integrated Development Environments such as the Microsoft Visual Studio and the Xcode offer some form of auto-completion, which provides for all functions and methods available to the developer, along with the description of the functionality and the parameters involved.

3.2.2 Noise Reduction

We use both types of noise reduction mentioned earlier: the Median Filter and the Bilateral Filter, for evaluation purposes. The smoothing method provided with OpenCV allows for both techniques to be used, and takes in 4 parameters. For Median only one parameter is used:

param1: The aperture width [odd number: 1,3,5...]

For Bilateral filtering **param2** is ignored and additionally:

param3: color sigma

param4: spatial sigma

3.2.3 Maximally Stable Extremal Regions

MSER is implemented in OpenCV in its most basic form. That is to detect maximally stable extremal regions and segment them with different colors. There are 8 parameters to initialize which affect the results of MSER detection:

- d** – Delta Threshold: Allows for more regions to be detected
- min_area** – Minimum Region Area
- max_area** – Maximum Region Area
- max_variation** – Maximum Variation of Regions
- min_diversity** – Minimum Diversity of Regions
- max_evolution** – Maximum Evolution of Regions
- min_margin** – Minimum Color Difference
- edgeBlurSize** – Sets edge blurring

The resulting MSERs are returned in a list (or sequence in OpenCV terms) and can be drawn on the original image or a new one.

3.2.4 Canny Edge Detection

For evaluation purposes and to help us make a decision whether or not MSER is the best approach for this problem, we implement edge detection and observe its behavior on text. The function available takes in two thresholds and the aperture size.

Threshold 1: Lower values (closer to zero) of Threshold 1 give less interrupts of the edges, meaning the edges will be more prolong.

Threshold 2: Higher values (closer to 255) of Threshold 2 minimize the amount of edges returned.

Aperture size: Size of the kernel: 1, 3, 5 or 7.

The output is an image with the detected edges which can be drawn or further used for processing.

3.2.5 LU Transform

The LU Transform could have many applications and has only recently been proposed and therefore is not part of any image processing library. We're interested in its results involving text on scene images, so implementation from scratch is required involving some basic matrix operations (multiplication and row operations) as discussed in the previous chapter.

After the Ω – Omega value has been obtained for each image patch, every pixel is compared against it (times a preset threshold **t**). All pixels values above this threshold maintain their intensity on the image, while those with values below are turned to black. The resulting image is used as a filter to aid in removing non-textual regions. The details of filtering are discussed in the following chapter.

3.2.6 Tesseract

Tesseract is an open source OCR-Engine initially written in C, and later improved and more coded was added in C++. Cross-compilation is needed to become available for usage on the iPhone, and many online communities deal with this matter exclusively. Tesseract version 3 was

used for this project with support of 32 languages. Once integrated with the application, the tesseract class will take an input image and output digitized text.

3.2.7 Google Translate

Google offers its web services through API (Application Programming Interface). Requests can be made with Ajax Queries. For this application we need a simple query of the type:

<http://ajax.googleapis.com/ajax/services/language/translate?q=%@&v=1.0&langpair=%@>

Parameter values are represented by “%@”. The first parameter q is the string to be translated. The langpair parameter specifies the language pair to be used for translation. For English to Greek translation the language pair would be: “en|el”. Leaving the first part empty (ie “|el”) means we want Google to detect the language to translate from, and translate to Greek. This would be a useful feature for our application considering we may not always know the language we are translating from, but due to limitations of tesseract we need a way to specify it so that it can perform the OCR. The application offers a list of languages available to the user to choose from, but we also implemented a feature to guess the language depending on the GPS location information (country). For example if someone is located in Barcelona, the program will automatically adjust the input language to be Spanish.

3.2.8 Parameters and Initialization

Values of MSER, Canny Edge Detector and LU Transform are set empirically after many tests on various images. Generally speaking, we tend to “allow” more things in, than cut regions out which could possibly contain text. There is no perfect value for each and every threshold because different images will have different components, but what is important is to set a value which is good for most cases.

Because of the complexity of the image processing methods discussed, we do not require the user to have any knowledge on what takes places and therefore no input is needed. Instead we set the values to what we believe will be appropriate for almost every scenario, after a series of tests performed on various images.

Median Filter: Kernel Size 3

Bilateral Filter: Kernel Size 7, Smoothing 40

MSER: We set delta – d at a low value of 3 (instead of default 5), to allow detection of more regions. Same goes for min_area. Max_variation is set to a lower value as well to discard regions with high variation (text regions are expected to have low variation).

Impact: Fewer non-textual regions are discarded, more regions to filter out detected.

Canny Edge Detection: Threshold 1 = 50 and Threshold 2 = Threshold 1 * 3 (150), aperture size of 3.

LU Transform: Window size = 32 and number of low frequency values to skip l = 8. Threshold t is set to 2.5.

Tesseract: The OCR-Engine requires an input image text language to be initialized. This is given either by the user (settings menu) or auto-detected by the application itself using GPS information.

3.3 Preliminary Results and Evaluation

This section demonstrates preliminary results with regards to output and speed. Tests were run on a 2.0GHz Dual Core Laptop using Visual Studio 2008+ in **DEBUG** mode, and the results are indicative of **MSER** and **Edge Detection and contour drawing** without any other processing involved. Time is the average time taken to perform MSER and Edge Detection after five tests. No noise reduction has been applied in any of the two techniques either, because the Canny algorithm already deals with that matter, and we're not interested at the effects of noise in MSER at this point. Images have been resized to resolution: 640x480.

Note that because the white background within the border of the sign is detected as a CoCo in canny edge detection, drawing it hides the characters within. Certain techniques could be applied to prevent this, but at this point we're more interested in the speed of each method. The table below shows the resulting images and processing time for MSER, Edge and Drawing with different images.



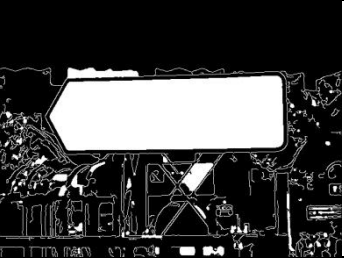

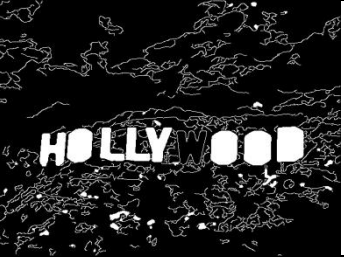



Input Image 1	MSER Result	Canny Edges
		
	Time (s) 0.175	Time (s) 0.045
Input Image2	MSER Result	Canny Edges
		
	Time (s) 0.177	Time (s) 0.052
Input Image 3	MSER Result	Canny Edges
		
	Time (s) 0.171	Time (s) 0.035

Table continues to the next page




Input Image 4	MSER Result	Canny Edges
		
	Time (s)	Time (s)
	0.168	0.035

Table 5: MSER and Canny Edge Results

3.4 Observations and Decisions

From the preliminary results as shown in the previous section, and performing a series of tests on various images, we took important decisions for this project. We concluded that MSER is a better solution and outperforms the Canny Edge Detection (CED) approach for text extraction.

Advantages of MSER over Canny Edge Detection:

- *Set and Forget:* Preset thresholds and parameter values work for all images tested. Canny Edge Detection did well in most cases because of text properties, but in several situations adjustment of threshold values was needed to detect all characters of text.
- *No background regions:* Because text and background appear in MSER+ and MSER- respectively, no processing is needed to separate and distinguish between the two, as opposed to CED from the results shown in Input Images 1 and 3.
- *Better Results:* MSER succeeded in finding all characters in all cases with fewer elements for filtering, while CED failed at times either by merging characters, or by not closing the regions.

Advantages of Canny Edge Detection over MSER:

- *Speed:* In most cases, CED was four to five times faster than the MSER detection algorithm. This allows for more time to perform filtering and text grouping, hence reducing the total time needed to present the user with results.

Decisions

Although CED was significantly faster than the MSER technique, the results were not qualitative enough to indicate success in building a robust text extraction system. MSER on the other hand proved promising and stable enough to be selected as the method of choice to be implemented and further improved in this project. CED proved the concept that text has special edge properties that make it easier to be picked up by an edge detection algorithm, and should not be neglected in situations where speed is of the essence. Adaptive thresholding is an option which could improve the results of CED and diminish its dependability on threshold values. In fact, many applications developed to this day use edge detection as the primary mean to detect text, especially in real time.

It is evident that choosing quality over speed will result in an application with no real-time processing and text extraction capabilities on a mobile phone, even of one with the technology of

the latest iPhone. At the end of the day, what is important is to have an application that can get the job done and serve its purpose well, which is to translate text for the user, in a reasonable amount of time. Real-time processing and text extraction with MSER is not impossible on a modern personal computer though, as we will show later in this thesis, and perhaps soon enough it will be possible on a mobile device as well. This along with the fact that MSER being used for text detection is relatively a new idea led us to this decision.

4. Basic model enhancement and text extraction

The basic model consisting of MSER segmentation or Edge detection is not enough for the task of extracting text from images. Further improvements need to be made, and certain filters have to be applied. This chapter is about these modifications to the algorithms and the extra methods required for the process of filtering.

4.1 MSER

Earlier we discussed the creation of the connected component tree of MSER regions. The tree is created as proposed by Nistér *et al* [24] in linear time. The code is then modified as proposed by Merino *et al* [7] to reduce the tree nodes to those of highest border energy. Figure 23 demonstrates the removal of the extra nodes. Computationally-wise this significantly reduces the amount of nodes we need to traverse to make out which regions is text or not, therefore the time needed for processing is also dramatically decreased.

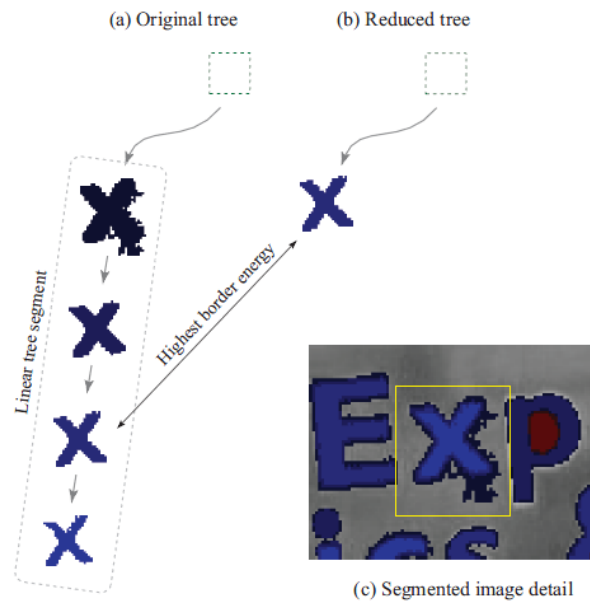


Figure 23: Linear Tree Segments Removal. Image Copied from [7]

4.2 Noise Reduction

As a way of improving the image taken from the user, noise reduction is used to smooth the image and free it of any noise. As described earlier, the median filter is a good and fast approach to eliminate noise, but Bilateral Filtering is an option to consider. We implement both of these noise reduction algorithms and compare the results in terms of speed and quality. Preliminary results indicated that as expected, Bilateral Filtering does a better job in reducing noise while preserving edges needed to correctly pick up text as MSER, but at the cost of computation time. Noise reduction step is applied right before MSER.

4.3 Filtering

The results returned by MSER are regions of interest which could contain text or not. Beyond this point there is nothing more segmentation could help with. We could try to send the results to OCR straight away, but that would make little difference in the output in cases of busy images. In spite of the fact that many regions representing other elements are still present on the image, a filtering process is required to make out what is text and what is not. There are two types of filters which we will be using



Figure 24: CoCo Bounding Boxes

for text extraction:

- **Region Filters:** These relate to shape, size and location properties of the regions.
- **Texture Filters:** Use of LU Transform to filter out non-textual regions.

We use common sense and properties of text to create filters that would help extracting text. The next two sections present the filters we applied with regards to a connected components bounding box. Figure 24 shows examples of connected components bounding boxes marked with red.

4.3.1 Region Filters:

Region filtering is performed in two passes. In the first pass, all of the connected components contours are examined against a set of conditions relating to size and shape properties:

Minimum size of bounding box in pixels: $2px \times 2px$

Minimum width: 60% of image width

Minimum height: 60% of image height

Center of gravity c of contour: $10\% \text{ of box size} < c < 90\% \text{ of box size}$

Boundary Length over Contour Area: < 1.4

Aspect Ratio (width/height) r : $0.05 < r < 6$

Note that when referring to boundary length we mean the perimeter of the contour. Every contour which passes the first round of filtering is considered as a candidate text region. A count of candidate text regions is held. After the first round, the mean width, height and boundary length of all CoCos bounding boxes is calculated. Then, in the second pass, every contour which does not satisfy the following conditions is discarded:

Width/Height of contour bounding box: $10\% < \text{mean contours Width/Height} < 500\%$

Boundary Length: $20\% < \text{mean contours Boundary Length} < 400\%$

The second pass could be considered somewhat too allowing, but we need to think of situations where different font sizes occur in an image (big title, small description). At the same time this also means that we are allowing large non-textual regions as well.

The reasoning behind this decision is that it is more sensible to have non-textual regions that will most likely be recognized as a symbol that we could ignore, rather than lose letters or words which the user wants translated. Furthermore, it is many times the case that characters may be joined together as one. OCR Engines can read such text but setting a strict limitation in Aspect Ratio and Mean Width/Height allowance would mean that these regions would be discarded.

In most cases, letters are expected to be of similar font size, and so after this round of filtering, most non-textual regions are removed.

4.3.2 Texture Filter:

After the LU Transform has been applied on the grayscale image, we subtract it the MSER resulting image. Every CoCo box in the MSER image which contains less than ten percent of information in the LU Transform Image cannot be considered as a region which contains text, and it is therefore discarded. We use this filter as a measure to remove regions with low texture response which happened to pass the first round of filtering. Ten percent is again very lenient but as explained before, under no circumstances should any region containing characters be removed.

4.4 Text Grouping

In busy images, it is expected that many regions will be detected as MSERs. In most cases, many of those regions will look like characters, and even pass the first round of filtering because they satisfy the set of conditions that we set. Regions with high texture response such as foliage or dirt will also pass the texture filtering part.



Figure 25: Original Image, MSER Image, Filtering Image, LU Transform, Filter Minus LU Image

Figure 25 demonstrates the results of: MSER, Region Filtering, LU Transform and Texture Filtering. Even after the first and second round of filtering, it is evident that many non-textual regions are still present.

Text grouping is a special filtering stage which takes into consideration text properties such as:

- Character Spacing: Usually characters have a constant distance between them
- Word Spacing: Following character spacing rule, words are usually at a distance of approximately a character width apart
- Character Location: Considering text is horizontal
- Character Similarity: A character is similar in size to another close to it.

With the above assumptions, we set the following conditions that must be true in order for regions to be considered part of a word:

- Two characters are considered part of a word if they are at most separated by a horizontal distance of two and a half times the width of the character.
- Two characters are considered part of a word if they are at most separated by a vertical distance of $\pm 10\%$ their height.
- A character's bounding box cannot be overlapping that of another character's. In the case of connected characters, only one bounding box appears.

- A character's height is at most twice or at least half of the height of a nearby character.
A character's width is at most four times or at least one fourth of the width of a nearby character.

There are a few special cases requiring more caution, which is why the conditions seem to be very lenient again:

- The letter I (or i) occupies at most one third of the width of the other characters and therefore the distance between that character and another is very small. It may also be the case that the distance seems to be bigger than what it should be (fixed size fonts).
- A word does not necessarily need to consist of two or more letters. Letters "A" or "I" usually form a word by themselves and so no character needs to be at a distance of width "A" from them. We need to consider isolated characters, but only if the distance horizontally or vertically indicates sensible for a sentence.

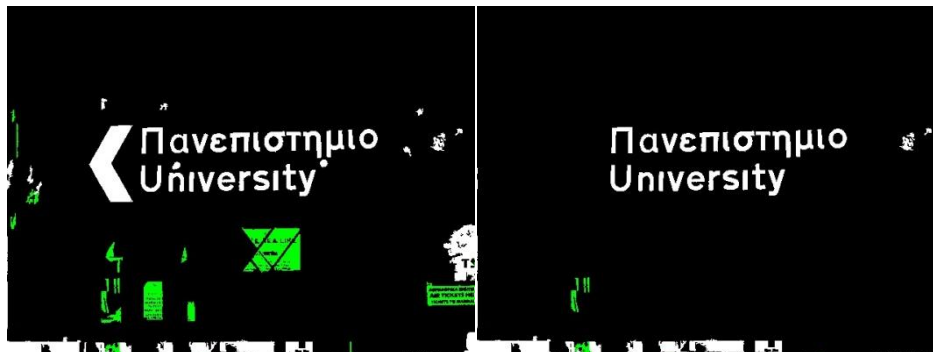


Figure 26: Region Filtered Image Result, Text Grouping Result

Figure 26 demonstrates the perceptual text grouping result on the image used before. There are still some regions left which do not represent text, but because of their shape nearby regions, they resemble text properties and cannot be removed. Nevertheless, this image is now good enough for OCR and the results can be filtered even further by removing strange characters.

4.5 Optimizations

This section discusses optimizations made to the code for better results and less processing time.

4.5.1 MSER and Filtering

The way our implementation of MSER works gives us results for MSER+ and MSER- regions. Two separate pictures are returned, those which presumably contain text on light background, and those which contain text on dark background.

Observing the results as shown in Figure 26, on a variety of different images, we noticed that many times when merged, green and white regions overlap or exist in close proximity. The initial thought was that this hinders the process of both region filtering and text grouping. For this reason, region filtering and text grouping was performed on each of the two images separately. After a series of experimenting we noticed that this is not the case.

Noise and non-textual busy regions usually produce regions in both images. Think of a tree and its foliage against a light background – the sky. The trunk and branches would turn up in the MSER- image, while if the leaves were light green, they would turn up in the MSER+ image, assuming other foliage and branches behind the ones in the foreground appear dark. By filtering the images separately we lose the opportunity to pick up the anomalies of these regions and disqualify them as textual regions. Furthermore large leaves would likely be considered as characters of a word for example. Filtering separately will produce two images that when merged will have few overlapping regions and no indication of the original object as whole and so further filtering of the merged image would have very little impact.

4.5.2 Analysis of the image

This project is concerned with scene images. It would not be a good idea to try processing an image which contains just text for the following reasons:

- Noise reduction may distort, merge or even hide characters
- Weak appearing characters (due to noise, blurring or bad lighting/resolution) may not be detected at the stage of MSER
- Processing, filtering and grouping, and then OCR takes time
- On such images Tesseract performs well

To decide whether a picture should be processed or not is not an easy task. One approach is to analyze the pixels of the grayscale image and look for a large count of different intensity levels. Usually, text on a monochrome background will have very few different changes in intensity. Because this step does not depend on the quality or size of the image, scaling to a small size would significantly speed up the process. This approach will of course fail if text characters are of different color, or if there are light effects present.

Another approach would be to binarize the image first. Otsu's binarization is thought to be one of the best binarization methods in image processing. The method assumes that there are two classes to be thresholded in the image and pixels belong to one of those classes (background or foreground). By separating those two classes so that their combined spread (intra-class variance) is minimal, it is able to calculate the optimum threshold. To express the method mathematically, assume that all pixels in an image are represented in G gray levels (0-255). Pixels with gray value G are denoted by n_i so the total number of pixels is: $N = n_1 + n_2 + \dots + n_G$. If we split all the pixels in two classes C_0 and C_1 denoting background and foreground with a threshold k [25]:

C_0 : pixels with levels $[0, \dots, k]$

C_1 : pixels with levels $[k+1, \dots, G]$

The probabilities of class occurrence:

$$\omega_0 = \Pr(C_0) = \sum_{i=0}^k p_i = \omega(k) \quad (5)$$

$$\omega_1 = \Pr(C_1) = \sum_{i=k+1}^G p_i = 1 - \omega(k) \quad (6)$$

Mean levels:

$$\mu_0 = \sum_{i=0}^k iPr(i|C_0) = \mu(k) \quad (7) \quad \mu_1 = \sum_{i=k+1}^G iPr(i|C_1) = (\mu_T - \mu(k)) \cdot (1 - \omega(k)) \quad (8)$$

$$\mu(k) = \sum_{i=0}^k iPi \quad \mu_T = \sum_{i=0}^G iPi$$

To calculate the optimal threshold k^* :

$$\sigma_B^2(k^*) = \max_{0 \leq k \leq G} \sigma_B^2(k) \quad (9)$$

Where

$$\sigma_B^2(k) = \frac{[\mu_T \omega(k) - \mu(k)]^2}{\omega(k)}$$

When we apply Otsu's method on the input image, we get a well binarized output which we can then analyze and decide if processing is needed or not. To decide whether or not to continue with the processing part, a more lenient form of region filtering is performed on the binarized image:

- Allow a number of characters to be less than 10% of the mean width, height and boundary length of all characters. This covers cases of common punctuation marks.
- Allow a CoCo to be almost the size of the image itself. Assuming the picture is now black and white, the background is detected as a connected component as well, but this is not a false positive.

After this round of filtering, if no connected components needed to be removed (zero), the image is declared as **OCR Ready**, and no processing takes place.

Consider the following images, and the binarization results. These are good enough for OCR when binarized and so there is no need for further processing:



Figure 27: Original Images and Otsu's Binarization results

If these images were to go through the whole processing stage of MSER, filtering, LU Transform Filtering and perceptual grouping, many characters such as small punctuation marks could end up being lost at the process. Furthermore, the time needed to perform all these tasks and OCR would not be justified, since there is no need to extract text, there is only text on the images. To further explain the process of binarization and the need for analyzing, we also provide a few images that need to be processed, and the Otsu binarization results in the following table.

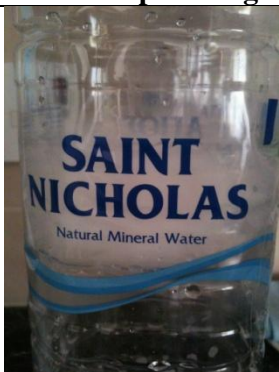
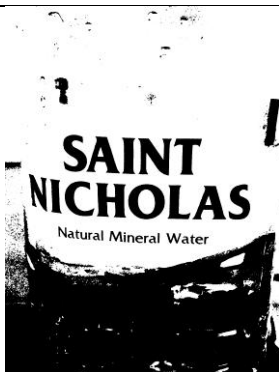
Input Image	Otsu Binarization	OCR Ready
		No: Filtering discovered regions that do not resemble text Tesseract OCR Engine would not be able to recognize text in this image

Table continues to the next page

		No: Filtering discovered regions that do not resemble text. In this case, it was caused the texture of the wood. Tesseract OCR Engine would perhaps not be able to recognize text in this image due to the wood texture.
		No: Filtering discovered regions that do not resemble text. Tesseract OCR Engine would not be able to recognize text in this image.
		No*: Non-text regions were detected (borders) *Tesseract OCR Engine would be able to recognize text in this image successfully.

Table 6: Binarization Results and Analysis

Observing the results in different images, we notice that in some cases where borders exist (Image 4), the image is not declared as OCR-Ready, even though Tesseract should have no problems with it. We could of course be less strict and allow for example one tenth of the total number of characters to be removed, but that would also allow an image such as Image 3 to go directly through OCR and give bad results. Nevertheless, this approach serves its initial purpose, which is to decide whether or not the image is a scene image or just a page of a book or a restaurant's menu.

The improved framework proved to significantly help the OCR-Engine in correctly identifying characters while reducing false-positives at the same time.

4.5.3 Google's Spell Checker

The result returned by the OCR-Engine may not always be 100% accurate, and therefore we need to make sure that what we send to the translator makes sense. We use Google's spell checker to correct the text returned by tesseract. In essence, we capture the "Did you mean:" corrected text returned by Google when we perform a simple search.

5. Mobile Application

This project is about scene text extraction and translation on the iPhone. This chapter is concerned with issues on the iPhone device, as well as the application developed, its features and user interface.

5.1 iPhone Device Issues

It is important to understand exactly what makes this a special case compared to performing the same task on a personal computer. There are two main reasons why development on the iPhone for this particular project differs from the implementation on a PC:

Processing Capabilities: Naturally mobile devices do not possess the processing power of a desktop computer. Memory is not as much of an issue as CPU speed is. Keeping in mind that the amount of time it takes for the application to perform processing, OCR and translation will be multiple of that taken on a desktop PC, the application needs to perform as efficiently as possible.

Mobile Camera: As stated earlier, newer iPhone models (3GS and 4) are equipped with a modern quality mobile camera of 3 and 5 megapixels, with auto focus and exposure features. Here, an important decision needs to be made.

Do we really need to be processing 3 million pixels? Resizing an image may cause degradation, and possibly loss of characters due to blurring or merging.

Solution: Resize the image to a smaller size, significantly reducing processing time, but not too small to jeopardize the quality of the results. Pictures taken using the iPhone 3GS and 4 cameras, are of 1536x2048 and 1936x2592 size respectively. Resizing the image to 1024x1365 (1.4 megapixels), effectively reduces the amount of pixels to process by more than half.

5.2 Application

The application for the iPhone has been developed in parallel to our implementation on the desktop PC. It targets devices with iOS version 4.2 and later.

5.2.1 Features

All of the features described in this thesis have been fully implemented for the application. In short, the application can:

- Load a Picture from the Library (Open File) or take one using the camera
- Analyze the picture and decide if processing is needed
- Process the image performing MSER and Filtering
- Save, Rotate, Undo Image
- Perform OCR and Translation on the resulting image. The user may copy the text to the clipboard.
- Increase speed by further resizing the image to a smaller resolution
- Determine Image Input Language using GPS Location Information

5.2.2 User Interface

The user expects the application to perform a simple task. Take a picture, extract text and translate it to the language of choice. Therefore, the user must not be required to perform any other intermediary steps or input any other information.

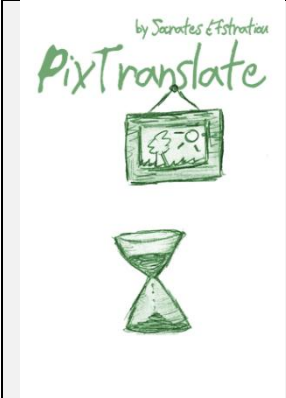


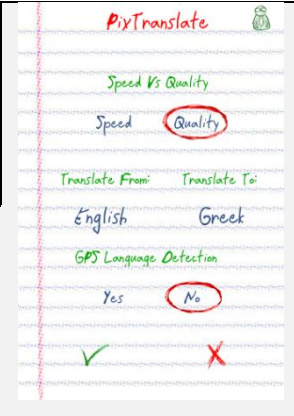



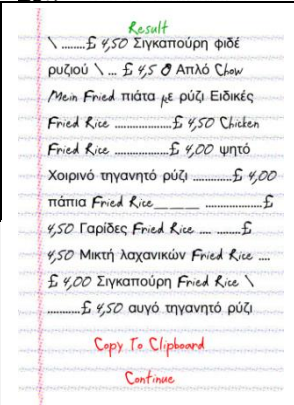
	<p>Loading Screen: This is the loading screen shown when the application is launched.</p>	
	<p>Main Screen: (Initial) From here the user can load a picture using the photo button. The rest of the buttons can be used once a picture has been loaded.</p> <p>Main Screen: (Picture Loaded) When a picture is loaded or taken from the camera, the user can rotate, save or restore the previous image. The checkmark button performs processing and Ocr, and the globe button performs the translation</p> <p>Settings Screen: From here the user can select the speed of the application (resizes the image), as well as the input and output (translation) languages. The user may also enable the GPS feature which auto detects the language to translate from.</p>	
	<p>Processing: Once the user taps the checkmark (process and OCR) button, a loading animation is shown until completion.</p> <p>Processing Result: The processed image replaces the original in the main screen.</p>	
	<p>OCR Result: The result is presented to the user in digitized text. This text can be copied to the clipboard.</p> <p>Translation: Tapping the globe (translate) button, the user is presented with the final translation result. This text can be copied to the clipboard as well.</p>	

Table 7: Application Screen Flow and Description

Table 7 provides a description of the screens shown to the user with the functions available in each screen.

Once the preferences have been set, the user should be able to perform the task in three simple steps:

- Load: The user loads a picture from the phone's library or takes one using the camera.
- Read: With a tap of a button the application extracts text, and the digitized text is presented to the user.
- Translate: The application translates the text and presents the user with the result.

Note that these are the screens to this point of writing this thesis. The final product screens may differ, but they will be along the lines of the ones presented here.

5.2.3 Final Product – Release

The application developed is planned to be released to the App Store in the near future. The user interface will be further polished, and some final fine tuning will be performed to prepare the application for release.

The application needs to be reviewed and evaluated by Apple Staff before it is allowed to enter the App Store. A developer's license is required to be able to build and publish an application for the iPhone. Enrollment costs fifty-nine (59) pounds per year for an individual. The evaluation period for the application by Apple is reportedly three to four weeks, after which the application should appear in the App Store and be ready to download.

Because this application was part of a project and a learning process, the first version of the application is planned to be available free of charge. At a later time, if we wish to improve and expand the application we might consider making it a paid application, at which case the price will most likely be that of fifty-nine (59) pence.

6. Experimental Results

Continuous testing is a very important process in this sort of applications, not only for adjusting and fine tuning, but also to observe and analyze results in different situations or by using different methods. The following series of tests were conducted using various random images containing text which we believe would adequately assess the performance of each algorithm.

In our experiments we used different scenarios and explain why and how the results led to various decisions we took with regards to the final application. An image is loaded to the application and the output of the algorithms is saved in images that can then be sent to OCR. In cases where time was recorded, C++ time library was used to calculate the time it took to execute an algorithm (MSER, Filter, and Grouping). The following table describes all of our following experimental results.

Table #	Experiment	Input/Parameter1	Input/Parameter2	Result1	Result2	Result 3
9	Noise Reduction	No Noise Reduction	Noise Reduction	Improved OCR		
10	MSER d	Delta – d = 3,4,5,6,7		Output Image		
11	MSER max_variation	Max_variation = 0.05, 0.10..0.25		Output Image		
12	OCR vs. MSER-OCR	Input Image	MSER Image	Expected OCR	OCR	MSER OCR
13	MSER OCR vs. Region Filter OCR	MSER Image	Region Filter Image	Expected OCR	MSER OCR	Region Filter OCR
14	MSER OCR vs. Texture Filter OCR	MSER Image	Texture Filter Image	Expected OCR	MSER OCR	Texture Filter OCR
15	MSER and Filtering evaluation			MSER Result-Time	Region Filter Result-Time	Texture Filter Result-Time
16	Filter and Text Grouping Results	Region Filter Image	Text Grouping Image	Expected OCR	Region Filter OCR	Text Grouping OCR
17	Application results vs. Ground Truth	Original Image		Application Result	Ground Truth	False Positives
18	Application results vs. Plain OCR	Original Image	Application Result Image	Expected OCR	OCR	Application Result OCR

Table 8: Experimental Results/Inputs/Parameters

All of the experimental results were obtained using our implementation on the PC, unless stated otherwise.

6.1 Noise Reduction

Noise reduction is a helpful image processing technique to eliminate certain types of noise in images such as the Amplifier Noise, the Salt-and-Pepper noise, Shot noise etc. We're interested in finding out if applying noise reduction is beneficiary enough to the application results to justify the extra processing time it requires.


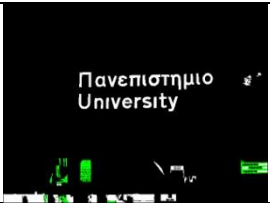
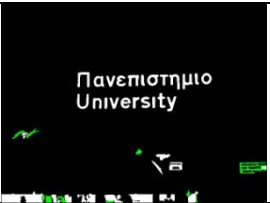






Input Image	No Noise Reduction	Noise Reduction	Improved OCR
			No – Different regions detected as MSERs and characters got smoother, but no change in OCR result.
			No – Fewer non-textual regions but smoothing caused letter I not to be detected as a MSER
			No – Detected 1 more character, but lost 3.

Table 9: Noise Reduction result vs. Original Image result

The above table's results are indicative of images with both dense and sparse text regions in them. In almost all of our tests we did not notice any significant improvement using noise reduction. In some cases while using the median filter we lost characters, mainly because the edges got weaker. Bilateral filtering had similar results although it took a lot more time to complete.

Noise reduction barely helped in the detection of more characters, while increasing the processing time especially on the iPhone. Images with excessive noise may exist, such as the ones taken in dark conditions, and in such cases noise reduction with bilateral filtering could help, but in general, performing noise reduction on ordinary images does not seem to improve on MSER detection or help with OCR. We evaluated the algorithms on the iPhone 3GS using images we took with its camera and resized to 1024x1365 resolution.

Input Image	Median Filter	Total Processing Time (s)	% of total Processing Time	Bilateral Filter	Total Processing Time (s)	% of total Processing Time
1	2.26	10.29	22%	6.06	14.09	43%
2	2.19	11.00	20%	6.05	14.96	40%
3	2.21	11.98	18%	6.05	15.82	39%

Table 10: Median Filter and Bilateral Filter execution time on the iPhone 3GS.

Table 10 shows that it took on average 20% and 40% of the total processing execution time just to perform noise reduction. It would normally take 8 seconds to finish processing (MSER, Filtering and Text Grouping), yet with noise reduction we increase that time up to 6 seconds.

Decision: Noise reduction could be omitted from the final product since the gain vs. speed ratio is not high enough.

6.2 MSER

In this section we examine and compare results of MSER alone and combined with other techniques. First we test MSER on various images with different parameters. These results help us set final values for those parameters. First, the delta parameter's behavior is examined, while all other parameters are kept constant. The default value of d is 5. Table 11 shows the results of different d values.

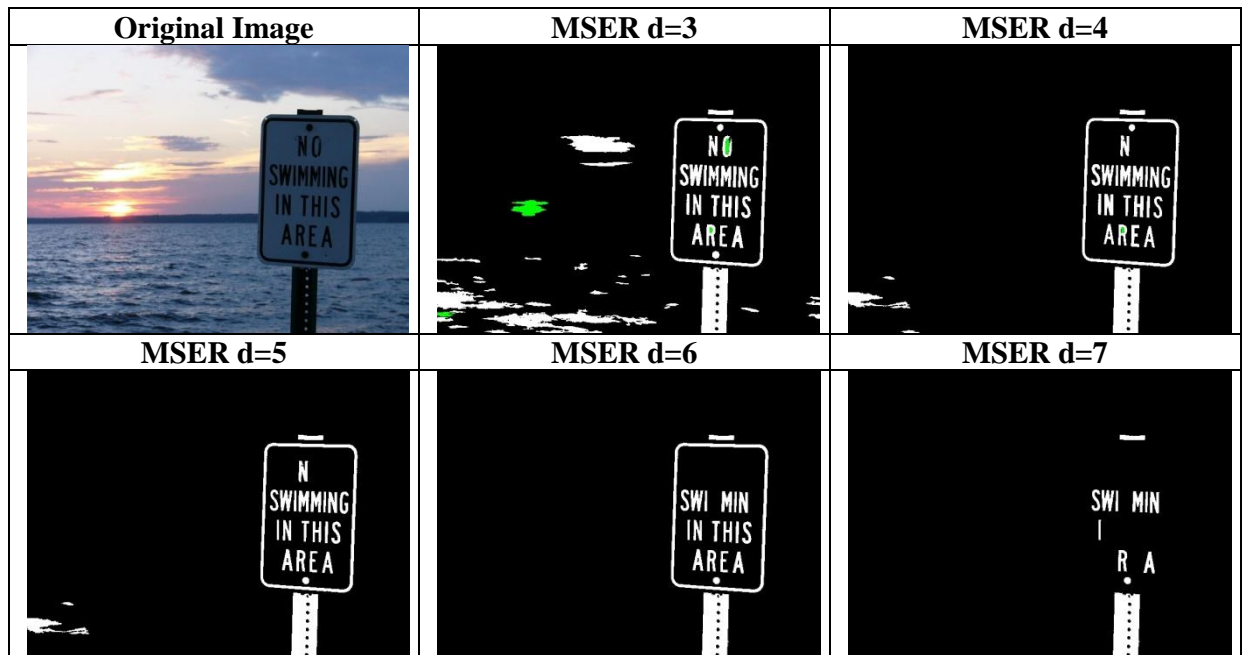


Table 11: MSER delta results

The bigger the value of d , the fewer MSERs it detects. The letter O in NO is slightly faded which is why it is lost with delta values bigger than 3. As we keep increasing the value of d we lose more non-text regions, but we also begin losing characters from the text we want to extract.

Decision: Based on this observation on various images we decided to keep delta at a low value of 3 so that we detect all regions of text.

Extracting text will then rely more on filtering, but it would otherwise be pointless to carry on if we lost characters to begin with. Furthermore, not many elements of a scene have strong MSERs like text has, so the odds are most non-textual regions will be not be detected at all. The input image in Table 6 above demonstrates that fact: The Sea and the sky have many shades and so during the thresholding phase the regions keep changing size. Solid objects like metal and text retain their region area size.

Next we want to examine the max_variation parameter behavior. The following table shows the results of MSER output as max_variation changes. The default parameter value is 0.25.







Original Image	Max_variation = 0.05	Max_variation = 0.10
		
Max_variation = 0.15	Max_variation = 0.20	Max_variation = 0.25
		

Table 12: MSER max_variation results

Notice how every time we increase the value of max_variation more and more elements are detected as MSERs. This happens because we tell the MSER algorithm to be more lenient and allow for regions to have bigger variation in pixel values.

Decision: We keep the max_variation parameter at a low value (0.05), since text pixels are likely not to have much variation amongst them. Note that keeping this value low does not necessarily mean other regions will not be detected as well unless they are stable or text will always be picked up no matter how it appears. This particular image shows that if a letter is occluded or has light or shadow effects on it (letter S in “STOP”), it is likely to change area size in different thresholds and therefore be discarded. Part of the letter S in this instance has been not been detected as a MSER.

The rest of the parameters did not seem to affect region detection too much, with the exception perhaps of min_area which needed to change from 60 to 20 to allow smaller regions to be detected. We found that in certain cases, smaller characters such as the letter “I” (especially true for sans-serif fonts) were found to have a small area compared to other regions detected, and would therefore get discarded.

Setting the optimum values for these parameters and keeping the rest to defaults, we have found that the algorithm had good results in extracting all text regions in almost every image given to it. In this way, we have an algorithm which does not depend on any further user input to perform well.

In Chapter 1 we posed and answered the question:

- Why use processing and not send the image directly to OCR?

This time we pose the question:

- Why not stop here and send this processed image to OCR?

To answer this question we needed to perform some tests using these images and the tesseract OCR-engine. We sent various images resulting from MSER detection, to find out whether or not the results were any better than those of pre-processing, and if so, to what extent. The following table presents OCR results pre-processing and post-processing. Only MSER is performed so far, no filtering. Correctly recognized characters are shown in bold.




Input/MSER Image	Expected Text	OCR	MSER-OCR
	STOP	*non-sense symbols*	*non-sense symbols*
	# Characters: 4	Success:	Success:
		0/4 – 0%	0/4 – 0%
Input/MSER Image		OCR	MSER-OCR
	Pure Butter ASSORTED SHORTBREA D	*non-sense symbols* alkers *non-sense symbols* Pure Butter ASSORTED ‘ SHORTB *non-sense symbols*	Wilkers Pure Butter ASSORTED SHORTIREAD
	# Characters: 85	Success:	Success:
		30/35 – 85%	33/35 – 94%
Input/MSER Image		OCR	MSER-OCR
	NO SWIMMING IN THIS AREA	*non-sense symbols*	'1~ SWIMMING N00 2, m rms An_fA *non-sense symbols*
	# Characters: 20	Success:	Success:
		0/20 – 0%	11/20 – 55%

Table 13: Original and MSER Image OCR Results

The results indicate an improvement of OCR output when performing MSER detection prior to sending the image to the OCR-engine. The second image shows that MSER had OCR detect three

characters more than what it would with the original image. It is important to have in mind that Tesseract performs some sort of binarization itself, which is why it could sometimes pick up a few of the characters, but it is not something to rely on. The third image would give no result if passed straight away to the engine. With MSER however, we detected 55% of the characters, but only one word would be usable for translation. The first image on the other hand clearly shows that neither of the pre- and post- processed images helped the OCR-engine recognize any of the characters.

MSER improves results but it is not enough to fully extract text. Non-textual regions and borders cause the OCR-Engine to return non-sense characters, and in some cases, no characters at all.

Decision: The solution to a successful extraction of text for the OCR-Engine is to perform filtering to remove non-textual regions.

Region Filtering is applied to the resulting MSER output image with rules as discussed in chapter 4. The next series of experiments show the improvement made by the use of these filters with regards to the output of the OCR-engine. No texture filtering or text grouping has been applied at this stage.

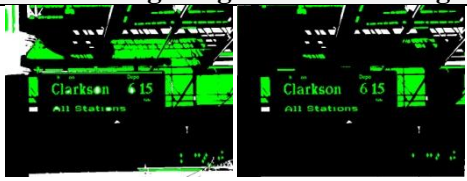

MSER Image/Region Filtered Image	Expected Text	MSER-OCR	Filter-OCR
	Clarkson 615 All Stations	*non-sense symbols* Clarkstn 615 M ; M. =V\$ A All Statlns 'i	*non-sense symbols* Clarkson 615 l ` V... =V\$ A _ All Stations 'i
	# Characters: 22	Success: 19/22 – 86%	Success: 22/22 – 100%
MSER Image/Region Filtered Image		MSER-OCR	Filter-OCR
	NO SWIMMING IN THIS AREA	SWIMMING N00 2, m rms An_fa	no SWIMMING m rms An_fa
	# Characters: 20	Success: 11/20 – 55%	Success: 12/20 – 60%
MSER Image/Region Filtered Image		MSER-OCR	Filter-OCR
	DANGER DONT RUN BEWARE DONT WALK BACKWARDS DEEP SHAFTS UNMARKED HOLES	A N G E R aewmsl nw1wAu< BACKLWARIS sHAFrsj~ Q i si UNMARKEI HOLES 9	DANGER DON T RUN BEWARE DONT WALK BMLKWARDS DEEP ° 'Q sums -1-si UNMARKED HOLES 9
	# Characters: 59	Success: 29/59 – 49%	Success: 51/59 – 86%

Table 14: MSER and Region Filtered Image OCR Results

There is a significant improvement in the results returned by OCR as shown in table 14. Input image 3 resulted in successfully finding 22 more characters than the MSER image. This happens mainly because of the extra non-textual elements that get removed during this process, and so the OCR has fewer regions to read and translate to text. Notice that inner contours in closed characters' connected components like the letter "D" and "O" also get removed in this stage, so this makes the text recognition even easier. Even though the results were better, the progress this far shows that again this is not enough.

Decision: MSER and Filtering complement each other, but there is more filtering to be done before the image is ready to be fed to the OCR-Engine.

Introducing the LU Transform filter, we examine the effects it has on the MSER image, to decide whether or not it is better than region filtering. This stage performs MSER and Texture Filtering only.



MSER Image/Texture Filtered Image	Expected Text	MSER-OCR	Filter-OCR
	Fort George	gl `o'='n ts, 1.5¢ Fm ceases	_¢, mis FORT GEORGE
	# Characters: 10	Success: 2/10 – 20%	Success: 10/10 – 100%
MSER Image/Texture Filtered Image	Expected Text	MSER-OCR	Filter-OCR
	NO SWIMMING IN THIS AREA	'1~ SWIMMING N00 2, m rms An_fA	SWIMMING rio m rms An_fA
	# Characters: 20	Success: 11/20 – 55%	Success: 10/20 – 50%

Table 15: MSER and Texture Filtered Image OCR Results

The results of the LU Transform as shown in Table 15 seem to help in cases where there are a lot of text-like elements that cannot be removed with region filtering. The results indicate improvement in some situations, like the first image in Table 8. Because a few non-text regions were removed, especially near the word "FORT", the OCR-engine was able to recognize the words successfully. In the second image though, only a few regions were removed, and instead we lost a character. Texture filtering is a good filter to have as backup on "busy" images. It will remove regions that other filters could not remove, but it is not so effective when the image does not contain too many elements, and the computation time required by the LU Transform on large images is something to think about when implementing text extraction on a mobile device.

Decision: The LU Transform complements MSER and could be used for text extraction, but it is better off being used as an "optional" feature on the iPhone application.

The following table shows results of MSER and Filtering (texture and region) on images, as well as the time taken to perform each process. This provides a good measure to understand the significance and value of each algorithm compared to the results it provides.









Original Image	MSER	Region Filtering	Texture Filtering
			
Time to complete:	0.204	0.012	0.629
			
Time to complete:	0.219	0.016	0.554

Table 16: MSER and Filtering evaluation

Both these pictures were of size 0.4 megapixels. The results indicate that the LU Transform and filtering takes on average three times as long to perform compared to MSER and Region Filtering combined.

So far we have examined the effects of MSER, Region and Texture filtering with regards to better character recognition from the OCR-Engine. We have found that non-textual elements and borders can cause the engine to fail returning some or all characters. The following set of results in table 17 indicate the improvement that perceptual text grouping offers.





Region Filtered Image/Text Grouping	Expected Text	Filter-OCR	Grouping OCR
 	TESCO express	_1 IE\$\$\$Q @xi2!eSS *'\f	I\$\$\$\$Q express " v
	# Characters: 12	Success: 4/12 – 33%	Success: 7/12 – 58%
Region Filtered Image/Text Grouping	Expected Text	Filter-OCR	Grouping OCR
 	NO SWIMMING IN THIS AREA	no SWIMMING m rms An_fa	NO SWIMMING IN THIS AREA
	# Characters: 20	Success: 12/20 – 60%	Success: 20/20 – 100%

Table continues to the next page


Region Filtered Image/Text Grouping		Filter-OCR	Grouping OCR
	DANGER DONT RUN BEWARE DONT WALK BACKWARDS DEEP SHAFTS UNMARKED HOLES	DANGER DON T RUN BEWARE DONT WALK BMLKWARDS DEEP ° ‘Q sums -1-si UNMARKED HOLES 9	DANGER DON T RUN BEWARE DON T WALK BACKWARDS DEEP SHAFTS UNMARKED HOLES
	# Characters: 59	Success: 51/59 – 86%	Success: 59/59 – 100%

Table 17: Filter and Text Grouping results

The results clearly show that with perceptual text grouping, OCR performs at its best. The first image result indicates that even though OCR could not understand what the decorative underlining under the word “TESCO” was, removing the small symbol under the word “express” as a result of text grouping helped in finding the whole word correctly. It is important to understand why text grouping could not remove that underlining from the text. Earlier we explained that text grouping considers regions which look like characters, and that have other regions nearby them that together could form a word. In this case, the parts of the underlining could be forming a word, and so they were not removed.

Results in image 2 and 3 prove the significance of text grouping. We went from an unreadable image for OCR, to a partially readable image with MSER, and finally to a fully readable image with text grouping.

To conclude this section we want to compare the final image as it comes out after text grouping with the ground truth image as shown in the following table. The ground truth image is the binary image which contains only the regions of text. For our purposes we obtain this image by manually extracting the text regions from the MSER image with simple editing.

Original Image	MSER	Final Result	Ground Truth	False Positives
				
				
				

Table continues to the next page






Original Image	MSER	Final Result	Ground Truth	False Positives
				

Table 18: Application results vs. Ground Truth

We used images with many non-textual elements, which would otherwise be unreadable if fed straight to the OCR engine. Compared to the ground truth, the application's final results are not too far off. Although we may still get false positives due to the fact that those regions look like text, we almost never lose characters. False positives can further be filtered out using the output of the OCR-engine itself, but losing actual characters will cause an inaccurate translation. In the next chapter we will evaluate the overall performance of the system, and discuss future improvements that could be made to the system.

6.3 Final Product

To conclude this chapter, a comparison of the application results with that of plain OCR and of other applications is sensible. We want to find out to what extent we have improved upon text recognition, of what significance this whole process was towards the original goal which was text translation on the iPhone.

6.3.1 Application vs. Basic OCR

First we want to test whether or not our application improves on basic OCR or not. We do this by feeding the OCR-Engine with the original image and the one we processed, and evaluate the results the engine produces. Table 19 indicates these results with images taken using the iPhone's camera.

The application clearly performs at its best when the image contains a busy background or light effects on the foreground. It enables OCR to pick up characters that would otherwise not be recognized. At the same time, a picture with mostly text on it is good enough for OCR. Our application will attempt to filter out any non-textual elements and this may help the engine make out the characters more easily, or it may lose characters due if they do not appear in a word style format.




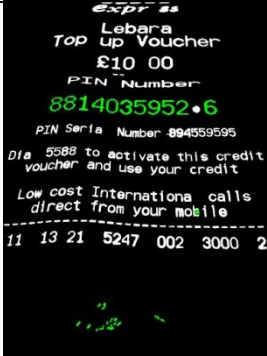


Original Image	OCR	Application Result	OCR
	*blank*		KTC 755 i
	Lebara TOD~up Voucher_ \$?9_;99 PIN Number 551403595235 PIN Serial Number 894559595 Dial 5588 to activate this credit v0Uçh°f and use your credit. Law GUST International ca\\s dff°m your mobile. '11 ~ 13:21 5247 002";%E)E)E) _- EZ;		Lebara Top up Voucher £10 00 PIN_Number 8814035952»6 PIN 5°"S N mber- 894559595 Dia 5538 T0 8ç'tl vate this credit v0U0h°" and use your credit Low COST Internationa ca\\s direct f°m your mobile Q1 13 21 5247 _5c]\$" \$f>f>\$"§
	\`1 J/@-		KLN/ 7

Table 19: Application results vs. Plain OCR Results

6.3.2 Application vs. other iPhone Software

In the second chapter we presented a few of the applications that already exist on the iPhone, and carry out the task of text extraction and translation. To the best of our knowledge these applications do not use any image processing techniques, with the exception perhaps of Word Lens. Instead, they send the image directly to the OCR-engine, and so results as discussed in the previous section are expected. We will compare the results of these applications with the one we developed.

CapnTrans & Prismo: As expected, these applications perform basic OCR on the image, and so results are identical to the tests we conducted in section 6.4.1. These applications are both paid applications, which means the user needs to pay a specific amount to purchase them from the Apple App Store.

Word Lens: We felt that Word Lens was superior to the other two applications and therefore we focused our tests on its results compared to those of our application.

Both applications did very good when given images that have text on a monochrome background with many other elements in the image as well. However, when presented with a complex image with busy background, our application did better. The following test uses the demo version of Word Lens which reverses the word instead of translating it.



Figure 28: Word Lens and Final Product result pictures

Figure 28 shows pictures taken pre- and post- processing using the two applications. In the first example, Word Lens only managed to pick up two words correctly, ("Sixth Edition"), while our application picked up most part of the word "Engineering" and made readable the words "A practitioner's Approach". The missing letters were filtered out due to the merging of the characters with light the background behind them. The second image shows that our application made the words "World Best Seller" readable while Word Lens only managed to read "World" correctly. The other words not in a horizontal line and text-like elements could not be read and were discarded.

Another problem Word Lens has is that it uses a dictionary for the words and therefore if a word is not found in the dictionary it replaces it with the nearest match, which would result in a wrong translation. Nevertheless, Word Lens is to our opinion the best amongst other OCR/Translation applications available in Apple's App Store. The big downside is as we mentioned earlier the fact that for each translation to another language, the user must buy the language pack which costs thirteen pounds to translate both ways (ie: English to Spanish and Spanish to English).

6.3.3 Application Benchmark on iPhone 3GS and 4

The final set of tests was performed on the two fastest iPhone models available at this time (3GS and 4) to compare the total processing time as shown in table 20.

Input Image:							
	iPhone Model:	3GS	4	3GS	4	3GS	4
	Processing Time (s)	16.07	7.46	10.16	7.69	3.50	2.04
	OCR Time (s)	1.46	1.06	4.87	3.63	2.79	1.81

Table 20: iPhone 3GS vs. 4 application performance

The results clearly show that the iPhone version 4 outperforms the 3GS version. On the iPhone 4 model, the total time required to Process and OCR is on average 10 seconds, while on the 3GS around 13 seconds. Note that the third input image was found to be OCR-Ready after binarization; hence no further processing was required.

7. Discussion

This chapter discusses and evaluates the performance of the basic and the improved frameworks, and the mobile application.

7.1 Basic Framework

In this project we have presented a way to improve on traditional basic-OCR in applications, as well as implemented text translation on the iPhone. We showed that OCR is a technology yet to be perfected, but there is a number of ways to improve the results. The proposed method involves implementation of the MSER detection method, which proved very promising in terms of successful text segmentation on an image. The comparison between this method and the widely used edge-based detection approach shows that MSER has the advantage when it comes to intuitive text detection. More specifically:

- Parameters can be set empirically to values that work best for text detection. The user will never need to change them again to improve the results.
- All text characters will be detected as MSERs, given the image is of good quality.
- Separation of objects on light background from objects on dark background.

The basic framework performs character recognition using Tesseract. The key benefits of using Tesseract is that it is a fast, accurate and offline OCR-Engine, widely accepted as one of the best ever developed. The implementation supports 32 languages, which serves the application purpose well.

The project was not only about text extraction and recognition, but also involved text translation. We decided that using an online service would be the best approach for this task, and chose Google's Translation service for the following reasons:

- No offline text translation solution provides a complete sentence translation feature, which means words would be translated one by one to the user, often generating non-sense text.
- Even on devices accessing the internet using 3G network subscription, the amount of data traffic for text is so small that no significant cost would be incurred.
- Google's Translate service is one of the best, free online translation services which continuously get improved by the online community. This means the results will be best in most cases.

The implementation was carried out both on a PC and the iPhone mobile device. We believe the framework was set up in a way to be very efficient and flexible.

- Uses state-of-the-art technologies for various tasks such as the OpenCV and Tesseract Libraries. This offers a large amount of extra features that may be needed in case of future improvement of the system.
- Modularized: The application performs each task of image processing in separate parts which can be modified or removed, without affecting the operability of the main system.
- Extensible: New features and functions can easily be added to improve the overall performance of the application, without need of significant changes in code.

- No further initialization of parameters is needed for the application to work well for text detection.

7.2 Framework enhancement

The basic framework at its original form does exactly what this project requires which is a simple text recognition and translation on the iPhone, with or without the use of MSER detection. Experimental results show that there is little improvement over basic-OCR when performing MSER detection on an image, and therefore there is a need for an improved model to achieve better results in this direction. We improve on our basic framework implementation to get the most out of the image processing techniques we chose for text extraction. We introduced a few key modifications to the original MSER implementation available from the OpenCV library such as the separation of MSER+ and MSER- regions in different images, and a reduced, hierarchically built connected components tree. These allowed for much faster execution times, as well as having image results which allowed for better manipulation towards text extraction.

The results clearly indicate that for a successful text extraction, filtering is required. With the introduction of specific filters for text extraction, we immediately noticed a significant improvement over the results we got back from MSER detection. Region filters removed non-textual regions which did not have the usual properties of text characters, while texture filters helped with the removal of false positives which resembled text. We chose to be lenient with the set of conditions of our filters, which meant that in our effort to keep all characters from being filtered out, we also had many false positives.

A new type of filtering needed to be introduced which took into consideration character-to-character and word-to-word relations. Perceptual text grouping helped even more with the removal of false positives, as well as making clear the parts where text existed. We achieved an improvement of up to sixty (60) percent in character recognition in some cases, compared to that we had only with region filtering.

Furthermore, having in mind that tesseract is a powerful OCR-Engine, with newer versions capable of some type of adaptive thresholding to better distinguish text, we wanted to save the user the time needed for processing, and just send the image straight to the OCR. We introduced a technique which uses Otsu's binarization and filtering to determine whether or not the image is OCR-Ready.

7.3 Mobile Application

The application developed for the iPhone features a modern and simplistic user interface and has a clear objective: To extract text from an image, and translate it to the user. No distracting features or buttons are present.

The iPhone offers great processing capabilities like most modern smart-phone devices, and so we settled with a solution that works well in a reasonable amount of time. The average time needed to perform processing, OCR and translation on the iPhone is about 15 seconds on the iPhone 3GS model, and 12 seconds on model 4. Future models will be even more powerful than these two

particular models, which not only means that processing time will be further reduced, but also that more image processing techniques could be introduced to improve the results.

7.4 Critical Evaluation

The approach followed in this project is clearly not the only way to go about text detection and extraction. We chose MSER over Canny Edge Detection but this does not necessarily mean that a successful implementation with edge detection is improbable. In fact, we showed that using an edge detection approach is very fast compared to the MSER approach, which means more time to perform filtering. MSER outperforms edge detection but it is not perfect either.

- Light/dark colored lines or elements intersecting with light/dark text characters (see Figure 28). This causes MSER regions to merge meaning there is no way to make that character readable any more.
- Light effects, shadows or occlusion on characters can cause partial or full exclusion of the region affected.
- Although MSER is not greatly affected by noise, characters with noisy edges will result in malformed regions, resulting in bad character recognition by the OCR-Engine.

In terms of filtering, we noticed that our region filters were in a way a little too “allowing”. Especially in very busy images containing plants and vegetation, or decoration on signs and walls, our implementation associates these regions with text and so more work is assigned to the perceptual text grouping method. The LU Transform proved a helpful filter, but because of its high computational requirements, it has been disabled in the final product.

A great deal of time was spent implementing and evaluating noise reduction on images. In the end, it proved that noise reduction has little impact on the detection of MSERs, and the computational time could be saved instead. However, it goes without saying that if the Canny Edge Detection algorithm was preferred, bilateral filtering would dramatically improve the results since it is an edge preserving image smoothing technique.

We believe the overall objectives of this project have been met. A system has been developed which extracts, reads and translates text from otherwise unreadable - by OCR-Engines - images. The improved framework is robust and requires no adjustments to parameters to be further optimized for different images. The next section describes ways our framework could be improved even more.

8. Future Work – Conclusion

This chapter discusses ways our improved framework could even further be improved, as well as the conclusions we made throughout this project.

8.1 Further Directions

In the first section of chapter 7 we explained how the basic framework is modularized and therefore extensible. This section is about a few suggestions we make that could be considered in order to further improve the system's results.

8.1.1 Improve on MSER

Earlier we discussed the problems underlying the MSER implementation, regarding overlapping regions. A good idea to prevent loss of characters due to merging when other elements of the same foreground color-to-background color intersect them would be to draw MSERs with different colors. This would of course require a more sophisticated approach to set a range of colors to use with MSER+ regions and another with MSER- regions.

8.1.2 Improve on Filtering

Good filtering is crucial for the successful extraction of text characters from an image. It could perhaps be improved if new sets of conditions were introduced. We applied what we believe were a good set of conditions for characters, but others could exist. Furthermore, with the use of Graphics Processing Units (GPUs), it is possible to perform calculations using in-built functions much faster. We had to implement the LU Transform from scratch, and for processing speed reasons we had to omit it from the final application on the iPhone, but many modern GPUs usually offer optimized matrix operations to the developer.

8.1.3 Improve on Text Grouping

The perceptual text grouping technique we implemented could further be improved with new sets of conditions. In exercise, it could be made even more intelligent by enabling it to look beyond simple character-to-character and word-to-word relations. For example, we set a minimum and maximum allowed space between characters and words, but a more intelligent system could keep track of the changes occurring in space distances within characters of a word. If similar size characters seem to have varying distances between them, they could be thought of false positives text-like regions and be discarded.

8.1.4 OCR Guided Text Extraction

Another thought is to use OCR to verify text-regions. Regions with questionable text resemblance could be fed to the OCR-Engine. If the engine gives back a non-valid (symbol) character, or the character comes with low confidence score, then the region could be discarded. This would of course have larger computational requirements, but considering the rapid growth of mobile technology, such a task is not impossible.

8.2 Conclusion

In this study we discussed the significance of text extraction from scene images, proving existing OCR-Engines to be unable to extract text from such images. We presented an approach for text extraction which involves Maximally Stable Extremal Regions detection. It proved to outperform the conventional approach which uses edge detection as its basis.

The basic implementation was extended introducing special sets of filters which help towards a better text extraction from complex and otherwise unreadable images by OCR-Engines. While the improved framework takes longer to perform text extraction than an edge detection-based approach, implementation on the iPhone device takes a reasonable amount of time to complete, and gives more satisfactory results than most existing OCR-Translation applications.

As discussed in the previous chapter, an absolute solution would perhaps need to employ more sophisticated filters and OCR capabilities at the same time. We believe this work sets foundations towards such a solution.

Bibliography

- [1] E. J. Wharton, K. Panetta, and S. S. Agaian, "Logarithmic edge detection with applications," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC '07)*, pp. 3346-3351, October 2007.
- [2] Canny J, "A Computational Approach to Edge Detection", *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol.8, 1986, pp.679-714.
- [3] M. G. Roque, R. M. Musmanno, A. Montenegro, E. W. G. Clua, Adapting the Sobel Edge Detector and Canny Edge Extractor for iPhone 3GS Architecture. *IWSSIP. 2010. September 17, 2010*.
- [4] Matas, J., Chum, O., Urban, M., and Pajdla, T. 2002. Robust wide baseline stereo from maximally stable extremal regions. In *British Machine Vision Conference, Cardiff, Wales*, pp. 384-393.
- [5] J. Digne, J.-M. Morel, N. Audfray, C. Mehdi-Souzani, The Level Set Tree on Meshes, *Proc. 3DPVT 2010, Paris*
- [6] Michael Donoser , Horst Bischof, Efficient Maximally Stable Extremal Region (MSER) Tracking, Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, p.553-560, June 17-22, 2006
- [7] Carlos Merino, Karel Lenc, Majid Mirmehdi, A Head-mounted Device for Understanding Text in Natural Scenes. *Fourth International Workshop on Camera-Based Document Analysis and Recognition. August 2011*.
- [8] A. T. Targhi, E. Hayman, and J. olof Eklundh, "Real-time texture detection using the LU-transform," in *CIMCV, 2006*.
- [9] C. Tomasi and R. Manduchi, Bilateral Filtering for Gray and Color Images. *Proc. Sixth Int'l Conf. Computer Vision*, pp. 839-846, Jan. 1998.
- [10] S. Efstratiou, Iphinger Olympics: Development Of An Iphone/Ipod Touch Application, *Thesis (BSc), University of Nicosia, May 2010*.
- [11] L. A. Fletcher and R. Kasturi. A robust algorithm for text string separation from mixed text/graphics images. *IEEE Trans. Pattern Anal. and Machine Intell.*, 10:910–918, 1988.
- [12] C. Merino and M. Mirmehdi. A Framework Towards Realtime Detection and Tracking of Text. Second International Workshop on Camera-Based Document Analysis and Recognition (CBDAR2007), pages 10–17, 2007
- [13] M. Leon, S. Mallo, and A. Gasull. A tree structured-based caption text detection approach. In *Fifth IASTED VIIP, 2005*.

- [14] P. Srimaiyee, G. Rama Mohan Babu, A. Srikrishna. Text extraction from heterogeneous images using mathematical morphology, *Journal of Theoretical and Applied Information Technology*, Vol 16. No. 1 -- 2010
- [15] M. Tanaka and H. Goto. Text-Tracking Wearable Camera System for Visually-Impaired People. *Proceedings 19th International Conference on Pattern Recognition (ICPR2008)*, 2008.
- [16] M. Mirmehdi, P. Clark, J. Lam, Extracting Low Resolution Text with an Active Camera for OCR, *Proc. of the IX Spanish SPRIP*, pp. 43-48, May 2001
- [17] N. Ezaki, K. Kiyota, B.T. Minh, M. Bulacu and L. Schomaker, Improved Text-Detection Methods for a Camera-based Text Reading System for Blind Persons, *ICDAR, 2005*, pp. 257-261.
- [18] J.Sushma, M.Padmaja Text Detection in Color Images, *International Conference on Intelligent Agent & Multi-Agent Systems 2009 IAMA-09* 22 - 24 July 2009 in Chennai, India
- [19] Kim, E., Lee, S., Kim, J.: Scene text extraction using focus of mobile camera. *Document Analysis and Recognition, International Conference (2009)* 166-170.
- [20] I. Haritaoglu, Scene Text Extraction and Translation for Handheld Devices, *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, 2001, Vol. 2, pp.408-413.
- [21] U. Bhattachatya, S. K Parui and S. Mondal, Devanagari and Bangla, Text Extraction from Natural Scene Images, *In Proc. ICDAR 2009*, pp. 171-17
- [22] M. O. Rahman, F. A. Mousumi, E. Scavino, A. Hussain, H. Basri. Real Time Road Sign Recognition System Using Artificial Neural Networks for Bengali Textual Information Box, *European Journal of Scientific Research*, 25(3):478-487, 2009
- [23] A. Clavelli, D. Karatzas, and J. Lladós. A framework for the assessment of text extraction algorithms on complex colour images. *In Proceedings of the 8th IAPR International Workshop on Document Analysis Systems*, pages 1926, Boston, MA, USA, 2010. ACM.
- [24] D. Nistér and H. Stewénus. Linear Time Maximally Stable Extremal Regions. *In Proceedings of European Conference on Computer Vision (ECCV)*, pages 83–196, 2008.
- [25] Otsu N. A threshold selection method from gray level histograms. *IEEE Trans. Syst. Man Cybern.* 1979;9:62–66.

Image Sources

Figure 1: Peter Facey - SU6020 : Sat Nav warning sign, Beacon Hill Lane, Exton. Available at: <http://www.geograph.org.uk/photo/386451>

Figure 10: Available at: <http://true-wildlife.blogspot.com/2011/02/cheetah.html>

Figure 14: Available at: <http://learningopencv.wordpress.com/2010/05/23/opencv-structure-and-content-overview/>

Signs in tables 9, 12, 14, 16, 18: Available at: <http://www.public-domain-image.com/objects-public-domain-images-pictures/signs-public-domain-images-pictures/>

Appendix: Source Code

```
"Pix_TranslateViewController.h"
(...)
static inline double radians (double degrees) {return degrees * M_PI/180;}
#define IMAGE_COUNT 19
#define w 32
#define l 8

typedef struct matrix {
    double mat[w][w];
} myMatrix;

typedef struct myContours {
    CvSeq *ptr;
    CvPoint cog;
} myCnt;
(...)
//Detect MSER and draw +/-
-(void)mser:(cv::Mat &image):(cv::Mat &mserPlus:(cv::Mat &mserMinus {
    (...)
    CvMSERParams params = cvMSERParams(d,20,14400,v,dv,200,1.010000,0.0030000,.3);
    IplImage timplng = image;
    cvExtractMSEr(&timplng, NULL, &contours, storage, params);
    for (size_t i = 0; (int)i < contours->total; ++i) {
        CvSeq *seq = *CV_GET_SEQ_ELEM(CvSeq , contours, i);
        //Draw MSERs
        if (seq->v_prev == NULL) {
            [self drawMSEr:mserPlus : mserMinus : seq : 0];
        }
    }
    cvReleaseMemStorage(&storage);
}
-(void) drawMSEr:(cv::Mat &mserPlus:(cv::Mat &mserMinus:(CvSeq *)seq:(int) level {
    (...)
    if (((CvContour *)seq)->color >= 0) {
        color_vec = cv::Vec3b(255,255, 255);
        plus = 1;
    }
    else {
        color_vec = cv::Vec3b(0,255, 0);
    }
    for (int j = 0; j < seq->total; ++j) {
        CvPoint *pos = CV_GET_SEQ_ELEM(CvPoint, seq, j);
```

```
        if (plus)
            mserPlus.at<cv::Vec3b>(pos->y, pos->x) = color_vec;
        else
            mserMinus.at<cv::Vec3b>(pos->y, pos->x) = color_vec;
    }
    (...)
}
//decide if contour is character - input: # of neighboring regions to check
-(bool) look:(int) direction:(int) contourIndex: (std::vector<myCnt> &) sContours: (int) depth {
    /* l=left 0=right*/
    (...)
    for (int i=0;i<depth;i++) {
        if (ptr2 != NULL) {
            cnt2 = cvBoundingRect(ptr2);
            /* height */
            if ((cnt1.height >= cnt2.height/2) && (cnt1.height <= cnt2.height*2.1) &&
                /* width */
                (cnt1.width >= cnt2.width/5) && (cnt1.width <= cnt2.width*5)
                &&
                /* horizontal distance */
                (abs(cnt1.x - cnt2.x) >= (int)(cnt1.width/2)) && (abs(cnt1.x -
                    cnt2.x) <= (int)(cnt2.width * 2.5)) &&
                /* vertical distance */
                (abs(cnt1.y - cnt2.y) <= (int) (cnt1.height * 1.1))) {
                    /* is not inner */
                    if ( ( (cnt1.x > cnt2.x) && (cnt1.x < cnt2.x+cnt2.width)) &&
                        ((cnt1.y > cnt2.y) && (cnt1.y < cnt2.y+cnt2.height)) )
                    {
                        inner = true;
                        break;
                    }
                    /* smaller and misplaced */
                    if ((cnt1.width*cnt1.height < cnt2.width*cnt2.height*0.6) &&
                        ((cnt1.y<cnt2.y-
                            cnt2.height*0.2)||cnt1.y+cnt1.height>cnt2.y+cnt2.height*1.5))) {
                        break;
                    }
                }
                res = true;
            }
            if (direction) contourIndex--;
            else contourIndex++;
            if (((contourIndex <= 0) && (direction)) ||
                ((contourIndex >= sContours.size()) && (direction)))
                if (thes) return false;
            else break;
        }
    }
}
```

```

    }
    }
    if (inner) res = false;
    return res;
}

bool contoursSortFunction (myCnt i,myCnt j) {
    return (i.cog.y<j.cog.y);
}

//check if region is character
- (void) textGrouping:(IplImage *)dst_img {
    int erased;
    do {
        (... )
        //sort contours by center of gravity
        std::vector<myCnt> sContours;
        for (ptr = contours; ptr != NULL; ptr = ptr->h_next) {
            myCnt tmpCnt;
            tmpCnt.ptr = ptr;
            cnt = cvBoundingRect(ptr);
            tmpCnt.cog = cvPoint(cnt.x + cnt.width/2,cnt.y + cnt.height/2);
            sContours.push_back(tmpCnt);
        }
        sort (sContours.begin(),sContours.end(),contoursSortFunction);
        //check if contour is character. look at neighbours
        CvRect cnt2;
        for (int i=0;i<(int)sContours.size();i++) {
            cnt = cvBoundingRect(sContours[i],ptr);
            if ((i>0) &&(i<sContours.size()-1)) {
                cnt2 = cvBoundingRect(sContours[i+1],ptr);
            }
            if ([self look:0 i:sContours :12] || [self look:1 i:sContours :12]) { }
            else {
                erased++;
                cvDrawContours(dst_neg, sContours[i],ptr, color, color, 0,
                    CV_FILLED, 8, cvPoint(0,0));
            }
        }
        (... )
    } while (erased);
}

```

```

//Region filtering
- (bool) filter: (IplImage *) dst_img : (bool) analyze {
    (... )
    for (ptr = contours; ptr != NULL; ptr = ptr->h_next, index++) {
        CvSeq *tmpPtr = ptr;
        bool inner = false;
        do {
            cnt = cvBoundingRect(ptr,0);
            cntArea = cvContourArea(ptr);
            cntBBArea = cnt.width * cnt.height;
            cntAreaRatio = cntArea/cntBBArea;
            cvMoments(ptr,&moment,0);
            boundaryLength = cvContourPerimeter(ptr);
            m_00 = cvGetSpatialMoment(&moment, 0, 0 );
            m_10 = cvGetSpatialMoment(&moment, 1, 0 );
            m_01 = cvGetSpatialMoment(&moment, 0, 1 );
            gravityX = m_10 / m_00;
            gravityY = m_01 / m_00;
            ratio = (double) cnt.width / (double) cnt.height;
            //satisfies text properties
            if ((cnt.width < dst_img->width * 0.6) &&
                (cnt.height < dst_img->height * 0.6) &&
                (cnt.width >= 2) &&
                (cnt.height >= 2) &&
                ((gravityX-cnt.x)/cnt.width>0.1) &&
                ((gravityX-cnt.x)/cnt.width<0.9) &&
                ((gravityY-cnt.y)/cnt.height>0.1) &&
                ((gravityY-cnt.y)/cnt.height<0.9) &&
                (boundaryLength / cntArea < c) &&
                (ratio > 0.05) &&
                (ratio < 6) ) {
                    meanBoundaryLength += boundaryLength;
                    meanWidth += cnt.width;
                    meanHeight += cnt.height;
                    meanElms++;
            }
            else {
                (... )
            }
        } if (meanElms > 0) {
            meanHeight /= meanElms;
            meanWidth /= meanElms;
            meanBoundaryLength /= meanElms;
        }
    }
}

```



```

//FILTER 2
for (ptr = last; ptr != NULL; ptr = ptr->h_prev) {
    cnt = cvBoundingRect(ptr,0);
    boundaryLength = cvContourPerimeter(ptr);
    if ((cnt.height < meanHeight * 5) && (cnt.height > meanHeight * 0.1)
        && (cnt.width < meanWidth * 5) && (cnt.width > meanWidth * 0.1)
        && (boundaryLength < meanBoundaryLength * 4.5) && (boundaryLength >
            meanBoundaryLength * 0.2)) {
        if (!analyze) cvDrawContours(dst_neg, ptr, colorBlack, colorBlack, 0,
            CV_FILLED, 8, cvPoint(0,0));
        else correct++;
    }
    else {
        if (analyze) {
            if (((cnt.width > dst_img->width * 0.8) && (cnt.height > dst_img-
                >height * 0.8)) ||
                (cnt.height < meanHeight * 0.1) ||
                (cnt.width < meanWidth * 0.1)) {
                    if (cnt.width < meanWidth * 0.1) punctuation++;
                    if ((cnt.width > dst_img->width * 0.6) && (cnt.height >
                        dst_img->height * 0.6)) {
                            largeRegions++;
                            correct++;
                            continue;
                    }
                }
            else {
                cvDrawContours(dst_neg, ptr, color, color, 0, CV_FILLED, 8,
                    cvPoint(0,0));
            }
            erased++;
        }
    }
    if (analyze) {
        cvReleaseImage(&gray);
        cvReleaseMemStorage(&mem);
        cvReleaseImage(&dst_neg);/*
        if ((correct == 0) || (largeRegions>1) || (punctuation > correct / 2)) return true;
        if (erased > 0) return true;
        else return false;
    }
}
(...)

```

```

- (bool) readyForOCR:(IplImage*) dst_img {
    IplImage *otsu, *gray;
    otsu = cvCreateImage(cvSize(dst_img->width, dst_img->height), IPL_DEPTH_8U, 1);
    gray = cvCreateImage(cvSize(dst_img->width, dst_img->height), IPL_DEPTH_8U, 1);
    cvCvtColor(dst_img, gray, CV_BGR2GRAY);
    cvSmooth(gray, gray, CV_MEDIAN, 3, 3);
    cvThreshold(gray, otsu, 70, 255, CV_THRESH_BINARY | CV_THRESH_OTSU);
    cvReleaseImage(&gray);
    if ([self filter:otsu :true] == false) {
        /* no filter took place */
        cvCvtColor(otsu, dst_img, CV_GRAY2BGR);
        cvReleaseImage(&otsu);
        return true;
    }
    else {
        cvReleaseImage(&otsu);
        return false;
    }
}

/* main processing method */
- (void)opencvProcess {
    IplImage* img = [self CreateIplImageFromUIImage: [self
        scaleAndRotateImage:image:View.image :0]];

    if (!(!only) && [self readyForOCR:img]) {
        restoreView.image = imageView.image;
        imageView.image = [self UIImageFromIplImage:img];
        cvReleaseImage(&img);
        return;
    }

    IplImage *img_g = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    IplImage *bil = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    cvCvtColor(img, img_g, CV_BGR2GRAY);
    cvCopy(img_g, bil, 0);

    IplImage* lu;
    if (!(!only)) { [luSwitch on] }
    IplImage *limg = cvCloneImage(img_g);
    [self LUTransform: bil :limg];
    cvSmooth(limg, limg, CV_BLMUR, 3, 3);
    lu = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
    cvCvtColor(limg, lu, CV_GRAY2BGR);
    cvReleaseImage(&limg);
}

```

```

if (luonly) {
    imageView.image = [self UIImageFromIplImage:lu];
    cvReleaseImage(&lu);
    cvReleaseImage(&img);
    cvReleaseImage(&bil);
    cvReleaseImage(&img_g);
    return;
}

cv::Mat mserImage(bil);
cvZero(img);
IplImage *dst_img = cvCloneImage(img);
cv::Mat drawImagePlus(img);
IplImage *img2 = cvCloneImage(img);
cv::Mat drawImageMinus(img2);
[self mserImage :drawImagePlus :drawImageMinus];
//[self mserTest:mserImage :drawImage];
IplImage imgPlus, imgMinus;
IplImage *resPlus, *resMinus;
imgPlus = drawImagePlus;
imgMinus = drawImageMinus;
resPlus = cvCloneImage(&imgPlus);
resMinus = cvCloneImage(&imgMinus);
[self removeInnerContours:dst_img :resPlus :resMinus];
cvReleaseImage(&resPlus);
cvReleaseImage(&resMinus);
if (filterSwitch.on) [self filter:dst_img :false];
//
//IplImage tmpIm = drawImage;
//dst_img = cvCloneImage(&tmpIm);

if (luSwitch.on) {
    IplImage *resMinusU = cvCloneImage(dst_img);
    IplImage *resMinusUtmp = resMinusU;
    [self LUFilter:resMinusU :lu];
    [self filter:resMinusU :false];

    if (resMinusU == NULL) {
        cvReleaseImage(&resMinusUtmp);
    }
    else {
        cvReleaseImage(&dst_img);
        dst_img = cvCloneImage(resMinusU);
        cvReleaseImage(&resMinusU);
    }
}

```

```

[self mser:mserImage :drawImagePlus :drawImageMinus];
//[self mserTest:mserImage :drawImage];
IplImage imgPlus, imgMinus;
IplImage *resPlus, *resMinus;
imgPlus = drawImagePlus;
imgMinus = drawImageMinus;
resPlus = cvCloneImage(&imgPlus);
resMinus = cvCloneImage(&imgMinus);
//[self filter: resPlus];
//[self filter: resMinus];
[self removeInnerContours:dst_img :resPlus :resMinus];
cvReleaseImage(&resPlus);
cvReleaseImage(&resMinus);
if (filterSwitch.on) [self filter:dst_img :false];
if (luSwitch.on) {
    IplImage *resMinusU = cvCloneImage(dst_img);
    IplImage *resMinusUtmp = resMinusU;
    [self LUFilter:resMinusU :lu];
    [self filter:resMinusU :false];

    if (resMinusU == NULL) {
        cvReleaseImage(&resMinusUtmp);
    }
    else {
        cvReleaseImage(&dst_img);
        dst_img = cvCloneImage(resMinusU);
        cvReleaseImage(&resMinusU);
    }
}

- (void)performTranslation {
    self.responseData = [[NSMutableData data] retain];
    NSString *langString = [NSString stringWithFormat:@"%s", langCode];
    NSString *textEscaped = [self ocrText
stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    NSString *langStringEscaped = [langString
stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    NSString *url = [NSString
stringWithFormat:@"http://ajax.googleapis.com/ajax/services/language/translate?g=%s&v=1.0&langpair=%s",
textEscaped, langStringEscaped];
    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString]
cachePolicy:NSURLRequestUseProtocolCachePolicy timeoutInterval:10];
    [NSURLConnection alloc] initWithRequest:request delegate:self;
}

```