Visual Marker Detection Algorithm for Mobile Phone Applications

Albert Lin, *Member, IEEE*

***Abstract*—An algorithm for detecting visual markers from images taken using camera phones was designed and implemented. The algorithm begins with pre-processing filtering, proceeds with marker feature detection and refinement, and concludes with data extraction. In addition, the algorithm utilizes many linear approximations and techniques to reduce complexity in favor of speed. Results verify the effectiveness of the algorithm.**

***Index Terms*—Linear Approximation, Marker Detection, Template Matching, Visual Markers.**

I. Introduction

WITH an increase in functionality and processing power in mobile devices, many personal electronic devices are beginning to offer camera functionality and image processing capabilities. In particular, nearly every mobile phone available on the market today is equipped with a VGA camera. This new trend and added functionality allows for new and innovative applications of mobile devices. This paper explores a visual marker detection algorithm for use in marker-based interaction applications of camera phones, as in [1]-[3].

Such applications in camera phones require fast detection and decoding, or even real-time processing, while offering very little processing power to perform the task. Thus, the visual marker detection algorithm proposed here aims to reduce complexity by making suitable approximations and assumptions given the nature of the marker format described in [1] (a sample marker image is shown in Fig. 1). By trading off the ability to handle markers in extreme, and thus rare, scenarios, the algorithm aims to deliver reasonable results with much decreased latency. Most notably, the algorithm assumes that the user will focus the marker at a moderate distance without extreme tilt. This assumption translates into the *square and parallelogram assumption* discussed in Section C.

The developed algorithm was implemented in MatLab and run against several training images. The results demonstrate a high degree of accuracy for the general case of images without much distortion, and diminished accuracy for images with a great deal of tilt or other distortion. As expected, the algorithm latency increased significantly when support for the extreme cases were added.

Manuscript received June 2, 2006. This work was supported in part by the Stanford Center for Image Systems Engineering (SCIEN).

A. Lin was with the Massachusetts Institute of Technology, Cambridge, MA 02139 USA. He is now with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: mrlin@stanford.edu).

II. Detection Algorithm Design

A. Overview

The detection algorithm consists of three phases. The Pre-processing Phase accepts the input image and converts it into a reduced format for manipulation. It enhances the image to retain relevant information, and otherwise discards unnecessary information. All subsequent phases process this image, and the original is never revisited. The Detection Phases proceed to detect distinguishing features of the makers – the corner squares and guide bars. These phases use multiple levels of filtering to iteratively reduce the work load for the final template matching step in Detection Phase II. Lastly, once the markers have been detected, interpretation of the data bits is relatively straightforward in the Data Extraction Phase. During Iteration, the entire algorithm is repeated, using a different set of input parameters based on the results of the previous run. The purpose of the Iterations is to extend the algorithm's scope of detectable distorted markers.

B. Pre-processing Phase

The Pre-processing Phase modifies the image such that the Detection Phases can efficiently find the markers. As a first step, the image is converted into an intensity image and sent through a low-pass filter to remove noise. Then the image's contrast is enhanced using a simple, yet intelligent, technique which correctly decides if a region is light or dark depending on its neighbors. Lastly, the image is sharpened to output a black and white image suitable for the latter phases. Fig. 1 shows an input image prior to pre-processing.

1) Noise Reduction:

Noise reduction is an important issue since the resolution in most camera phones is quite poor. In addition, the limited storage and processing capabilities of mobile phones often implies lossy compression formats. Thus, noise reduction is critical in correctly determining dark and light pixels.

In accordance with the main goal of reducing latency, a simple, yet effective, low pass filter is used to reduce noise. A filtering kernel[1] of 3×3 1's was used to reduce noise and

[1] All filtering kernels were normalized in implementation such that the sum of all elements is 1.

fluctuations in color intensities.

Such filtering does blur the image, causing some edges to smear across pixels. Since the average size of a bit square in the marker is only a few pixels, such blurring could become a problem. Most severely, small regions of white enclosed by large regions of black are much darker than other white regions due to the low camera quality. With blurring, these small regions become even darker, while small regions of black surrounded by large regions of white become lighter. In such a case, small black regions can actually become lighter in intensity than small white regions. This poses a problem when deciding whether a region is "black" or "white". Thus, an intelligent contrast enhancement is required to compensate for this effect.

2) Relative Contract Enhancement:

As described in the previous section, relative contrast enhancement helps reverse the blurring effects of low pass filtering which might otherwise cause bits to "flip" when using an absolute intensity threshold. In addition, images taken from a camera phone may have non-uniform lighting, causing shadows in one area but not in others. In such a case, it is possible for "white" in one area to be darker in intensity than "black" in a different area of the image. Using an absolute intensity threshold would classify such pixels incorrectly. Relative contrast enhancement addresses both these issues by considering the pixels neighbors, both far and near, to better decide whether the pixel is darker than its neighbors or lighter.

The relative contrast enhancement technique implemented in this algorithm uses very simple filter kernels for increased speed. First, a pixel's 3×3 neighborhood is considered and the average intensity is found. Then, the average intensity of the four immediate neighbors of this neighborhood is found (3×3 neighborhood to the left of this 3×3 neighborhood, and 3×3 neighborhood to the top of this neighborhood … and so forth). The intensity of the pixel in question is then enhanced proportionally to the ratio of the two. Thus, this technique enhances contrast relative to its surroundings. Gray surrounded by white will be pushed darker, while gray surrounded by black will be enhanced lighter.

Specifically, this can be implemented using a 3×3 kernel of 1's to find the average intensity for a pixel's neighborhood. A second filter is run on this average intensity image to find the average intensity of the four neighboring neighborhoods. The pre-normalization kernel[1] is

$$k_{avg\ neighborhoods} = \left[\begin{array}{cc|ccc|cc} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & \underline{0} & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right].$$

This kernel sums each of the four neighboring neighborhood averages already computed from the previous filter[2].

Finally, having computed the average intensity of the pixel neighborhood and its adjacent regions, the ratio is found and the pixel's intensity is multiplied by this ratio. Thus, if the pixel's average intensity is darker than its surroundings, its enhanced intensity will be forced lower, and vice versa. But if the pixel's average intensity is the same as its surroundings, its enhanced intensity will remain relatively unchanged.

This relative contrast enhancement technique very effectively prevents pixels from being misinterpreted. The output of this step is shown in Fig. 1.

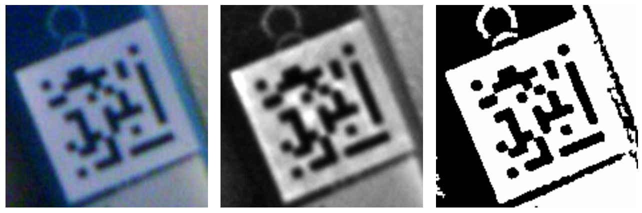

Fig. 1. (Left) Sample Marker Image. (Middle) After noise reduction and relative contrast enhancement. Light regions surrounded by dark regions have become brighter. The result is most apparent when comparing two markers within a non-uniformly lit image. (Right) After extreme sharpening and conversion to black and white. Images are shown at 50%.

3) Extreme Sharpening:

The last step in the pre-processing phase consists of converting the intensity image into a black and white image. This step discards unused information embedded in the image that would otherwise increase processing time. For example, color information is not used in this algorithm, and the rate of transition from dark regions to white regions is also not of interest. Thus, a black and white image with sharp transitions will suffice.

By sharpening the image, edges blurred by the camera and by the low pass filtering step become more pronounced. By using extreme sharpening, each pixel is forced to take on either an extremely high intensity value or extremely low intensity value, thereby converting the image to black and white after an appropriate threshold.

The sharpening kernel[1] used is

Note that the implicit normalization of this kernel when filtering actually multiplies all the values by a factor of

$$k_{sharpening} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & \underline{8.3} & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

approximately 3 due to the small center weight. Thus, the small choice of the center weight significantly increases the ratio of the intensity of a bright pixel surrounded by darker

[2] A faster implementation of this averaging could be to pick out just the four points of interest and average them, instead of computing the sum of multiplications that includes 45 zeros. The specific speed difference will be system dependent. Our tests indicated no significant speed increase when using this technique over straightforward kernel filtering.

pixels to the intensity of a dark pixel surrounded by brighter pixels, hence extreme sharpening.

Lastly, the image is threshold to yield a black and white image. All following phases operate only on this black and white image and its derivatives. The output image is shown in Fig 1.

C. Detection Phase I – Corner Squares Detection

The detection of the marker is split into two phases. First, the three corner squares of the marker are found. Then in the second phase, the bottom right guide bars are detected. Corner squares detection utilizes the first of many simplifying approximations characteristic of this algorithm. Fig. 2 shows the outputs of this phase.

1) Distortion Independent Template Matching:

Potential corner squares of the marker are found via template matching[3]. Unfortunately, "squares" are not rotation independent, nor are they tilt independent. Thus, it seems inevitable that some information regarding the rotation and tilt of the marker must first be found before detection can continue. However, each "square" can be approximated to first order by a circle (which can be inscribed within the square) given the quality of the camera phone images. Unlike squares, circles are rotation independent. Furthermore, the tilt distorted square is still well approximated by a circle, which in a sense ignores the far edges and corners of the square that are most susceptible to the tilt distortion effects and focuses on the center "body" of the square. Thus, by using a circular template to approximate the desired squares, the problem is much simplified and direct processing can begin.

The template kernel used here consists of a circle roughly 3 pixels in radius. The values within the circle are negative to detect for black by penalizing for white. Outside the circle is a 2-pixel wide ring of 0's. The 0's are effectively a "don't care" and do not contribute to the score regardless of the pixel value. Beyond this ring is another ring 3-pixels wide of positive values. This outer most ring checks and rewards for an enclosing white border, which is characteristic of the corner squares of the marker.

This template correctly identifies all regions that consist of roughly a black 6-pixel center with surrounding white. Since this returns not only the corner squares of interest, but also other similar textures in the image, further elimination is performed. To reduce the amount of work for the following elimination and filtering steps, a dynamic threshold is implemented here to control the number of matches returned. The resulting number of potential corner squares returned falls between 15 to 25 points with great probability.

2) Likelihood Filtering:

Likelihood filtering is fundamentally the process of iterating through each potential and eliminating ones that are not likely corner squares. The previous step looks and identifies suitable matches on an individual basis. Thus, this step performs additional filters by looking at pairs of matches, and ultimately at triplets. Thresholds and checks are designed such that the probability of a false negative is significantly low.

[3] Template matching kernels are not normalized in implementation. Often, the absolute value of peaks can provide information on the nature of the image region, thus allowing more accurate interpretations of the peaks.

The filtering step consists of many substeps. First, for any two points, consider them as the diagonal corner squares in the marker (the top right and bottom left corner squares). Check to the distance between them. Then compute an estimate location for the third corner square (the top left square). Check for the existence of a black square in the vicinity of the estimated location by referring to the output of the previous template matching step. Lastly, if a black square is indeed found in the expected location, estimate the location of the bottom right corner of the marker. Check to make sure this corner point lies on (or near) a black region by referring to the black and white image. Then, using these 4 corner points, check for a white border around the potential marker area. Consider these three corner squares as a potential triplet if it passes all the above checks (and keep the 4th corner point as an estimate for the next phase).

The most important part of this filtering step is to quickly and effectively filter down the number of potentials before the next phase. Since it iterates through pairs of points and a subset of triplets, many approximations are made to simplify and speed up the calculations for each pair/triplet. The most important approximation is the *square and parallelogram approximation*. The fundamental problem is: Given two points that are diagonals of a quadrangle, how can the remaining two points be found? The marker can be of any quadrangle shape given an unknown distortion, tilt, and size, and thus, the problem is actually unsolvable with the information given. By making the *square approximation*, this problem can be solved with almost no effort, avoiding a long series of processing that would otherwise be necessary to extract tilt information. Assuming the quadrangle is now a square, the third corner square of the marker must be either a 90^o or 270^o rotation of any corner point about the diagonal midpoint (since it is unknown which point of the pair is the top right point and which is the bottom left point). And rotation of a point (relative to the diagonal midpoint) by 90^o or 270^o is simply multiplying it by a 2×2 matrix consisting of only 0's, 1, and -1, which is essentially reordering and inverting one of the coordinates. Thus, the estimated third corner square location is very quickly found.

Once given the estimated location, a search region is defined and the search begins to find a black square near the estimated location. A distance matrix of weights inversely proportional to distance is pre-computed and stored in a separate file. This allows the algorithm to quickly find the nearest black square (as returned by the previous template matching step). With this new information of the actual location of the third corner square, the bottom right corner of the marker can be estimated. Since more information is now available, we assume a slightly more relaxed shape – the parallelogram. To find the bottom right corner point, simply rotate the third corner square around the diagonal midpoint, which is again just a reordering and inversion of the coordinates.

Results demonstrated that this approximation can find the corner squares of the marker exactly and estimate the bottom right corner to within 5 pixels. The error for the estimate and the probability of a miss increase with tilt. But for a wide range of tilt (~50^o from the perpendicular), this approximation holds and the algorithm can quickly find the corner quadruplet and proceed with filtering and eliminating potentials. For higher degrees of tilt, the algorithm results in some false negatives (misses).

D. Detection Phase II – Guide Bars Detection

At this point, the algorithm has already detected likely markers and can begin to read off the data. However, there are some false positives since the cumulative criteria thus far is only that the four black regions must form a rough parallelogram. In addition, since the bottom right corner is only estimated, there may be some bit errors near the bottom right corner with error probability proportional to tilt. Thus, this phase detects for the guide bars in the marker as a further measure to filter out false positives and also to improve the location accuracy of bottom right corner estimation. Fig. 2 shows the output of this phase.

1) Guide Bars Estimation:

Similar to the problem encountered before, detecting guide bars requires tilt and actual size information of the marker. Computing this can be intensive and minimally beneficial to accuracy. Thus, approximations are again made to linearize and simplify the problem. The guide bars consist of a "horizontal bar" and a "vertical bar", which due to tilt are no longer orthogonal to each other. But, by separating them, each can be manipulated individually and then combined to well approximate distortion due to tilt.

The location of the guide bars has been estimated previously. Now, the size and the angle of the guide bars are estimated. The approximation herein is fundamentally approximating a tilt distorted guide bar (a single guide bar) as a rotated version of a non-tilt distorted guide bar. This approximation is good (in the mean square error sense) for bars with long aspect ratios since the errors only occur at the short edges. Since the given guide bars do have a long aspect ratio, this approximation holds very well.

2) Dynamic Template Creation:

First, a truly horizontal bar template is created based on the size information available from the previously determined four corner points. The horizontal guide bar template is set to 5/11th (a little less to leave some margin) the width of the marker. Then, by calculating the slope between the correct two corner points, the required angle of rotation can be found without the use of the complicated Radon or Hough transform. Similarly, the vertical guide bar is dynamically constructed to be 7/11th the height of the marker, and then rotate appropriately. The two guide bar templates are merged together to form the final guide-bars template. By separating the guide bars for manipulation, the final merged template can attain incident angles that cannot be easily obtained starting from a template with both bars. As before, the guide-bars template here also has negative weights for anticipated black regions, a ring of positive weights for detection of white space around the black regions, and regions of "don't care's" between the two to account for sizing and alignment discrepancies. Then template matching and thresholding is used to identify true markers.

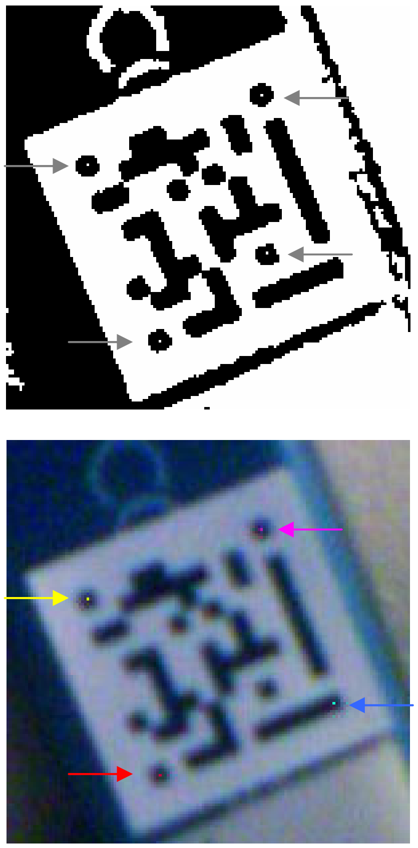

Fig. 2. (Top) Image illustrating corner squares detection via template matching. The three corner squares of the marker are correctly detected. In this case, another data bit square also with surrounding white is detected. (Bottom) Marker detection after both detection phases. The four corners here very accurately determine the marker. The Red and Magenta points signify the diagonal pair. The Yellow point signifies the top left corner which was detected using the square approximation. And the Blue point is the found location of the bottom right corner through template matching. Images are shown at 100%, and superimposed over black/white and color images for visualization only. Indicator arrows were added to facilitate detection of the single pixel outputs.

3) Location Refinement:

With one pass through this 2nd Detection Phase, almost all false positives are removed. In the rare cases where the marker appears to have a marker within itself (the data bits within form a shape similar to mini guide bars), this algorithm returns both; otherwise this technique eliminates all other false positives.

In addition, this technique also provides a new, more accurate estimate of the bottom right corner of the marker (based on the guide-bars template matching results). With this result, the algorithm can proceed to read off the data bits.

However, in cases of high tilt (~ > 50^o) the location of the bottom right corner of the marker may not be accurate enough to read all the bits with 100% accuracy (estimated overall bit accuracy is 95% at high tilt). Thus, this entire phase can be repeated, using the new coordinate information to better estimate guide bar sizes and angles, which will create a better template and find a more accurate result. This is of course very costly, but at this point, there are very few markers left and the iteration runs quickly through all the remaining valid marker entries. The algorithm runs this second Detection Phase for a total of three times, with the majority of the time spent on the first iteration. Results indicate that this iterative location refinement technique can increase corner location accuracy from ±5 pixels to within ±1 pixel. For small markers, this improvement in accuracy may be a big proportion of the marker length.

Having determined the four corner points that define the marker, data extraction can begin.

E. Data Extraction Phase

The data extraction phase actually includes a filter to remove redundant marker counts, though rare. The reason why this redundancy check is included here is because it is optional. Given the scoring mechanism for measuring our performance, it was deemed slightly beneficial (in terms of expected score) to include this redundancy checker. However, since the expected values with or without the checker are so close, it is really a question of risk aversion.

Here, the term redundant means two markers that are found to overlap, but are not identical repeats. Thus, they have a different origin and consequently different data output. Thus, using the checker to remove redundancies increases the variance in the score. The risk is higher, but the reward is also higher. Without the checker, one of the markers is likely correct, while the other is likely wrong, yielding a low variance, a low risk, and a low net reward. Since the number of markers at this point is somewhere between 1-4 (due to redundancies), the cost of running this checker is insignificant. Thus, it is a matter of personal preference.

Since this checker is included in the final version of the algorithm, it is briefly described here. The checker finds the midpoints of each potential marker. Since no two markers can overlap, it is necessarily the case that no two midpoints can be closer than the length of the marker. Thus, if such a case is detected between a pair of markers, the marker with the largest area is retained. As described before, certain markers seem to have a marker within a marker. Though rare, given a redundancy is detected, this is the most likely cause of the redundancy. Thus, keeping the largest marker correctly eliminates the marker subset false positive. The area of a quadrangle is approximated by the product of its diagonals.

1) Inverse Coordinate Transform:

A common method of processing markers involves transforming the marker from the image coordinates to the marker coordinates (reverse tilt-distortion, thus resulting in a frontal view of the marker). However, as seen in the previous sections, this algorithm has cleverly circumvented such intensive calculations thus far. Rather, the algorithm here will transform a subset of coordinates from the marker coordinate system to an approximated image coordinate system.

In addition, each marker is on average 100 pixels wide and 100 pixels tall in the camera phone image, thus containing a total of 10,000 pixels. Thus, transforming a marker into the image coordinate system will require transforming 10,000 coordinates. On the other hand, finding the points of interest in the marker coordinate system first, then transforming only those 121 coordinates into the image coordinate system only requires 121 computations. Furthermore, transforming from marker coordinates to image coordinates can be simplified greatly by making a series of disjoint linear approximations to perform the transformation. Thus, the algorithm only computes these 121 points.

The four corners of the marker in image coordinates corresponds to the points (1,1), (1,11), (11,1), and (11,11) in marker coordinates (all lie on the Z=0 plane). Thus, to transform all the integer points that form the 11 × 11 grid in marker coordinates into image coordinates, simply find points linearly along the line connecting the top left corner to the top right corner (in image coordinates). These points define the start of each column. Similarly, find points linearly from the bottom left to the bottom right. These define the end of each column. Now, for each column, linearly calculate points along the line connecting the top of the column to the bottom of the column. Note that this technique does not necessarily produce columns that are all parallel, which is desirable since this freedom can largely account for distortion due to tilt. The set of points found are then approximations to the transformation of the marker coordinates to the image coordinates.

2) Bit Decision:

For each of the 121 points found above, now decide if the bit is black or white. This is done by referring to the black and white image after pre-processing. A majority vote in a 3×3 window determines the bit decision.

Lastly, a find sorting of the bits is done to remove bits that are not data bits (such as the corner squares and the guide bars). Alternatively, these bits can be omitted directly during coordinate transformation, but the cost is small to include them. Thus, they were included as a check and for easier programming. Fig. 3 shows the outputs of Data Extraction.

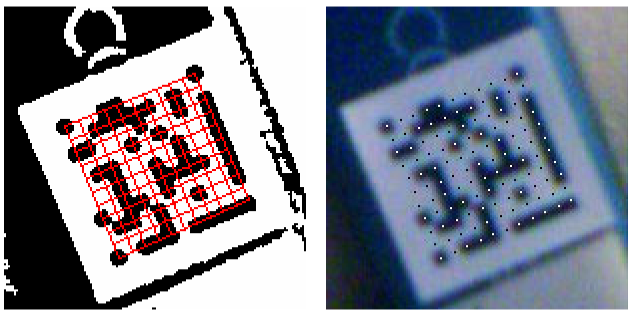
Fig. 3. (Left) Image illustrating the linear approximation algorithm to find the bit locations in the image coordinates. Bits are sampled at the intersections. (Right) Results of bit decisions superimposed on the original input image. White dots indicates a logical '1' (corresponding to a black square) and black dots indicates a logical '0' (corresponding to a white square). Images are shown at 75%.

F. Iterations

Finally, the algorithm has run one pass through the image, found the markers, and extracted the data. With one pass, the algorithm has a high marker detection accuracy. However, if even higher accuracies are desired, the algorithm can be run again with a different set of input parameters to detect the more extreme markers (highly distorted, unusually small or large…etc).

The greatest weakness of this algorithm is that it assumes the size of the marker will be around 100 pixels by 100 pixels. Since this may not be the case depending on camera distance to the marker, the algorithm is set to run a total of three times. The first time it checks for normal markers, the second pass it checks for small markers, and the last pass it checks for large markers. As expected, checking for small markers takes the most amount of time since it requires a finer search; and detecting large markers takes the least amount of time. In addition, the algorithm has checks to dynamically adjust the parameters into the second and third run depending on the results of the previous runs. It also terminates early if all three[4] markers are found.

III. Results

The algorithm was designed using 6 training images and then tested again another 6 training images, for a total of 12 test images. The results are briefly summarized below.

TABLE I
RESULTS OF TEST RUN ON 12 IMAGES

Final Score	**1909/1909**
Marker Detection Rate	100%
Bit Reading Accuracy	100%
Marker False Positives Rate	0%
Marker Repeat Rate	0%
Run Time Total	~180s
Run Time per Image	> 10 s and < 20s Avg: ~15 seconds
Average Run Time per Marker	~ 10 seconds

Tests were run on a Dell Optiplex Pentium 4 3.2 GHz machine. Run times are system dependent.

Some output images are shown in Fig. 1-3 for visualization purposes. Note that many of these images are generated for the sole purpose of visualization and that the algorithm does not necessarily include all of the computations hinted in these visualizations. The above results are run with all visualizations off.

Overall, the algorithm is a great success. It detects all of the markers and bits with 100% accuracy, and runs each image in less than 20 seconds. Since the goal is fast speed processing, the next section examines some speed-accuracy tradeoff options available to this algorithm should the user desire to alternate the operating point.

[4] It is assumed that the maximum number of markers in an image is 3, and the minimum is 1.

IV. Analysis

By adjusting parameters as well as iterations of the Location Refinement and of the entire algorithm, the accuracy of the algorithm can be traded off for speed. Fig. 4 illustrates some design trade off points. As expected, this algorithm runs fairly fast when moderate accuracy is required. To handle extreme cases, some of the approximations may start yielding noticeable errors, and thus the algorithm must work significantly harder to compensate for this. Thus, the algorithm takes much longer to run to reach 100% accuracy. This is expected of an algorithm designed for high speed with approximations tailored to handle the general, everyday cases.

This analysis also helps identify the critical steps in the algorithm which deliver most of the algorithm's accuracy, or the steps in the algorithm which consume the most time. Table 2 lists the average runtimes for each step using a stopwatch program running parallel to the algorithm. The stopwatch program is not expected to significantly affect the program runtime. The runtime breakdowns shown in Table 2 are consistent with the speed trade-off points in Fig. 4.

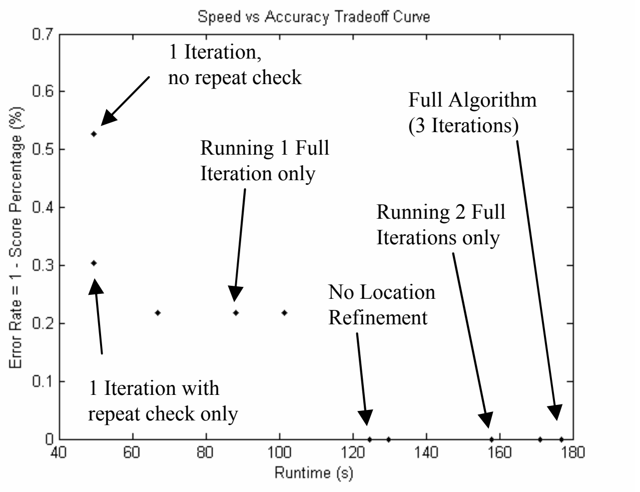

Fig. 4. Speed versus Accuracy (Runtime vs. Error Rate) Tradeoff curve. The graph illustrates a few operating points obtained by adjust parameters and iterations in the algorithm. 12 test images were used in the run. Runtime represents the total time over the 12 images. The algorithm can be adjusted to run at 100% accuracy with as little as 120 seconds (10 seconds per image). And if the extreme cases of tilt were not present, then the algorithm could run at 30-60 seconds (3-5 seconds/image) (80% accuracy here corresponds to 100% accuracy for general, non-extreme markers). The steps (discontinuities) in the curve hints that the algorithm only errs by missing markers, never by deciding bits incorrectly. Results confirm that for the markers detected, the bit accuracy is 100% for all cases. Thus the error percentage is crudely quantized into steps corresponding to 83 bits = 1 marker.

TABLE II
AVERAGE RUNTIMES BREAKDOWN

Total (per image)	**~15.4s**
Preprocessing Phase	~2.1s
Noise Reduction	~0.1s
Relative Contrast Enhancement	~1.7s
Extreme Sharpening	~0.3s
Detection Phase I	~2.9s
Corner Squares Template Matching	~1.1s
Likelihood Filtering	~1.8s
Detection Phase II (first iteration)	~5.6s
Guide Bars Estimation	~0.6s
Dynamic Template Creation/Matching	~5.0s
Detection Phase II, 2 more times (a.k.a Location Refinement)	~4.8s total (~2.4s each)
Data Extraction Phase	Insignificant
Inverse Coordinate Transformation	Insignificant
Bit Decisions	Insignificant

Tests were run on a Dell Optiplex Pentium 4 3.2 GHz machine. Run times are system dependent. The time reported are total times spent in each step for three iterations of the whole algorithm. The test images contained three markers over a complex background. Times shown are for a single image.

V. CONCLUSION

In conclusion, the algorithm proposed herein takes several approximations utilizing characteristics and features of the marker to simplify the processing complexity to deliver high accuracy at fast speeds. At one end of the speed-accuracy curve, the algorithm can detect with 100% all the markers and bits at about 15 seconds/image. And at the other end of the curve (at reasonable accuracy), the algorithm performs at ~80% at about 5 seconds/image.

ACKNOWLEDGMENT

The author thanks Prof. Bernd Girod, Aditya Mavlankar, and Chuo-Ling Chang, as well as the Stanford Center for Image Systems Engineering, for their help and guidance.

REFERENCES

[1] M. Rohs, “Real-World Interaction with Camera-Phones,” at the 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004), Tokyo, Japan, November 2004.

[2] M. Rohs, F. Mattern, Visual Code Recognition for Camera-Equipped Mobile Phones. [Online] Available: http://www.vs.inf.ethz.ch/res/proj/visualcodes/index.html .

[3] M. Rohs, “Marker-Based Interaction Techniques for Camera-Phones” in 2nd Workshop on Multi-User and Ubiquitous User Interfaces (MU3I) at IUI 2005, San Diego, California, USA, January 2005.