# Identifying and Reading Visual Code Markers

Oren Feinstein, Electrical Engineering Department, Stanford University

***Abstract*— A visual code marker is an 11x11 element pattern with two fixed guide bars and three fixed cornerstones, which is used to uniquely label everyday objects such as CDs, DVDs, books, etc. Each marker has an 83-bit data code that can be captured using any digital camera and transmitted over the Internet to a central database that stores more information about the labeled object. This paper describes a method to find the locations of visual code markers in a 24-bit RGB image and read the 83-bit data embedded in each marker. Alternatives at critical stages of the algorithm are discussed. Many different images are shown to prove robustness.**

***Index Terms*— digital image processing, visual code markers, adaptive thresholding, rotation and scale invariance.**

## I. Introduction

The goal of this project is to find all of the visual code markers in a 24-bit RGB image regardless of orientation using MATLAB. Upon identifying a visual code marker, the origin is reported as well as the 83-bit data embedded in the marker.

After explaining the method used in this study, the results are shown and discussed.

## II. Methodology

### A. Definition of Visual Code Marker

A visual code marker is defined as an 11x11 square pattern with two guide bars and three cornerstones as shown in Figure 1 below. The 83 data bits are enclosed in red, and are read column by column, from top to bottom. A black square represents a data bit of 1, and a white square is a data bit of 0. The vertical guide bar is seven elements long and the horizontal guide bar is five. The origin (0,0) is defined as the upper left cornerstone, while the upper right cornerstone and lower left cornerstone are (10,0) and (0,10), respectively.

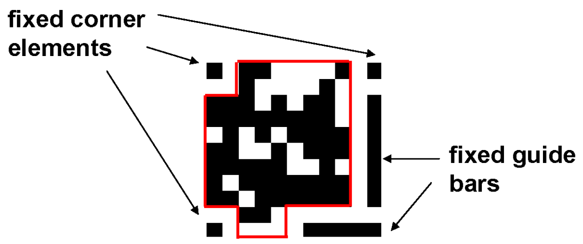

**Fig. 1.** A visual code marker example.

Manuscript received June 02, 2006.
O. Feinstein is with the Electrical Engineering Department at Stanford University, Stanford, CA 94305 USA (email: feinstei@stanford.edu).

### B. Overview of Algorithm

The algorithm implemented can be described as a five-stage process (see Figure 2). The first stage takes the input image and pre-processes it to produce a cleaner image for the next stage. The second stage creates a binary image by using a row-by-row adaptive threshold. The third stage labels all of the unique regions and extracts useful information such as eccentricity, orientation, area, etc. The fourth stage searches for a candidate vertical guide bar, and if found, searches for the horizontal guide bar and the three cornerstones. Using the three cornerstone pixel coordinates as well as the centroid of the horizontal guide bar, the fifth stage defines the 11x11 element plane and outputs the 83 data bits for each marker found.

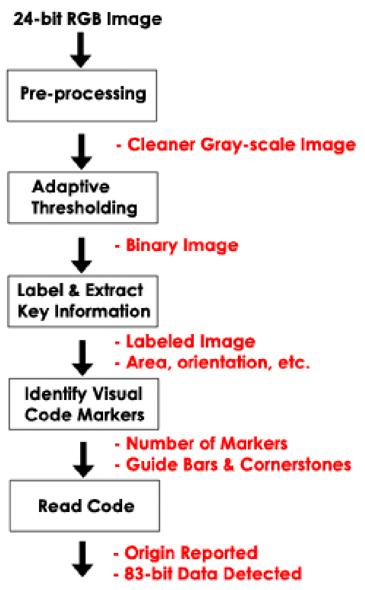

**Fig. 2.** Five-stage algorithm with outputs of each phase in red.

### C. Pre-processing

Since color is not needed in identifying the visual code markers, the first step in the algorithm is to convert the 24-bit RGB input image to a gray-scale image. However, instead of using the MATLAB function *rgb2gray*, [1] suggests weighting the gray-scale image as half red and half green because the blue component of the image has poor quality for sharpness and contrast. This optimization is also less computationally intensive and saves a fraction of a second in the overall run-time.

Occasionally, the input image can be noisy, thus further pre-processing is needed. For this application, the most frequent type of noise is blurring either due to motion of the camera while acquiring the image or a low quality camera as is common to cellular telephones. With that knowledge, a high pass filter is needed to counteract the low pass characteristics of the image.

Even if there is no noise in the image, a high pass filter is used to further emphasize the transitions between black and white, which is prevalent throughout the visual code marker. A 3x3 kernel is constructed to convolve with the gray-scale image:

$$\begin{matrix} 0 & -5 & 0 \\ -5 & 21 & -5 \\ 0 & -5 & 0 \end{matrix}$$

But, since this high pass filter strongly amplifies high frequency components, a simple 3x3 average filter is developed to attempt to maintain the fidelity of the image:

$$\begin{matrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{matrix}$$

The combination of the high pass filter followed by an average filter preserves the quality of the image and simplifies the task of the adaptive threshold since the visual code marker gets boosted in the image. Figure 3 shows the results of pre-processing for a given input image.

An alternative to the high pass and average filter combination was to use the scale-by-max color balancing technique on the gray-scale image. The idea was to boost the white components of the visual code marker and leave the black components relatively unaffected. However, the results were poor at best – the thresholding stage did not define the transitions between black and white nearly as well as the high pass and average filter.

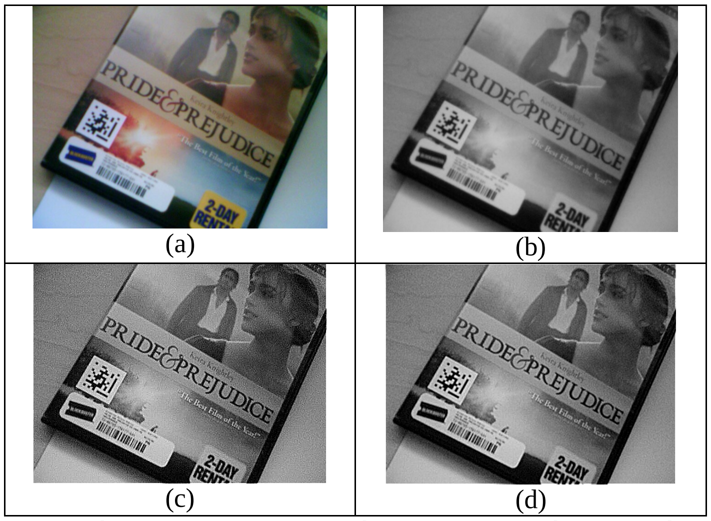

**Fig. 3.** The pre-processing stage. a) 24-bit RGB input image. b) gray = (red + green) / 2. c) Gray image after using high pass filter. d) Gray image after using high pass filter and then average filter.

*D. Adaptive Thresholding*

After pre-processing the image, the next step is to create a binary image as shown in Figure 4, where dark pixels are represented by the value 1 and light pixels are 0. According to [1], an adaptive threshold with a zigzag traversal of the scan lines is best. However, upon implementation, undesirable effects occur at even scan lines. Thus, a reasonable adaptation to [1] is to traverse row-by-row without traversing in a zigzag manner. Specifically, an average $g_s(n)$ is kept while passing through the image:

$$g_s(n) = g_s(n-1) * (1 – (1/s)) + p_n$$

where $p_n$ is the gray value of the current pixel, s = (1/8) * width of the image, and $g_s(0)$ = (1/2)*255*s. The pixel value binary_img(n) is selected by:

$$\text{binary\_img}(n) = \begin{cases} 1, \text{ if } p_n < g_s(n)/s * (100-t)/100 \\ 0, \text{ otherwise} \end{cases}$$

with t = 28.

Using a constant threshold does not produce results like the adaptive threshold. Particularly, edges are not defined in the right places and there are more false positives detected as dark pixels. No morphological operations can be used to create a cleaner binary image because erosion may remove the cornerstones since they are innately small and dilation can connect the guide bars. Also, both erosion and dilation destroys the data in the code marker.

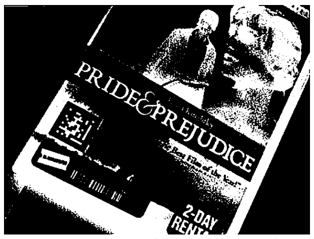

**Fig. 4.** Binary image created by adaptive threshold stage.

*E. Label and Extract Key Information*

The binary image is then labeled uniquely according to 8-connected white regions. The MATLAB function *bwlabel* is used for labeling by taking two passes through the image. The first pass traverses row-by-row or column-by-column and assigns preliminary labels for each region found. It may occur that two regions with different labels could be the same region, so a second pass handles equivalencies by assigning a final label.

After labeling the binary image, we must extract key information for each region: area, orientation, eccentricity, centroid, major axis length, and minor axis length. MATLAB's *regionprops* function is used for this step. The key information is critical in defining metrics to search for candidate guide bars and cornerstones.

### F. *Identify Visual Code Markers*

To find all of the visual code markers in the labeled image, the first step is to iterate through all of the unique labels and find potential vertical guide bars. This is accomplished by filtering out any regions that do not satisfy eccentricity and area conditions. For instance, the eccentricity as defined by MATLAB should be close to 1 for a guide bar. Also, if a human cannot see and read the visual code marker, then the computer likely will not either, thus there must be a constraint on size. Since the vertical guide bar is 7 elements long, a good threshold for size is typically 25 pixels or roughly 4 pixels per element.

After finding a potential vertical guide bar, its orientation is used to locate a horizontal guide bar and three cornerstones. Only when the two guide bars and three cornerstones are found can a visual code marker be successfully classified.

The process employed to find the horizontal guide bar and three cornerstones is exemplified in Figure 5. The technique essentially is to a project a line based on the orientation angle of the guide bars past the expected location of the target object. Then, the algorithm searches a window for the target object. If the object is found, the algorithm searches for the next target object, otherwise, it starts moving the window in the direction closest to the target object.

The algorithm first searches for the horizontal guide bar by projecting a line from the centroid of the vertical guide bar. Then, if the horizontal guide bar is found, it projects a line from the centroid of the vertical guide bar in the opposite direction to find the upper right cornerstone. After finding the upper right cornerstone, a line is projected from the centroid of the horizontal guide bar to search for the lower left cornerstone. Finally, a decision needs to be made on how to search for the upper left cornerstone. Either the algorithm can project a line from the lower left cornerstone or it can project a line from the upper right cornerstone. Due to perspective in some images, the choice is made by taking the minimum of the absolute value of the orientation angle of the two guide bars to minimize the error. Therefore, the algorithm projects the line from the lower left cornerstone if the absolute value of the orientation angle of the vertical guide bar is less than the absolute value of the orientation angle of the horizontal guide bar, otherwise it projects from the upper right cornerstone.

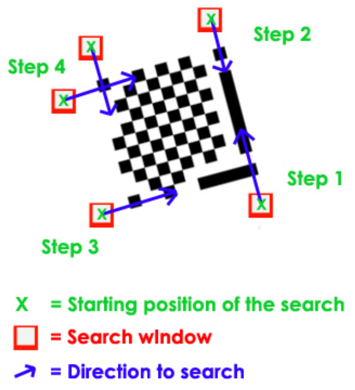

**Fig. 5.** Algorithm to search for the horizontal guide bar and cornerstones.

### G. *Read Code*

After obtaining the two guide bars and the three cornerstones, four point correspondences define the visual code marker plane. As in [1], the four points are the centroid of the horizontal guide bar and the centers of the three cornerstones:

| Element | Image Coordinate | Code Coordinate |
|---|---|---|
| upper left cornerstone | $(x_0,y_0)$ | (0,0) |
| upper right cornerstone | $(x_1,y_1)$ | (10,0) |
| horizontal guide bar | $(x_2,y_2)$ | (8,10) |
| lower left cornerstone | $(x_3,y_3)$ | (0,10) |

A code coordinate (u,v) can then be mapped to an image coordinate (x,y) by

$$x = (au + bv + 10c) / (gu + hv + 10)$$

$$y = (du + ev + 10f) / (gu + hv + 10)$$

where the parameters a through h are calculated from the four point correspondences as follows:

$$dx_1 = x_1 - x_2, \qquad dy_1 = y_1 - y_2$$
$$dx_2 = x_3 - x_2, \qquad dy_2 = y_3 - y_2$$

$$sig\_x = 0.8x_0 - 0.8x_1 + x_2 - x_3$$
$$sig\_y = 0.8y_0 - 0.8y_1 + y_2 - y_3$$

$$g = (sig\_x*dy_2 - sig\_y*dx_2) / (dx_1*dy_2 - dy_1*dx_2)$$
$$h = (sig\_y*dx_1 - sig\_x*dy_1) / (dx_1*dy_2 - dy_1*dx_2)$$

$$a = x_1 - x_0 + gx_1 \qquad d = y_1 - y_0 + gy_1$$
$$b = x_3 - x_0 + hx_3 \qquad e = y_3 - y_0 + hy_3$$
$$c = x_0 \qquad f = y_0$$

To read the code, the algorithm traverses code coordinates column-by-column from top to bottom starting with code coordinate (0,2). Since the visual code marker has built-in white space surrounding the cornerstones and guide bars, not all of the 11x11 elements in the marker are read. Specifically, the algorithm only reads the data code coordinates. For example, the left-most two columns have 7 rows of valid data, the next three columns have 11, and the next four columns have 9, totaling 83 data bits.

## III. Results

To prove the robustness of the algorithm a set of 12 training images were provided. However, the training set provided could not possibly be sufficient to test the many possible code marker and camera orientations, thus 37 more images were added to the training set.

The algorithm ran on average between a fraction of a second and three seconds, and had an accuracy of nearly 96%. The two images that the algorithm failed to detect the visual code marker were extreme cases where the camera was either too close with a severe perspective or at a perspective that skewed the cornerstones and made the horizontal guide bar appear much larger than the vertical guide bar.

Figures 6-16 show pairs of images the algorithm succeeded on. The left image of the pair is the 24-bit RGB input image, and the right image is a binary image that highlights the vertical guide bar of each visual code marker found.

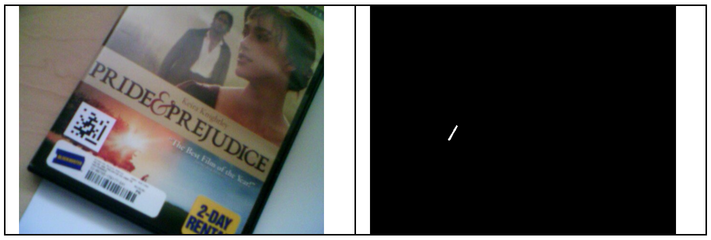
**Fig. 6.** training_8.jpg

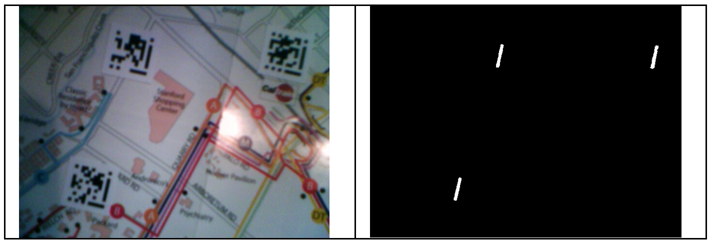
**Fig. 7.** training_9.jpg

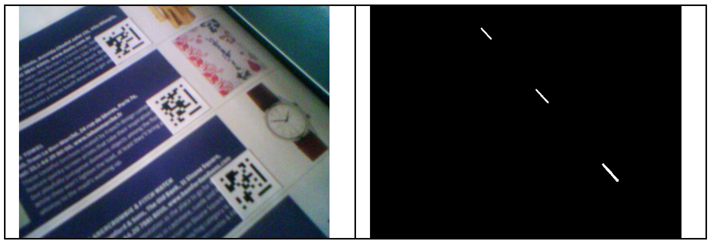
**Fig. 8.** training_10.jpg

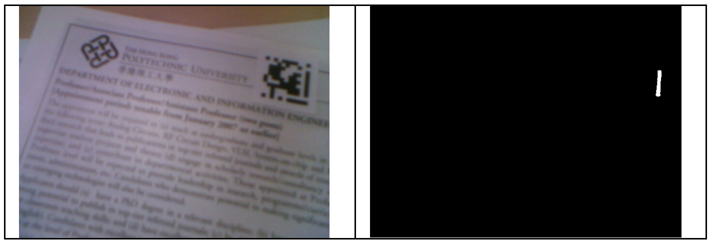
**Fig. 9.** training_11.jpg

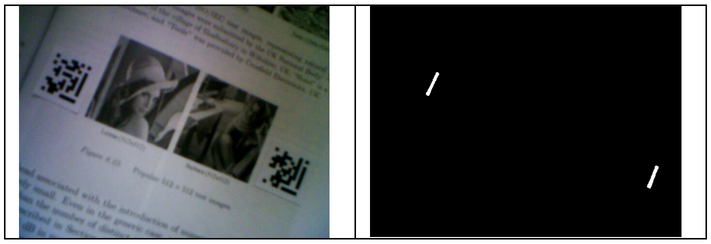
**Fig. 10.** training_2.jpg

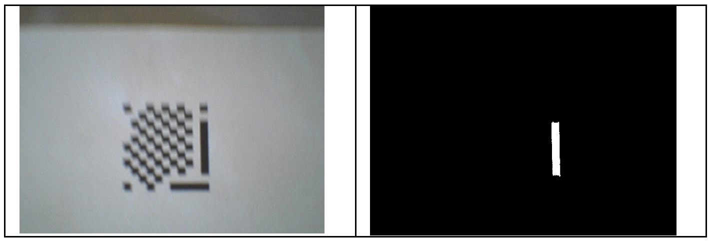
**Fig. 11.** training_29.jpg

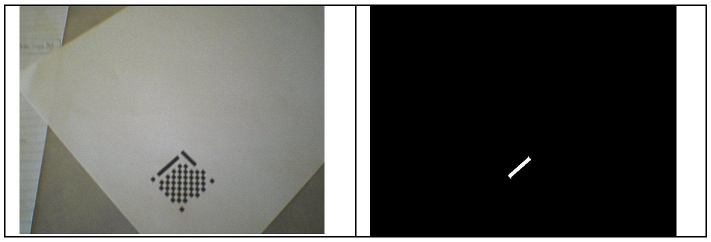
**Fig. 12.** training_32.jpg

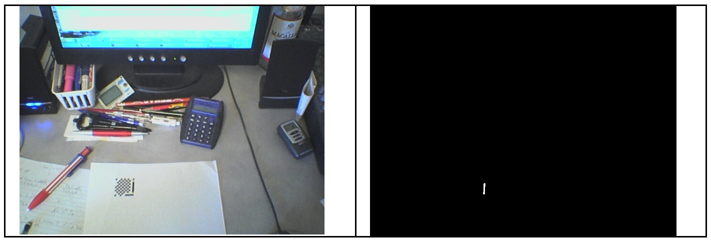
**Fig. 13.** training_45.jpg

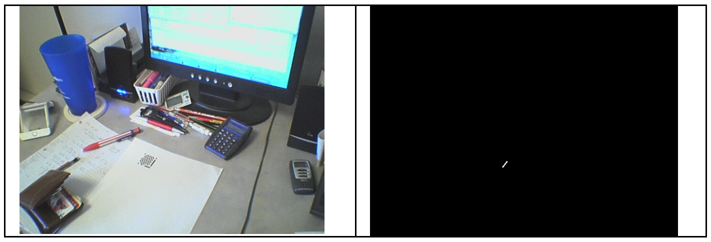
**Fig. 14.** training_46.jpg

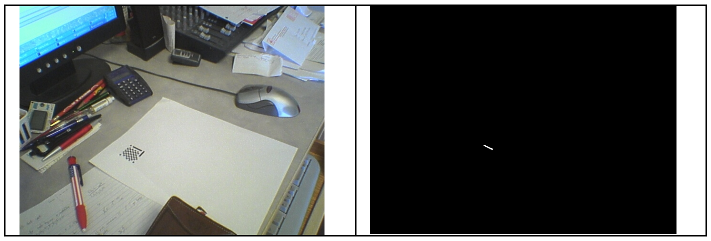
**Fig. 15.** training_49.jpg

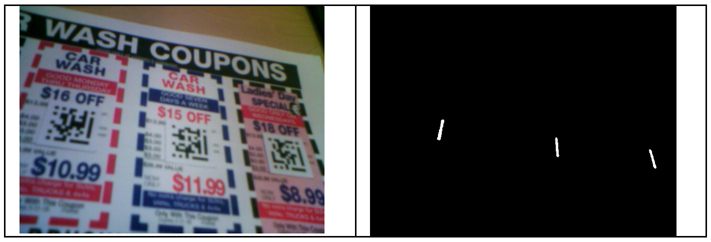
**Fig. 16.** training_5.jpg

## IV. Conclusion

An algorithm to identify visual code markers in 24-bit RGB images has been proposed. Employing a five-stage process, the algorithm identifies the origin of any visual code marker in the image and reads the 83-bit data associated with each marker. Various alternatives at certain stages were discussed. Several results were shown to prove the robustness of the algorithm. On average, the program did not run longer than 4 or 5 seconds, making it near real-time for this particular application. The accuracy was very near perfection.

Further work for this project could include optimizing the code to run faster, creating an even cleaner binary image from the adaptive thresholding stage, and exploring alternatives for identifying visual code markers that are severely skewed.

## Acknowledgements

Many thanks to Professor Bernd Girod, Chuo-Ling, and Aditya for conducting an excellent image processing class.

## References

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

[2] M. Rohs, “Visual Code Widgets for Marker-Based Interaction”, IWSAWC ’05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems – Workshops (ICDCS 2005 Workshops), Columbus, Ohio, USA, June 6-10, 2005.