Visual Code Marker Detection

May Zhou, Kanako Hayashi, Yuki Konda

I. Introduction

THE purpose of this project is to detect and read visual code markers in color images taken by camera phones. We are given a set of twelve training images, each containing up to three code markers with arbitrary location and orientation. Each code marker is an 11 x 11 array of black and white elements, 83 of which represent variable data bits. The remaining 38 elements are fixed and consist of three corner elements and two guide bars. Figure 1 shows a code marker with standard orientation, where the two guide bars are in the lower right corner.

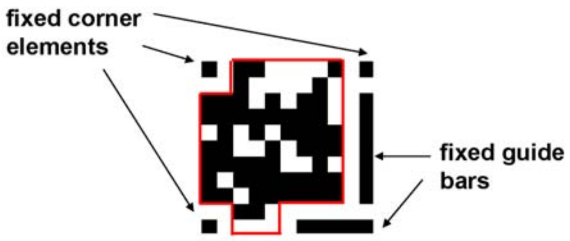


Figure 1. Code marker in standard orientation.

Our task is to determine the center pixel of the upper left corner element for each code marker and extract the 83 bits of data encoded. This task is complicated by several factors. First, each code marker element can span multiple pixels and have variable size, depending on its relative location within the image, and the angle, distance, and perspective from which the image was photographed. Second, each code marker can be rotated and distorted by unknown transformations, which must be reverse-engineered from the given image. Third, the contrast and visual quality of camera phone images can be quite low, thus making it difficult to apply simple color segmentation to distinguish black and white elements. Fourth, code markers can be interspersed among background objects of similar color, size, and shape, which make it difficult to detect genuine code marker regions. Finally, our processing time is restricted to no more than one minute per image.

In this project, we develop a completely original, robust system that combines edge detection, morphological processing, region labeling, object recognition, quadrant classification, and projective transformation, to accurately detect and read visual code markers. A system flow chart is given in Figure 2. During development, we tested and rejected many other techniques, including color segmentation, template matching and area correlation, matched filtering, and local statistical analysis in both spatial and frequency domains. Our current system achieves 100% accuracy on all twelve training images, as well as an augmented set of 36 additional images obtained from 90º rotations of the original set. The system requires no more than 10 seconds per image on a 3.6 GHz machine. Sections II through VI describe the steps in our algorithm. Section VII provides performance results and runtime measurements. Section VIII summarizes the project and our contributions.

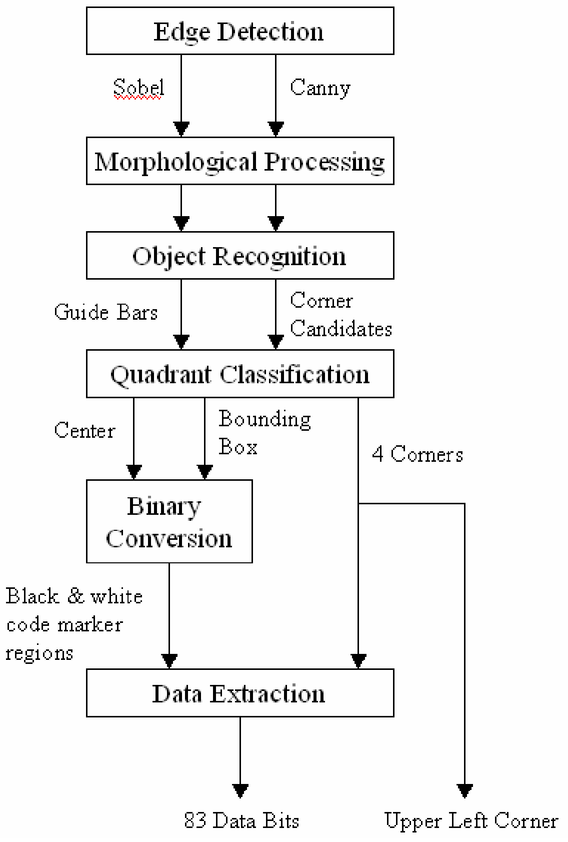


Figure 2. System flow chart.

II. Edge Detection and Morphological Processing

The first step in our algorithm consists of edge detection followed by edge refinement through morphological processing. This step extracts object boundaries, which are the relevant features needed to detect code markers.

A. *Edge Detection*

To apply edge detection, we first convert the color RGB image to grayscale using the formula

$$Y = 0.2989\, R + 0.5870\, G + 0.1140\, B \tag{1}$$

where Y is between 0 and 255.

We then apply two different edge detectors to target the two types of subregions that are fixed in each code marker: guide bars and corner elements. Sobel edge detection is used to enhance straight lines, which are characteristic of guide bars. Canny edge detection is used to preserve small round regions, which are characteristic of corner elements. Both operators are applied to the grayscale image to produce two separate edge images, which are processed in parallel in the next step of our algorithm.

The Sobel and Canny operators were carefully selected for their edge detection properties. Sobel favors the well-defined linear edges of guide bars but disrupts small closed contours and overlooks weak edges. Canny preserves small closed contours for corner elements, even when edges are weakened by poor contrast, but tends to be overly sensitive and produce too many spurious edges. These properties can be explained as follows. Sobel uses two filter kernels, one for horizontal gradients and another for vertical gradients, which together pick out linear and piecewise linear edges:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Canny finds edges by looking for the local maxima of the gradient calculated using the derivative of a Gaussian filter. Two thresholds are used to detect strong and weak edges, resulting in a higher incidence of spurious edges. Figures 4-6 compare Sobel and Canny results for training image 1, along with results from three other edge detection operators that were tested and rejected. Roberts was rejected for producing too many small broken edges, Prewitt for its weaker gradients and relative inferiority to Sobel, and Laplacian of Gaussian for its oversensitivity and enlargement of contours.

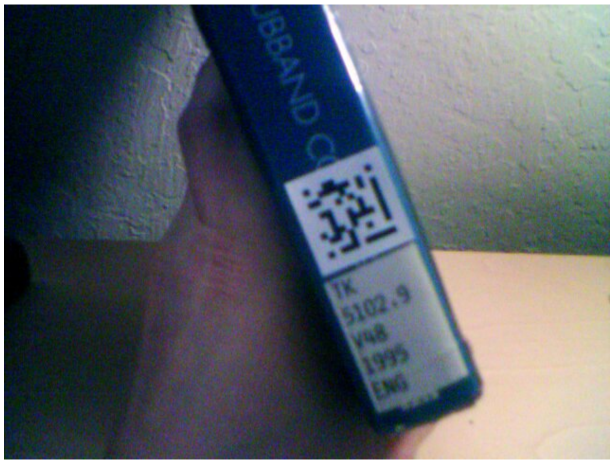


Figure 3. Original image.

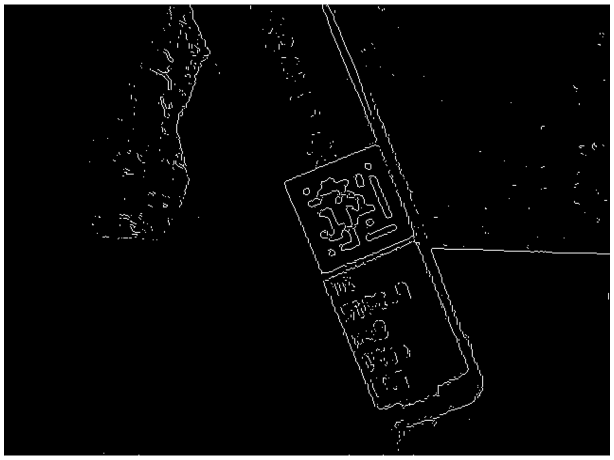


Figure 4. Sobel edge detection.

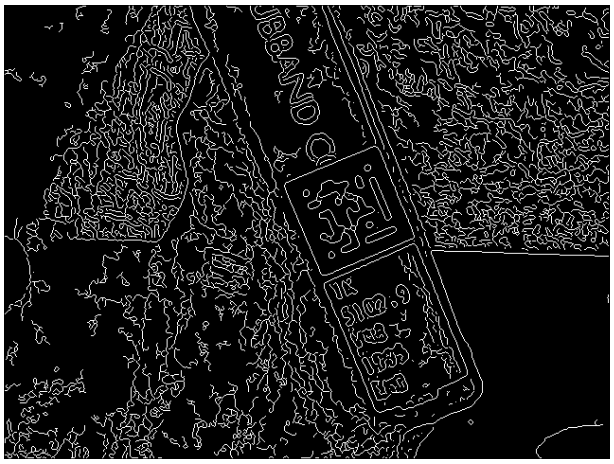
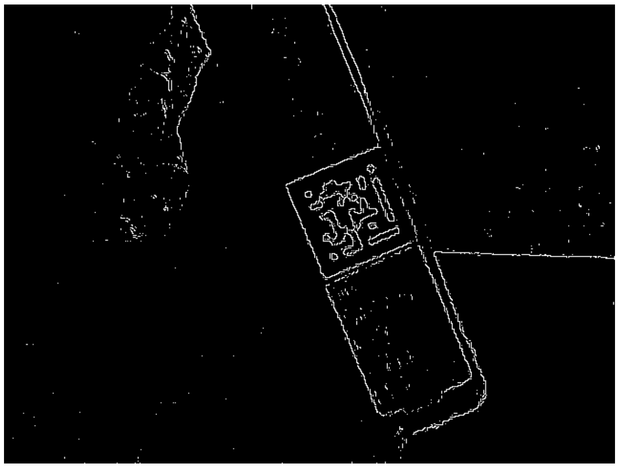
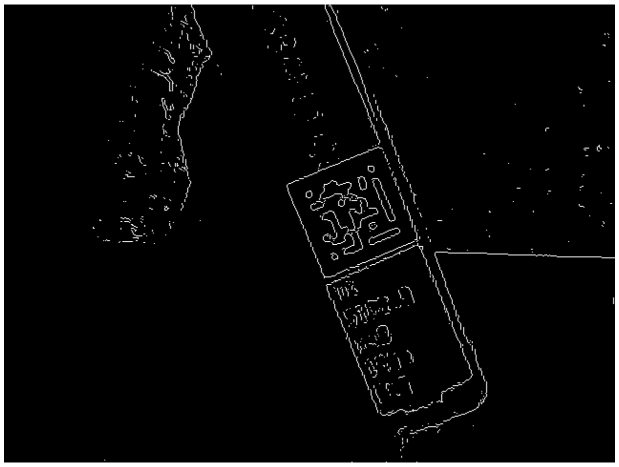


Figure 5. Canny edge detection.


(a)


(b)

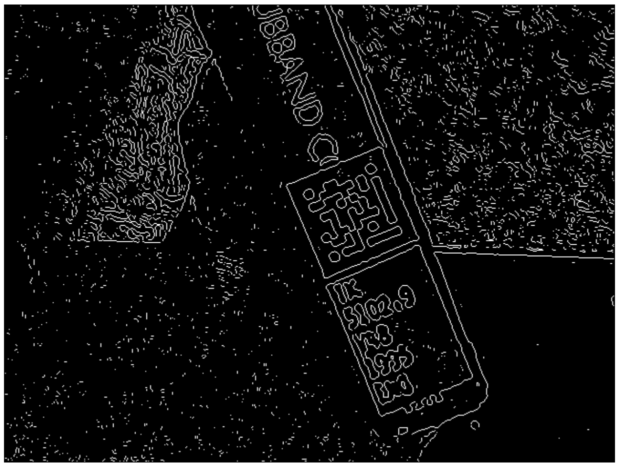
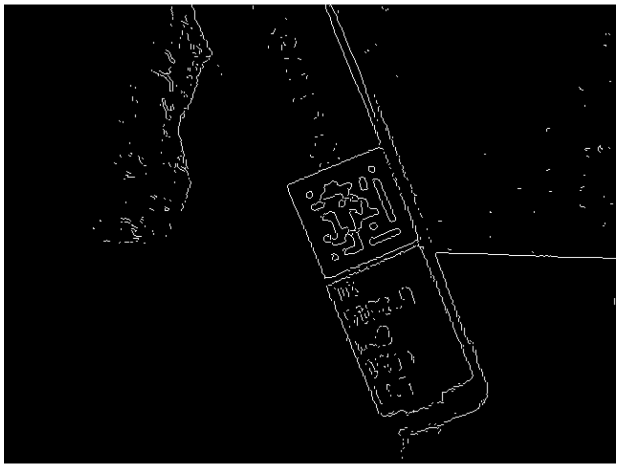
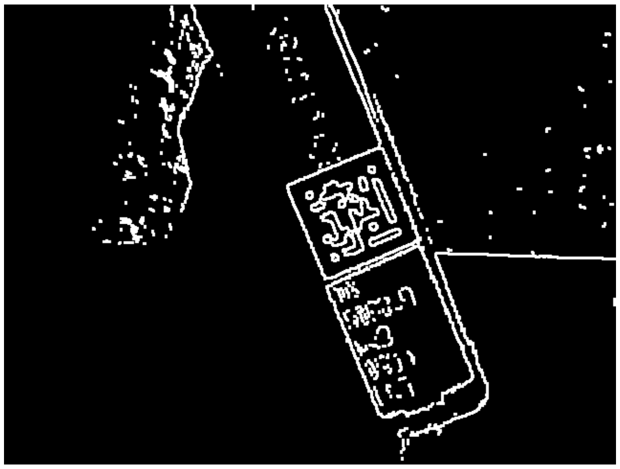

(c)

Figure 6. Rejected edge detection schemes. (a) Roberts, (b) Prewitt, (c) Laplacian of Gaussian.


(a)


(b)

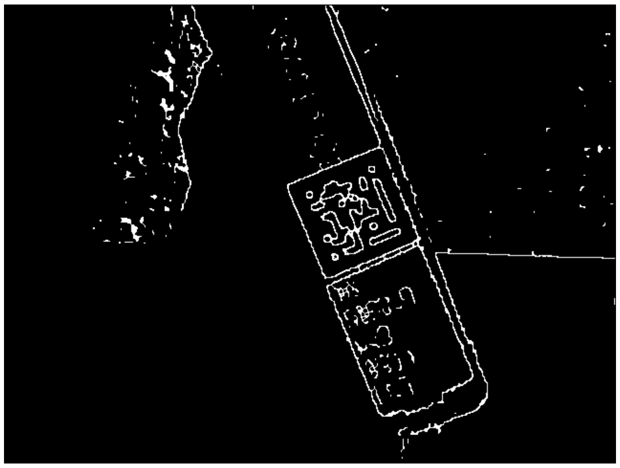

(c)

Figure 7. Morphological processing. (a) Removal of isolated pixels. (b) Dilation. (c) Thinning.

B. *Morphological Processing*

Three morphological operations are applied to both edge images to smooth and refine detected edges. First, isolated pixels are removed by eroding with the structuring element

$$\Pi_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Next, the image is dilated with

$$\Pi_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

to smooth out edge discontinuities. This results in a thickening of all edges. To compensate, the final operation thins edges to lines of minimal width. The thinning operation preserves the Euler number, which is equal to the number of objects in the region minus the number of holes in those objects. Figure 7 shows the result after each morphological operation for training image 1.

III. Object Recognition

Object recognition is performed on the two edge images to detect guide bars and corner elements. Regions in each image are labeled using 8-connected neighborhoods. Figure 8 shows the region-labeled Sobel image for training image 8. Region labels are differentiated by color.

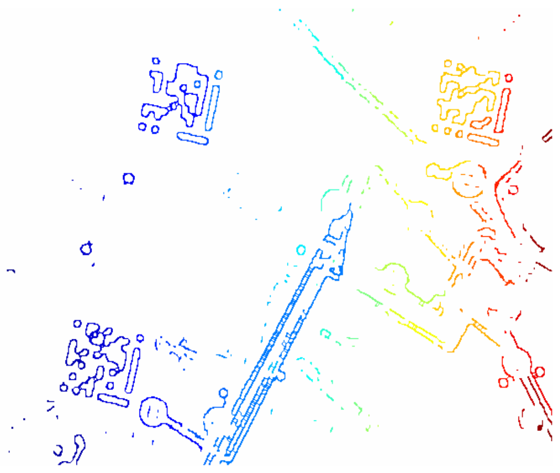


Figure 8. Region labeling.

Each region is then modeled by an ellipse with the same normalized second central moments. The attributes of these ellipses are thresholded to identify the correct regions corresponding to guide bar pairs in the Sobel image and candidate corner regions in the Canny image. Both individual and pairwise region attributes are thresholded for guide bar pair recognition, whereas only individual region attributes are used for candidate corner recognition. All threshold values are empirically determined from the given training images and are listed in Table I.

TABLE I
THRESHOLD VALUES FOR OBJECT RECOGNITION

γ_1^{maj}	30
γ_2^{maj}	90
γ_1^{maj}	6
γ_2^{maj}	20
α	85
ε	0.9
δ_1	55
δ_2	22
β	28

A. *Guide Bar Pairs*

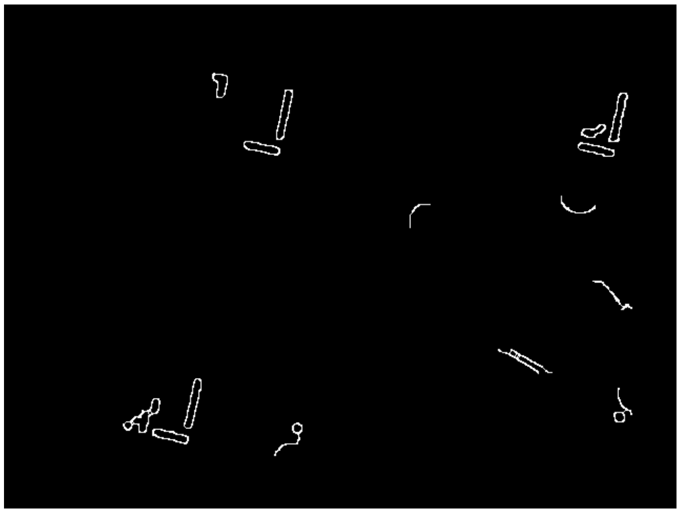
To recognize and extract guide bar pairs, we use two sets of thresholds on the Sobel image. The first set acts on individual region attributes: major axis length (M), minor axis length (m), area (A), and eccentricity (E) for each corresponding ellipse.

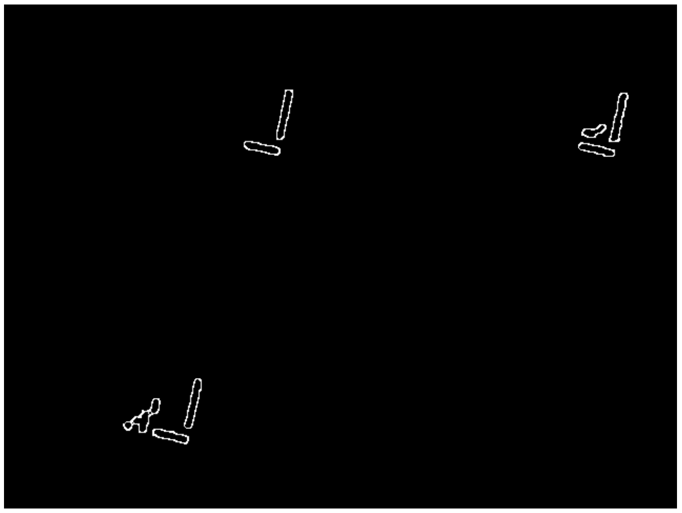
$$\gamma_1^{maj} < M < \gamma_2^{maj}$$
$$\gamma_1^{\min} < m < \gamma_2^{\min}$$
$$A > \alpha$$
$$E > \varepsilon$$

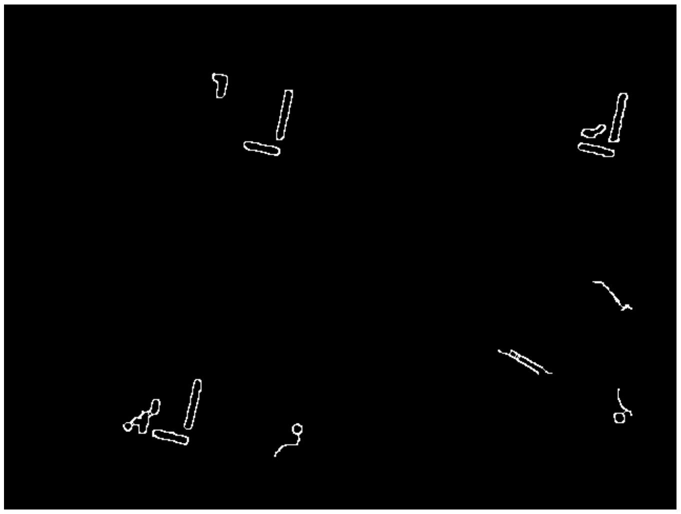
The second set acts on pairwise region attributes: distance between centroids (D) and difference in orientation (θ) between two adjacent regions.

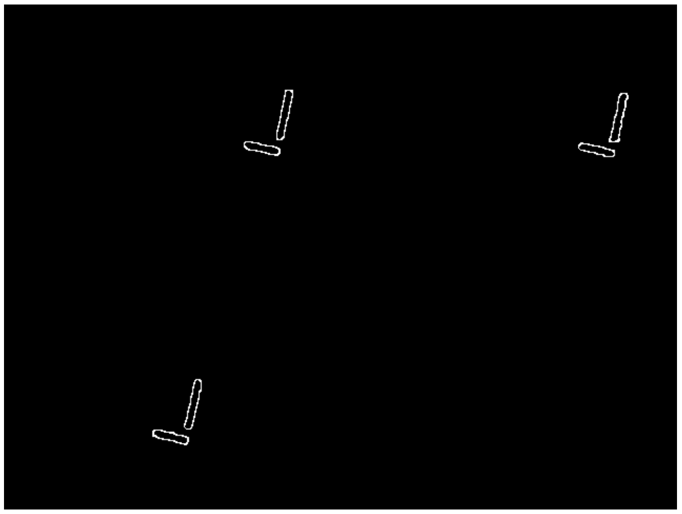
$$\delta_1 < D < \delta_2$$
$$|\theta| < \beta$$

The final result is a precise determination of the exact regions that correspond to all guide bar pairs in the image. Figure 9 illustrates the step-by-step elimination of extraneous regions using the two sets of thresholds for training image 9.


(a)


(d)


(b)


(e)

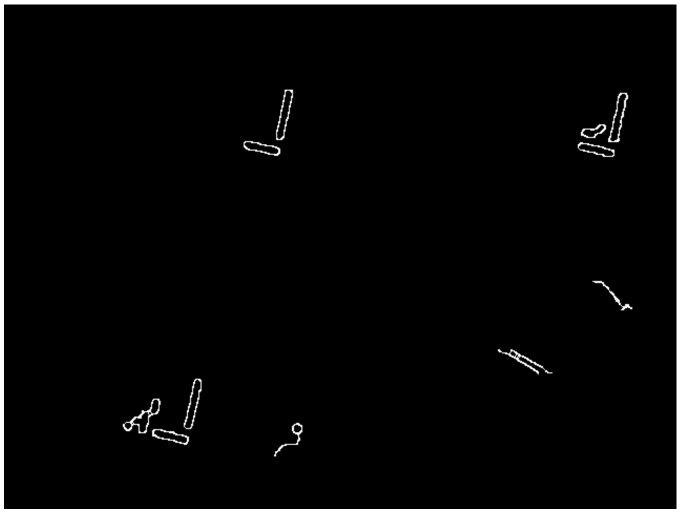

(c)

Figure 9. Extraction of guide bar pairs by thresholding. (a) Major axis length. (b) Minor axis length. (c) Area. (d) Eccentricity. (e) Distance between centroids. (f) Orientation difference.

B. Corner Elements

To detect the most likely candidate regions for corner elements, we use the Canny image and threshold on individual region attributes, namely the area and eccentricity of each corresponding ellipse. Unlike guide bars, corner elements are difficult to extract accurately using only region attributes, due to their small size, indistinct shape, and relatively wide variation in attributes. To avoid confusion with background objects of similar size and shape, and to maintain robustness of threshold values, we extract only a list of candidate regions and leave a precise determination of corner elements to the next step in our algorithm.

IV. Quadrant Classification

In this step, we introduce a new classification scheme based on the orientation of guide bar pairs. We define quadrants Q1 through Q4 as shown in Figure 10. As will be seen shortly, this quadrant classification scheme is the key to all subsequent calculations and processing steps, and provides an organized mathematical framework for the rest of the algorithm.

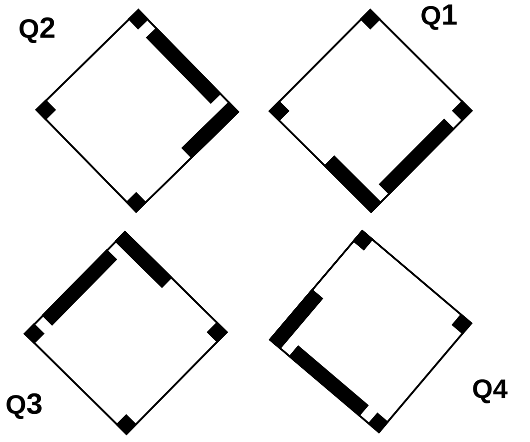


Figure 10. Quadrants Q1-Q4.

Let θ_L be the long bar orientation, θ_S be the short bar orientation, (x_L, y_L) be the centroid location of the long bar, and (x_S, y_S) be the centroid location of the short bar. The orientations are measured counterclockwise from the horizontal and are restricted to $|\theta_L| \le 90$ and $|\theta_S| \le 90$. Negative orientations indicate clockwise measurement from the horizontal. The coordinate system is defined with the x-axis extending to the right and the y-axis extending downward. Quadrant assignments are as follows.

Q1: $\theta_L \ge 0$
$x_L \ge x_S$

Q2: $\theta_L < 0$
$y_L < y_S$

Q3: $\theta_L \ge 0$
$x_L < x_S$

Q4: $\theta_L < 0$
$y_L \ge y_S$

A. Center Calculation

Using the quadrant assignments, we now estimate the center of each code marker region. This data is needed to find corner locations, which are the ultimate values of interest. Let (x_c, y_c) be the center for a given code marker.

$$x_c = \frac{1}{2}(x_{c,L} + x_{c,S})$$
$$y_c = \frac{1}{2}(y_{c,L} + y_{c,S}) \tag{2}$$

where $(x_{c,L}, y_{c,L})$ are the center coordinates estimated from the centroid of the long bar, and $(x_{c,S}, y_{c,S})$ are the center coordinates estimated from the centroid of the short bar. For the sake of robustness, we take the average of these two estimates as our final center location. If l_L and l_S are the long bar and short bar lengths, respectively, then the two center estimates are

$$x_{c,L} = x_L + k_{x,L}\,\delta_1 \cos\left(90 - |\theta_L|\right)$$
$$y_{c,L} = y_L + k_{x,L}\,\delta_1 \sin\left(90 - |\theta_L|\right) \tag{3}$$

$$x_{c,S} = x_S + k_{x,S}\,\delta_2 \sin\left(|\theta_S| + \Delta\theta\right)$$
$$y_{c,S} = y_S + k_{y,S}\,\delta_2 \cos\left(|\theta_S| + \Delta\theta\right) \tag{4}$$

where

$$\delta_1 = \frac{11}{2}\left(\frac{l_S}{5}\right) \tag{5}$$

$$\delta_2 = \sqrt{34}\left(\frac{l_L}{7}\right) \tag{6}$$

$$\Delta\theta = \tan^{-1}\left(\frac{3}{5}\right)$$

The multiplicative sign factors $k_{x,L}, k_{y,L}, k_{x,S}, k_{y,S}$ are quadrant dependent and reflect the code marker orientation.

Q1:	$k_{x,L} = -1$	$k_{x,S} = +1$
	$k_{y,L} = -1$	$k_{y,S} = -1$
Q2:	$k_{x,L} = -1$	$k_{x,S} = -1$
	$k_{y,L} = +1$	$k_{y,S} = -1$
Q3:	$k_{x,L} = +1$	$k_{x,S} = -1$
	$k_{y,L} = +1$	$k_{y,S} = +1$
Q4:	$k_{x,L} = +1$	$k_{x,S} = +1$
	$k_{y,L} = -1$	$k_{y,S} = +1$

Note that in Eq. (5) and (6), $\frac{l_S}{5}$ and $\frac{l_L}{7}$ serve as estimates of the average sidelength of each code marker element.

Our current implementation approximates the guide bar lengths l_L and l_S in Eq. (5) and (6) with the major axis lengths of the corresponding ellipses for the guide bar regions detected by object recognition, as described in Section III .

B. *Corner Calculations*

Using the center estimates and quadrant assignments, we can now estimate the four corner locations and match them to the correct candidate regions detected by object recognition. Let us label the corners C1 through C4 according to the canonical orientation, as shown in Figure 8, where the guide bars are in the lower right corner.

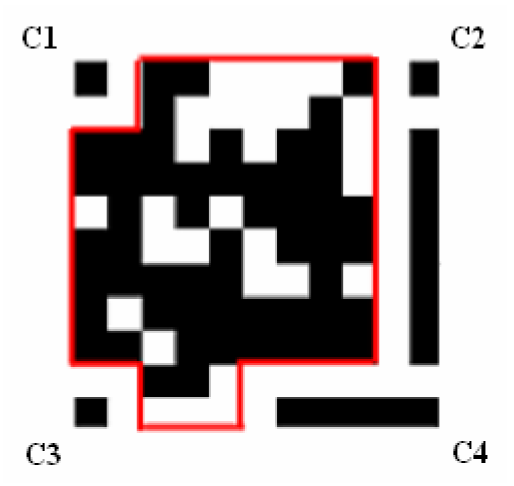


Figure 11. Corner labels C1 - C4 for canonical orientation.

Since some corners require information that is not available until other corners are found, we must calculate the corners in a precise order, namely C4, C2, C3, and C1.

Corner calculation algorithm:

1. Find C4.

$$x_4 = \frac{-y_S + y_L - a_S x_S + a_L x_L}{a_L - a_S} \tag{7}$$

$$y_4 = y_S - a_S(x_c - x_S)$$

where

$$a_S = \tan(|\theta_S|)$$
$$a_L = \tan(|\theta_L|)$$

$(x_L, y_L), (x_S, y_S)$ are the centroids of the long bar and short bar, and θ_L, θ_S are the orientations. Eq. (7) calculates C4 as the intersection of the two guide bars.

2. Find C2.

$$x_2 = x_L + k_{x,2}\,\gamma_2 \cos(|\theta_L|)$$
$$y_2 = y_L + k_{y,2}\,\gamma_2 \sin(|\theta_L|) \tag{8}$$

where

$$\gamma_2 = 4.5\left(\frac{l_L}{7}\right)$$

and $k_{x,2}$ and $k_{y,2}$ are multiplicative sign factors that are quadrant dependent.

Q1:	$k_{x,2} = +1$ $k_{y,2} = -1$
Q2:	$k_{x,2} = -1$ $k_{y,2} = -1$
Q3:	$k_{x,2} = -1$ $k_{y,2} = +1$
Q4:	$k_{x,2} = +1$ $k_{y,2} = +1$

Eq. (8) uses $\frac{l_L}{7}$ as an approximation for the average sidelength of each code marker element. The factor of 4.5 is motivated by the fact that corner C2 is 5 elements away from the long bar centroid and 4.5 is a robust, conservative estimate that avoids overshooting the actual corner location. We note that undershooting is safe since there is a white buffer region that is one element wide around C2.

3. Find C3.

$$x_3 = x_4 + k_{x,3}\ \gamma_3 \cos(|\theta_S|)$$
$$y_3 = y_4 + k_{y,3}\ \gamma_3 \sin(|\theta_S|) \tag{9}$$

where

$$\gamma_3 = 7.5\left(\frac{l_S}{5}\right)$$

and $k_{x,3}$ and $k_{y,2}$ are quadrant dependent, as illustrated in Figure 10.

Q1, Q2: (i) $\theta_S \geq 0$

$$k_{x,3} = -1$$
$$k_{y,3} = +1$$

(ii) $\theta_S < 0$

$$k_{x,3} = -1$$
$$k_{y,3} = -1$$

Q3, Q4: (i) $\theta_S \geq 0$

$$k_{x,3} = +1$$
$$k_{y,3} = -1$$

(ii) $\theta_S < 0$

$$k_{x,3} = +1$$
$$k_{y,3} = +1$$

Eq. (9) uses $\frac{l_S}{5}$ as an approximation for the average sidelength of each code marker element.

Just as in step 4, we use the factor of 7.5 as a robust, conservative estimate to avoid overshooting the actual corner location since corner C3 is 8 elements away from the short bar centroid. As before, undershooting is safe since there is a white buffer region that is one element wide around C3.

4. Find the matching candidate corner objects for C2 and C3.

Using the C2 and C3 estimates calculated in steps 2 and 3, select the closest candidate corner objects from the candidate set obtained by object recognition (see Section III-B). Update corner coordinates (x_2, y_2) and (x_3, y_3) using the centroids of these objects.

For additional robustness, our implementation includes an extra sanity check to handle the possibility of spurious guide bars obtained from object recognition. We verify that the updated corners (x_2, y_2) and (x_3, y_3) are within a radius of $2\left(\frac{l_L}{7}\right)$ and $2\left(\frac{l_S}{5}\right)$ of the estimated corners.

5. Find the angle between guide bars.

$$\theta_0 = \left(\frac{|\theta_L| + |\theta_S|}{2}\right)$$

6. Find C1.

$$\begin{aligned} x_1 &= x_c + k_{x,1}\ \gamma_1 \cos(|\theta_1|) \\ y_1 &= y_c + k_{y,1}\ \gamma_1 \sin(|\theta_1|) \end{aligned} \qquad (10)$$

where

$$\gamma_1 = \left(\frac{l_S + l_L}{2}\right)$$

and $k_{x,1}$ and $k_{y,1}$ are multiplicative sign factors that are quadrant dependent.

Q1:
$$k_{x,1} = +1$$
$$k_{y,1} = -1$$
$$\theta_1 = \frac{\theta_0}{2} + |\theta_L|$$

Q2:
$$k_{x,1} = -1$$
$$k_{y,1} = -1$$
$$\theta_1 = \frac{\theta_0}{2} - |\theta_S|$$

Q3:
$$k_{x,1} = -1$$
$$k_{y,1} = +1$$
$$\theta_1 = \frac{\theta_0}{2} + |\theta_L|$$

Q4:
$$k_{x,1} = +1$$
$$k_{y,1} = +1$$
$$\theta_1 = -\frac{\theta_0}{2} + |\theta_L|$$

Eq. (10) calculates γ_1 as a conservative estimate of the distance between the corners C1 and C4.

In an ideal square code marker, this distance is $\left(\frac{11}{2}\right)\sqrt{2} = 7.778$ elements long. Since the code markers in the camera phone image are distorted and since there is a one-element wide white buffer region around C1, it is safer to use an underestimate of this distance by taking the average of the two guide bar lengths, which is essentially 6 elements long.

7. Find the matching candidate corner object for C1.

First threshold the set of candidates based on area, distance from the estimated center, and deviation angle from the estimated center. The threshold values are calculated empirically from the given training set. For each of the remaining candidate corner objects, calculate the ratio of its distance from (x_2, y_2) to its distance from (x_3, y_3). Select the candidate with the distance ratio that is closest to 1.

The extra thresholding used in step 7 is necessary for robustness since our estimate of C1 is the least accurate out of all the corners, due its dependency on other estimated values and the resulting accumulation of estimation errors. Figure 12 compares the estimated corner locations from steps 2, 3, and 6 with the actual locations of the corner elements in training image 9.

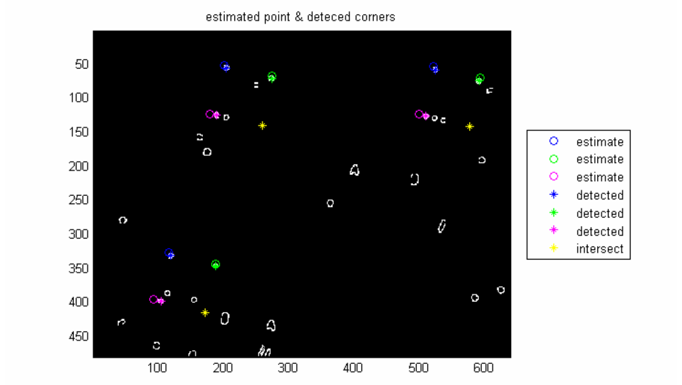


Figure 12. Comparison of estimated corner locations and actual corner elements.

From step 7, we obtain one of the required outputs for our algorithm, namely the center pixel for the upper left corner element in each code marker. This, along with the other three corner coordinates, are used in a later stage of the algorithm (see Section VI).

V. Binary Conversion

This step of the algorithm converts images to black and white in order to read the binary data bits embedded in code markers. A naive approach would simply convert the entire color or grayscale image to black and white. However, because of the poor contrast and low quality of the original image, doing so will degrade and often destroy the code marker regions. Figure 13 shows the result of naively applying binary conversion to the entire grayscale image for training image 2. Neither of the code markers appear in the black and white image.

Figure 13. Naive binary conversion.

We solve this problem by finding bounding boxes for each code marker region and performing localized binary conversion only within these regions.

A. *Bounding Box Calculation*

Using the estimated center coordinates (x_c, y_c) from Section VI-A, we calculate a bounding box for a given coder marker:

$$[\, x_c \pm b,\ y_c \pm b \,]$$

where

$$b_W = 1.3 \max\left\{ \frac{11}{2}\left(\frac{l_L}{7}\right),\ \frac{11}{2}\left(\frac{l_S}{5}\right) \right\}$$

This b is a robust overestimate of half the sidelength of the code marker, using $\frac{l_L}{7}$ and $\frac{l_S}{5}$ as estimates of the average code marker element size. The factor of 1.3 was empirically found to guarantee that the entire code marker fell within in the bounding box.

B. *Grayscale Threshold Calculation*

To convert any grayscale image to black and white, we need to set a threshold between 0 and 255 so that grayscale intensities above this value are quantized to white (1) and those below are quantized to black (0). To get a robust estimate of the optimal threshold for a given code marker, we take a window W of size b_w x b_w centered on (x_C, y_C), where

$$b_W = \min\left\{ \frac{11}{2}\left(\frac{l_L}{7}\right),\ \frac{11}{2}\left(\frac{l_S}{5}\right) \right\}$$

is a robust underestimate of half the sidelength of the code marker, so that no part outside the code marker falls within the window. We then calculate

$$\tau = \frac{1}{2}(\max_{w} I + \min_{w} I)$$

as the grayscale threshold, where I is the pixel intensity value, and apply localized binary conversion on the entire bounding box region. Figure 14 shows the result for training image 2.

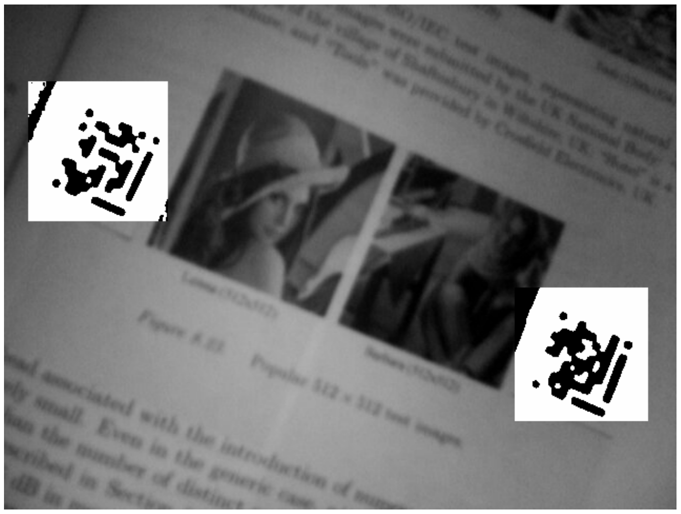


Figure 14. Localized binary conversion.

VI. Data Extraction

The previous steps in the algorithm successfully determine the image areas that correspond to marker regions and their vertices. The remaining task is to extract the bit information from the code marker regions.

A. *Projective Transformation*

Projective transformation will find out the position of the observer in the three dimensional space from given vertices of a four-sided figure, and maps the image to a scene seen from another observer's view. Using the vertices provided, it is possible to fix the code marker region into a clean square. Please refer to http://en.wikipedia.org/wiki/Projective_transformation for more discussion.

i. Shifting the vertices

Given (x,y) coordinates of the center of each corner pixel, we must find out the coordinates that corresponds to position just outside of the corner pixels to include the entire encoded region. Otherwise, about half of the edge rows and columns will be cut out.

We proceeded by assuming that a circle can represent each corner pixel. For three of the four corners, we have the area of black dots. For the bottom right corner, we must calculate the area from length of the short guide bar. Letting r be 1/2 of 1/5 of the length of the short bar, area can be approximated by pi*r^2.

The function MoveVertexBack takes three points in its parameters and moves back the first point along the line that bisects the angle created by three points. The area information is passed to determine the distance to move vertices back. Before calling the function, we multiply the area by 0.6 to get the best result, which balances the assumption that corner pixels are circles. The summery is tabulated later in the results section.

ii. Determining the maximum supported magnification

Because each code marker image is only of certain sizes, maximum meaningful transformed image size is bounded. MATLAB's imtransform returns image of maximum size it can support when specified an overly large output size. It is always more beneficial to magnify the image as much as possible to retain all possible information. We also need even number of pixels between pixels so that same numbers of pixels exist between an edge and the first pixel read on all sides.

In order to find the maximum magnification factor that meets the condition, we specify the output to 1000 pixels and look at the size of the returned image. We then reverse calculate the maximum supported magnification.

After we get the maximum magnification factor, we need the output image to have factor*11+11 pixels: 11 pixels to be read and factor*11 pixels to be skipped in between.

iii. Running the projective transform

After determining the parameters, we conduct the projective transform to map a lopsided figure into a square.

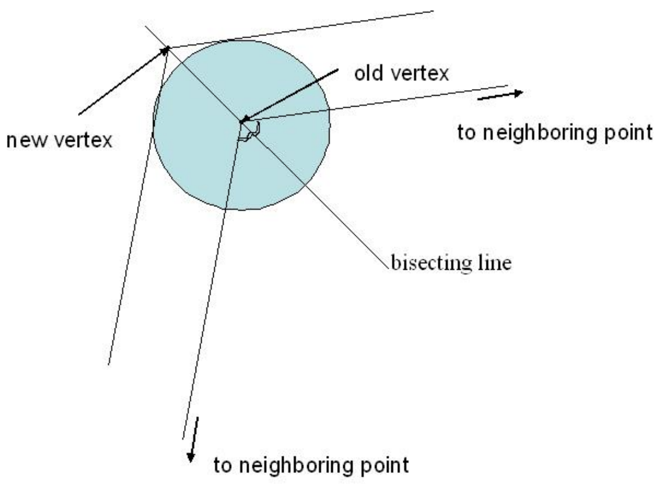


Figure 15. Vertex moving routine

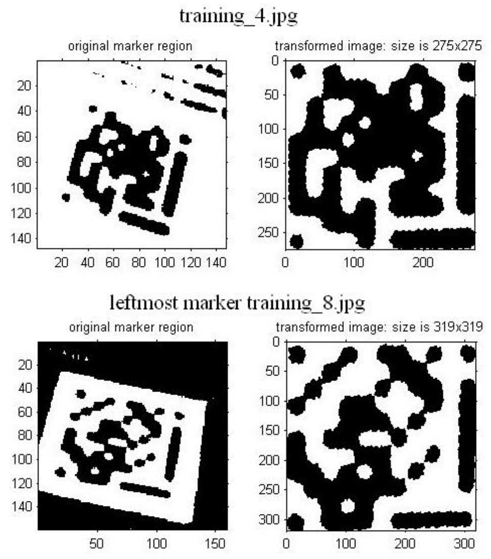


Figure 16. Example projective transformation

B. *Magnified marker processing*

Before reading the bits of the magnified and squared image, it is sometimes beneficial to enlarge the islands of each color in order to not miss them. We performed dilation of small islands that went above the threshold of 1/8 of the resized marker area that corresponds to one bit. Such areas were recursively dilated until it occupied 1/2 of the block. This process mattered in training_9.jpg and training_10.jpg, and we set the threshold values by comparing the intermediate images.

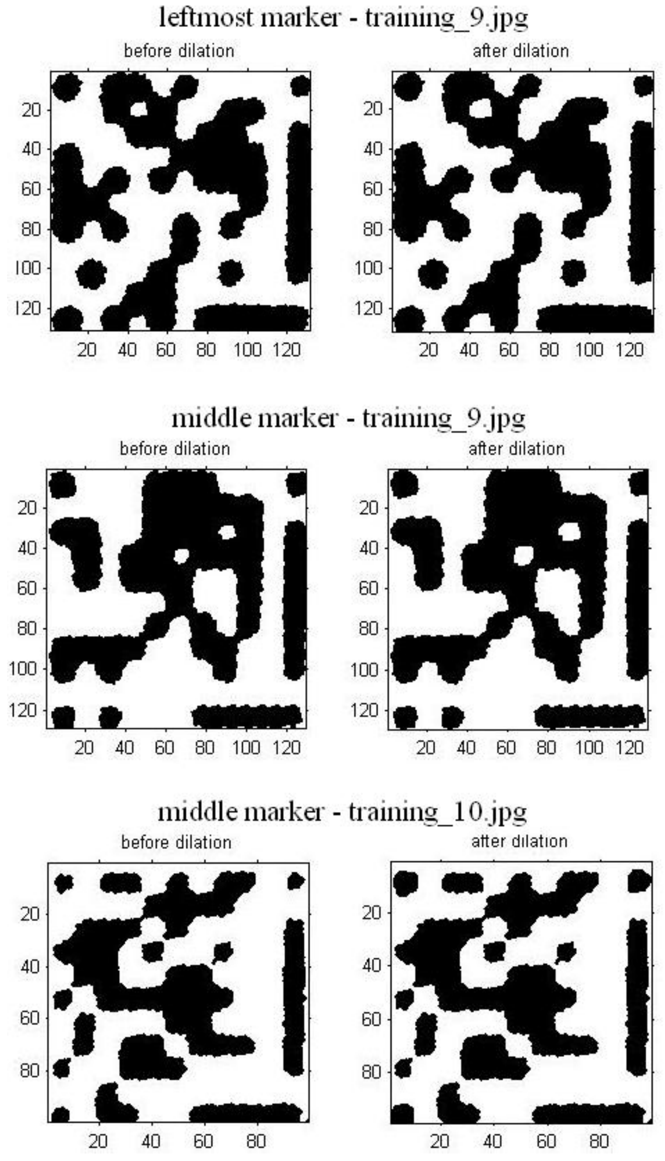


Figure 17. Before/after dilation

After dilation of small islands, we remove obviously non-marker regions of the code markers. We saw that some markers have white rows and columns around the edge pixels of the images, although in theory, there should be no non-marker pixel inside the given and processed vertices. We check to see if more than 97% of the first several rows or columns are white, and if so, we reject the row and column. The number of rows and columns checked depends on the magnification factor. The threshold value was chosen based on experience, as this part mattered mostly to training_4.jpg, training_8.jpg, and training_10.jpg. After trimming, the image is then resized to have the right number of rows and columns.

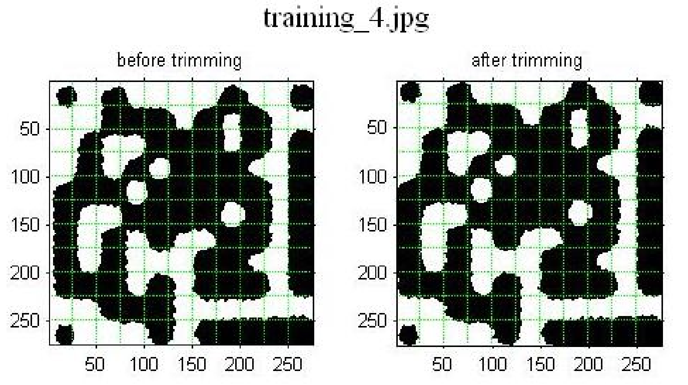


Figure 18. Before/after trimming

C. *Picking the color of the blocks*

With the largest possible representation of code marker properly reshaped, realigned, and resized, we are ready to proceed to reading the bits.

The algorithm reads a small window that sits at the center of each block. We subdivide the image to have 11x11 of (factor+1)*(factor+1) pixels large blocks. Each block has odd number of rows and columns, thus guaranteeing single center pixel. The algorithm looks at the center pixels and its neighbors to determine block color. We ultimately chose a window size of 25 pixels that has the best results of all the block sizes considered, based on the testing results displayed later in the report.

Table2 Conflict frequency

with the use of adaptive magnification factor and factor to multiply corner area = 0.6						
window size	full processing		no dilation		no trimming	
	conflicts	bits missed	conflicts	bits missed	conflicts	bits missed
9	3	0	3	0	40	9
25	16	0	19	0	102	8
49	33	0	45	0	217	6

When more than two colors coexist in the neighborhood, we look at the block colors to the top and left.S If two neighbors are of the same color, we assume that their color has spilled over to the current window and choose the opposite color for the current block.

If two colors are different, we take a majority vote of the members in the current window to determine the block color.

This algorithm provides more robustness than either looking only at the center pixel or running conflict resolution on the entire block. By just looking at the center pixels, we may miss color changes. Taking the conflict resolution on the entire block will almost always lead to conflicts, and lets the errors to propagate to subsequent blocks.

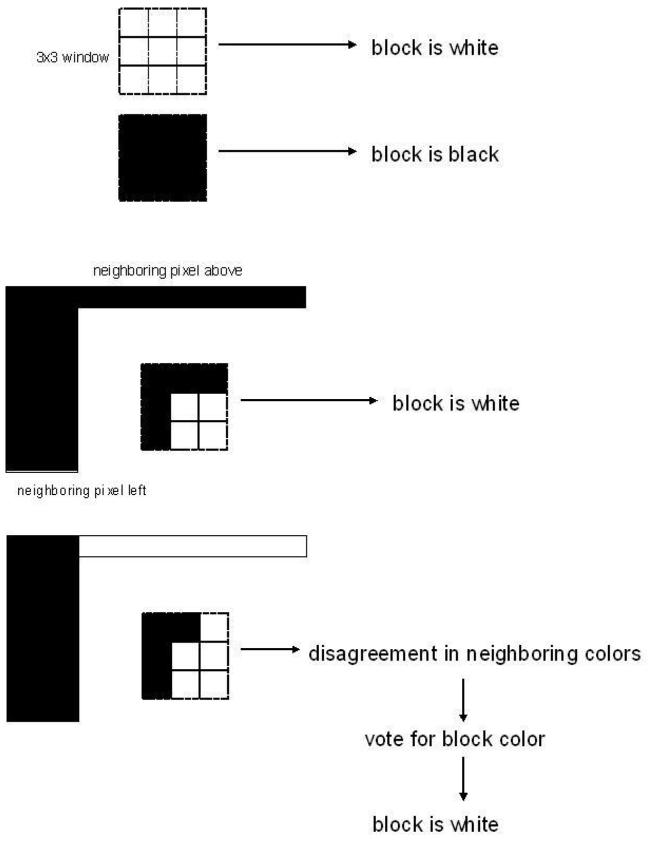


Figure 19. Conflict resolution go here

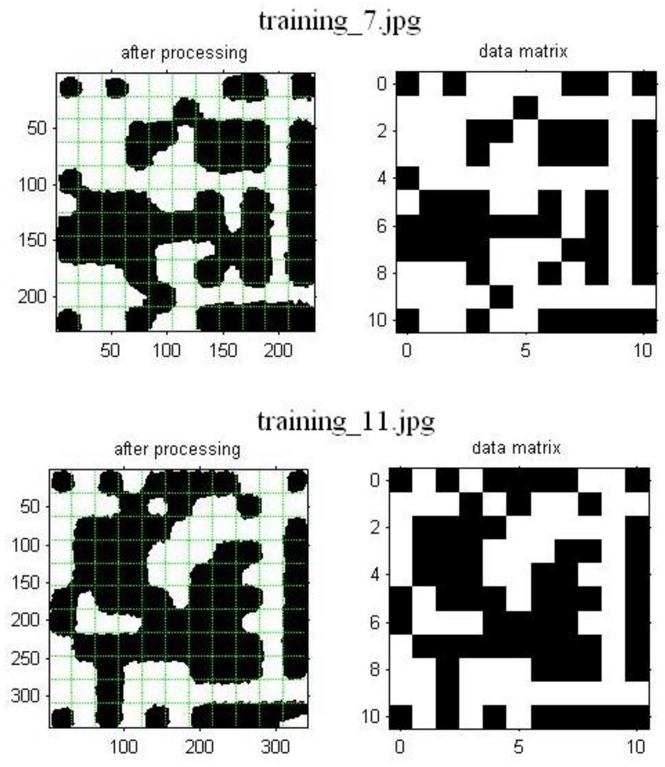


Figure 20. Example figure of squared and magnified marker region and corresponding checkerboard

VII. Results

The algorithm described in Sections II through VI achieves 100% accuracy on all 12 training images, with no repeats, misses, false alarms, or incorrect bits. To test for robustness, we ran the algorithm on 36 additional images generated by rotating each original image by 90, 180, and 270 degrees. On this augmented set, the accuracy achieved is still 100%.

For the set of 12 original training images, the total runtime on a 3.6 GHz machine is 94 seconds, with a maximum time of 10 seconds for an individual image. This performance falls well within the runtime restriction of one minute per image.

VIII. Conclusion

In this project, we have developed a completely original, robust system for visual code marker detection that incorporates edge detection, morphological processing, region labeling, object recognition, and quadrant classification to automatically extract and read code marker elements from relatively low-quality, low-contrast camera phone images. The cornerstone of the algorithm is quadrant classification, which provides a unifying mathematical framework for geometric approximations to calculate the orientation and corner coordinates of any code marker region. Using this framework, code marker data extraction is reduced to a standard projective transformation, with conflict resolution to handle pixel bleeding. The algorithm does require any offline training and achieves accurate near-realtime performance on standard desktop machines. Finally, its robustness has been confirmed by additional testing on perturbed images from the given training set.

Appendix

May Zhou	Quadrant Classification
	Code Marker Detection
	Center Calculation
	Corner Calculation Algorithm
	Bounding Box Determination
	Binary Conversion
	Conflict Resolution Algorithm for data extraction
	Writing the report (except for Section VI)
Kanako Hayashi	Edge Detection
	Morphological Processing
	Object Recognition
	Code Marker Detection
	Quadrant Classification
	Corner Calculation Implementation
	Creation of figures & diagrams for report
Yuki Konda	Extraction of code marker bits
	Writing Section VI (data extraction)

References

[1] R. Gonzalez and R. Woods, Digital Image Processing, 2nd ed., Prentice Hall, 2002.
[2] A. Jain, Fundamentals of Digital Image Processing, Prentice Hall, 1989.
[3] B. Girod, Lecture Notes for EE368, Spring 2006.