Detecting Visual Codes (May 2006)

Paul Reynolds, Bryan Brudevold, and Paul Baumstarck

***Abstract*—The detection of binary visual codes is an integral commercial application of computer vision, facilitating the transmission of digital information from the physical to the electronic domains. Our detection method uses repeated elimination of image areas to identify likely guide bars. Guide bars are used to predict the locations of the bounding bits, and further calculation yields a refined estimate of the code's location and bit values. Final processing eliminates candidate visual codes that do not fit the fixed pattern to a high enough degree.**

***Index Terms*—Visual system, visual codes, binary sequences, pattern recognition.**

I. Introduction

VISUAL codes have been designed to convey binary information in a two-dimensional array. They were designed as an efficient way to store and retrieve information about the objects on which they are attached. The visual code is designed for quick and reliable identification and decoding by computer imaging. In this paper, we present one such method.

II. Algorithm design

A. Visual Code Detection

Our initial approach to the problem of locating and reading the visual codes was to select regions that contained a high number of strong corners. We used Harris corners with a high threshold then labeled possible visual code locations as those places where high number of these corners were clustered together. Fig. 2a shows a test image where this algorithm worked very well in selecting only regions that contained a visual code. However, Fig. 2b shows another test case where the image background also contained many strong corners, thus making it impossible for the detector to generate any useful estimate of where codes might be. We rejected the method because of its poor performance on this image and other similar images.

At first we sought to correct for this by adding in color search. We hoped that the high volume of color in images would help reject corners that were not part of the black-and-white visual codes. This method proved to be unreliable because the codes themselves were not perfectly gray and because gray features that were not part of the codes were not eliminated; thus this method was abandoned.

In another approach, we tried to exploit the regular black and white bar-shaped structures in the codes by taking the Hadamard and the Discrete Cosine Transforms of the image in various block sizes. Using labeled images showing the positions of the codes, we trained a multi-layer neural

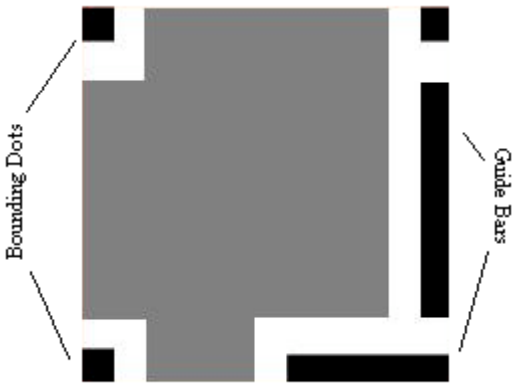


Fig 1. The visual code pattern to be detected. Black and white dots indicate fixed values on the pattern while gray dots indicate data bits that can be either 0 or 1.

a

b

Fig 2. Output of preliminary visual code detection work using Harris corners: red dots indicate corners and blue lines indicate regions suspected of containing codes. (a) shows the detector functioning well on training_2.jpg while (b) shows the detector functioning very poorly on training_5.jpg.

Fig 3. Output black-and-white image for training_5.jpg, the "car wash" image (pixels classified as black by the algorithm are labeled white on this mask).

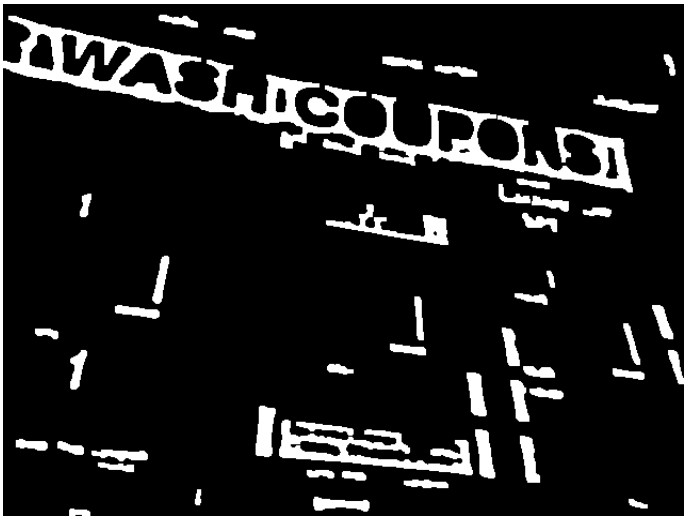


Fig 4. Output of passing black and white image through the rectangular guide bar detector.

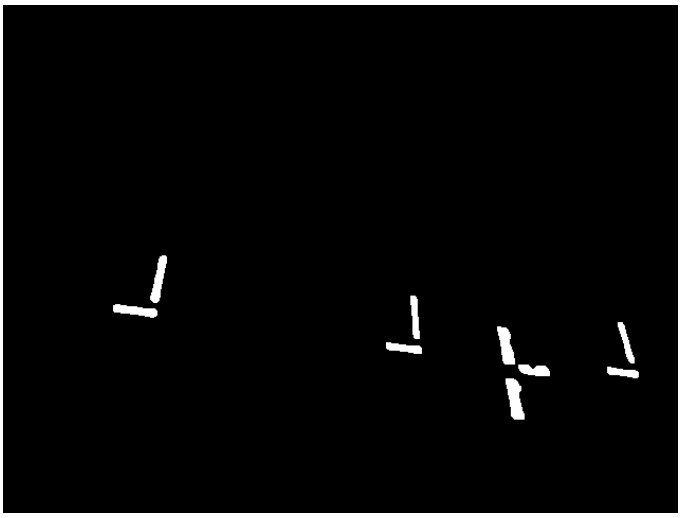


Fig 5. Results of selecting only guide bars from the previous image that appear in the correct groupings and orientations for true guide bars.

network to recognize the codes using the transform inputs. However, this method also had little success and was rejected.

Finally, we decided to implement a detection algorithm following the method presented by Michael Rohs [1]. This algorithm is built around first finding the guide bars and then using their positions and orientations in order to find the whole visual code.

The first step in the algorithm was to detect the black guide bars by converting the image to binary or black-and-white. To do this, we de-noised the input by convolving each color channel with a Gaussian kernel. Next we obtained local averages, r_{ave}, g_{ave}, and b_{ave}, by convolving each channel with a normalized 20x20 kernel. The final black-and-white image was constructed using the formula:

$$(r < (r_{ave} - 0.01)) \cdot (g < (g_{ave} - 0.01)) \cdot (b < (b_{ave} - 0.01))$$

This method only considers pixels which are less than the local average in all color channels as being black pixels, and thus possible guide bar pixels. Fig. 3 shows the binary output image for the test image used previously in Fig. 2b. A region labeling operation was then performed to identify contiguous blocks of white pixels (which, in these images, signify black regions of the original color image).

This step typically returned far too many regions, so to further refine our estimate of guide bar locations we selected only those regions that met certain criteria. The first criterion we used was size: we rejected all regions with less than a certain number of pixels. The second criterion was eccentricity, or the ratio of the longest to the shortest dimension of the region.

We determined the eccentricity of each region by removing the mean and then taking the singular value decomposition (SVD) of the block's pixels. The ratio of the largest singular value to the smallest singular value yielded a measure of eccentricity. Blocks with small or very large eccentricities were removed from consideration as possible guide bars. These two criteria significantly reduced the number of possible guide bars, and the surviving blocks from the example image in Fig. 3 are shown in Fig. 4.

The next step was to exploit the grouping and orientation of the guide bars, specifically the requirements that they always appear in pairs, oriented orthogonally to each other, and with the long guide bar in line with end of the short guide bar. We implemented this step by considering each region and saving only bars with a second guide bar located at the end of the first bar and at a nearly perpendicular angle (a moderate tolerance was allowed because of perspective distortion to the angle). The surviving pairs from the example image are shown in Fig. 5. Some further testing was also done on the relative lengths of the guide bars and on the direction of rotation from the long to the short bar (both of which were calculated from the left singular vectors that were output in the SVD step).

Fig 6. Red "x"s mark the preliminary estimates of the locations of the four bounding points for each valid visual code in Fig. 5 using the size and orientation of the guide bars only for prediction.

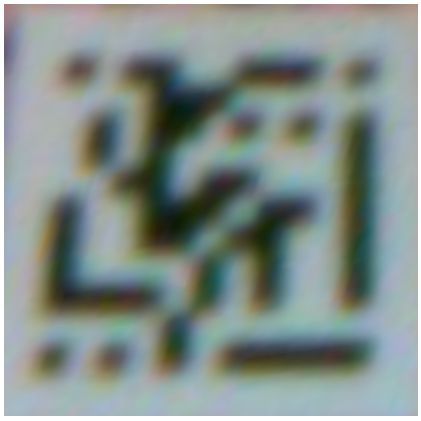


Fig 7. Cropped, bilinearly-interpolated square image obtained using the corner points in Fig. 6c.

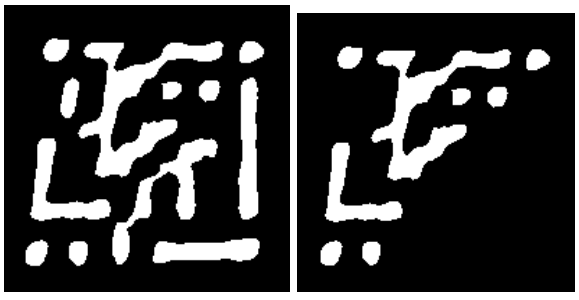


Fig 8. (a) shows Fig 7 converted to black and white using local image mean. (b) right shows the remaining patches of Fig 7a after selecting only regions with low eccentricity. Refined bounding point estimates are derived as the centers of the white regions closest to the three non-guide bars corners of the image.

B. Finding Bounding Corners for Each Code

Given the position of a possibly valid set of guide bars, the other three corners ("bounding bits" in Fig. 1) of the code had to be found. First we used the position and dimensions of the known guide bars to guess where these points would lie if the visual code was under a projective transformation, not taking into account perspective or other possible distortions. Example outputs of this step are shown in Fig. 6 for the three valid visual codes in the training image. In each of the three cases, the only previously known point was that in the lower right corner point that lies on the guide bars. The other three points were the result of the parallelogram extrapolation. Fig. 6 shows that this first estimate was quite good but not perfect.

To improve our results, we used these four initial points to extract a square, bilinearly-interpolated section of the image. We relaxed the four points to capture a larger portion of the image, and an example transformed image is shown in Fig. 7 for the visual code and points shown in Fig. 6c.

We then ran our black-and-white classification algorithm again on these cropped images and obtained a series of binary images with white regions indicating sections of black. This first output is shown in Fig. 8a. We wanted to find the true positions of the bounding dots which appear in these images as isolated circular regions. Therefore we ran the SVD-eccentricity check on each block and passed only those with a low eccentricity. The regions that survived this step for the image in Fig. 8a are shown in Fig. 8b.

To find the true locations of the bounding dots we selected the mean of the surviving blocks that were closest to the three corners of the image (excluding the guide bar corner). Then we converted from the cropped corner pixel coordinates to the original, un-cropped image pixel coordinates. This gave us a final estimate for the position of the visual code, the guide bars, and the three bounding bits. The output of this step for the codes in Fig. 6 is shown in Fig. 9. The upper-left corner of each code is marked with a blue circle while the other bounding points are marked with red "x"s. These secondary estimates of the bounding bits were quite accurate.

Now that we had the position and bounding shape of each code, we extracted another bilinearly-interpolated square image using the four corrected corner points and converted the image to black and white. The method we used here for black and white conversion, however, differs from those used before and requires special discussion.

C. Decision Threshold Calculation

In order to convert the cropped, magnified visual code to black and white, we initially tried thresholding the image by its mean value. However, in visual codes with a preponderance of 0's or 1's for data bits, the image mean could end up being perilously close to the average value of the 0's or 1's, and this could conceivably lead to some bits being read incorrectly. This is illustrated below with an example data set X which is the union of samples from two Gaussian populations: 5000 samples were drawn from a 0-mean, unit-variance Gaussian and 1000 samples were drawn from a unit-variance Gaussian with a mean of 6. The histogram of this population is shown in Fig. 10.

The mean of this population is 0.9905, and a vertical bar is drawn on the graph for this value and is labeled _0. Clearly _0 lies within one standard deviation of the mean of the 0-mean Gaussian, so it would erroneously classify many samples from that group as belonging to the set of 1's.

Our solution was to "correct" the decision threshold in an iterative manner. First we computed the mean of the set as above. Then we constructed two subsets that were the elements of X separated into groups that were above or below that threshold:

$$X_0 = \{X_i : X_i < \mu_0\}$$
$$X_1 = \{X_i : X_i \geq \mu_0\}$$

Then we calculated the next decision threshold μ_1 as the average of the individual means of these two subsets:

$$\mu_1 = \frac{\overline{X}_0 + \overline{X}_1}{2} = \frac{\sum_{i=1}^{|X_0|} X_{0,i}}{2|X_0|} + \frac{\sum_{i=1}^{|X_1|} X_{1,i}}{2|X_1|}$$

For the Gaussian populations, we found _1 to be 1.8454, which is clearly moving away from the 0-mean population, as desired. Continuing the iterative calculations, we see that the decision threshold converges after only three more applications at 2.9970, which is almost exactly equal to the average of the two populations' means (3). For such a value to be convergent it must formally satisfy the equation:

$$\mu_\infty = \frac{\sum_{\{i:X_i<\mu_\infty\}} X_i}{2|\{i : X_i < \mu_\infty\}|} + \frac{\sum_{\{i:X_i\geq\mu_\infty\}} X_i}{2|\{i : X_i \geq \mu_\infty\}|}$$

All of the values of μ_k and their corresponding sizes of the subset X_0 are shown below for the Gaussian example:

$$\mu_k = \{0.9905, 1.8454, 2.7172, 2.9730, 2.9970\}$$
$$|\{X_i : X_i < \mu_k\}| = \{4191, 4849, 4985, 4998, 4998\}$$

These values of μ_k are also drawn on Fig. 11 and clearly show that the result of the iterative correction has been to locate the "trough" in the histogram in between the means of the two populations. This has had the desired effect of moving the decision threshold away from the mean of any dominant population in the sample.

We did not investigate the effects of this iterative calculation in depth, but we verified that the convergent value need not be unique. By seeding the above calculations with a value of _0 = 6, the sequence converged at _3 =3.0006, which ended up classifying two more points as 0's than did the previous threshold. While these two solutions are distinct, they are only trivially distinct, so we did not concern ourselves with this effect in implementation. Also, to avoid possibly oscillatory conclusions, in actual implementation we only ran the decision threshold algorithm up to _4.

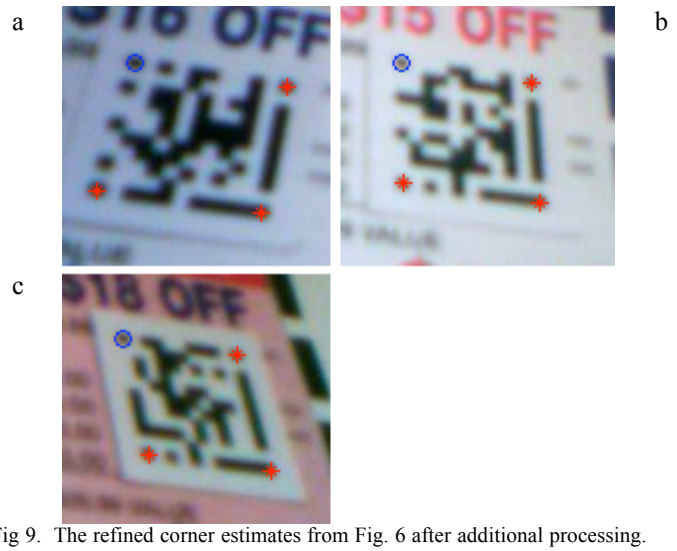


Fig 9. The refined corner estimates from Fig. 6 after additional processing.

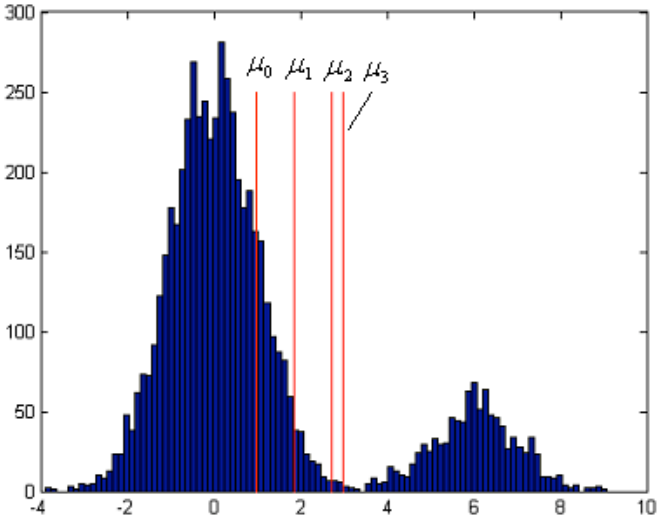


Fig 10. Histogram of combined Gaussian populations to illustrate an example of the action of the iterative decision threshold correction algorithm. Corrections move the threshold away from the mean of the 0 population and into the "trough" in between the two populations.

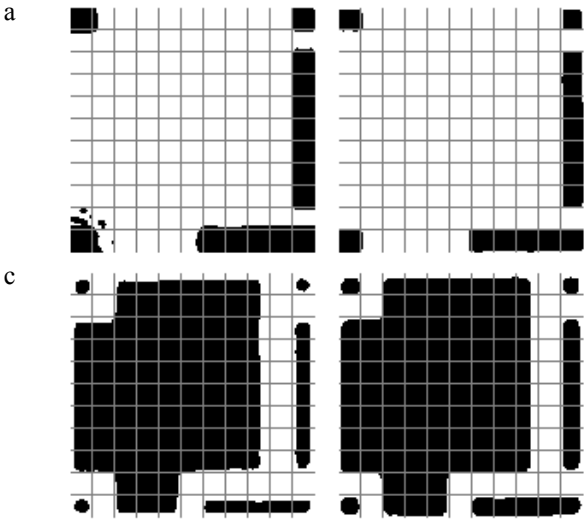


Fig 11. Effects of mean versus iterated thresholding. (a) shows a code with all 1 data bits that was converted to black and white using the image mean while (b) shows conversion using the corrected threshold. (c) and (d) show the same for the case of a code with all 0 data bits.

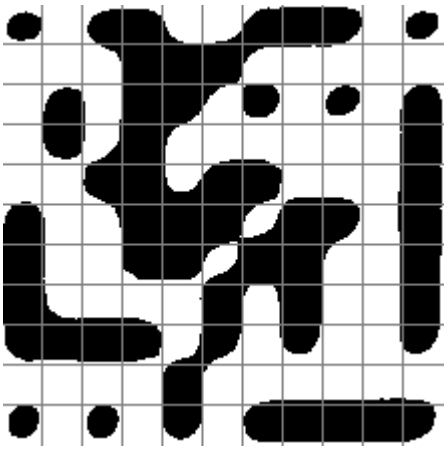


Fig 12. Cropped, bilinearly-interpolated, decision-thresholded output for the code shown in Figs. 6c and 9c using refined corner point estimates. Gray lines show the expected boundaries between bits of the visual code.

The effects of this corrected threshold on actual data is illustrated in Fig. 11 which considers the case of two codes with either all 1 or all 0 data bits. Fig. 12a shows the all-1's code converted to black and white using the image mean as the threshold, and clearly the result is that an undesirable
number of white pixels has been read as black since the image mean is much closer to the mean of the white value. Figure 11b shows the same image read using the corrected threshold, and the flaws in 11a have been greatly ameliorated. The code with all 0 bits shown in Figs. 11c and 11d shows similar improvement thanks to the corrected threshold: fewer black pixels are erroneously read as white ones.

D. Reading and Verifying Code Bits

Now given a position for each code, we extracted a final bilinearly-interpolated, square image using the four corrected bounding points. We chose to extrapolate to an image of size 220x220 since this would allot a 20x20 region for each bit of the visual code. Then we converted the image to grayscale and, using the decision threshold calculation discussed above, we converted the image to binary. The output of this step for the same code featured in Figs. 7 and 8 is shown in Fig. 12. Gray lines were drawn on this figure to show the expected boundaries between pixels of the code. Clearly the actual pixels fit very well inside of their expected locations.

However from the figure it is also clear that the entirety of each pixel's region may not be filled with a majority of the correct white or black value. Thus we averaged only the middle 10x10 patch of each pixel's region and evaluated it with respect to a threshold to obtain the final 0 or 1 value for that bit.

After this processing step, we obtained an 11x11 bit pattern for the whole image. As a final check we tested how well these bit values corresponded to the expected fixed pattern described by the guide bars and bounding bits shown in Fig. 1. In particular, each visual code has 23 fixed white bits and 15 fixed black bits. We checked the difference of the bits we read off with the fixed pattern, and we only passed codes that read at least 20 of the 23 white bits correctly and 13 of the 15 black bits correctly. This final check proved essential since sometimes all of our processing up until this point would still feed spurious images into this last part of the program.

TABLE I
RESULTS FOR ORIGINAL TRAINING IMAGES

Training Image	Score	Possible	Percent Score	Execution Time (sec)
1	83	83	100%	5.5
2	166	166	100%	8.9
3	249	249	100%	12.3
4	83	83	100%	5.9
5	249	249	100%	12.6
6	83	83	100%	5.5
7	166	166	100%	12.4
8	83	83	100%	5.4
9	249	249	100%	12.6
10	249	249	100%	12.5
11	83	83	100%	5.8
12	166	166	100%	12.4
Mean Execution Time (sec):				9.3

Results summary for the original training images.

III. RESULTS

A. Given Test Images

After adjusting some of the parameters of our detection algorithm, we obtained perfect results on the twelve test images provided on the EE368 web site. Our detection program found every visual code and read its data bits with 100% accuracy with no repeats or false positives. The output of the "evaluate" program for our code is shown in Table I. The average elapsed computation time per image was 9.3 seconds, and this was well below the absolute time limit of 60 seconds per image, so we decided not to seek to optimize our code.

B. Original Test Images

In order to ensure that our detection algorithm was working to an even higher degree, we created new visual codes and took additional sample images for testing. The extra codes we generated and the full set of additional sample images are shown attached in the Appendix as Images 1-30. In order to test the limits of the abilities of our detection algorithm, most of our images contained extreme cases of background clutter, perspective and scaling variation, special-case bit patterns, and/or lighting conditions.

Our algorithm achieved perfect scores on 25 of the 30 additional test images, failing only on the most challenging images. A summary of the results of the evaluate program on these additional images is shown in Table II, and a note accompanies each image that produced errors, explaining why it failed. The average execution time on the supplementary set was 8.2 seconds per image, with a maximum execution time of 18.8 seconds, well within the time constraint of 60 seconds per image.

The algorithm failed to locate several codes in images that featured sudden changes in illumination across the

TABLE II
RESULTS FOR SUPPLEMENTARY TRAINING IMAGES

Appendix A Figure	Score	Possible	Percent Score	Execution Time (sec)
1	166	166	100%	9.1
2	249	249	100%	18.7
3	166	166	100%	8.5
4	166	166	100%	9.4
5	83	83	100%	5.2
6	166	166	100%	9.0
7	83	83	100%	5.1
8	249	249	100%	11.9
9	166	166	100%	9.6
10	83	83	100%	5.2
11	249	249	100%	11.8
12	166	166	100%	12.8
13	83	83	100%	6.0
14	166	166	100%	8.7
15	83	83	100%	5.7
16	83	166	50%	10.0
1 code missed: uneven lighting on code due to a shadow				
17	166	166	100%	10.7
18	83	83	100%	5.2
19	83	83	100%	9.1
20	0	83	0%	1.7
1 code missed: too large, blurry				
21	166	166	100%	8.5
22	83	83	100%	5.8
23	0	83	0%	5.6
1 code missed: uneven lighting on code due to bright lighting				
24	83	83	100%	5.3
25	166	166	100%	8.8
26	166	166	100%	11.7
27	83	83	100%	5.0
28	166	166	100%	14.4
29	0	166	0%	3.4
2 codes missed: too small				
30	78	83	94%	4.9
5 bit errors: extreme red light illumination variance				
Mean Execution Time (sec):				8.2

Results summary for the supplementary training images.

body of the code (an example is Image 16 which shows a dark shadowing covering half of the code). In Image 30, a few data bits were reversed due to an intense red light incident on the code, but the code itself was still located correctly. Our algorithm works by considering only the mean illumination conditions over the entire code, and this deals well with gradual illumination changes across the entire image. However, a much more sophisticated black-and-white detection scheme would be needed to detect stark illumination changes across a single visual code.

Our detector also failed on some extremely large or small codes. The small codes (Image 29) could not be found because the guide bars blended into a single black region which the detector could not resolve as two separate bars. Very large codes (Image 20) were not always found because the guide bars are sometimes split into several pieces by our detector's initial step of conversion to black and white. This was because there were large neighborhoods of all-black pixels, so the local averaging produced some black pixels that were read as white pixels since they were above their local mean. In order to handle these cases, we would need to adjust our black-and-white classification scheme to work appropriately at low and high scales, which would require estimating the size of the codes before identifying all of their bits. However, the algorithm already handles a reasonably wide range of scale variation so we did not deem this problem critical enough to fix for the final submission.

IV. CONCLUSIONS

Overall, the algorithm is very robust in identifying visual codes and can handle a wide variety of difficult cases. This allows the user to be less concerned with exact positioning and will eliminate the need for rescanning codes. However, should a second scan be needed, the computation time is within a few seconds and the user will not have to wait long.

APPENDIX A. DIVISION OF LABOR

For much of the project, coding was typically done with all members present and contributing, though each member was primarily responsible for specific segments of the project. The division was as follows:

Bryan Brudevold – initial probable guide bar location, including black location and eccentricity check.

Paul Reynolds – secondary guide bar check, including guide bar pairing, and initial and final corner point approximations.

Paul Baumstarck – Perspective correction, thresholding and visual code reading.

APPENDIX B. SUPPLEMENTARY IMAGES

Appendix B shows the set of supplementary training images we used and appears attached as the next two pages.

REFERENCES

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

Image 1	Image 2	Image 3

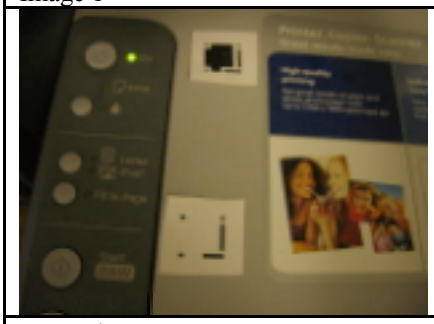
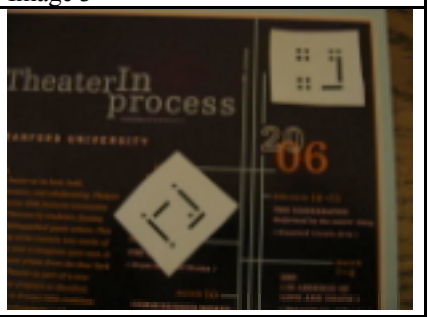

		

Image 4	Image 5	Image 6

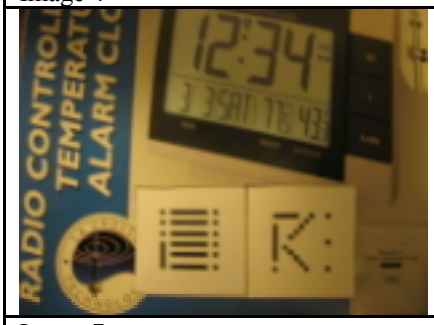
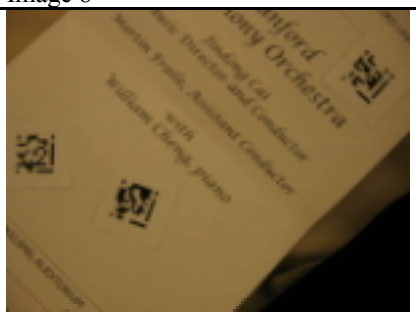
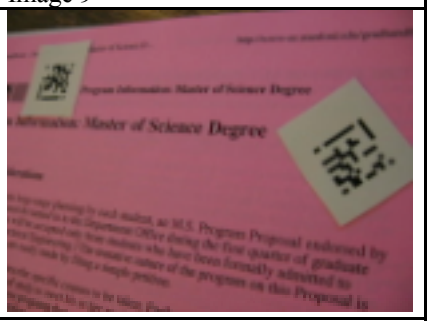



Image 7	Image 8	Image 9
		
Image 10	Image 11	Image 12

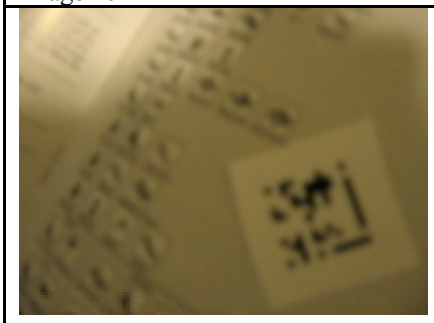
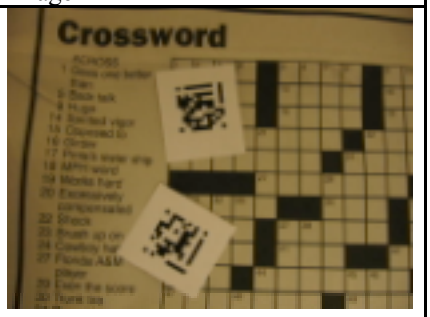

		

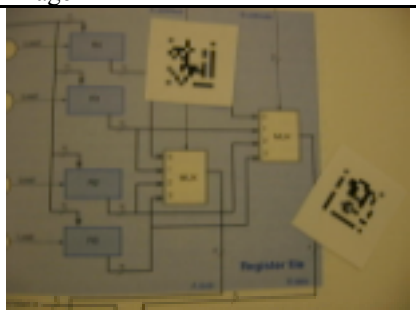
Image 13	Image 14	Image 15
		

Image 16	Image 17	Image 18

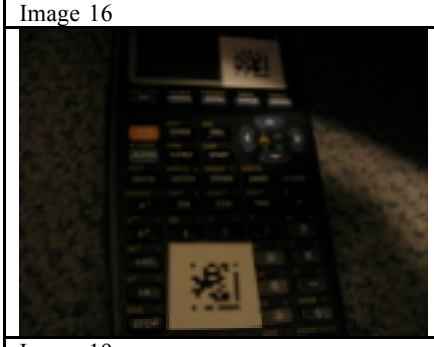
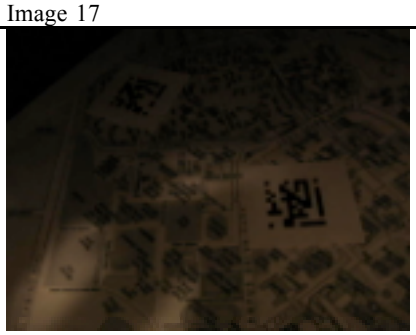
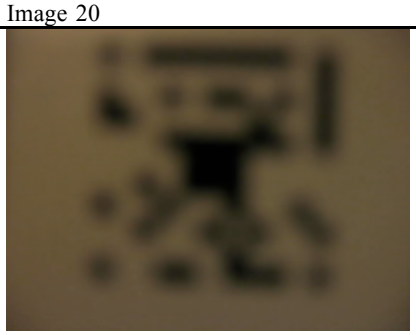
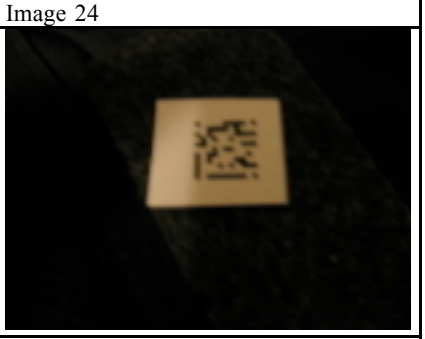

	

Image 19	Image 20	Image 21
		
Image 22	Image 23	Image 24
		
Image 25	Image 26	Image 27

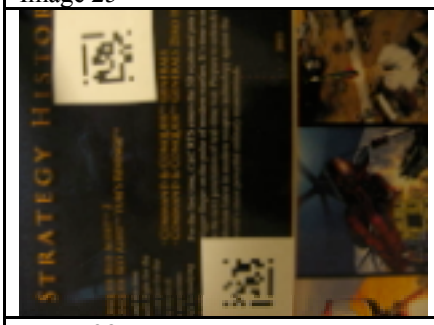
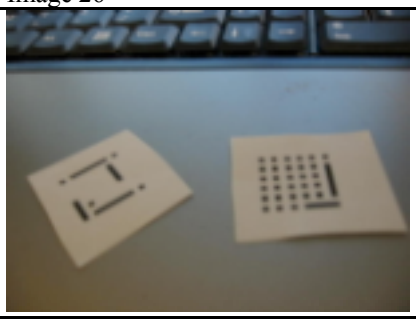
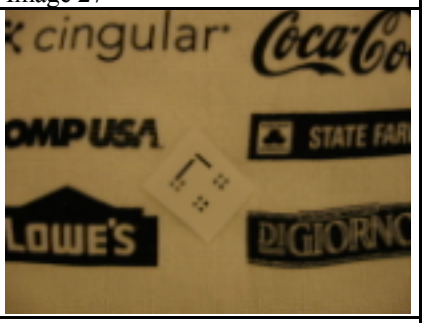

		

Image 28	Image 29	Image 30