VISUAL CODE MARKER SEGMENTATION AND DATA EXTRAPOLATION

Bradford Bonney, Evan Millar
Electrical Engineering Department
Stanford University
Stanford, CA 94305

Abstract

The purpose of this research is to automatically segment and read visual code markers. These markers are found in standard 640x480 pixel color images that were acquired with standard VGA cell phone cameras. In this paper, we investigate an approach to automatically identify and read visual code markers. The experimental results show that the algorithm discussed in this paper successfully segments visual code markers and reads all the data correctly. The average time for the segmentation and data parsing of twelve test images was 2.12 seconds per image.

1. Introduction

Visual code markers have a wide range of applications, including the automated tasking of applications in cell phones via the built-in camera [1]. With the increasing number of cell phones on the world stage, and the integration of CCD imagers into even the most modest of cell phones, the ability to have automated visual code marker detection and reading capabilities is a wonderful feature enhancement. While we are not concentrating on the applications of our developed algorithm, the applications of such routines need not be ignored.

The visual code markers that we consider are 2-dimensional arrays. The array consists of 11x11 elements. Each element is either black or white. As shown in the figure below, we fix the elements in three of the corners to be black. One vertical guide bar (7 elements long) and one horizontal guide bar (5 elements long) are also included. The immediate neighbors of the corner elements and the guide bar elements are fixed to be white. This leaves us with 83 data elements, which can be either black or white [2]. Figure 1 shows an example of the visual code marker we are trying to detect and read in the following algorithm.

Our initial experimental results show that it is, in fact, possible to locate multiple visual code markers embedded in low resolution, color images, and subsequently process the data they contain. We were successfully able to locate visual code markers in 23 test images, which contained anywhere from one to five visual code markers.

2. Corner Detection

In order to properly extract the data contained within each visual code marker, the four corners of the marker must first be identified. We refer to each of the corners based on their corresponding compass directions: northwest, northeast, southwest, and southeast. To determine the location of these coordinates, a four-step approach was used. First, the reverse "L" (two fixed guide bars) was identified within the image. Next, the northeast and southeast coordinates were labeled. Using geometric and algebraic calculations, the southwest coordinate was calculated next. Four, and finally, the northwest coordinate was found. If at any stage in the process a valid location was not identified, the potential marker was discounted as a false positive. Figure 2 shows the image to which the in-depth breakdown and discussion of our algorithm will be applied.

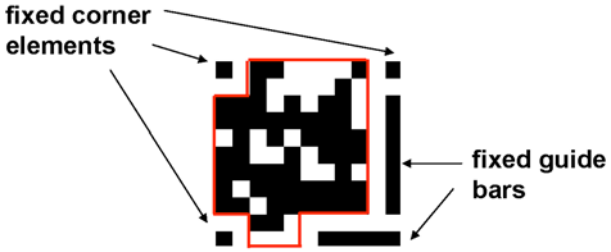

Figure 1: Visual Code Marker

Figure 2: Sample image with visual code markers

2.1. "L" Guide Bars for Northeast / Southeast Detection

The fixed lengths and proportions of the guide bars make them a logical choice for initially identifying potential visual code markers. Additionally, the relatively

solid-black on solid-white nature of the visual code markers makes working with binary images ideal [3-4]. In order to convert the color images to binary images, a thresholding algorithm needed to be applied. However, a simple mean value threshold could not be used due to potential shadowing and other local image intensity characteristics.

To make our thresholding algorithm as invariant to image intensity as possible, local thresholds were utilized. Our local threshold window was 40 by 40 pixels, and was applied in a non-overlapping fashion. The threshold was applied to each of the red, green, and blue channels of the color-RGB image separately. To achieve the final binary, thresholded image, all three channels were combined using a logical AND operator. This created a locally thresholded, binary version of the original input image as seen in Fig. 3.

Figure 3: Binary version of input image after application of a three channel local threshold

Clearly, the "L" shaped guide bars remain after the local threshold has been applied. Unfortunately, there is still a great deal of noise in the image. To eliminate this noise, we divided each mass of connected pixels into labeled regions as seen in Fig. 4. We cycled through each region and calculated the ratio of its major axis to its minor axis. The major axis and minor axis were calculated using

Figure 4: Labeled regions of binary visual code marker image

an ellipse that had the same second moment as the labeled region. This calculation was performed automatically using the built-in MATLAB® functionality [5].

Without any knowledge of the camera's intrinsic characteristics, we could not be certain of any perspective transformations, nor correct for them. However, knowing that an original visual code marker had vertical guides with dimension ratios of 7:1 and horizontal guides with dimension ratios of 5:1, we kept the region as a potential guide bar if the major-to-minor axis ratio was greater than 3:1. This allowed for some leniency when dealing with distortions incurred during the local thresholding process, as well as with markers in which the object plane and the image plane did not coincide – an assumption that was made during processing. Figure 5 shows the potential guide bars following this initial noise cancellation step.

Figure 5: Binary image: regions that remain have major-to-minor axis ratios greater than or equal to 3

To further reduce noise, we took advantage of the fact that the guide bars were consistently solid black (or in the case of Fig. 5, solid white, as they are the regions of potential interest). Since the guide bars are solid, we remove from consideration any region that has holes. If the region is not solid, it is no longer a potential guide bar. Additionally, due to the guide bars straight-lined nature, the area of their convex hulls are roughly equal to the area of the guide bar regions. By eliminating from consideration all regions whose convex hull area differs greatly (by more than 30%) from the area of the region area in question, we are left with Fig. 6 as the locations of potential guide bars.

Now that a majority of the noise has been successfully removed, and the visual code marker's guide bars still remain, we can proceed with determining which regions are guide bars. To do so, we exploit the relative spatial positioning of the guide bars and the fixed corner elements. Again by cycling through the remaining binary regions, we test to see if there is a binary region that exists a proper distance from the end of each guide bar given by (1).

$$d_x = \left(\frac{M}{2} + m\right)\cos(\theta),\ d_y = \left(\frac{M}{2} + m\right)\sin(\theta) \qquad (1)$$

Figure 6: Binary image: potential guide bars after removal of non-solid and non-convex regions

where d_x and d_y are the distance from the center of a region in the x and y directions, M is the major axis of the region, m is the minor axis of the region, and θ is the orientation of the region.

This specific calculation aims at locating the 7:1 guide bar, for if binary regions exist at distance d in both directions from the region of interest (finds both the 5:1 guide bar and the northeast fixed corner), then there is a good chance that the region is part of a visual code marker. To isolate the northeast block, we compare the calculated guess location (d_x,d_y) to the binary threshold image in Fig. 3. Figure 7 shows the visual code markers' "L" guide bars and the northeast fixed corner element. At the conclusion

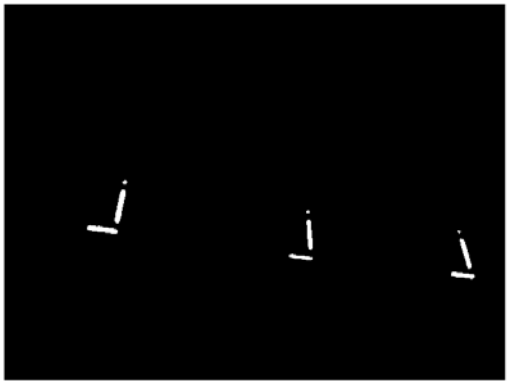

Figure 7: "L" guide bars and northeast fixed element isolation for all visual code markers in the input image

of this portion of the algorithm, all three markers have been identified and, since we know which regions are grouped together – linked to the 7:1 guide bar – each marker will subsequently be treated individually. The northeast and southeast corner locations are easily determined by the center of mass of the northeast fixed element, and the equivalent within the 5:1 guide bar.

2.2. Southwest and Northwest Corner Detection

Once the northeast and southeast corner coordinates are successfully determined, the next step is to isolate the southwest corner of the visual code marker. Since we already know which regions in Fig. 7 belong together, we will work with each potential visual code marker separately. While in this example the only remaining regions happen to be visual code markers, this is not necessarily the case, and therefore each of the following steps must also provide legitimate coordinates for corner locations or else the potential marker will be flagged as a false positive and ignored.

As with isolating the northeast and southeast region locations, a distance from the center of the 5:1 region to the estimated location of the center of the southwest fixed corner element is calculated by (2),

$$d_x = \left(3 \cdot \frac{M}{2}\right)\cos(\theta),\ d_y = \left(3 \cdot \frac{M}{2}\right)\sin(\theta), \qquad (2)$$

where d_x and d_y are the distance from the center of the 5:1 region in the x and y directions, M is the major axis of the region, and θ is the orientation of the region.

After the estimated location of the center of mass of the southwest fixed element is calculated, we select the region closest to that pixel location that is found in the original binary, locally threshold image. We keep the region as a positive match if the area of the region has approximately the same area as the northeast fixed corner element. Otherwise, we discard the potential visual code marker as a false positive.

With three corners positively identified, we need only to find the remaining northwest corner before data parsing can take place. To do so, we operate under one primary assumption: the visual code markers will appear in the image as orthogonal projections, not perspective projection. Visual code markers with perspective projection can be identified if the degree of their projection is not severe. This assumption was made after visual inspection of the initial twelve test images.

With our assumption of orthogonal projections in hand, the distance between the northeast and southeast coordinate was calculated and applied to the southwest coordinate. Similarly, the distance between the southwest and southeast coordinate was calculated and applied to the northeast coordinate. The average location of these two results yielded our initial estimate for the northwest corner of the visual code marker. Again, we searched within the binary, locally thresholded image in Fig. 3 for the region closest to the estimated marker coordinates. If the region's area was similar to (less than twice) the area of the northeast and southwest fixed element regions, then we considered the region the proper northwest fixed element. Otherwise, we considered the visual code marker a false positive and ignored the marker. Figure 8 shows all three visual code marker guiding elements (the guide bars and the fixed corner elements). Once we have these bounding conditions, we are ready to proceed with parsing the data contained within each visual code marker.

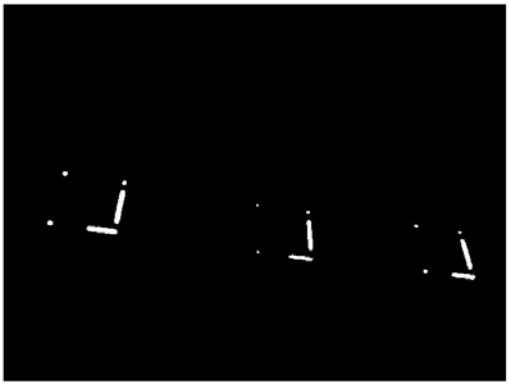

Figure 8: Final guiding elements found in original image

3. Marker Data Parsing

Once we have identified the 4 corners of a marker in the image, we begin our data-parsing algorithm. This algorithm's task is to (1) filter the image to maximize readability of the marker, (2) use the locations of the four corners of the marker to calculate where we should sample the image in order to read each data square on the marker, and (3) return a vector of length 83, containing the final data read from the marker.

3.1. Filtering Original Image for Data Retrieval

In order to be able to read the data contained in a marker we need to be able to separate the "black" portions from the "white" portions. The challenge in doing this arises from the fact that although the marker was printed on white paper with black ink, the actual pixel values in the digital image of this marker can be very far from black and white. Many factors—such as ambient light, noise introduced by low quality image sensors, blurring, and chromatic aberration, etc.—contribute to this problem. So a challenge exists in deciding which pixel values should be considered "white" and which pixel values should be considered "black".

In order to accomplish this we tried four different approaches, each with varying degrees of success. Each of the four approaches involved first converting the original image to grayscale. Since we are not concerned with color information in this task, converting to grayscale simplifies the calculations, and eliminates superfluous data. The first approach was a simple threshold at a hard coded threshold value. Although this approach is not adaptive and relies on multiple assumptions it is simple and fast. Unfortunately it did not work very well because of the issues mentioned in the previous paragraph. The second approach was a slight improvement on the first: a threshold value was set to the mean of the image. This is an improvement on the first approach because it adapts somewhat to the overall lighting conditions of the image. A mean-thresholding technique alleviates some of the problems (with ambient light for example); however, it introduces other problems such as large areas of black or white in the image skewing the mean gray value in one direction. The third approach extracted a local mean value for small blocks throughout the image, and subsequently thresholded each block with it's own local mean. We experimented with the block size but it was apparent that without knowing the size of the visual code makers in the image, we would run into the same types of issues we encountered with our second approach. In general, the problem is that the amount of black or white in the local blocks contributes to the mean value much more than any ambient lighting conditions.

Our fourth and final approach takes advantage of our knowledge of the location of the corners of the marker. If we know the value of a typical black spot on the marker and a typical white spot on the marker, we can assign a threshold value based on that particular marker's white value and black value. Figure 9 shows a single visual code

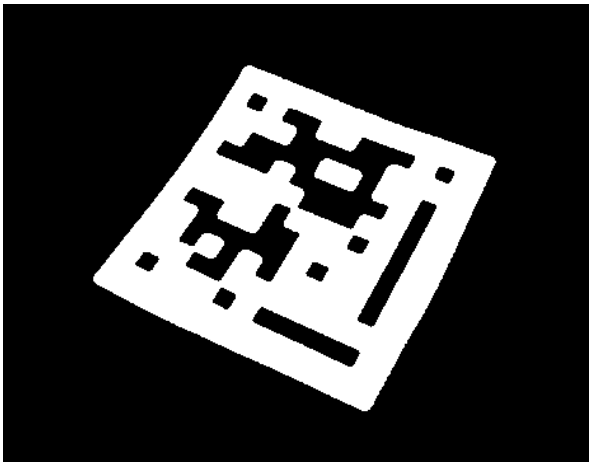

Figure 9: Binary visual code marker after local threshold

marker after the local threshold has been applied. Fortunately we know that the corner element pixels, of which we know the locations, all represent black values for that particular marker. We also know that if we move 1/10 of the distance between two adjacent corners we will land on a white block by virtue of the marker design as seen in Fig. 10. By taking mean values over four "black" areas,

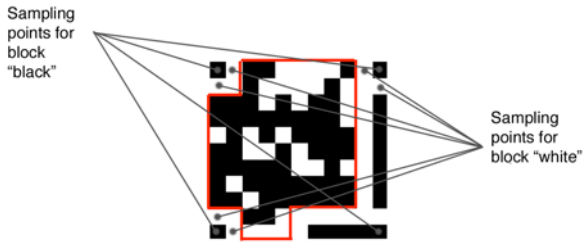

Figure 10: Visual code marker with "white" and "black" sampling points

and six "white" areas, and averaging those values, we are able to approximate a mean value that is specific to each visual code marker, as well as a threshold that isn't affected by the amount of black or white values within that marker. This method worked very well as it was highly adaptive, and was based on very solid assumptions of the marker layout. This method does not work if our corner points are wrong, but if our corner points are wrong we should have already discounted the potential marker as a false positive.

3.2. Reading Data by Sampling

Once we have a binary image that results from the thresholding described above, it is a matter of locating the center points of each block in the marker grid and sampling at and around each of those points. We decided in the beginning that, because in addition to rotation and scaling there could be a large range of perspective transformations on the markers, it would be unnecessarily difficult to try to rectify the image to bring the marker back to an orthogonal and upright position. We knew that if we had the corner points of the image we could interpolate between them to approximate the center point of each block on the visual code marker grid. This approximation can break down at extreme perspective distortions, but for our application purposes, it is sufficient.

The algorithm finds the sampling points by starting at the northwest corner and takes steps of one-tenth the distance from the northwest to northeast corner. By then moving our starting point to one-tenth of the distance between the northwest and southwest corners, and the ending point to one-tenth of the distance between the northeast and southeast corners, we could again interpolate between those to points to read the second row. This process continues across the entire visual code marker until we have sampled all 11 rows of the grid.

In order to actually retrieve the data contained within the markers, two sampling approaches were tried. The first approach was to simply check whether the center point of each cell in the grid was a black or white pixel in our binary image. The second approach was to sample a circle of diameter less than or equal to the step distance of that row and taking the majority. We hypothesized that the second approach would be more robust, but slower. In reality taking into account pixels around the center point not only slows down the sampling process but it also introduces data that is less reliable than the center point of the block. Our results were less accurate when taking into account the set of pixels near the center point than they were when relying completely on the center point. In the end we decided to use the faster, simpler, and more accurate sampling approach. Figure 11(a) the single point grid that represents our sampling location, and Fig. 11(b) is the overlap of that mask with a specific visual code marker.

(a) (b)

Figure 11: (a) Sampling grid (b) visual code marker with overlapping sampling grid

After the grid has been sampled, we make a vector of the 83 visual code marker points in which we are interested as outlined in Figs. 1 and 10.

4. Results

In order to verify our algorithm's effectiveness in segmenting and reading data from images containing visual code markers, twenty-three images were tested. These twenty-three images (twelve of which were provided by teaching staff) covered a wide spectrum of potential scenarios including a varying number of markers, differing projections (orthogonal, affine, minor perspective), and even cases in which a portion of the marker was not fully located within the test image. Four of these test images are displayed in Fig. 12.

Figure 12: Four examples of test images that have multiple visual code markers at various projections. Some of the markers are not fully encapsulated by the image, and these markers were all successfully rejected as false positives

For each of the twenty-three test images, all visual code markers were successfully segmented. There were zero false positives, zero false negatives, and zero repeated marker locations. Only one individual marker's data was not read correctly due to an extreme perspective projection. Figure 13 shows the one marker that was successfully

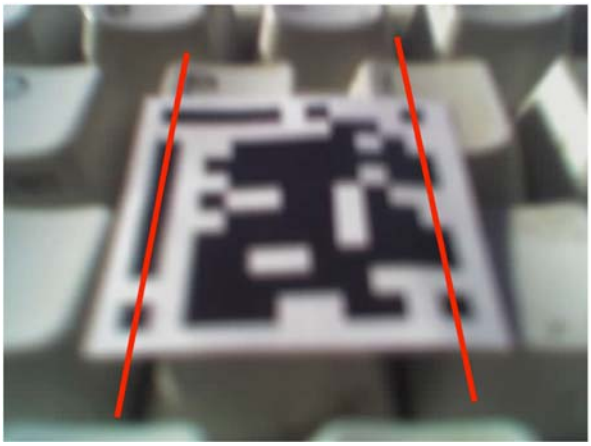

Figure 13: Visual code marker image that properly segmented but failed to correctly parse the data due to the perspective projection

segmented, but not successfully parsed for data due to the large perspective projection – a projection that does not maintain the parallel relationship between lines. Red lines have been added to the image to emphasize the perspective project, as the human brain tends to automatically correct for this transformation without our conscious knowledge.

The average time for processing the twelve provided images was 2.12 seconds per image, and the average time for processing the eleven additional test images was just under 3 seconds per image.

5. Conclusions

Our algorithm's approach to the segmentation and parsing of data from visual code markers embedded within color images has been shown to work for multiple test cases. Only in larger perspective projection does our fast algorithm begin to break down during data parsing. Throughout all of our testing, we never returned a false positive, false negative, or repeated any visual code markers. This high-speed approach shows that it is possible to properly segment and parse information from visual code markers with accuracy and precision, while not requiring massive computational times.

6. References

[1] ETH Zurich, Department of Computer Science, Institute for Pervasive Computing. Visual Code Recognition for Camera-Equipped Mobile Phones. 26 May 2006. 31 May 2006. <http://www.vs.inf.ethz.ch/res/proj/visualcodes/>

[2] EE368 Class Project, Spring 2005-2006. Visual Code Marker Detection. 26 May 2006. 31 May 2006. <http://www.stanford.edu/class/ee368/project.html>

[3] Gonzalez, Rafael C., Richard E. Woods, and Steven L. Eddins. Digital Image Processing using MATLAB. Pearson Prentice Hall. Upper Saddle River, New Jersey: 2004.

[4] Girod, Bernd. "Binary Image Processing." EE368 Course Lecture, Department of Electrical Engineering. Stanford University, May 2006.

[5] The MATLAB® function mentioned is the *regionprops(…)* function is found in the image processing toolbox.