OpenAR

OpenAR is a very simple C++ implementation to achieve Marker based Augmented Reality. OpenAR based on OpenCV and solely dependent on the library. OpenAR decodes markers in a frame of image. OpenAR does not implement Marker tracking across frames. Also OpenAR does not implement Template matching for Marker decoding.

Image Segmentation

With most applications of computer vision, one of the primary and most important tasks is to isolate the foreground from the background objects. This task is often referred to as segmentation or thresholding and its performance in isolating elements will determine how successful features can be extracted from the image. Since we are using Black over white markers, a binary image would suffice.

The color images retrieved from the camera must be converted to grayscale before being converted to binary. For this, we make use of opencv function-

```
cvCvtColor(color_img,gray_img,CV_RGB2GRAY);
```

Here `color_img` is the input color image, gray_img is the output of the function and `CV_RGB2GRAY` is flag which indicates the function to perform grayscale conversion from RGB color space.

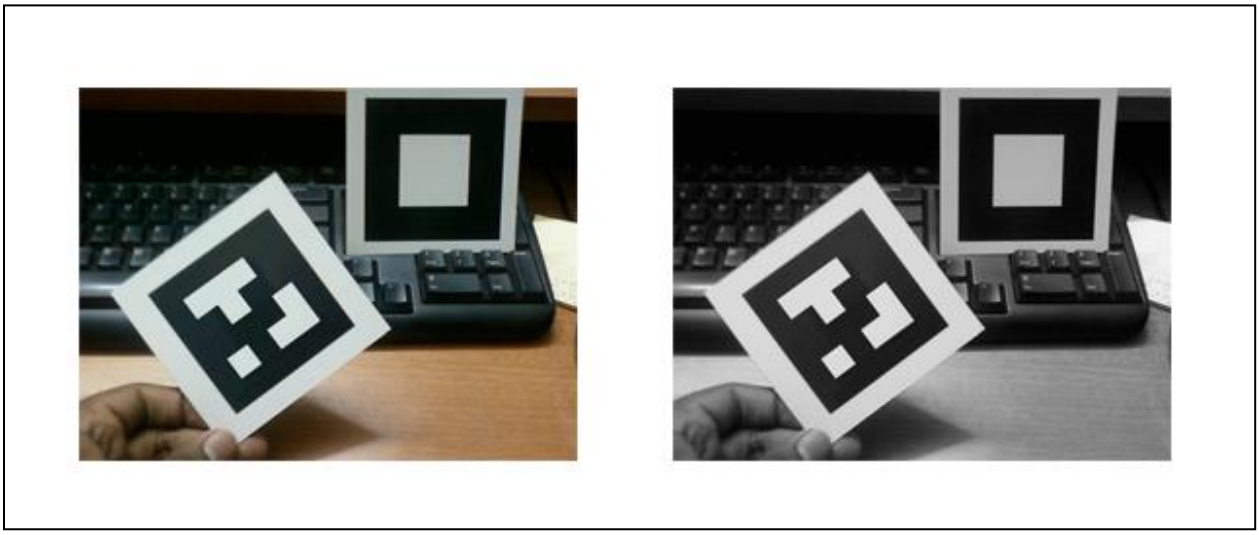


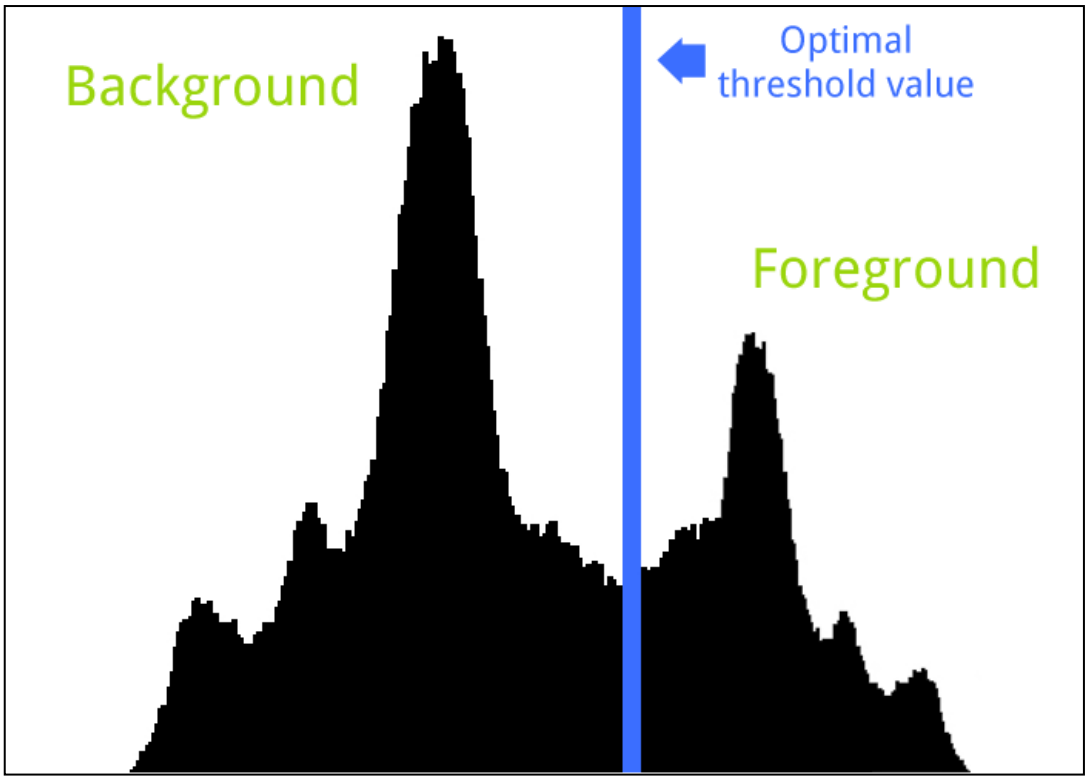
Figure:

Dynamic thresholding

Normally the static threshold operation provides satisfactory results. But under varying conditions for brightness, contrast and exposure the pattern will not be recovered quite easily. Static thresholding may erase valid points or pixels, that hold key to successful detection of patterns. With edges and corners not available the pattern detection may fail or report a false detection.

In order to overcome the drawbacks of static thresholding we implement Otsu method of dynamic thresholding.

The Otsu method works on the theory that there are two peaks in the grey values of an image's histogram, one representing the background and the other representing either the foreground or an object. Otsu makes the assumption that the lowest mid-point between these two peaks is the optimal threshold value, see figure.



The optimal threshold value is calculated from the Otsu method using the following equations:

$$\omega(k) = \sum_{l=1}^{k} p(l)$$

$$\mu(k) = \sum_{l=1}^{k} l \cdot p(l)$$

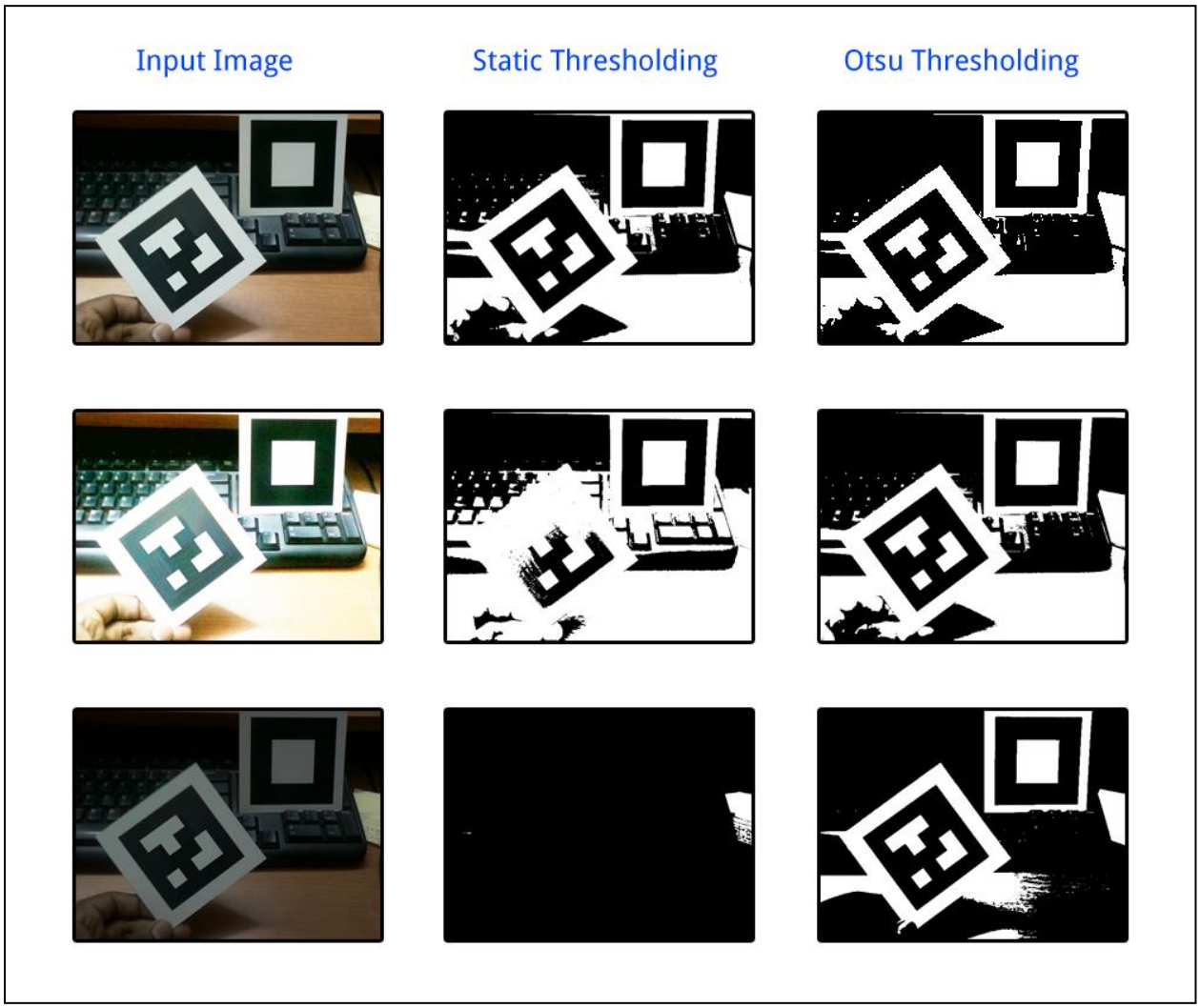
$$\mu T = \sum_{l=1}^{N_{max}} l \cdot p(l)$$

Using the previous equations, the final calculation can be expressed as:

$$\sigma_B^2(k) = \frac{(\mu T \cdot \omega(k) \cdot \mu(k))^2}{\omega(k)(1 - \omega(k))}$$

Where:
k = The histogram index position in the range 0 to 255.
p = The normalised histogram value for the current index position.
ω and μ = The first and second order cumulative values of the normalized histogram.
μT = The total mean level of the image.

A final comparison of how Otsu's algorithm performed against a static threshold is shown in figure.

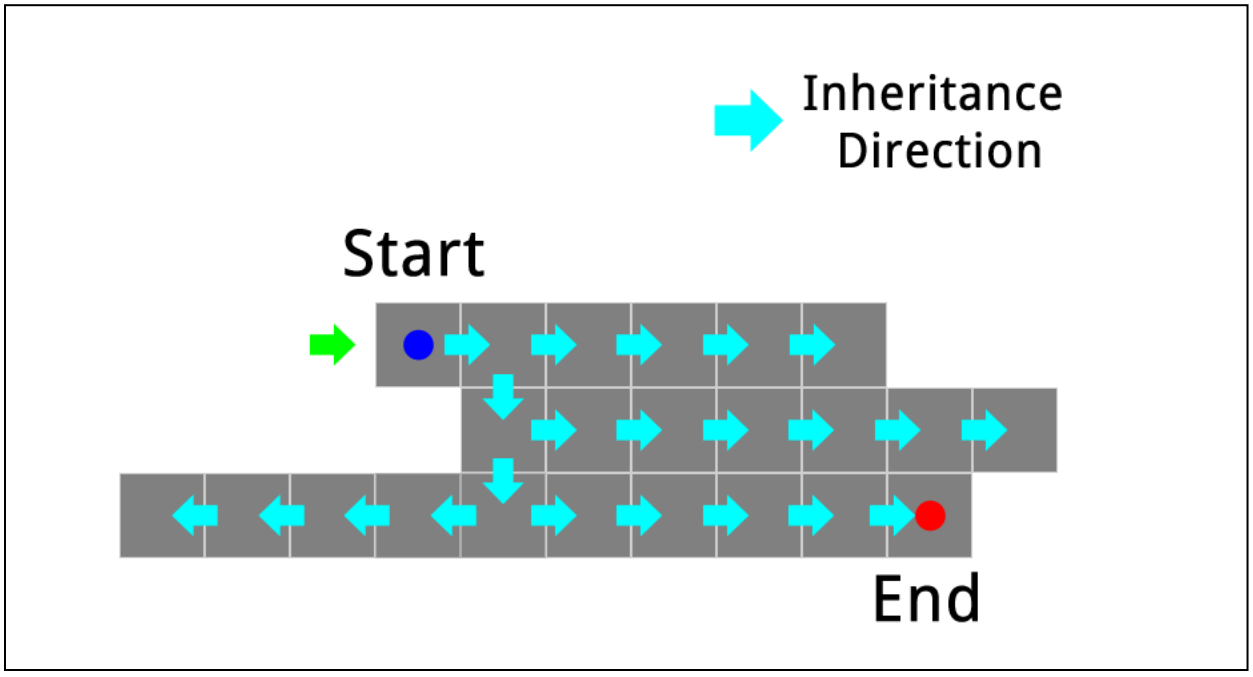


The first column shows the same scene with different lighting. The middle column shows the result after using a static threshold value and the third column shows the result with the Otsu dynamic thresholding method. It can clearly be seen that the Otsu method out-performed the static method, as the markers were more pronounced and also the noise caused from the shadows was greatly reduced.

Connect Components

The key step in identifying the marker in the program is based on the connected component analysis. The primary method used is the parent-child linking. By raising a flag for every connected pixel, we group them and fit a bounding box around them.

The initial grouping begins whenever the scan hits a black pixel. After this, we search for neighboring black pixels linked on the same line.



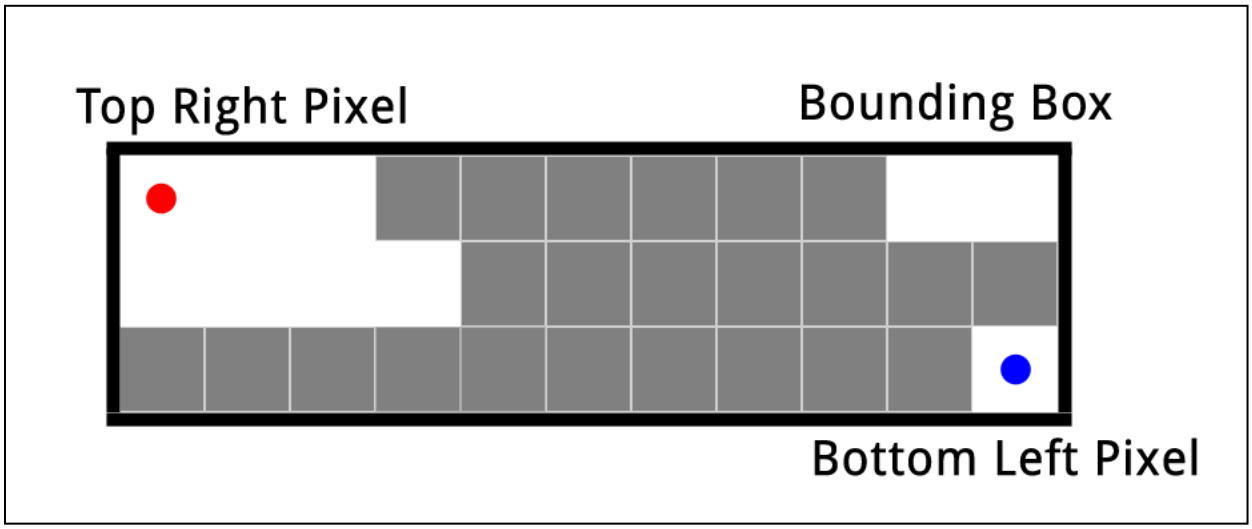
The process is described in the following steps:

1. Starting at the bottom-left pixel of the image, each row is scanned until a black pixel is found.
2. When a black pixel is found, it is flagged as being processed and a bounding box is created around it.
3. The four unprocessed neighboring pixels (above, below, left and right) are recursively scanned and processed and the extents of the bounding box are adjusted accordingly.
4. When no more black pixels can be found during the recursive process, the current bounding box is added to the list of regions that may potentially contain a marker.
5. The scanning of rows continues until another unprocessed black pixel if found, then steps 2 to 4 are repeated.

The pixel gains its inheritance by its surrounding pixels as shown in the above diagram. While scanning a bounding box is adjusted to accommodate the pixels. This is done by feeding the current pexel to the function-

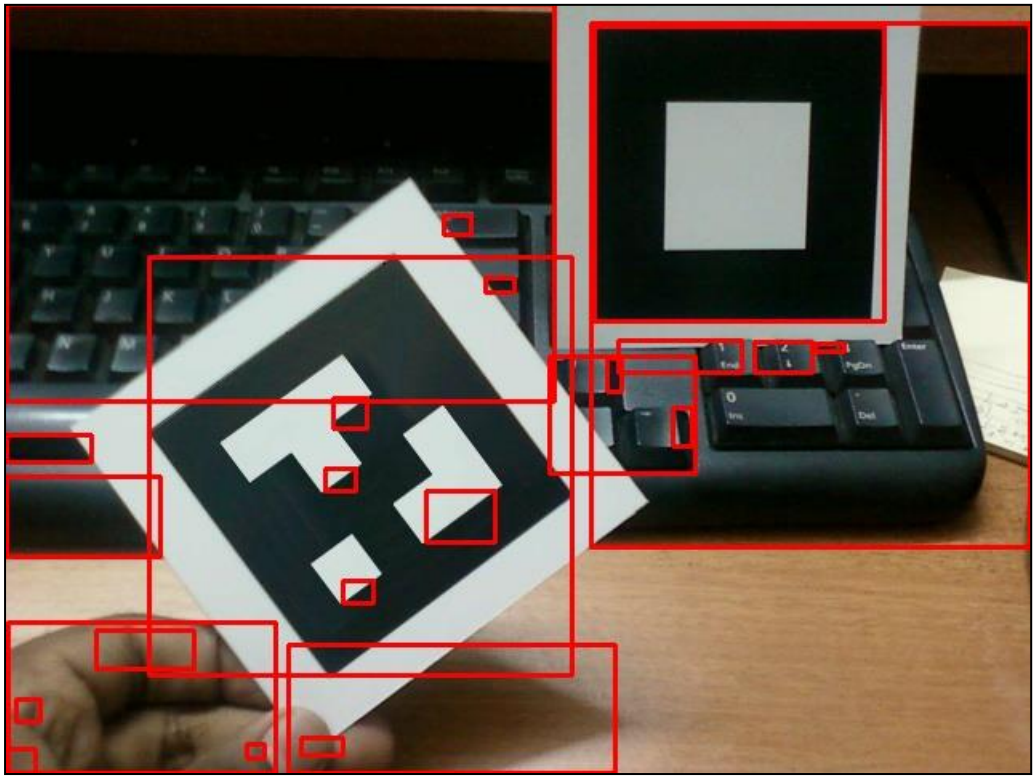
```
adjustboxcorners(x,y,cornerA,cornerB);
```

The points (x,y) indicate the present pixel. CornerA and CornerB represent the right-most and the left-most in-line pixels as shown below-



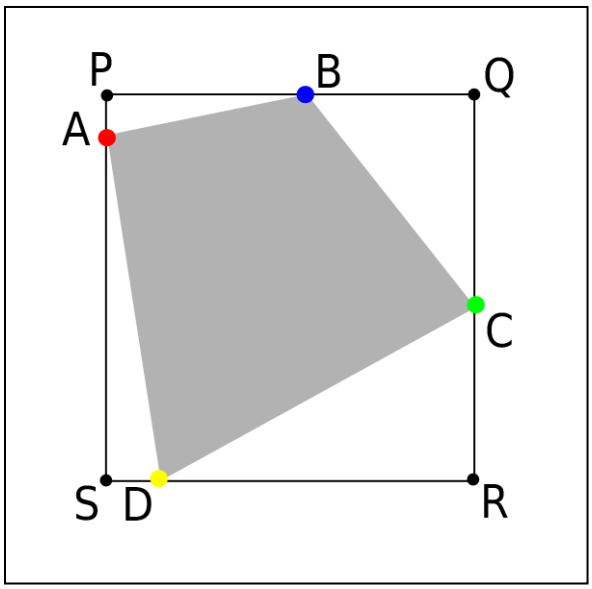
Considering these pixels assures that the bounding box encloses all the pixels of the same group. Note that whenever a pixel is found to belong to group, a test for corner is carried out and is saved for further processing.

Connected components applied to a sample image:



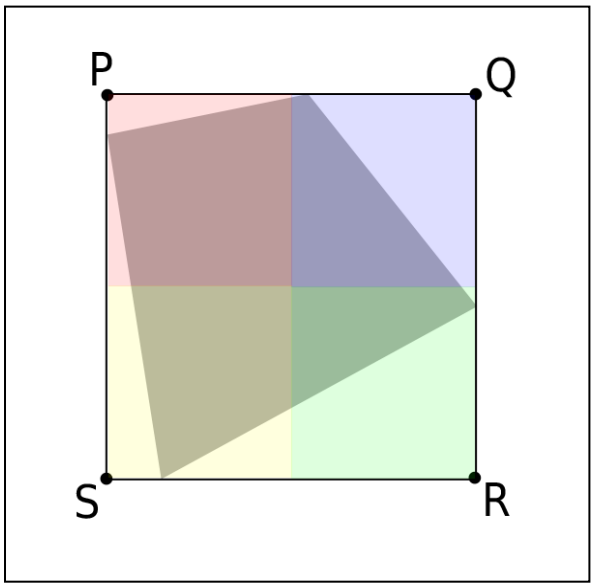
Quadrilateral detection

Once the bounding box around the regions is obtained, the region corners could be identified by working out the furthest edge pixel from each of the bounding box corners.



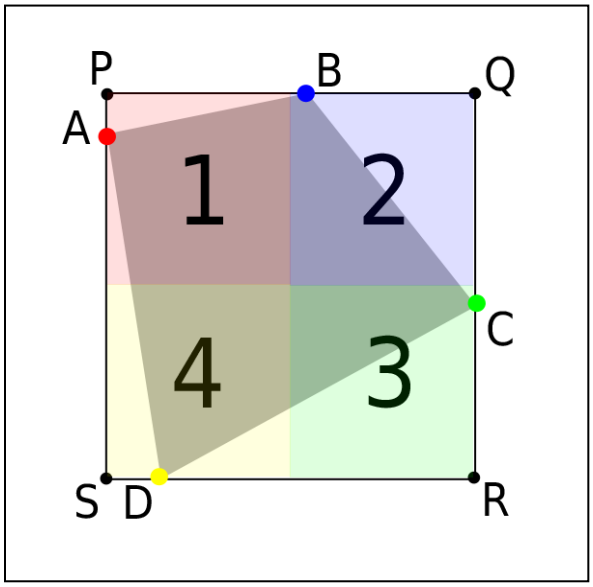
These pixels are necessary so as to recover the pattern free of distortion. From previous connected component analysis we gather the information about the bounding box of a region. This is nothing but the points P, Q, R and S. Using these points and the list of corners within the region, we have to identify the corners of the quadrilateral. This is achieved by following the method described below.

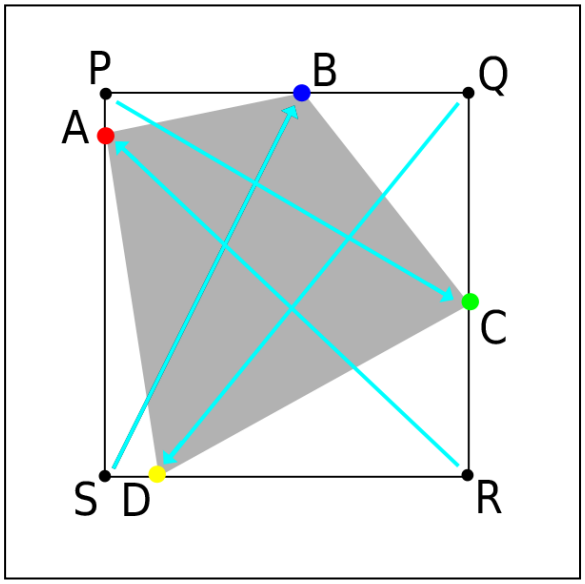
Firstly we divide the region into 4 equal parts along the center (namely 1, 2, 3 and 4).



Now each part is separately considered and a corner farthest to the points of bounding box is chosen. A corner is classified to a particular region depending upon which half of the bounding box it is present in.

This can be explained with the diagrams below-





From above figures we see that the:

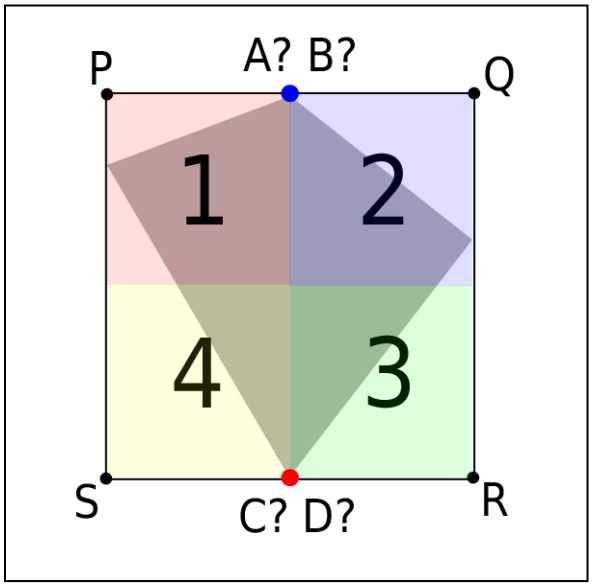
-Point C is the farthest corner from P (Part 1).
-Point D is the farthest corner from Q (Part 2).
-Point A is the farthest corner from R (Part 3).
-Point B is the farthest corner from S (Part 4).

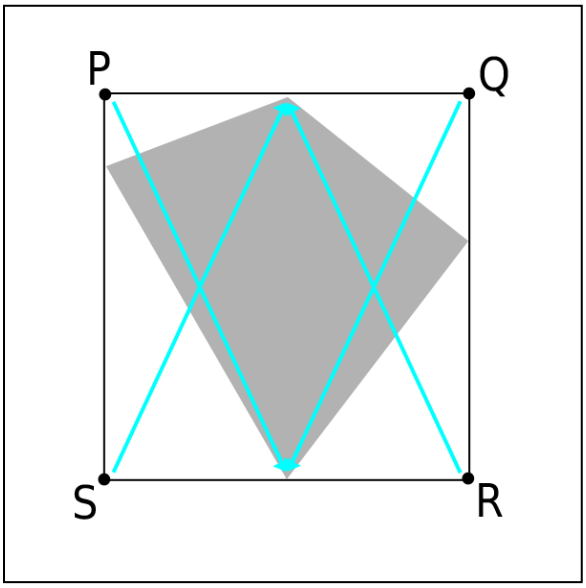
Hence A, B, C and D are chosen as the corners of the quadrilateral which will be use to remove the projective distortions.

The function `marker_points(total_corners,corners_list,cornerP,cornerR,A,B,C,D)` returns the corners of the quadrilateral.

`total_corners` are the total number of corners in the region
`corners_list` is the list of corners in the region
`cornerP,cornerR` are the corners of the bounding box
(Note: It is enough to provide only P and R, if necessary Q and S can be calculated from them)
`A,B,C,D` are the corners of the quadrilateral region.

This technique works for all cases but except one. This happens when the corners are at the center of these parts, as a consequence the algorithm will be unable to decide between the corners.

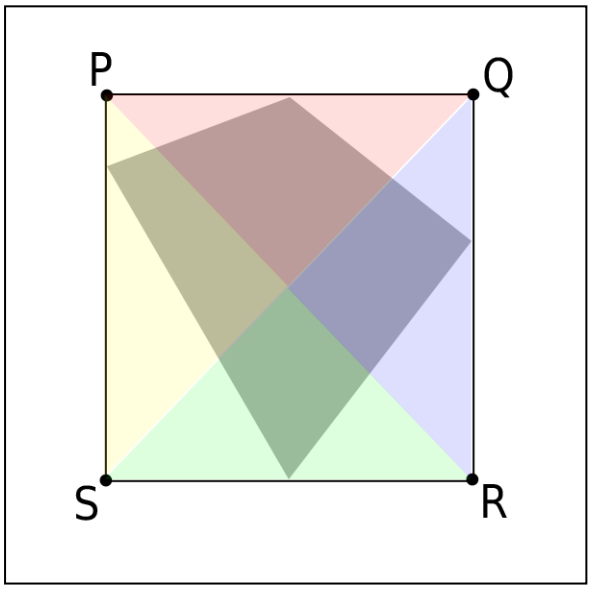




In this situation, a corner will be farthest one to two regions and thus giving rise to ambiguity and errors.

However, we can overcome by slightly modifying the technique to choose different regions. If the detection of the four corners fails, we subject the region to a second technique explained below. Using this method we should recover the corners and in any orientation.

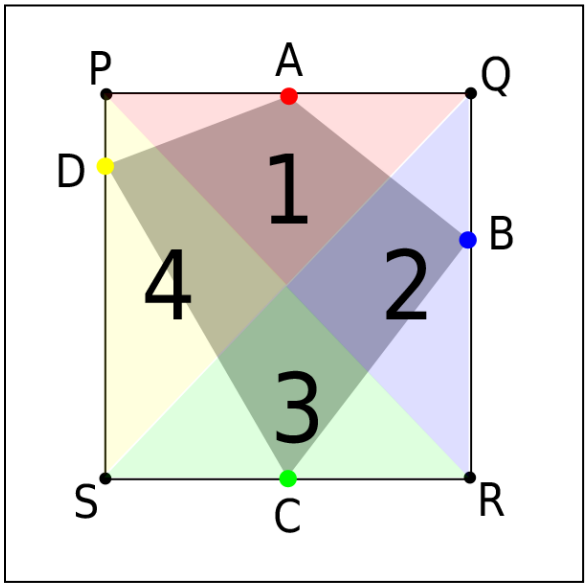
The modified method involves dividing the region into 4 equal parts along the diagonals.

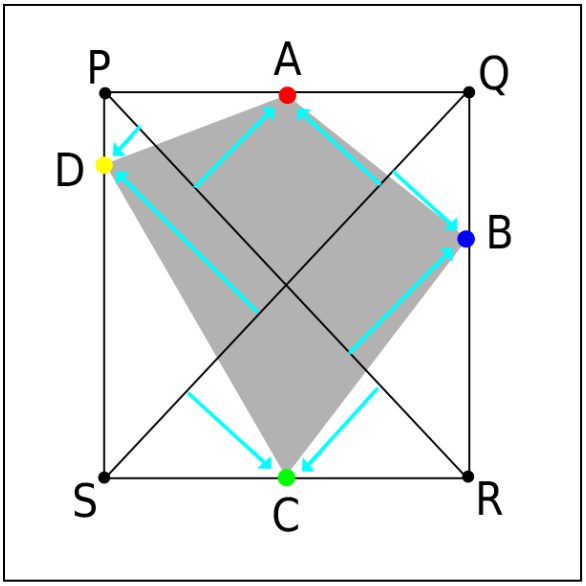


Now instead of calculating the farthest corner from the bounding box, we have to make use of a slightly more complicated calculation. In this method we identify the diagonals of the bounding box. Using diagonals as reference and the 4 parts of the region we identify 4 corners uniquely.

We classify every corner in the region depending upon the value of the perpendicular distances from the diagonal.

- If both the perpendicular distances are positive, it is Part 1
- If both the perpendicular distances are positive, it is Part 3
- If one of the perpendicular distance is positive (from PR) and the other negative, it is Part 2
- If one of the perpendicular distance is positive (from SQ) and the other negative, it is Part 4





As we see, the 4 corners that previously could not be recovered have been identified.

The key is to identify a corner which is at <u>maximum perpendicular distance</u> from *both* diagonals in each part.

From the above diagrams we have:

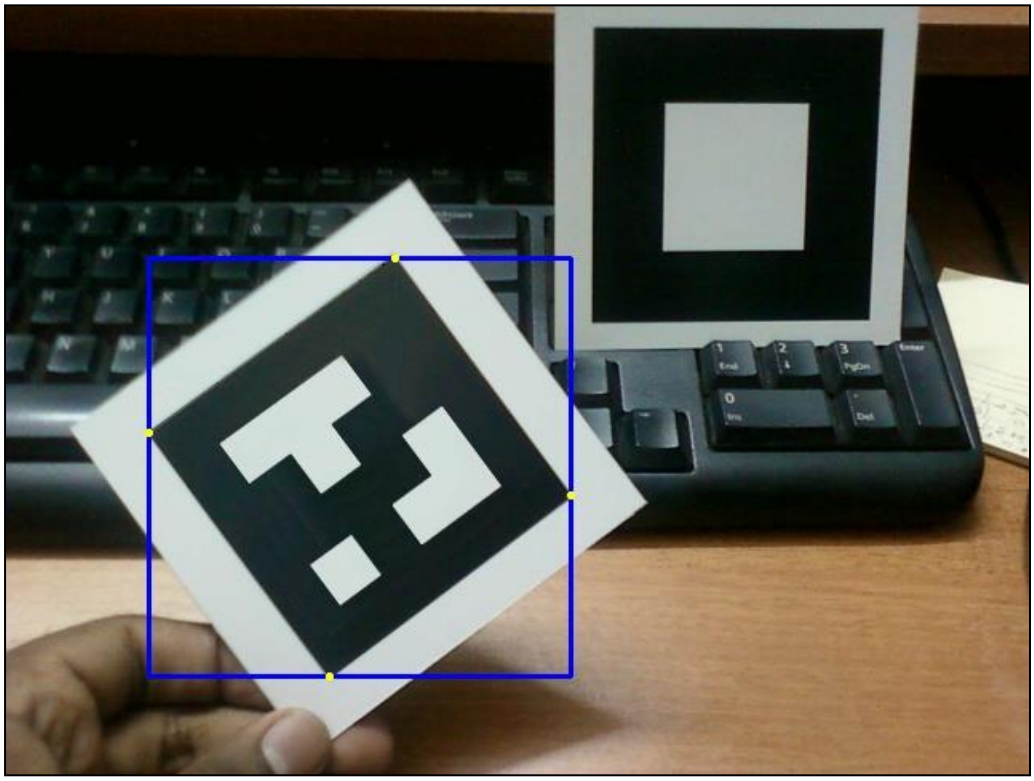
-Point A is perpendicularly farthest corner from diagonals in Part 1.
-Point B is perpendicularly farthest corner from diagonals in Part 2.
-Point C is perpendicularly farthest corner from diagonals in Part 3.
-Point D is perpendicularly farthest corner from diagonals in Part 4.

The function `another_region_points(total_corners,corners_list,cornerP,cornerR,A,B,C,D)` returns the corners of the quadrilateral.

`total_corners` are the total number of corners in the region
`corners_list` is the list of corners in the region
`cornerP,cornerR` are the corners of the bounding box
(Note: It is enough to provide only P and R, if necessary Q and S can be calculated from them)
`A,B,C,D` are the corners of the quadrilateral region.

This by utilizing method 1 and method 2 in ambiguous cases, we detect the 4 corners of the quadrilateral. These 4 corners will be used further to recover the bit pattern.

Quadrilateral corners detected from a sample image:



Corner Detection

There are numerous methods to detect corners or feature points in an image. Although opencv had a corner detecting function `cvGoodFeaturesToTrack()`,instead we made use of a much faster algorithm. The algorithm detects a corner by scanning through the periphery of a rectangular window around the pixel. This is explained using the following image:

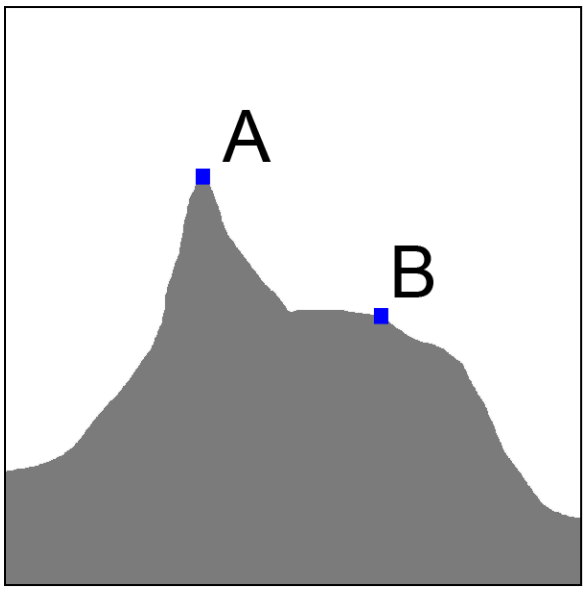
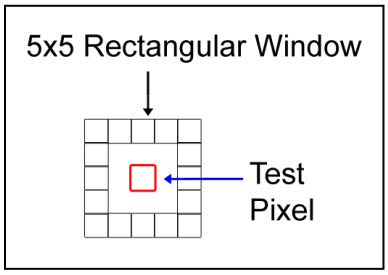


Figure : Points A and B are subjected to the corner detection process.

The function that carries out the corner detection is –

```
bool find_corner(char* img_data,int img_width,int x,int y);
```

The img_data is a pointer to the image data storage and the img_width is the width of the image (which comes in handy to jump to required data in the memory). The variables x and y are the co-ordinates of the pixel in interest.



The algorithm considers a simple rectangular window around the given pixel as shown in the figure. Then it scans on the periphery of this rectangular window counting all the black pixels. If the number of black pixels around the point is less than a fixed threshold, then the function confirms that it as a corner. The size of the rectangular window and the fixed threshold can be altered to suit the user.

```
const int wind_sz = 5;
```

The size of the rectangular window is 5.

```
if(sum <= 7)
```

This condition sets the threshold for the number of black pixels.

Thus the applying the function to points A and B we see that only A is detected as a corner (*For the sake of simplicity, black is shown as gray in the images).

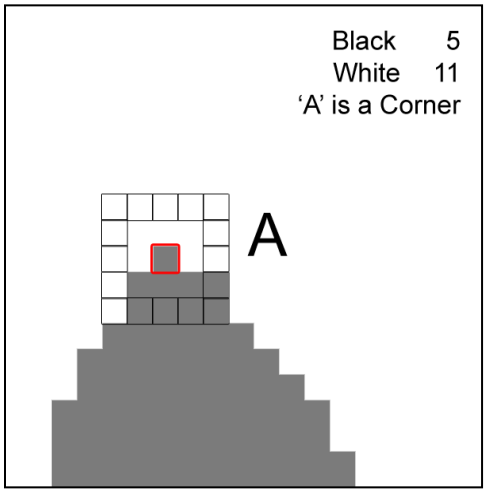
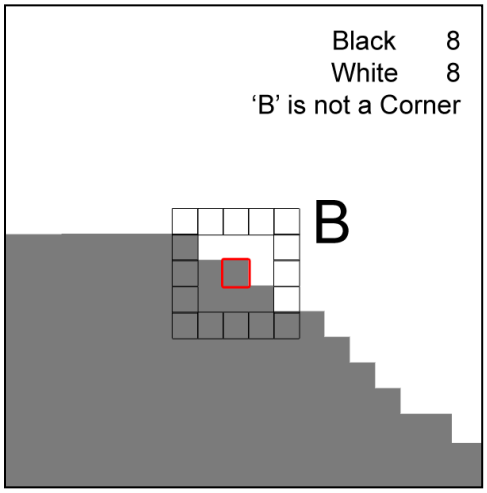
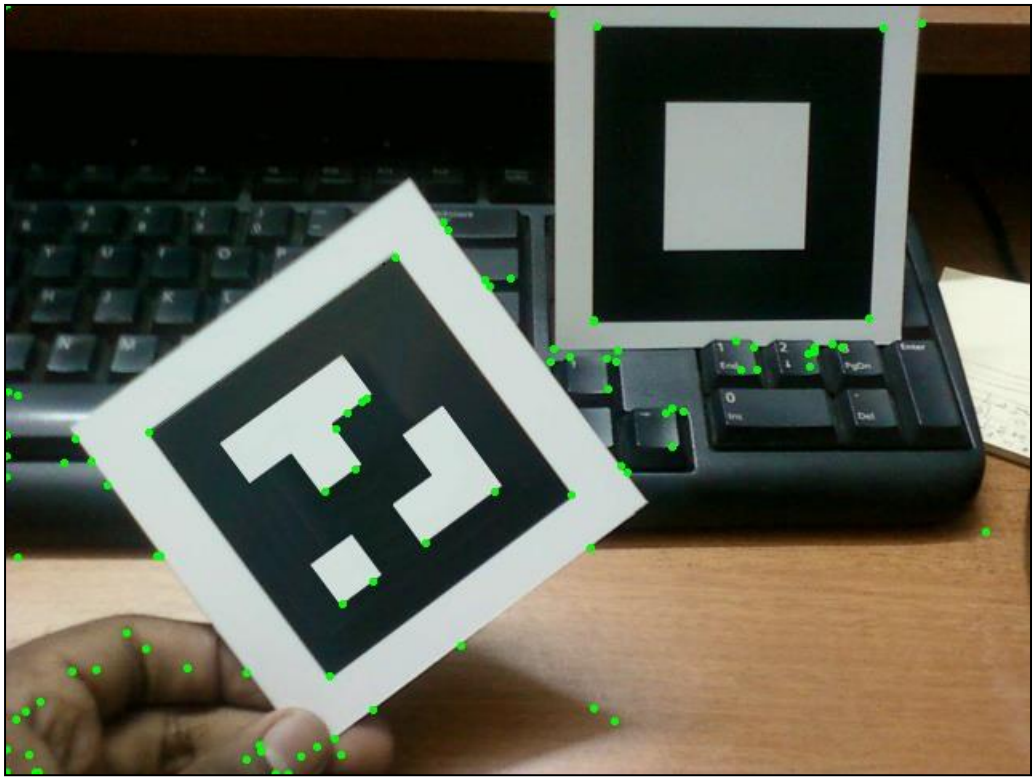


Figure: Zoomed Image



Figure

Corner detection applied to sample image:



Marker Identification

By applying previously discussed Connect components and corner detection algorithms we obtain a list of candidates for marker detection.

The projective distortion of these potential markers must be compensated [fig] in order to detect valid markers. In order to achieve this, we need apply the 2D-2D projective mapping or planar Homography.

Projective Mapping

The *projective mapping*, also known as the perspective or homogeneous transformation, is a projection from one plane through a point onto another plane. The 2-D projective mappings studied here are a subset of these familiar 3-D homogeneous transformations.

Basically we are transforming a set of points in an image to another set of points in an image, which sounds like a mapping from two dimensions to two dimensions. But this is not exactly correct, because this transformation is actually mapping points on a two-dimensional plane embedded in a three-dimensional space back down to a (different) two-dimensional subspace. Think of this as being just what a camera does, the camera takes points in three dimensions and maps them to the two dimensions of the camera imager. This is essentially what is meant when the source points are

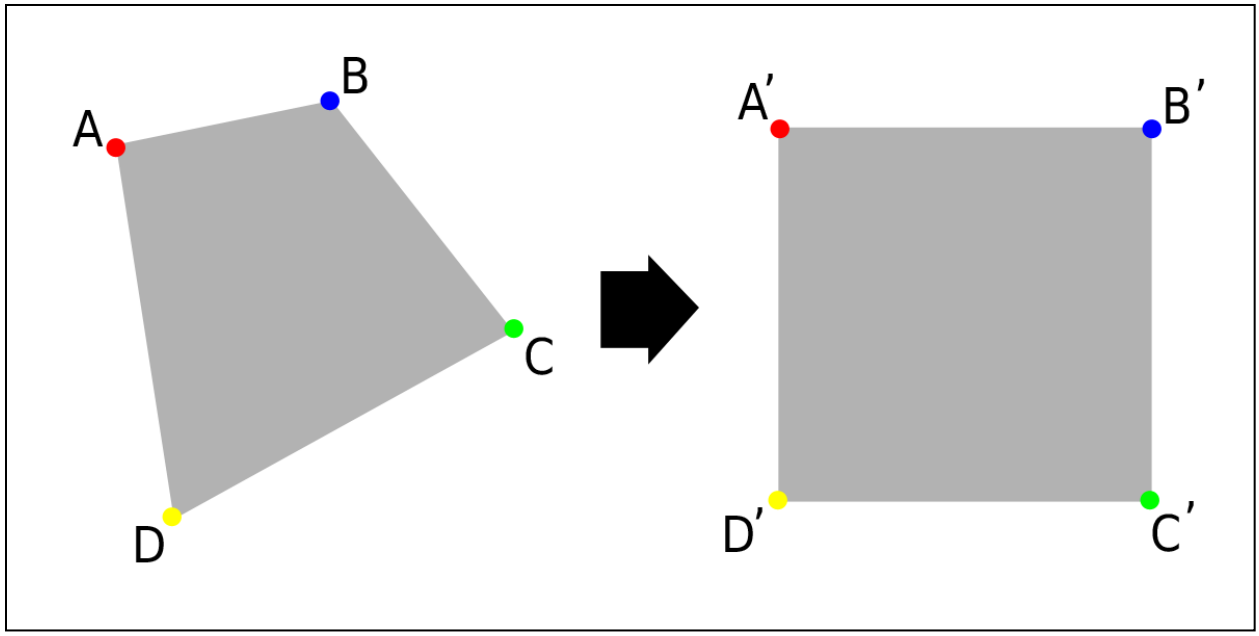


Figure : Projective mapping

taken to be in "homogeneous coordinates". We are adding an additional dimension to those points by introducing the *w* dimension and then setting all of the *w* values to 1.

The general form of a projective mapping with set of points (u, v) mapped to another set of points (x, y):

$$x = \frac{au + bv + c}{gu + hv + i}$$

$$y = \frac{du + ev + f}{gu + hv + i}$$

These equations can also be represented in the homogeneous matrix notation:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ q \end{bmatrix} . \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

where $(x, y) = (x'/w\,, y'/w)$ for $w \neq 0$, and $(u, v) = (u'/q\,, v'/q)$ for $q \neq 0$. As discussed, we will equate $i = 0$,

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ q \end{bmatrix} . \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 0 \end{bmatrix}$$

For simplicity, we shall denote the matrix on the left as H:

$$H = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 0 \end{bmatrix}$$

$$\therefore \begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ q \end{bmatrix} . H$$

With $i = 0$, we are left with 8 unknowns. By using the four corners of the reference square and the four corners of the quadrilateral, eight equations can be defined. Each corner mapping produces the following equations:

$$x_k = \frac{au_k + bv_k + c}{gu_k + hv_k} \implies u_k a + v_k b + c - u_k x_k g - v_k x_k h = x_k$$

$$y_k = \frac{du_k + ev_k + f}{gu_k + hv_k} \implies u_k d + v_k e + f - u_k y_k g - v_k y_k h = y_k$$

for $k = 0,1,2\ and\ 3$.

The equations can be re-written as:

$$\begin{bmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

The linear solution can be solved using Gaussian elimination or by calculating the inverse of the 8x8 matrix (which will not be discussed here).

With this understanding, we will make use of OpenCV functions to calculate the H matrix. We utilize the OpenCV function `cvGetPerspectiveTransform()`.

```
CvMat* cvGetPerspectiveTransform(
       const CvPoint2D32f* pts_src,
       const CvPoint2D32f* pts_dst,
       CvMat* map_matrix
);
```

The `pts_src` and `pts_dst` are arrays of 4 points. The transformations will be saved to the 3 by 3 array `map_matrix` which is nothing but the H matrix.

As we know the potential marker(-a quadrilateral) must be mapped to a square. The points A,B,C and D have to be mapped to A',B',C' and D' respectively. Hence we have:

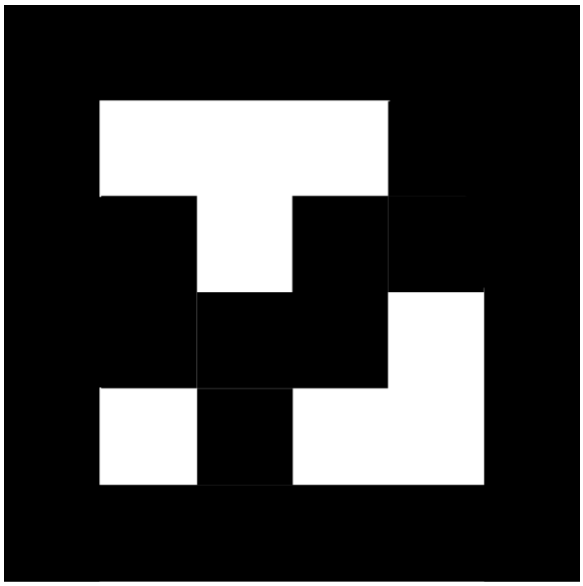
`pts_src[0]=` A (u_0, v_0);
`pts_src[1]=` B (u_1, v_1);
`pts_src[2]=` C (u_2, v_2);
`pts_src[3]=` D (u_3, v_3);

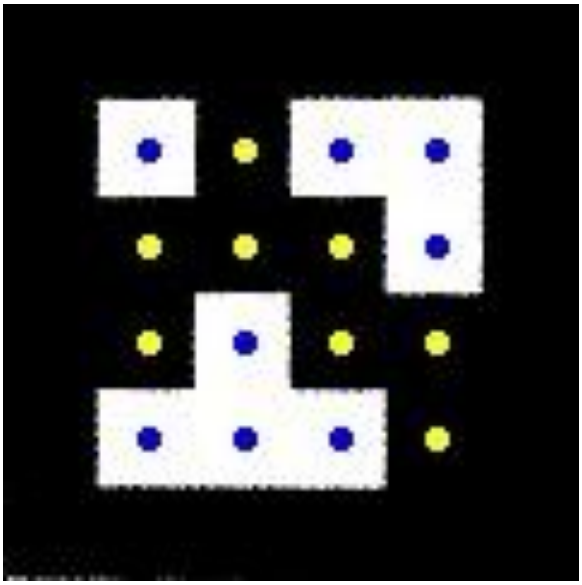
`pts_dst[0]=` A' (x_0, y_0);
`pts_dst[1]=` B' (x_1, y_1);
`pts_dst[2]=` C' (x_2, y_2);
`pts_dst[3]=` D' (x_3, y_3);

Thus by feeding the above values to `cvGetPerspectiveTransform()`,we can retrieve the H matrix.

Marker Decoding

The marker is a two dimensional bit pattern containing binary code. The default marker used is shown in fig .





The marker is divided into 4x4 matrix where each cell represents a single bit in the identification code.

The function `int find_marker(IplImage* trans_img)` calculates the unique marker identification number. The input the function is the squares recovered from the quadrilateral detection and projective mapping.

The marker identification numbers are calculated by squaring the weights of the bit position.

```
value += (i*i);
```

Thus depending upon weather the pixel is white are black within the region, weights are squared and accumulated. This way a marker is uniquely identified.

2D Augmentation

It is now possible to achieve augmentation but only for 2D object viz. an Image. By utilizing the position of the corners we can modify the image so as to suit the real world. All we have to do is use the inverse H matrix and a image to be augmented. By properly mapping the points in the image to the frame we can achieve 2D augmentation.

A sample 2D augmentation example is shown below-

References for this section:

Paul Heckbert- *Projective Mappings for Image Warping*
Shahzad Malik -*Robust Registration of Virtual Objects for Real-Time Augmented Reality*
Learning OpenCV, Perspective Transform p169-p172.O'Reilly Media.

References and Further reading:

1. [Book] Learning OpenCV - Computer Vision with the OpenCV Library
   By Gary Bradski, Adrian Kaehler
   First Edition
   http://shop.oreilly.com/product/9780596516130.do

2. ARlib – C++ Augmented reality library
   by Danny Diggins

3. Features from accelerated segment test
   by Edward Rosten
   http://www.edwardrosten.com/work/fast.html
   http://en.wikipedia.org/wiki/Features_from_accelerated_segment_test

4. Connected Components Analysis
   http://en.wikipedia.org/wiki/Connected-component_labeling
   http://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm

5. Perpendicular Distance of a Point from a line
   http://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line
   http://www.intmath.com/plane-analytic-geometry/perpendicular-distance-point-line.php
6. Solutions to Equation of a Line
   http://en.wikipedia.org/wiki/Linear_equation
   http://doubleroot.in/straight-line-position-of-a-point-relative-to-a-line-examples/#
   http://www.emathzone.com/tutorials/geometry/position-of-point-with-respect-to-line.html

Important OpenCV Online Resources

http://www.seas.upenn.edu/~bensapp/opencvdocs/ref/opencvref_cv.htm