AR Tag Detection, Decoding, Tracking, and Superimposing image and cube on the tag : ENPM673

Rahul Karanam
Robotics Graduate Student
University of Maryland, College Park
College Park, MD
rkaranam@umd.edu

I. Introduction

This report explains the process of detection and tracking of a AR Tag(April tag - fiducial marker) using homography and perspective projection.

We will be looking about how to detect the AR tag, decoding the tag to get its orientation and id. First section will explain about detecting the tag from the sequence and decoding information from it.After detecting the tag, we then superimpose the testudo image (refer Fig-4) for all the frames in the given video (refer the docs directory). Later sections discussed how to project cube on the AR tag in all the sequences of the video using projection matrix.

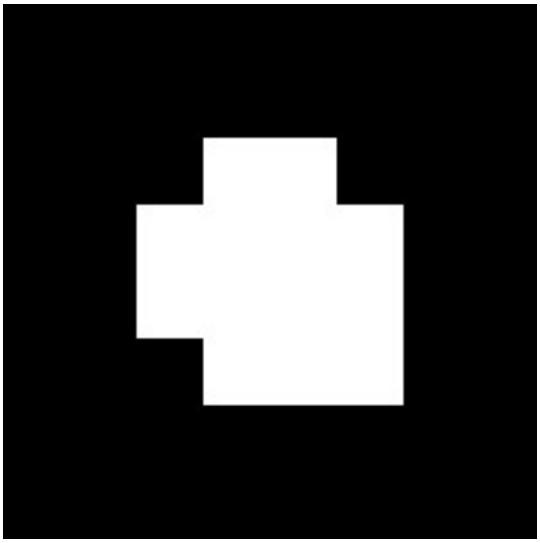

Fig. 1: AR Tag Reference

II. DETECTION OF AR TAG

Pre-processing of the image:

We'll show you how to find the corners of the AR tag and edges using Fast Fourier Transform in this section.

Before detecting the corners of the AR tag, we need to pre-process our image in order to smoothen it out to get better edges and corners.

1) First, we take the video as a input frame by frame, then convert the image to a grayscale image.
2) Apply a gausian blur of (7,7) kernel along with opening morphological operation in order to get rid of outliers and small black spots found in the frame.
3) We convert the above image to a binary by applying thresholding with a range of 170-255.
4) We then apply histogram equalization to our image in order to improve the contrast of the filtered image. I have used CLAHE(Contrast Limited Adaptive Histogram Equalization) for doing the histogram equalization.

Finding edges of the image using Fast Fourier Transform(FFT):

Now after pre-processing our image, we then find edges using Fourier transform. The main advantage of finding edges using Fourier transform is it gives more finer details and faster than normal convolution process. Please find the below process how to find edges of the image using FFT.

1) First we convert our image to frequency domain using the np.fft and then shift the center to zero frequency as our image has origin at left corner(opencv).
2) We find the magnitude of our frequency and then apply a circular mask to block all the lower bounded frequencies which will give out the edges. I have taken my radius for the mask as 100.
3) After the mask, we need to apply the fft inverse to get back to time domain.
4) Please look at the Fig 2 which shows the frequency domain,mask and the edges of the image.

Finding the corners of the tag in the each frame:

Now we have applied the mask and got the edges of the image using FFT in the above method. Now we find the corners of the tag using min-max approach. I have used Shi-Tomasi(goodfeaturestotrack) corner detector to detect the corners in the image. I have selected the best

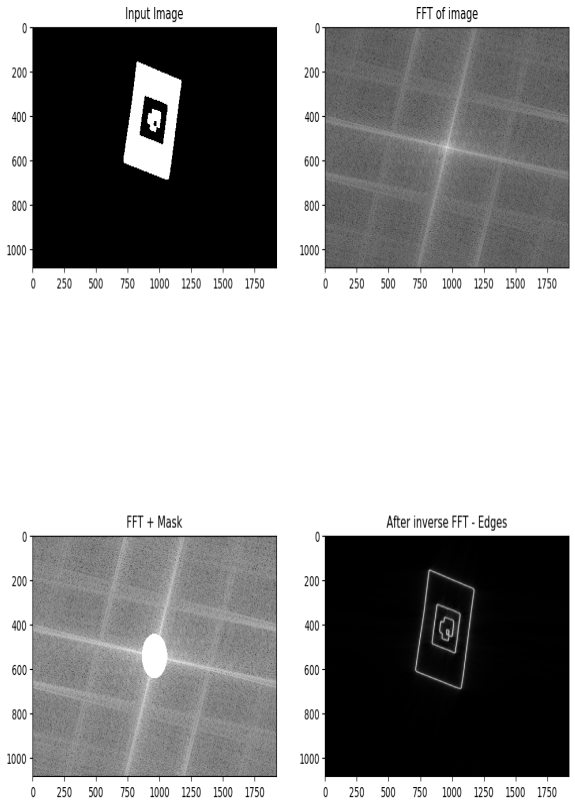

Fig. 2: Finding Edges using Fourier Transform

20 corners. It returns a output of 20 best corners on the image.

Now we then find the indices of x-min,y-min,x-max,y-max in the corners. After finding these min-max corners, I have created line equations which I'll loop it again over my corners and get the corners which only lie inside the boundary. After the first iteration, I get a certain list of corners , I again loop over those points while removing my already visited points(min-max coordinates of the paper).

This loop will return the four corners of the AR tag. The main issue after finding the corners was to arrange them in a cyclic order or in a clockwise direction as it will be easy to map these corresponding points later doing homography. I have sorted the points by first sorting the columns and then finding the left and right most points of the sorted array. Now we just find the distance between them and sort them based the distance. The order which I have followed is top-left,top-right,bottom-right,bottom-left.

Please refer Figure 3 for your reference about the corners. I have plotted lines in order to show the boundary of the detected tag using cv2.line.

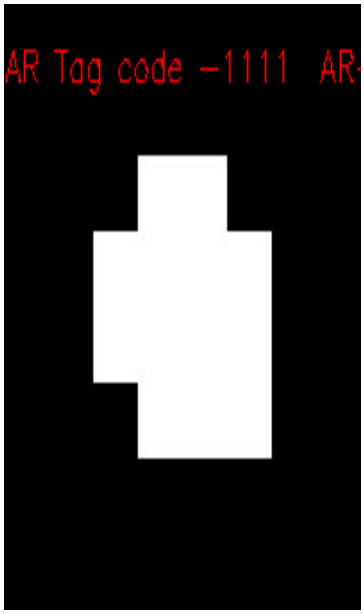

Fig. 3: Decoding the Tag from reference image from frame

Decoding the AR tag

Now we need to decode the AR tag in order to check the orientation a.k.a pose of the tag as it will be used later in the sections. Please refer to the reference image of the AR tag which will be used as a reference for decoding the tag.

Please find below the steps in order to decode the tag and get its orientation and tag-id

1) Convert the image to gray scale , and then divide our image into 8 x 8 grids.
2) Create a array of checking only the outer corners starting from 0 till 15.
3) Now we substitute all the pixel with the mean greater than 127 to 1 and other pixels as 0.
4) Now we check for every corner of the 8 x 8 grid.
5) If there is a white pixel present in the top-left corner first we check the orientation of the tag and then we take the 4 x 4 grid inside the tag to decode it.
6) we flatten the 4 x 4 array and convert it to a string to get the tag-id as binary where the left most bit is the least significant bit and right is the most significant bit.
7) We then reverse the order of the tag id as we get the values in the reverse order.
8) Please refer to the refernce image where the white pixel is at the bottom-right which is the upright position.
9) After applying the above decoding procedure we get the tag id as 1111 (15).

Fig. 4: Finding corners

Fig. 5: Testudo Image used for super imposing onto the tag

III. TRACKING OF AR TAG

As per our above frame , we can find four corners of our tag using the above mentioned method.

We will be doing two different tasks here in this section. First we will superimpose the testudo image onto our tag and the later one is to project a 3D cube onto the tag using homography and projective geometry.

In this sub-section we will be going through the process of superimposing our reference testudo image on a tag using our own custom homography and warp-perspective functions.

A. SUPERIMPOSING IMAGE ONTO TAG

Below are the steps for a single frame in the sequence which can be later extended to all the frames.

1) First we load our reference testudo image and our frame.
2) We find the corners of the tag using the get-corners method which returns the four corners of the tag from the frame.Please refer to the above section 2.1 for finding the corners of the tag from a frame.
3) Then we find the homography matrix between our tag corners and refernce tag corners.
4) After finding the homography , we then give this matrix to our custom warp-perspective function which will multiply my homography matrix will all the x,y coordinates and get into the world coordinates.
5) The ouput from the warp-perspective function will be a 160 x 160 tag which will then be decoded to get the orientation and tag-id.
6) We then rotate our testudo image as per the orientation our tag given by decoding it.
7) In order to superimpose the testudo image onto the tag, we need to find homography between these two in order to move from one plane to other.
8) Finding homography can be found after these steps.It generates a 3 x 3 matrix given four corresponding points of both the tag corners and testudo corners.
9) We compute homography between testudo-corners and tag corners from the frame.
10) The image will be warped given the above matrix, destination shape i.e we are performing forward warping.
11) Forward warping results in noise over the warped image, in order to compensate that i have used the morphological operation i.e closing to remove the noise.
12) To Superimpose the testudo , I have created a blank frame with the tag corners using drawContours method from opencv.
13) I have used bitwise-OR between my warped image and blank frame to get the superimposition of testudo.

Finding a homography matrix

To discover the solution to these system of equations, we utilize Singular Value Decomposition to get the homography matrix given four sets of data.

$$\underset{8\times 99\times 1}{A\ X} = 0$$

The dimension of the homography matrix is 3 x 3.

Fig. 6: Super Imposing the testudo to the tag

The eigen vectors of $A.AT$ make up the matrix, while the eigen vectors of $A.T.A$ make up matrix V and Σ matrix consists of singular values of $A.A^T$ or $A^T.A$ based upon the size of (m,n).

U - Left Singular Vectors of A V - Right Singular Vectors of A Σ - Non-zero singular values of A (found on the diagonal entries) are the square roots of the non-zero eigenvalues of both $A.A^T$ and $A^T.A$.

The singular values should be in decreasing order.

We find the eigen vectors of $(A.A^T - \lambda * I) = 0$ or $(A^T.A - \lambda * I) = 0$.For finding out the Σ we find the eigen values and take a square root of the values and insert in the diagonal of the matrix.

After calculating the SVD matrix from the python code, we extract the eigen vector of V which is having the smallest singular value.This is the solution to the equations or the Homography matrix.

To reduce the number of parameters from 9 to 8 , we divide the last element with our matrix which will result in less number of equations.

I have faced issues when superimposing the testudo onto the tag as some of my corner points are not accurate which is resulting in shrinking and shifting my image. I have increased my fft mask to 240 and increased the number of pixels in the goodfeaturestotrack to 50.

B. PROJECTING A CUBE ONTO TAG

In this section we are going to project a 3-dimensional cube onto the tag using projection matrix.

Fig. 7: Projecting Cube onto the Image

Below are the steps to project the cube onto the tag for a single frame which can be extended to all the frames in the sequence.

1) We get our corner points of our tag as similar to the above process.
2) I have selected my cube dimensions as 160 where I have used these 2d points first to superimpose my cube onto the tag.
3) First, we find the homography between the cube 2d points and tag corners
4) Then we find projection matrix using the homography matrix and camera intrinsic parameters(K).
5) The projection matrix will be a 3 x 4 matrix , we use the projected matrix and multiply with the (x,y,z,1) of our cube 3d coordinates in order to get x,y,z which is a 3 x 8 matrix.
6) We scale them to x/z and y/z to get a 2 x 8 array which contains the our new x,y coordinates in the projected space.
7) After getting the coordinates we then draw lines between these points using cv2.line to draw our cube onto the tag.

Finding the Projective Matrix

Intrinsic Matrix Parameters of the camera - K Projection Matrix Parameters of the camera - P Rotation Matrix Parameters - R Translation Vector Parameters - T

$P = K[R|T]$
$B = K^(-1) * H$

1) First we find the homography matrix using the corresponding points from the corners and the cube dimensions
2) We find the B_hat by taking the dot product of the Homography matrix and the inverse of Intrinsic matrix (K).
3) After computing B_{hat} , we then find the$lambda(\lambda) - (2/norm(b1) + norm(b2))B = [b1b2b3]$scale factor which is the first two column vectors of the B_{hat}
4) $B_hat = [\lambda*b1\lambda*b2\lambda*b3]$ which is equivalent to [r1,r2,t]-rotational vectors and translation vectors.
5) To get r3 we take cross product of r1 and r2.
6) Finally, we take a dot product of the $camera - matrix(K)and[R|t]$to get our projection matrix.

Fig. 8: Input Image for Edge Detection

C. Dexined vs CANNY Edge Detection

Canny edge detection only focuses on changes present locally, but it doesn't have a semantic understanding (context of the image). This limits the accuracy and fails to capture edges which appear on local contexts.

Semantic understanding is imperative for edges detection and to obtain this, the deep learning models like CNN learn them as a part of their features and update their edge detecting kernels according to the context/semantic of the picture.

I have tested the above edge detection methods on the above figure 8 and the output from the dexined and canny can be seen in figure 9 and 10.

A state-of-the-art technique called holistically nested edge detection or HED is a learning-based end-to-end edge detection system that uses a trimmed VGG-like convolutional neural network for an image-to-image prediction task. This was an enhancement over using a basic deep learning model.

But these architectures have an inherent drawback. As the number of layers in the deep learning framework grow, the issue of vanishing gradient arises, and the features detected at very deep levels are often not proving as much impact as expected.

This is a common problem in image semantic segmentation. To overcome this issue, the ideas of skip connections and upscaling were introduced into the well-known UNet semantic segmentation model.

DexiNed - Dense Extreme Inception Network for Edge Detection is a state-of-the-art edge detection deep learning model which utilizes these concepts for creating an architecture that caters to just edge detection? Apart from the normal convolutions, pooling and batch normalization, there are parallel skip connections implemented with upscaling.

DexiNed can be split into two parts:

Dexi – Acting like the encoder with skip connections Skip connections ensure that the features at one level are sent as inputs to the layers at much higher depth hence ensuring mitigation of the vanishing gradient issue.

And;

USNet – Upsampling Network

They are a sequence of one convolutional and deconvolutional layer that up-sample features on each pass.

Another important differentiation between Dexined and other Deeplearning Edge detectors is that DexiNed generates thin edges. This is due to the three strategies of upsampling: bi-linear interpolation, sub-pixel convolution and transpose convolution.

Training was performed on BIPED Dataset. Testing benchmark is on MDBD, BSDS500, BSDS300, NYUD, and PASCAL-CONTEXT.

The output of Dexined is far better than the canny as it refines the edges and makes the line smoother.

Please refer to this github repository for the above code base.

$https://github.com/karanamrahul/AR - tag.git$

REFERENCES

[1] ENPM 673, Robotics Perception Theory behind Homography Estimation Document
[2] https://ieeexplore.ieee.org/document/9093290
[3] https://github.com/bnsreenu/pythonformicroscopists

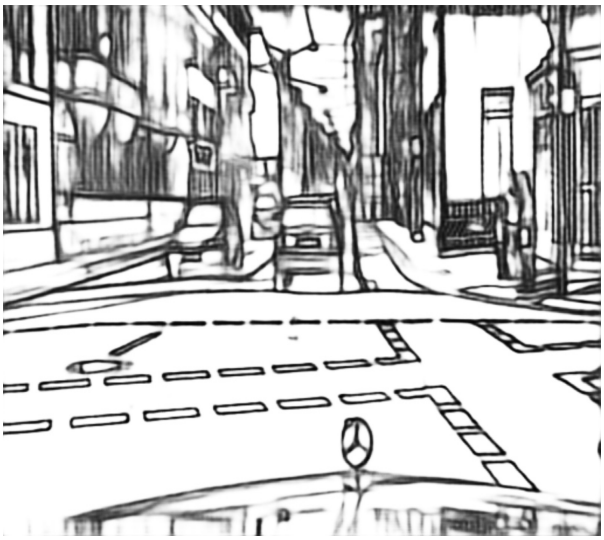

Fig. 9: Edge Detection using Dexined

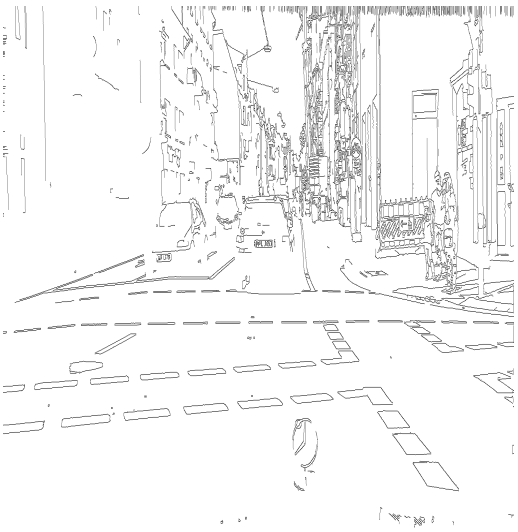

Fig. 10: Edge Detection using Canny