ENPM673 Project 1

Justin Albrecht and Brian Bock

Spring 2020

Table of Contents

Note

In an effort to reduce the propagation of problematic sexualized objectification of women in the world of computer science[1] [2], we elected to replace the given stock image "Lena" with photos of our dogs (Figure 1). In addition, this makes our project a bit more unique, and we hope these photos bring you even a fraction of the joy that they bring us.

(a) Tucker

(b) Hailey

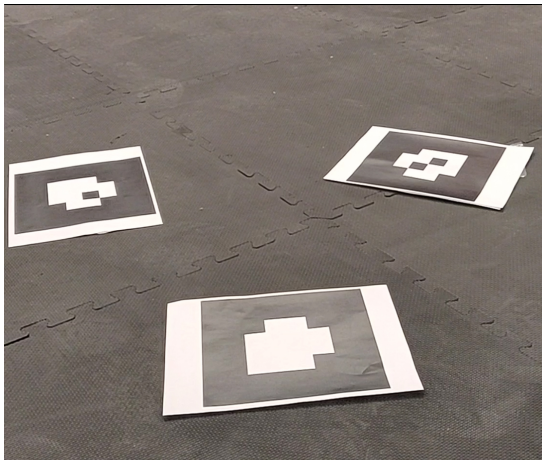
(c) Tessa

Figure 1: The dogs used as stock images

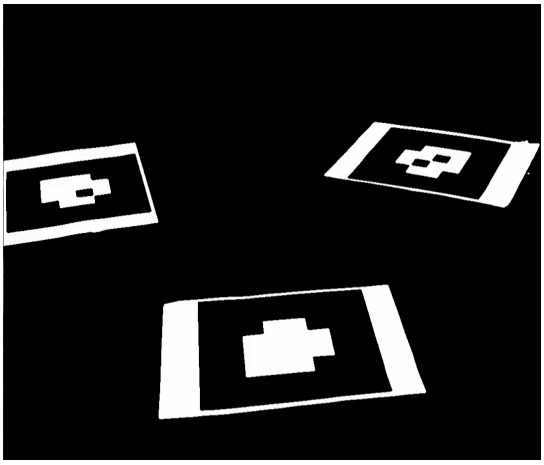
1 Corner Detection

1.1 Initial Filtering

We evaluated the video frame by frame, treating each frame as an individual static image. We first converted the frame to a greyscale color space. We experimentally determined the ideal grayscale threshold that made just the AR tag clearly visible in the frame and eliminated all undesired image elements (Figure 2). You can view a threshed version of the video at https://youtu.be/HG7GqCJOxOs. If you watch this video closely, you'll see some of the motion blur artifacts that made some frames more challenging to analyze.



(a) Original Video Frame



(b) Binary frame threshed from gray scale

Figure 2: Example photo showing the result of a binary threshing

[1] https://en.wikipedia.org/wiki/Lenna#Criticism
[2] https://www.theatlantic.com/technology/archive/2016/02/lena-image-processing-playboy/461970/

1.2 Contour Detection and Filtering

On the binary image we call `cv2.findContours`. This function returns both the list of all contours in the image as well as their hierarchical relationship. Using the hierarchical relationship, we filter out any contours that do not have a parent or child contour. By further filtering these contours so that only the largest 3 remain, we reduce most noise and erroneous contours. This leaves us with only the contours that are associated with the black region of the AR tag. Figure 3 shows, in green, all of the contours that are found in the image, and, in blue, only the contours that remain after filtering.

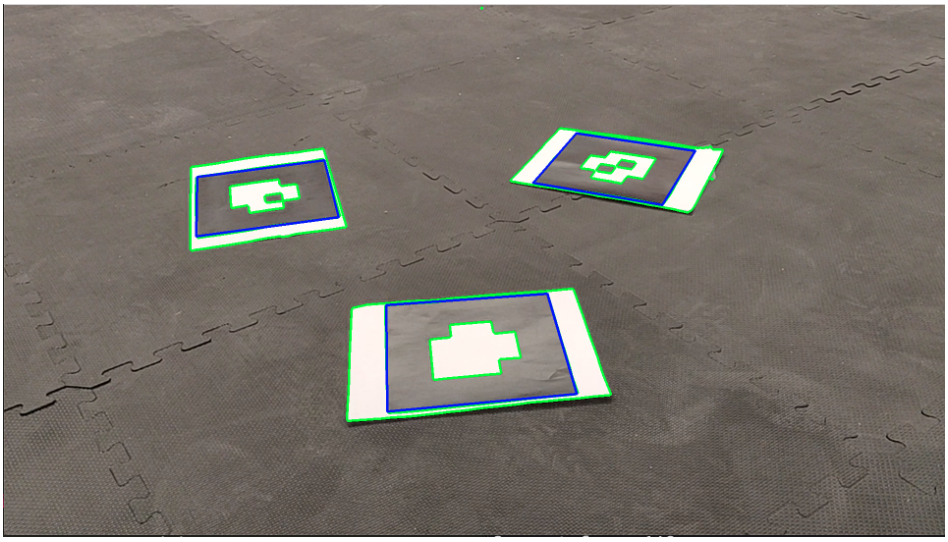


Figure 3: Video frame with contours shown

1.3 Getting Corners

After the contours have been filtered so that only the edges of the black region of the tag remain we call the our function `approx_quad` on the contour. This takes in the contour and tries to approximate the region as a quadrilateral.

Using the function `approxPolyDP` and filtering for only polygons that have four sides we can ensure that we have found the corners of the tag.

We store the corners in a python list of lists called corners, where the length of corners will be equal to the number of detected tags a given frame. Inside each element of corners is a list of four points that correspond to the corners. `approxPolyDP` orders the corners so that they are arranged counter-clockwise, but the starting corner may change from frame to frame.

2 Decoding

After we have defined the corners of the black region of the tag we need to use homography to warp the image from the camera coordinate frame into a square coordinate frame that will allow us to decode the tag.

2.1 Homography Computation

To find a homography matrix that will allow us to convert between our camera cordinates and the square coordinates we need eight points. Four of the points are the corners of the tag that we have from detection. The other four points are the corners of our square destination image. The dimensions of the destination are somewhat arbitrary but if it is too large we will start to have holes, since there are not enough pixels in the source image, and if it is too small we will have a lower resolution and will start to lose information. To fix this we found the number of pixels that were contained within the four corners of the tag and took the square root of that number to get an ideal dimension dim. The four corners are then [(0,0), (0, dim), (dim, dim), (dim, 0)]

Now that we have the eight points we can generate the A matrix below:

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 * xp_1 & y_1 * xp_1 & xp1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 * yp_1 & y_1 * yp_1 & yp1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2 * xp_2 & y_2 * xp_2 & xp2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 * yp_2 & y_2 * yp_2 & yp2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3 * xp_3 & y_3 * xp_3 & xp3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3 * yp_3 & y_3 * yp_3 & yp3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4 * xp_4 & y_4 * xp_4 & xp4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4 * yp_4 & y_4 * yp_4 & yp4 \end{bmatrix} \tag{2.1}$$

From this A matrix we can solve for the vector h, which is the solution to:

$$Ah = 0 \tag{2.2}$$

By reshaping our vector h, we can find the homography matrix:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \tag{2.3}$$

To solve for the vector h, we can use Singular Value Decomposition or SVD. This will decompose our A matrix into three matrices, U, Σ, and V.

To perform the decomposition the following steps need to be followed:

1. Find eigenvectors (w_i) and eigenvalues (λ_i) for $A^T A$.
2. Sort λ_i from largest to smallest
3. Compose V as an $n \times n$ matrix such that the columns are w_i in the order of sorted λ_i
4. Create the list of singular values (σ_i) from the square root of the non-zero λ_i. $\sigma_i = \sqrt{\lambda_i}$
5. Compose Σ as an $m \times n$ matrix where the ith diagonal element is σ_i and all other elements are zero.
6. Use the relationship $Av_i = \sigma_i u_i$ to solve for the columns of U.

Note:
v_i is the columns of V
u_i is the columns of U

Using the steps above, we can decompose A into U, Σ, V.

After this decomposition we know that our h vector can be approximated by the column of V that corresponds with the small singular value in Σ. Since the singular values are ordered from largest to smallest, h can be approximated as the last column of V

$$h = V[:, -1] \tag{2.4}$$

2.2 Creating the Square Image

After we have computed the homography matrix we need to generate a new set of points in the square frame, and then create a new image using the pixels from source image but in their new location. For any point in the camera coordinates X^c (x,y), we can compute that point in the square coordinates X^s (x',y'), using the following equations:

$$sX^s = \begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.5}$$

$$x' = \frac{sX^s[0]}{sX^s[2]} \tag{2.6}$$

$$y' = \frac{sX^s[1]}{sX^s[2]} \tag{2.7}$$

We can generate new points for all (x,y) in a given frame. The points generated are floats with multiple decimal places. There are several possible approaches to remedy this, including averaging the pixel values around the closest four pixels, picking the closest neighbor, or even simply truncating the floating point value as an integer. We decided to use this last approach for its simplicity and because the grid size of the tags is so large compared to any pixel that it this loss of resolution does not affect the decoding of the tag.

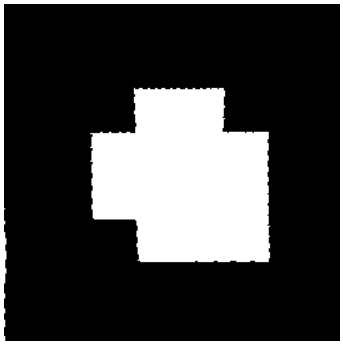
It is also important to note that any points outside of the boundaries of the corners of the tag will be outside of the dimensions of our destination image. We can generate a new image by taking only the points that lie in the range of our destination image and reconstruct an image pixel by pixel in the new coordinates. The resulting images looks like this (Figure 4):

(a) Tag 1 (b) Tag 2 (c) Tag 3

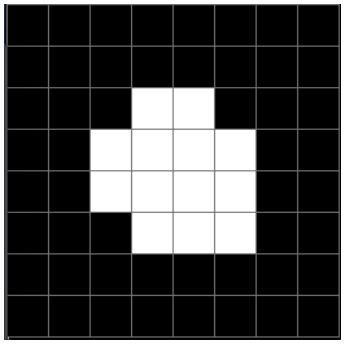
Figure 4: View of each tag after homography has been performed

2.3 Decoding the Tag into a Binary Matrix

Now that we have each of the tags as a square image we can start the process of decoding them into a binary matrix. We know that the tag is made up of an 8x8 grid so we divide the square into 64 sectors and create a new matrix that is 0 if that sector is black and 1 if that sector is white. In order to determine if the sector is black or white, we threshold the square image using an experimentally determined value into a binary image. We then take the mean of that sector and if the mean is closer to 0, then the value for that sector is 0 and if the mean is closer to 255, the value for that sector is 1.



(a) Threshed Tag



(b) Decoded Tag

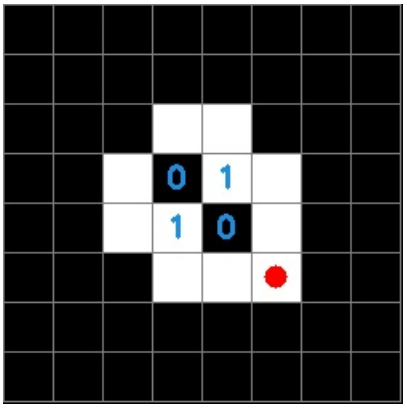
Figure 5: (a) is the square image after applying the threshold, (b) shows the 64 sectors as either black or white depending on the mean of that sector

The process above gives the example 8x8 binary matrix shown below (Equation 2.8)

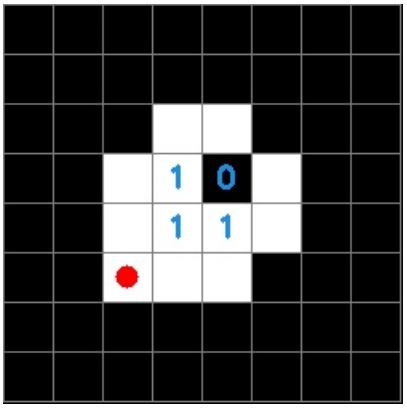
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

2.4 Decoding the Orientation and Tag ID

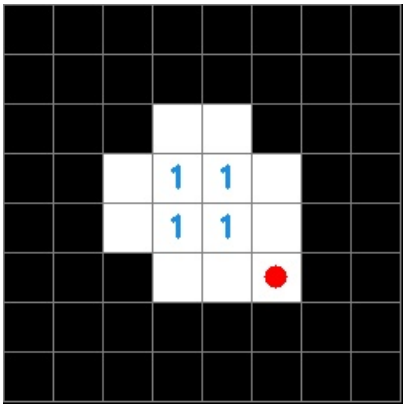
Now that we have a data structure for the information on the tag it is possible to determine both the orientation of the tag and the tag ID. The orientation is determined by a white square in any of the zero indexed positions [2,2], [5,2], [5,5], [2,5]. The ID is then derived from the inner four sectors of the tag and is composed of a four character string made up of ones and zeros. The order of characters is counter-clockwise and the first character is defined by the sector opposite the corner that defines the orientation. The images below show a red circle in the sector that defines the rotation and the resulting tag ID.



(a) Tag 1: '0101'



(b) Tag 2: '0111'



(c) Tag 3: '1111'

Figure 6: Representation of the decoded tag's orientation and ID

To determine the tag ID, we look the inner 4 elements of our square binary matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.9}$$

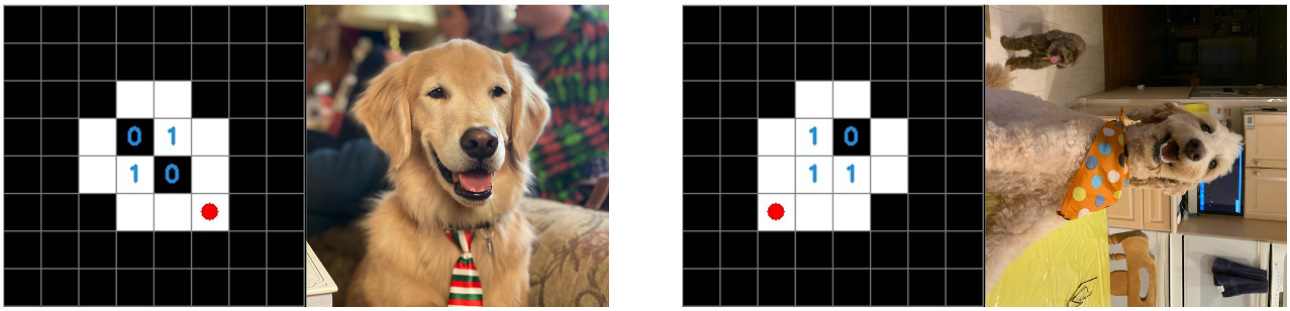
This is read counterclockwise with the start position determined by the orientation. In this example, the tag ID would be 0111.

3 Superposing an image

After we have determined the position, orientation, and ID of all the tags in a given frame we can then superpose the tag with our images of dog. The steps to do this are as follows:

1. **Match the tag ID to an image**
   The IDs of the three tags and their corresponding photos are hard coded at the start of the program. We simply check the tag ID string to see if it matches any of the existing tags and assign the new image to be it's corresponding dog image. It would be trivial to add additional tags and photos to this system.

2. **Rotate the image**
   The function we use to decode a tag returns a number of rotations necessary to match the rotation of the tag. This number is either 0,1,2,3. We can then apply either no rotation or a multiple of 90° to the dog image using the function `cv2.rotate()` (Figure 7).



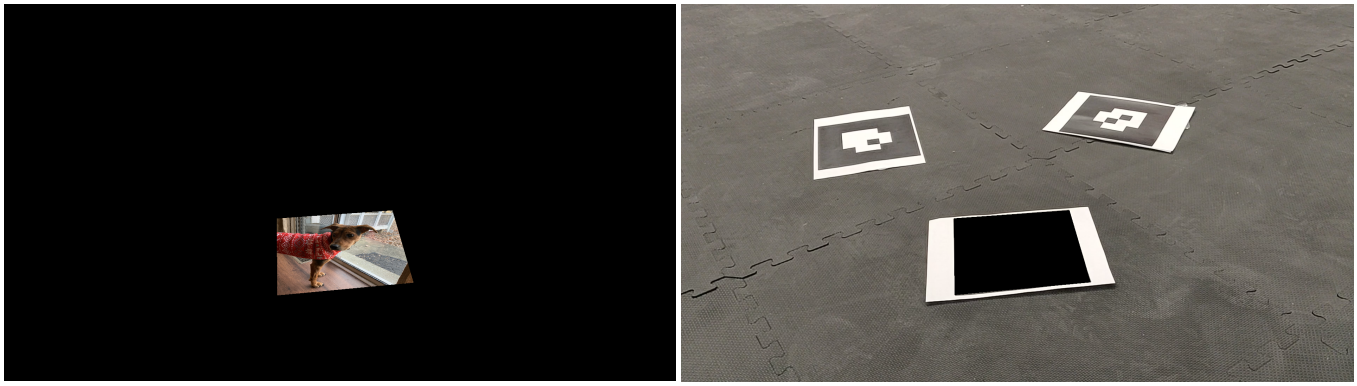
(a) Upright Image

(b) Right turned image

Figure 7: Example showing the rotation of an image based on AR tag orientation

3. **Use homography to generate a new image**
   We then use the same process as above to generate a warped version of our dog image. In this case however instead of generating a square image of arbitrary size, we instead create an image that is the same size as the original frame. When we generate our new homography matrix we need to use the corners of the frame in addition to the corners of the tag. We can then follow the same process as in equation: 2.5, to generate a new image that is the same size as our frame but has a warped image of the dog and is black everywhere outside of the frame (Figure 8a).

4. **Blank the original region of the tag**
   In order to perform the next step we need to blank the region of the image where the tag is. We can achieve this by drawing a filled black contour in the region defined by the corners of the tag. The output of drawing this contour is show in Figure 8b

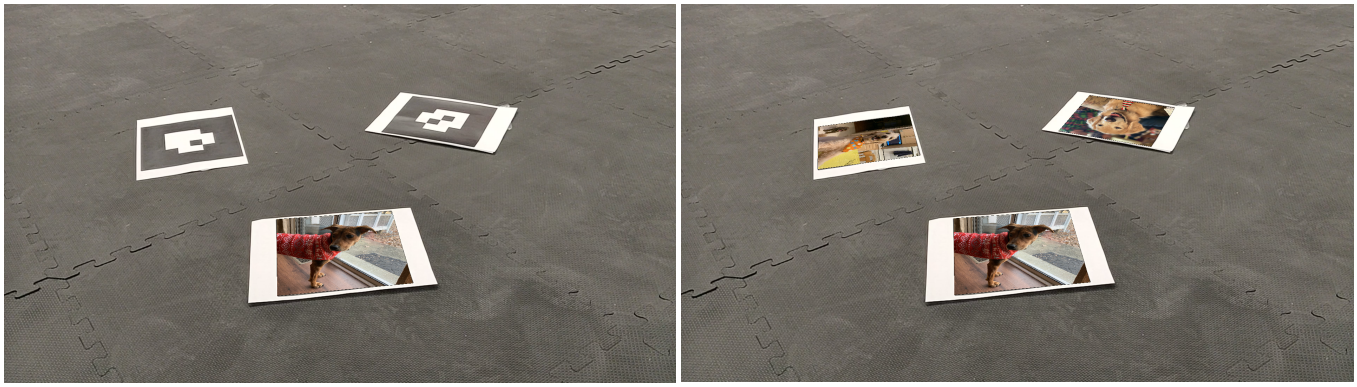


(a) Warped dog picture in location of tag.

(b) Tags replaced with black region

Figure 8: Two images that will be added together

5. **Add the new image to the existing frame**
   Now that we have the image generated from homography and the original frame with the blank region we can use the function `cv2.bitwise_or()` to add them together. Since the first image (Figure 8a) is black (0) everywhere except the tag and the second image (Figure 8b) is black (0) only in the tag region, the output is the dog superposed onto the tag (Figure 9a). Repeating this for all three tags yields Figure 9b.



(a) One tag replaced with a dog image

(b) All tags replaced with dog images

Figure 9: Output of the superposition

4 Superposing a Cube

The coordinates of the corners of our cube, in the world frame, are:

$$\text{Corners}_w = \begin{bmatrix} x_1, & y_1, & 0 \\ x_2, & y_2, & 0 \\ x_3, & y_3, & 0 \\ x_4, & y_4, & 0 \\ x_1, & y_1, & -d \\ x_2, & y_2, & -d \\ x_3, & y_3, & -d \\ x_4, & y_4, & -d \end{bmatrix} \tag{4.1}$$

where d is the height of the cube. The base of the cube is flush with the floor plane of the world frame, so the first four points have zero $\hat{Z}$ component. The last four points bound the top face of the cube, and exist d away from the floor plane. Our task is to find these points in the image frame so that they can be displayed in the image.

At the start, all we know are the coordinates of our tag corners in the camera frame:

$$\text{Corners}_c = \begin{bmatrix} x_{c_1}, & y_{c_1} \\ x_{c_2}, & y_{c_2} \\ x_{c_3}, & y_{c_3} \\ x_{c_4}, & y_{c_4} \end{bmatrix} \tag{4.2}$$

as well as the camera intrinsic matrix:

$$K = \begin{bmatrix} 1406.084 & 0 & 0 \\ 2.207 & 1417.999 & 0 \\ 1014.136 & 566.348 & 1 \end{bmatrix} \tag{4.3}$$

We start by computing the homography transform (H) that converts our points from the image frame (skewed) to the world frame (square). This is the same computation done earlier to make our tags square for identification. We use this to convert the tag corners (the base of our cube-to-be) from the image frame to the world frame. This gets us the first four points of Equation 4.1. The other four corners (the top face of our cube) are those same points just shifted by $-d$. In the world frame, the Z axis points into the floor, so this negative is necessary to make the cube exist above the floor plane. We now need to convert the coordinates from the camera frame back to the world frame. To do this, we'll need the Projection Matrix (P).

4.1 Finding the Projection Matrix

$$P = K \begin{bmatrix} R & | & t \end{bmatrix} \tag{4.4}$$

To get P, we need to evaluate all of the components and sub-components that it is comprised of.

$$H = K\tilde{B} \tag{4.5}$$

$$B = \widetilde{B}(-1)^{|\widetilde{B}|<0} \tag{4.6}$$

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \tag{4.7}$$

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \tag{4.8}$$

$$\lambda = \left(\frac{||K^{-1}h_1|| + ||K^{-1}h_2||}{2} \right)^{-1} \tag{4.9}$$

$$r_1 = \lambda b_1 \tag{4.10}$$

$$r_2 = \lambda b_2 \tag{4.11}$$

$$r_3 = r_1 \times r_2 \tag{4.12}$$

$$t = \lambda b_3 \tag{4.13}$$

$$R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix} \tag{4.14}$$

4.2 Generating Top Corner Points

We then use the projection matrix P (Equation 4.4) to convert these new points from the world frame to the image frame, where they can be superposed on the image. To do this, we start with our homogeneous coordinates for the bottom corners, X_{c1}, in the camera frame:

$$X_{c1} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.15}$$

These become scaled homogeneous coordinates in the world frame:

$$sX_s = H \cdot X_{c1} \tag{4.16}$$

$$sX_s = \begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} \tag{4.17}$$

We need to normalize these so the last row is 1:

$$X_s = \frac{sX_s}{sX_s[2]} \tag{4.18}$$

Where $sX_s[2]$ is the 3rd (0 index) row of sX_s

$$X_w = \begin{bmatrix} X_s[0] \\ X_s[1] \\ -d \\ 1 \end{bmatrix} \tag{4.19}$$

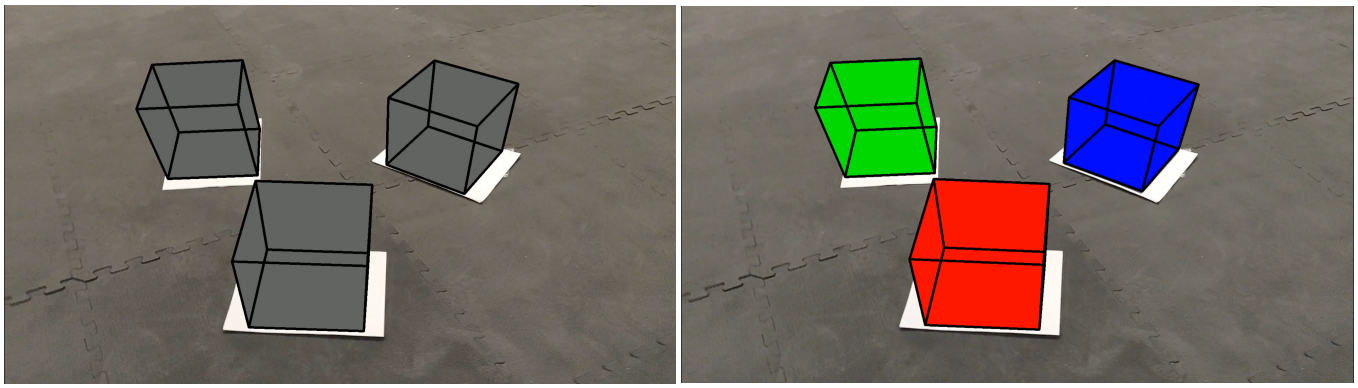
Now that we have shifted the points in z in the world frame we need to use the projection matrix P to get the top corners X_{c2} back in camera coordinates.

$$sX_{c2} = P \cdot X_w \tag{4.20}$$

$$X_{c2} = \frac{sX_{c2}}{sX_{c2}[2]} \tag{4.21}$$

4.3 Drawing Lines and Contours

Once we have the corner points, we use `cv2.drawContours` and `cv2.line` to create first the faces and then the edges of our cubes (Figure 10a). This creates the shaded wire-frame aesthetic that characterizes our project. Cube faces can also be colored to correspond to their unique tag IDs (Figure 10b).



(a) Cubes rendered on the tags

(b) Colored cubes rendered on their specific tags

Figure 10: Monochromatic and ID-specific colored cubes rendered on their tags

5 Smoothing

Due to noise and blur in the video, the base of our cubes (and then also their top points), shift considerably from frame to frame. This makes the cubes appear very jittery. You can see an example of this here: https://youtu.be/S5xvTcTTBsw
To combat this, we implemented inter-frame smoothing.

5.1 Initial Efforts - Frame Interpolation

We initially attempted to smooth the video by interpolating additional frames to create a video with a higher effective framerate. We took two frames, and created 10 blended frames, each 10% more of the next frame and 10% less of the previous frame. In this way, each pair of the original video frames became 12 new frames (the original, 90%O+10%N, 80%O+20%N ..., 10%O+90%N, and the next). This produced smoother video motion, but didn't aid in the cube jitter issue.

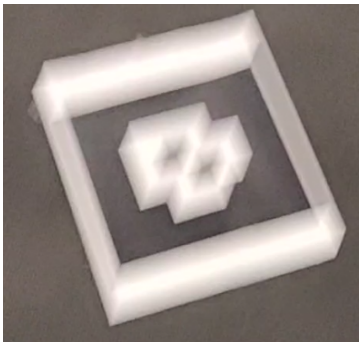
```
for q in range(1,10):
            a=.1*q
            b=1-a
            half_frame = cv2.addWeighted(frame_array[i], b, frame_array[i+1], a, 0)
            # add the half frame to the array
            new_frame_array.append(half_frame)
```

5.2 Improved Smoothing - Inter Frame Averaging

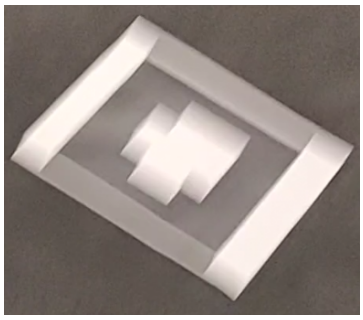
We then moved to our final smoothing solution. Unlike in the original videos, which processed the video frame by frame as it was read from the file, this attempt started by reading in the entire video into an array of frames. This resolved a tricky issue with `video.set()` which does not permit backtracking. We later abandoned this resource intensive approach for a simpler one - We now read in the first few frames (however many are needed to get started) and then read in only the "frontier" frame for each successive iteration.
Using a look-ahead and look-behind method, we average the corner coordinates for the top and bottom of the cube with their positions over several previous and post frames. The computed average for each corner is used as the points for that frame. This process is repeated for all subsequent frames. As a check against undesired cube behavior, we examine this average and compare it with the original position. If the average is too far away from the original position, we assume there was some error (the wrong point averaged due to excessive motion/blur), and the original position is used instead. This smoothing function works for an arbitrary number of preceding and post frames, allowing for varying amounts of smoothing. The more frames that are averaged results in a smoother, less jittery cube, but may cause cube drift, where the cube position slides slightly from the tag due to camera movement in neighboring frames. Cube drift becomes more noticeable as the frame count increases. See the 15 frame smoothed videos in Section 7 for examples of this.

6 Known Issues

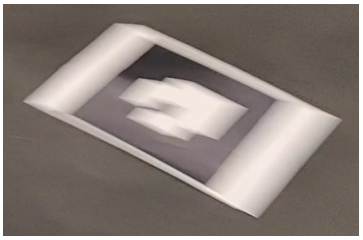
There are a few minor known issues with our implementation. In some frames, the motion blur is so extreme that it becomes impossible to distinguish between the AR tag and the paper (Figure 11). This causes the tag to be dropped for that frame, and it's cube or dog will disappear.
Similarly, we experience issues where the extreme motion blur obfuscates the key regions of the AR tag, and we fail to identify it properly (Figures 11a, 11c). This results in either a dropped tag, or the wrong dog image displayed.
In some frames, some portion of an AR tag is off screen. Since the tag doesn't have all four corners visible to the camera, it is not recognized and dropped.



(a) Example 1



(b) Example 2



(c) Example 3

Figure 11: Some examples of extreme motion blur

7 Videos

7.1 1 Tag

Dog: https://youtu.be/eMDw7v8iKVM
Dog and Contours: https://youtu.be/u8Eu2P4GyU8
Cube: https://youtu.be/832ytyWuLog
Cube and Contours: https://youtu.be/AOtB_EfQM8M
Cube and Dog: https://youtu.be/XJA6ZihT7dM
Cube, Dog, and Contours: https://youtu.be/5Ye-M5kQ49s

7.2 1 Tag - Smoothed

Cube (4 frames, with color): https://youtu.be/e78oBha6Y-k
Cube (5 frames): https://youtu.be/ptrkv0mcJ-o
Cube (15 frames): https://youtu.be/4-SMcVtxhTY

7.3 2 Tags

Dogs: https://youtu.be/W07xNjdikj4
Dogs and Contours: https://youtu.be/MZroDtzSx2U
Cubes: https://youtu.be/qDmahYQLtnI
Cubes and Contours: https://youtu.be/Y0AVKwwF5Rs
Cubes and Dogs: https://youtu.be/G0JwnAY8btk
Cubes, Dogs, and Contours: https://youtu.be/SjvsdsYbnNc

7.4 2 Tags - Smoothed

Cubes (4 frames, with color): https://youtu.be/LD1vFiq3ZNg
Cubes: (5 frames) https://youtu.be/WOoJEzgwbFA

7.5 3 Tags

Dogs: https://youtu.be/6kUJMu_lp6s
Dog and Contours: https://youtu.be/hTnjIdVy1As
Cubes: https://youtu.be/S5xvTcTTBsw
Cubes and Contours: https://youtu.be/pob9dMti9o0
Cubes and Dogs: https://youtu.be/ERwL77zEsw8
Cubes, Dogs, and Contours: https://youtu.be/j1lANCtlTnI

7.6 3 Tags - Smoothed

Cubes (4 Frames, with color): https://youtu.be/bwa-wyenpoI
Cubes (5 Frames): https://youtu.be/UzQukBC6EkI
Cubes (15 Frames): https://youtu.be/ZHo5AcK5vZE

8 Code

You can view the files related to this project at: https://github.com/jaybrecht/ENPM673-Project1