Gebze Technical University

Department of Computer Engineering

CSE 463

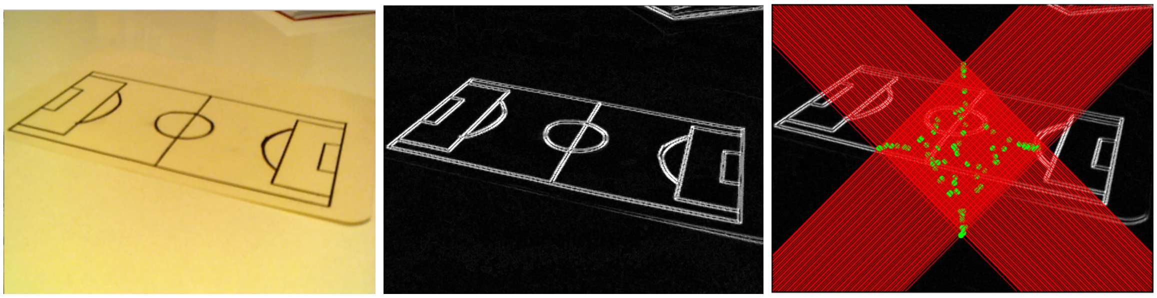
(Introduction to) Computer Vision

Spring 2020

HW1

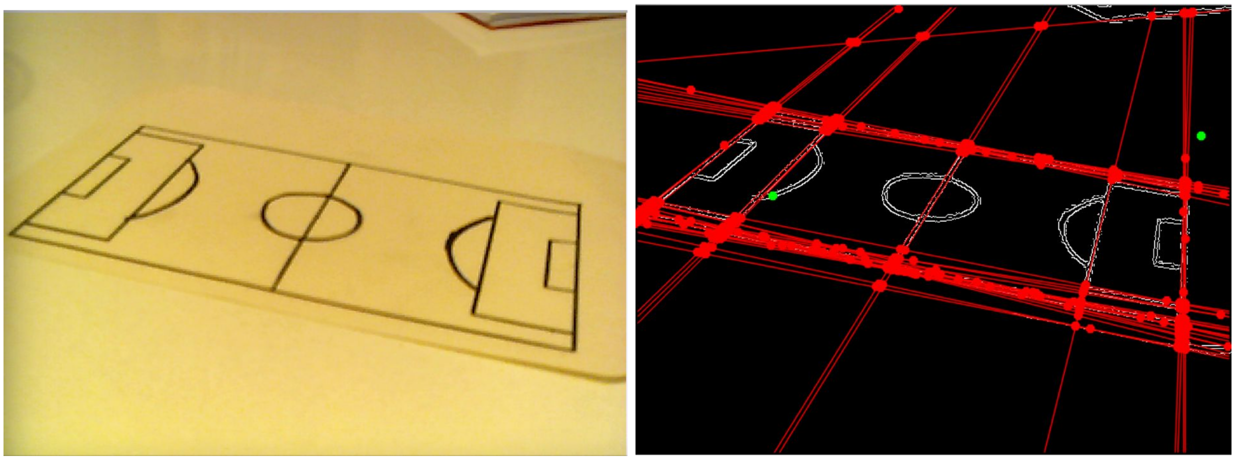
1- Finding Main Dominant Lines

Before applying the Hough Line Transform(HLT), first an edge detection preprocessing is recommended. In the beginning, I experimented with the Sobel using the cv.Sobel() [1].



Applying gaussian blur > grayscale > sobel > hough. The HLT doesn't yield a meaningful result because there is too much noise in the image. Small thresholds result in too many lines. Bigger thresholds lessen the lines but we lose the meaningful lines too. Sobel gives good lines at first glance but creates noise in the image which confuses HLT.

After experimenting with Sobel, I tried for cv.Canny[2] next. According to OpenCV documentation it applies to Sobel also. As I understand Canny solves the noise use during the step of thresholding. Which gives us a good image without noise to send to HLT.



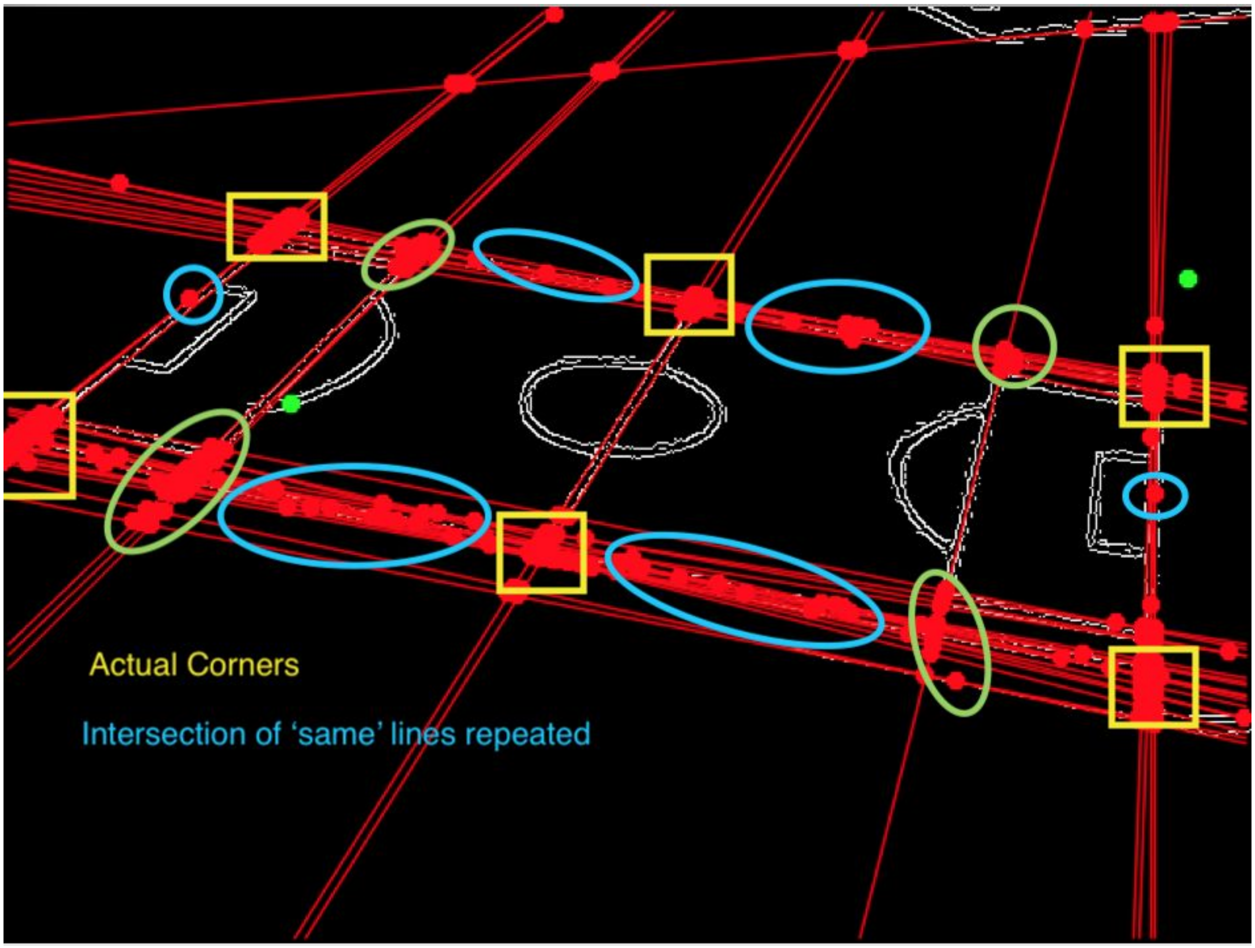
As seen above applying HLT after Canny instead of just Sobel yields better results. We can see the dominant lines better. We can adjust the Hough parameters heuristicly to acquire a less chaotic image. (Green & Red dots are explained later in the report)

2- Finding Intersections and Corners

This was the most enjoyable part of the assignment. Because I had to come up with a clever solution to find the corners without using any OpenCV functions. At first, I wrote a function to check if two lines intersect at some point. Then iterate over all the lines that HLT returned and marked all the intersections using cv.rectangle[3]. You can see the red dots in the image above. They are 4 by 4 squares.

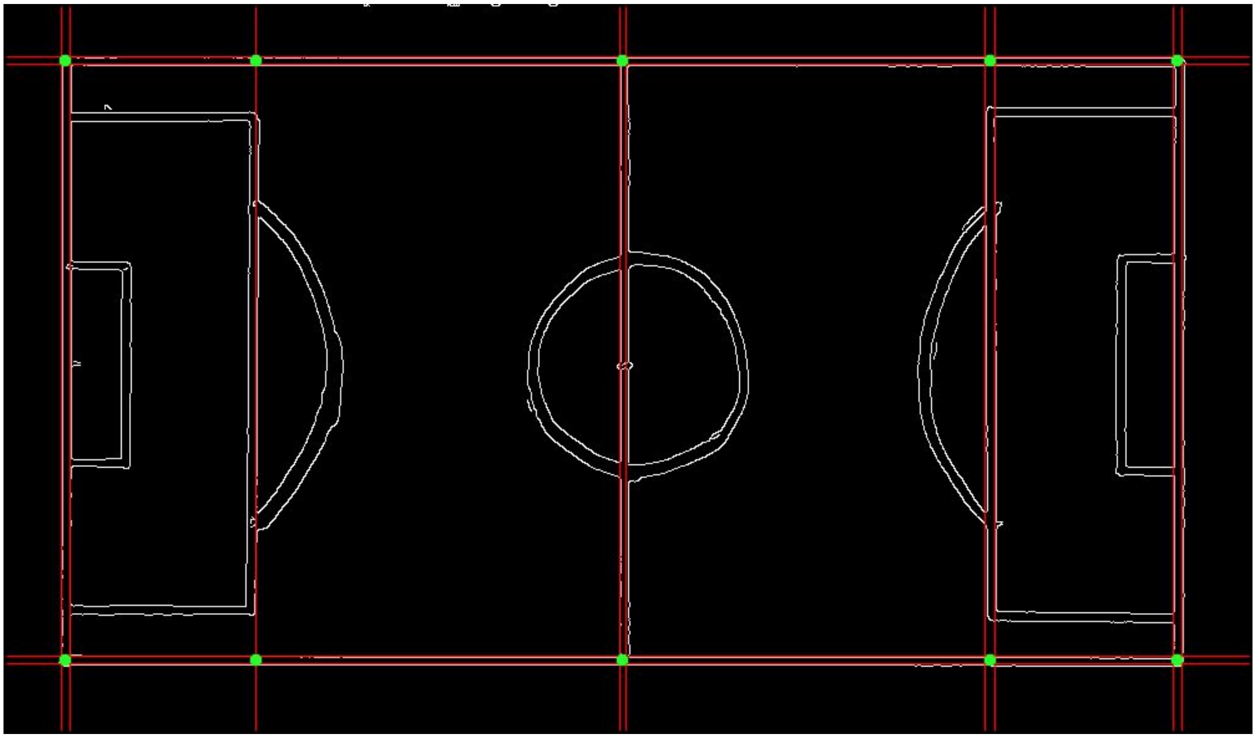
Now we have all the intersection points at hand, but some of them are out of the frame. This is because some of the lines intersect outside of the frame(lines at infinity). At that point I was not sure how to utilize them so I weeded them out and only kept the intersections that I can observe on the frame.

I observed 2 kinds of dominant points of intersections. One of the groups was the actual corners of the field. The other one was the intersection of very close and almost parallel lines.



For my method I only require the points that are marked with the yellow squares, also note that green circled ones can also be useful.

Before proceeding any further, I wanted to perform the same steps on the model image that we will use for calculating our transformation.

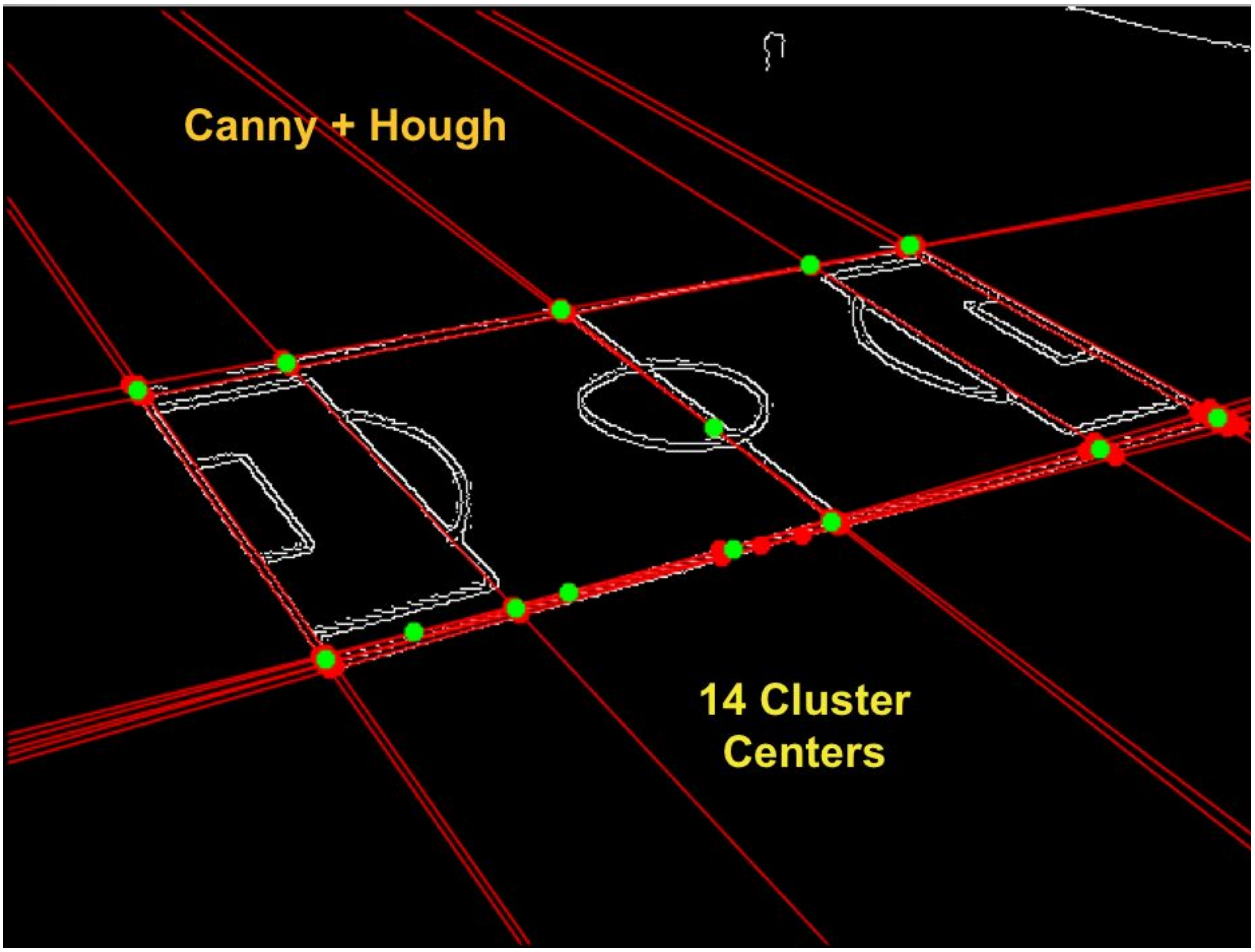


To create this frame, I applied the same procedures upto HLT. But the HLT threshold is almost 1.5 times higher than the live capture frames get applied. HLT parameters are decided heuristicly by trial and error.

2.1- Applying K-Means On Junction Points

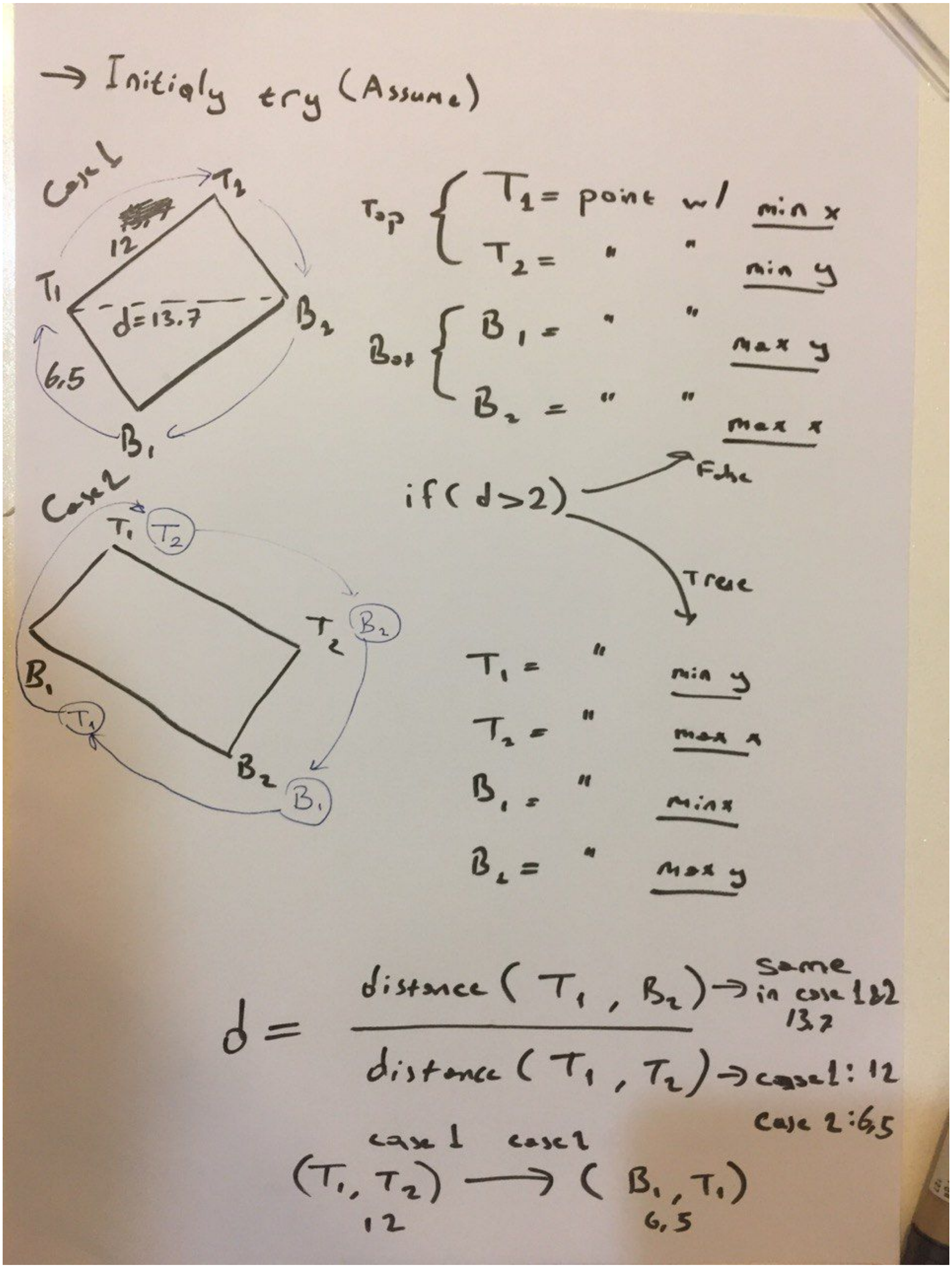
Now to get actual corners, I ran K-Means clustering on **the infinite and the non-infinite points**. For the model image number of clusters is 10, and since the data has no outliers, k-means cluster centers are perfectly placed at the corners.

For the live feed, the number of clusters should be higher than 28 as I observed. N choosen between 28 and 40 yields workable results.



2.2- Determining The Corners

After applying K-Means on all points, only the non-infinite points are kept. Next step is to send these points to the get_frame_corners() function. I'll explain this function by drawing in the following page.


Case 1
T1
T2
12
d=13.7
B2
6,5
B1
Case 2
T1
T2
T2
B2
B1
T1
B2
B1

→ Initialy try (Assume)

Top { T_1 = point w/ min x
T_2 = " " min y

Bot { B_1 = " " max y
B_2 = " " max x

if(d>2) → False

→ True

T_1 = " min y

T_2 = " max x

B_1 = " minx

B_2 = " max y

$$d = \frac{distance(T_1, B_2)}{distance(T_1, T_2)}$$

→ same in case 1&2 13.7

→ case1: 12

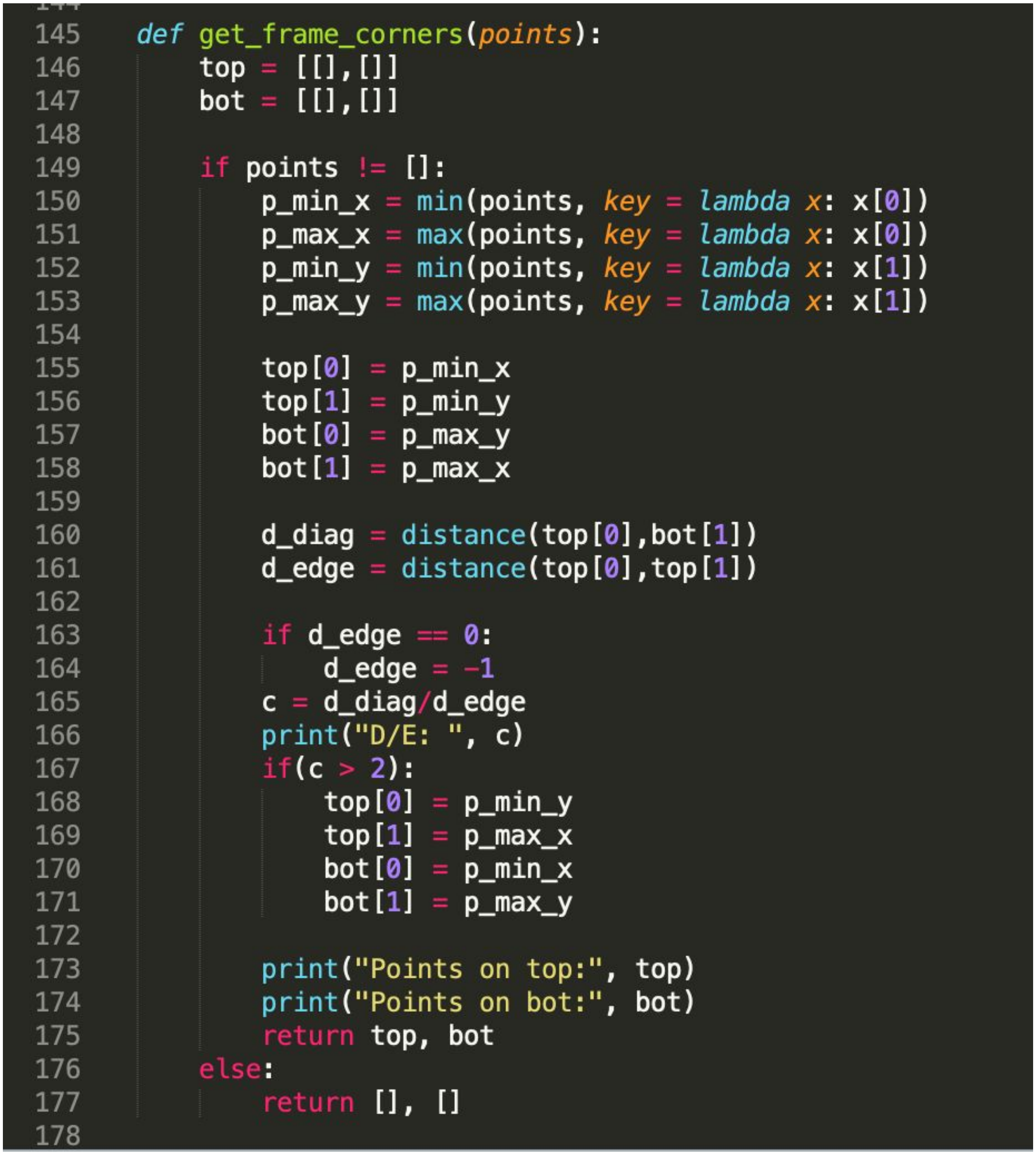
Case 2: 6,5

case 1 case 2

$(T_1, T_2) \longrightarrow (B_1, T_1)$

12 6,5

Python implementation:

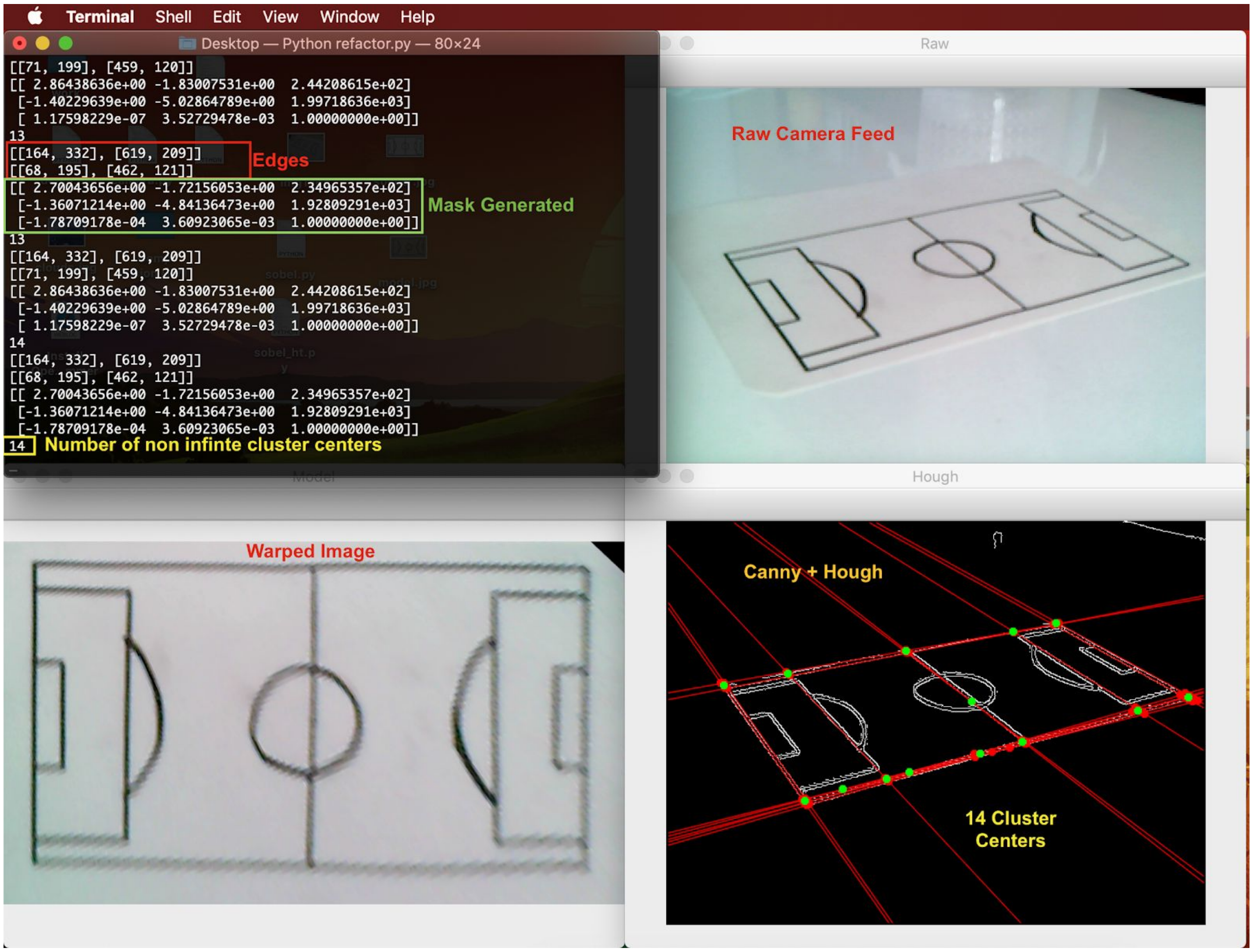


```python
def get_frame_corners(points):
    top = [[],[]]
    bot = [[],[]]

    if points != []:
        p_min_x = min(points, key = lambda x: x[0])
        p_max_x = max(points, key = lambda x: x[0])
        p_min_y = min(points, key = lambda x: x[1])
        p_max_y = max(points, key = lambda x: x[1])

        top[0] = p_min_x
        top[1] = p_min_y
        bot[0] = p_max_y
        bot[1] = p_max_x

        d_diag = distance(top[0],bot[1])
        d_edge = distance(top[0],top[1])

        if d_edge == 0:
            d_edge = -1
        c = d_diag/d_edge
        print("D/E: ", c)
        if(c > 2):
            top[0] = p_min_y
            top[1] = p_max_x
            bot[0] = p_min_x
            bot[1] = p_max_y

        print("Points on top:", top)
        print("Points on bot:", bot)
        return top, bot
    else:
        return [], []
```

3- Estimating Homography and Warping

Now we have 4 corners of the model image and 4 corners that we calculated from the live feed frame. We give them to the cv.findHomography function and retrieve the mask from it.

Then we call cv.warpPerspective() function to apply the mask on our live feed frame and get a perspective transformed image.



```
[[71, 199], [459, 120]]
[[ 2.86438636e+00 -1.83007531e+00  2.44208615e+02]
 [-1.40229639e+00 -5.02864789e+00  1.99718636e+03]
 [ 1.17598229e-07  3.52729478e-03  1.00000000e+00]]
13
[[164, 332], [619, 209]]
[[68, 195], [462, 121]]
[[ 2.70043656e+00 -1.72156053e+00  2.34965357e+02]
 [-1.36071214e+00 -4.84136473e+00  1.92809291e+03]
 [-1.78709178e-04  3.60923065e-03  1.00000000e+00]]
13
[[164, 332], [619, 209]]
[[71, 199], [459, 120]]
[[ 2.86438636e+00 -1.83007531e+00  2.44208615e+02]
 [-1.40229639e+00 -5.02864789e+00  1.99718636e+03]
 [ 1.17598229e-07  3.52729478e-03  1.00000000e+00]]
14
[[164, 332], [619, 209]]
[[68, 195], [462, 121]]
[[ 2.70043656e+00 -1.72156053e+00  2.34965357e+02]
 [-1.36071214e+00 -4.84136473e+00  1.92809291e+03]
 [-1.78709178e-04  3.60923065e-03  1.00000000e+00]]
14
```

4- Final Notes and Extras

4.1- Video Demo

A video showing a run example can be viewed here:
https://www.youtube.com/watch?v=LfU9K9izaSE

4.2- How Can This Be More Efficient

- Using probabilistic HLT will result in higher frame rates.
- An outlier detection procedure can be applied to weed out the odd intersection points, DBSCAN is very fitting for this problem.
- K-Means number of clusters can be determined programmatically.
- HLT & Canny parameters are heuristic parameters, meaning they can be optimized for some situations. There are many methods to do this, Erdoğan Hoca would explain them.

Thanks for reading!