GTU Department of Computer Engineering
CSE 463 - Fall 2024
Homework 1 Report

Emirkan Burak Yılmaz
1901042659

Contents

Introduction

This task is divided into two segments. Initially, we need to identify and implement the homography by utilizing three corners and the point at infinity formed by the intersection of two parallel lines, followed by applying the transformation. In the subsequent part, the objective is to identify the players and draw circles around them, ensuring no overlap occurs between the circles and players. Various approaches were explored to meet these requirements, detailed information of which can be found in the Jupyter notebook titled homography.ipynb

Part 1

The proposed algorithm comprises the following steps:

- Corner Detection
- Extraction of parallel lines and determination of their intersection point at infinity
- Estimating the homography
- Transformation of the source image using the homography

Corner Detection

I have experimented with several approaches to detect the four main corners of the soccer field. Initially, I attempted to utilize **Canny Edge detection** to identify the corners, followed by locating the contours. While this method worked well for images of soccer fields captured from close to a bird's-eye view where all the field lines are clearly visible, it proved less robust for images taken from other angles.

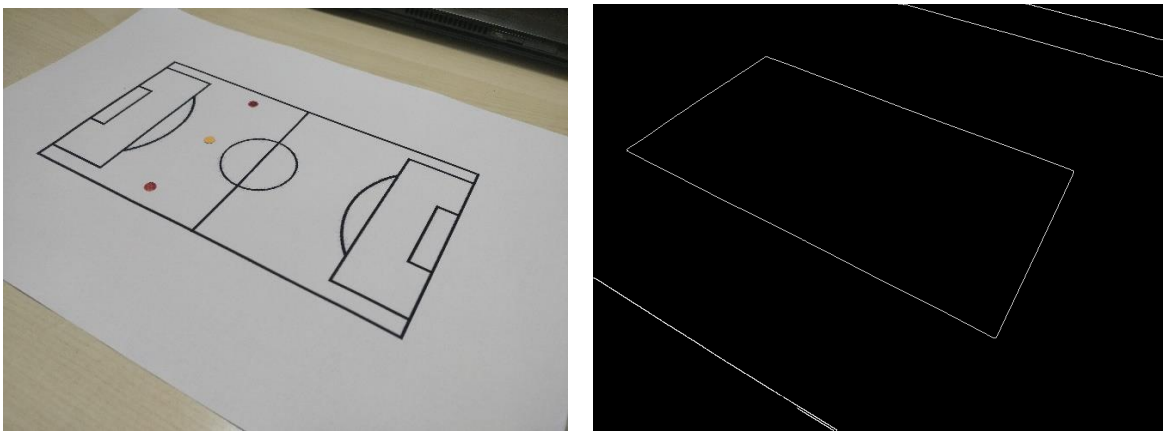


Figure 1: Performing Canny edge detection followed by contour extraction

Another approach involves leveraging the fact that the soccer field comprises many parallel lines. Thus, line detection can be applied to find the intersection points, which represent the corners. Initially, I applied **Canny edge detection** followed by **Hough transformation** to extract line information. While this method effectively identified the corners, it necessitated a selection process to extract the corners specific to the soccer field. Due to this requirement for further refinement, I consider another approaches.

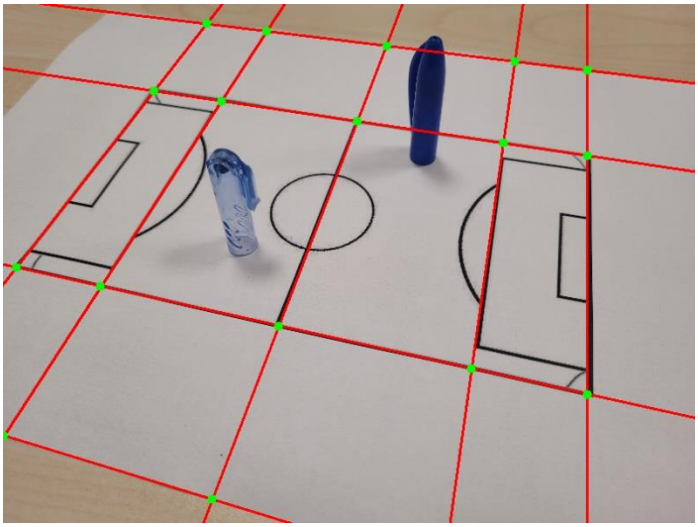


Figure 2: Detecting lines via Hough transform and identifying their intersections as potential corners

I also experimented with the **SIFT** algorithm to detect local features in the images. While it successfully identified key points and matched them, these points didn't necessarily correspond to corners. Since the assignment specifically requires the detection of corners to define two parallel lines, this approach didn't meet the criteria.

Ultimately, I devised a solution where markers were placed on the soccer field image to represent the corners. I selected green and red as distinctive colors on the soccer field and positioned dots at the corners accordingly. I then printed the image with these markers on a piece of paper and detected the red and green dots as corners in the image.

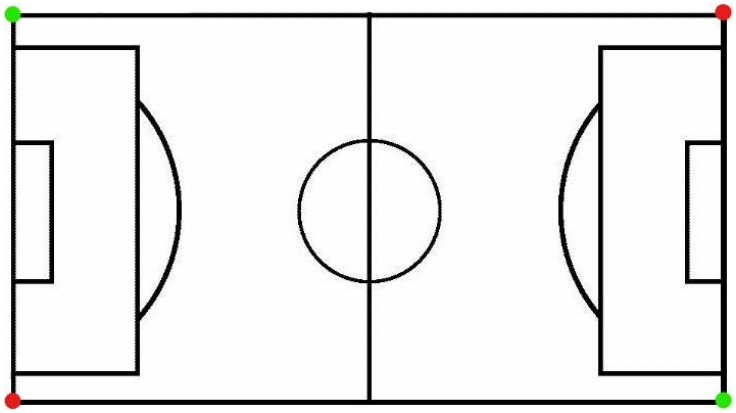


Figure 3: Red and green markers positioned at the corners of the soccer field

By employing marker dots, I can detect and arrange the corners accordingly. Using two distinct colors and positioning them diagonally was sufficient to establish the correct order. The corners are arranged as top-left, top-right, bottom-left, and bottom-right.

Using the Point at Infinity

In mathematics, a projective plane extends the concept of a plane by introducing additional points at infinity, allowing parallel lines to intersect at a single point known as the vanishing point or the point at infinity, contrary to the Euclidean plane where parallel lines do not intersect.

After detecting and ordering the corners of the plane, we can establish the equations of the lines using the top two points and the bottom two points, ensuring that these lines are parallel to each other. By representing these lines in homogeneous coordinates, their point of intersection can be determined using the cross product of their respective line equations. This is because the cross product yields a vector orthogonal to both lines, which represents their intersection point.

Calculation of the Homography

To derive the homography equation, we start by defining a transformation matrix H that maps points from one plane to another.

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 4: Homography equation

```python
def calculate_homography(points1, points2):
    # Constructing the matrix A
    A = []
    for (x, y, z), (x_prime, y_prime, z_prime) in zip(points1, points2):
        A.append([-x, -y, -z, 0, 0, 0, x * x_prime, y * x_prime, z * x_prime])
        A.append([0, 0, 0, -x, -y, -z, x * y_prime, y * y_prime, z * y_prime])
    A = np.array(A)

    # Solve the linear system using SVD
    _, _, V = np.linalg.svd(A)
    h = V[-1] / V[-1, -1]  # Normalize to make h33 = 1
    H = h.reshape(3, 3)

    return H
```

Figure 5: Custom homography function

The problem I've encountered is that the intersection points may not consistently align. One line could extend to positive infinity while the other reaches negative infinity, resulting in incorrect transformations due to mismatches between points at infinity.

Transformation Results

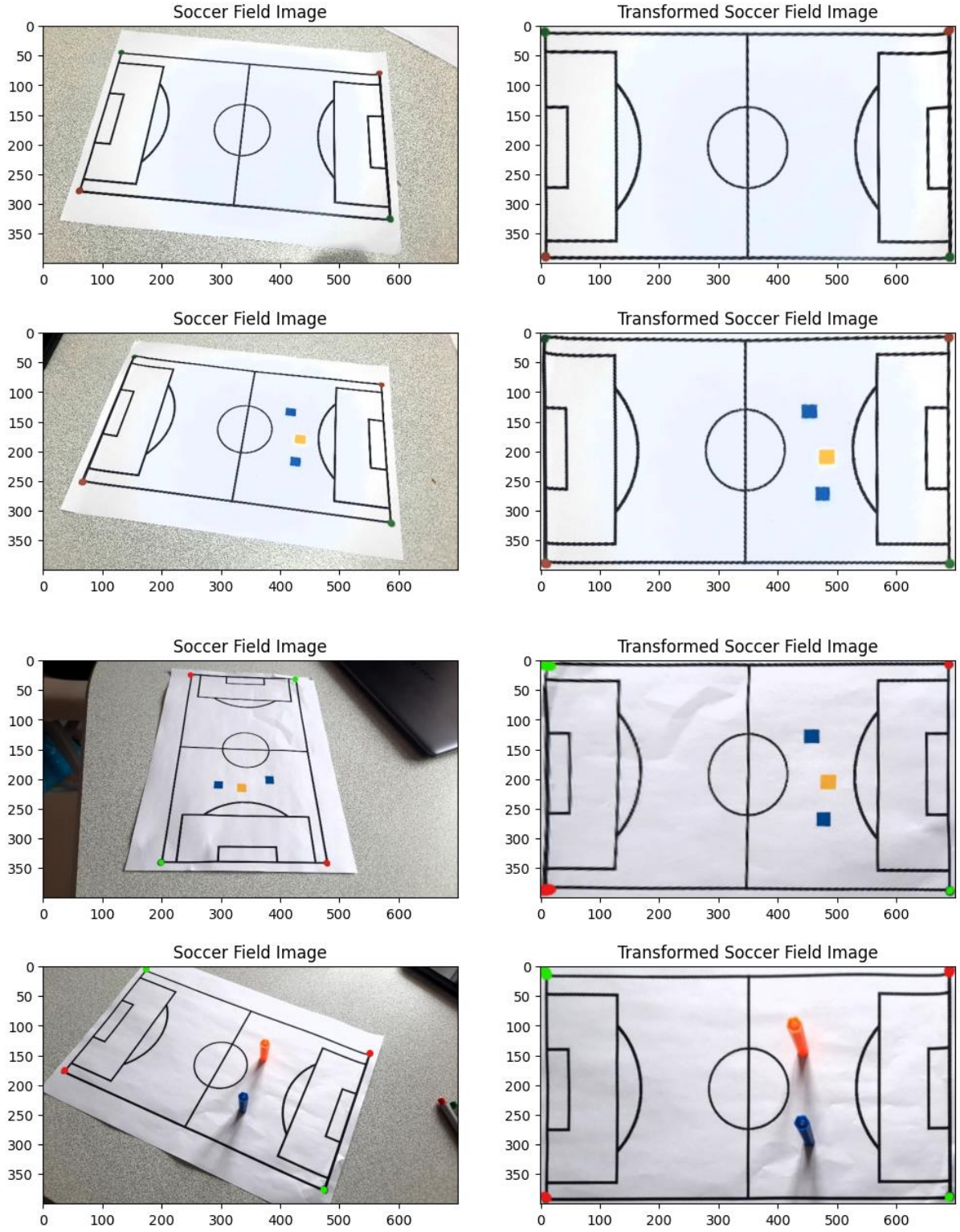


Figure 6: Applied homography transformation

Part 2

The proposed algorithm comprises the following steps:

- Corner detection (same as part 1)
- Estimating the homography
- Applying the homography to the source image
- Player detection
- Drawing circle around the players
- Apply the inverse transformation

Detecting the Players

The players are distinguished by the colors orange and blue. Hence, player detection can be achieved by generating color masks and subsequently identifying contours. Specific lower and upper RGB value ranges are set, classifying any pixels falling within this range as belonging to a player.

```
# Convert the image to the HSV color space
hsv_img = cv2.cvtColor(transformed_img, cv2.COLOR_BGR2HSV)

# Define lower and upper range for orange color in HSV
lower_orange = np.array([5, 100, 100])
upper_orange = np.array([15, 255, 255])

# Create a mask to isolate orange regions
orange_mask = cv2.inRange(hsv_img, lower_orange, upper_orange)

# Define lower and upper range for blue color in HSV
lower_blue = np.array([100, 100, 100])
upper_blue = np.array([120, 255, 255])

# Create a mask to isolate blue regions
blue_mask = cv2.inRange(hsv_img, lower_blue, upper_blue)

# Combine the masks to detect both orange and blue players
players_mask = cv2.bitwise_or(orange_mask, blue_mask)

# Find contours in the combined mask
players_contours, _ = cv2.findContours(players_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Filter contours based on area (size) to remove noise
min_player_area = 500
players_filtered_contours = [contour for contour in players_contours if cv2.contourArea(contour) > min_player_area]

draw_circle_around_player(players_filtered_contours)
```

Figure 7: Detecting players using color masks

Drawing Circle Around the Players

Following player detection, a circle is drawn with careful consideration to avoid overlap with the player. To achieve this, a custom draw circle function is designed. This function calculates the beam

length and loops accordingly. Notably, only pixels close to the soccer field background are printed, while the rest are left as is. This strategy prevents overlap between players and circles.

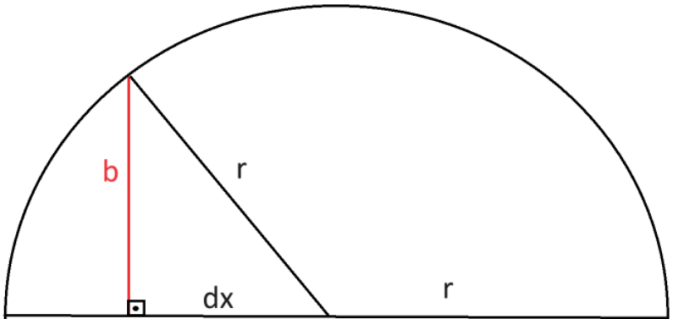


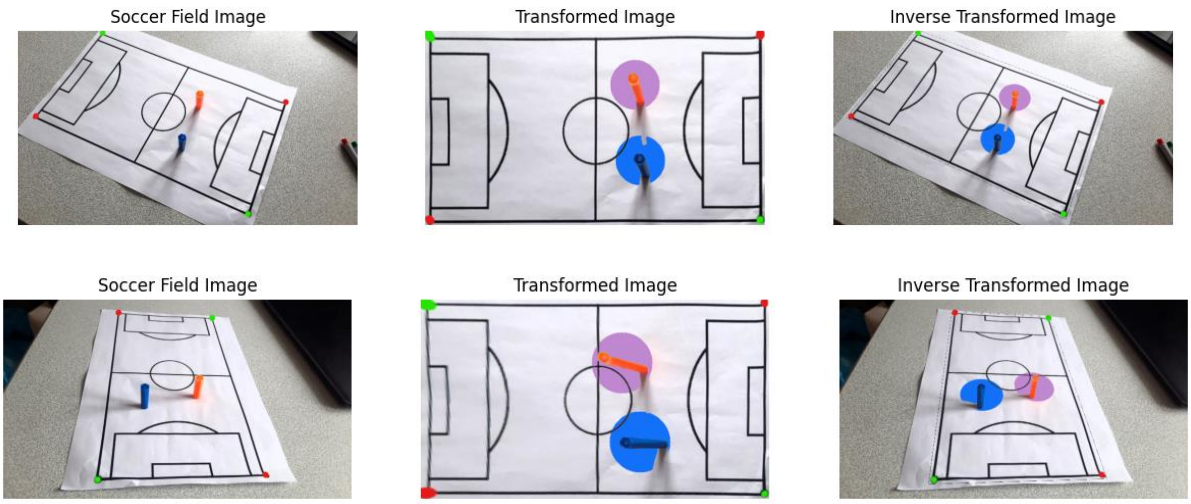
Figure 8: Lenght of the circle beam

```python
def draw_circle_around_player(contours):
    for i, contour in enumerate(contours, start=1):
        print(f"player{i}")
        # Calculate the centroid of the contour
        M = cv2.moments(contour)
        if M["m00"] != 0:
            cx = int(M["m10"] / M["m00"])
            cy = int(M["m01"] / M["m00"])

            radius = 50
            for dx in range(-radius, radius + 1):
                # Calculate the length of the beam
                b = int(math.sqrt(radius**2 - abs(dx)**2))
                for dy in range(-b, b + 1):
                    x = cx + dx
                    y = cy + dy
                    if 0 <= x < gray_img.shape[1] and 0 <= y < gray_img.shape[0] and not is_overlapping(gray_img, x, y):
                        cv2.circle(transformed_img, (x, y), 1, (0, 255, 255), -1)  # Draw circle in orange color
```

Figure 9: Custom function for drawing circles

Results



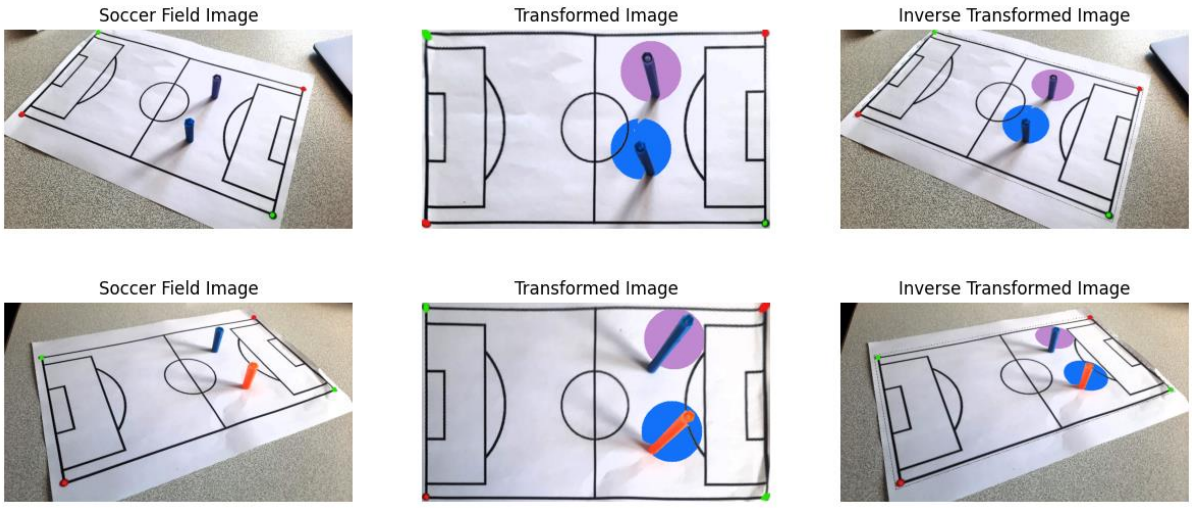


Figure 10: Creating a circular boundary around the identified players