Augmented Reality Applied to Images of Football Games

Daniel Marques
up201503822@fe.up.pt
Pedro Reis
up201506046@fe.up.pt
Tiago Magalhães
up201607931@fe.up.pt

Faculty of Engineering
University of Porto
Porto, Portugal

Abstract

This paper documents a program that allows the overlaying of visual information into images of football games. This information includes textual notes like the distance between the ball the and the goal line, and virtual marks such as the offside line during an attack or a circle that represents the minimum distance the opponents must respect during a free kick. The implementation uses *OpenCV* in *Python* and integrates image processing techniques for field detection, camera calibration, and finally the rendering of the visual information. The detection of the field is done through color masking a range of green with the image. The camera calibration stage can either be manual or automatic. When using the manual approach, the user selects the four feature points prompted in the console and a homography is calculated. When using the automatic alternative, the field lines and their intersections are detected using *Canny* and *Hough Transform* for edges and lines, the points from the intersections are matched with the reference points with a brute force algorithm, and the homography is calculated. Either approach requires the user to select other relevant points, specific to the user need, for instance, selecting the ball for the free kick example or the player to display the offside line. The rendering process blends the overlay with the original image, taking advantage of the field detection to draw the visual marks behind the players and referee.

1 Introduction

Football is the most popular sport in the world [5], with an estimated market size in Europe of 28.4 billion euros in the 2017/2018 season [3]. In an industry of this size, it's expected that viewership follows the same pattern with fans watching the matches broadcast throughout the world.

Football matches represent a market where computer vision techniques have great potential, both for broadcast stations and referees. Fans watching the broadcast enjoy statistics such as the distance to the goal line in a free kick, which can be seen by fans as an informal heuristic on whether or not it will result in a goal. Referees can be further aided in enforcing rules with these visual marks, such as confirming or denying an offside situation or checking if the opponents are respecting the minimum distance in a free kick.

On average, a referee makes around 245 decisions per game with only 5 of them being incorrect [2]. However, as each decision can be crucial to a match's result, there should be a concern in increasing this accuracy. A computer vision based system with augmented reality can help referees make better decisions by not relying solely on his immediate view of the game.

Aside from the rule enforcement benefit, there is also a benefit for broadcast stations. As computer vision allows them to create more informative and interesting content, fans might prefer a station over another thus increasing its viewership share, or new fans can emerge as rules become visual and easier to understand.

This work focuses on developing a computer vision based solution with *OpenCV* in *Python* to enhance the analysis of still shots of offside and free kick situations. The presented solutions can be divided into manual and automatic approaches for the camera calibration stage. However, this automation still constitutes an open problem as the presented solution is still quite limited, requiring parameter tuning and not working on all cases. The manual alternative works for any image, as long as the required feature points can be selected. Both approaches perform field segregation and manual entity selection.

2 Techniques

In this section, we'll describe the various computer vision and augmented reality techniques used in the implementation of our solution. These include field segregation and homography calculation. Each sub-section will introduce a technique, its theoretical background and relevant code snippets of its implementation.

2.1 Field Segregation

Many of the techniques explored in this project work best when applied only on the field and disregard other entities, such as the stands, the audience and the players themselves. In our case, this is also useful to be able to draw overlays such as lines and circular areas underneath the players and referees, making for a much more seamless and immersive experience.

The way we have chosen to execute this segregation is based on color. Since we know that the football field will always be a shade of green we can define a range of green colors and extract pixels that are within that range to form out the football field segregation layer. However, this method comes with two major disadvantages: (1) it might work poorly if the equipment of either of the teams contains a shade of green; (2) due to weather and lighting conditions this method might not work as intended, requiring the range of colors to be adjusted. Some of these issues could be solved by the use of more advanced techniques such as automatic dominant color detection [4].

The color-based segregation was done in HSV color space as it can separate the *luma* (image brightness), from *chroma* (color information). Thus, making the process of selecting a color range more intuitive when compared to an RGB color space. As such, in this project, all color range checks are done in HSV format version of the original image. OpenCV offers ways to convert from one color space to another. The code snippet below demonstrates how to convert an image from RGB to HSV.

```
hsv = cv2.cvtColor(src_img, code=cv2.COLOR_BGR2HSV)
```

Once the color range is defined we can create a mask that will be used to extract the field layer from the original image. It is also useful to obtain a mask for the layer that contains objects that are not part of the field, in order to reconstruct the original image. To do this we simply flip the previously created field mask. These masks can be created with the following code snippet:

```
# color range
lower_green = np.array([35, 60, 60])
upper_green = np.array([65, 255, 255])
# layer masks
field_mask = cv2.inRange(hsv,lower_green,upper_green)
player_mask = cv2.bitwise_not(field_mask)
```

These masks are an image where if a pixel is white then it is within the specified range and if the pixel is black it isn't. We can then take these images to extract the layers from the original image, this can be done with the bitwise operation *bitwise_and* of the image and the masks which are used to remove the pixels that are not part of the layer. This can be done with the following code:

```
field_layer = cv2.bitwise_and(src_img, src_img,
    mask=field_mask)
player_layer = cv2.bitwise_and(src_img, src_img,
    mask=player_mask)
```

Resulting from this are the segregated layers, containing only the objects that pertain to them.

(a) Field layer (b) Player layer

Figure 1: Images of the layers

2.2 Manual Feature Point Selection

Having a homography that represents the transformations from the real world to an image allows us to precisely draw the visual marks on the image and get additional information. With it, we can draw the visual marks with the correct perspective, position and scale. By using the inverse homography, we can calculate distances in the real world, such as the distance from the ball to the goal line.

The manual approach to selecting feature points requires the user to select the four points prompted in the console in clockwise order, as seen in Figure 5. Specifically, in the offside line example we've decided to use the right-side penalty area, whereas in the free kick example we prompt for the left-side goal area corners. This was done to ensure our implementation would work on either side of the field and with whichever corners.

The following code block calculates a homography between the real world and the image, and its inverse which does the opposite:

```
h, status = cv2.findHomography(ref_pts, img_pts)
h_inv = np.linalg.inv(h)
```

To apply a homography to a point or to an image we've used the following functions, respectively:

```
# Get the ball position in the real world
ball_rw = cv2.perspectiveTransform(ball_im, h_inv)
# Transform real world overlay into image space
overlay_img = cv2.warpPerspective(overlay_rw, h,
    shape)
```

2.3 Automatic Feature Point Detection

In order to automate the process of introducing the feature points needed for a homography, we need to detect the 4 corners that make up the penalty (large) area of the football field.

There are various methods to do this and we have chosen to base our approach on edge and line detection techniques, specifically Canny and Hough Lines, respectively. To achieve better results we apply these techniques only to the field layer so as not to obtain interference from outside objects that could introduce unwanted data, making the whole process much more complicated. To improve the edge detection results of Canny we first pass the field layer image through a Gaussian blur filter with a 3x3 kernel. This process can be done with the following code snippet:

```
field = cv2.GaussianBlur(field, (3, 3),
    cv2.BORDER_DEFAULT)
edges = cv2.Canny(field, 100, 300)
lines = cv2.HoughLines(edges, 1, math.radians(1.7),
    150, None, 0, 0)
```

One thing to note is that different images will likely require tuning of the parameters used both for Canny and HoughLines. However, these parameters are the ones we are currently using for our demonstration set of images.

With this, we have several feature points. Nonetheless, many are too similar, resulting from the field lines being so thick they generate two different lines and, therefore, various intersection points near each other. To clean up the points we simply loop through them and delete the ones which are within a fixed number of pixels (20 pixels). However, we need to match them with the reference points to calculate the homography. We've decided to take a brute-force approach with some heuristics. Firstly, we generate and loop through all the permutations of feature points and reference points. For each permutation, we check two heuristics: (1) the polygon is convex; (2) the vertices are in clockwise order. Since we are looking for a quadrangular area, it makes sense that the polygon should be convex. As for being clockwise, since the reference points are in clockwise order, to ensure the homography is correct, the reference points should also be in clockwise order. Because a solution might be encountered by chance, we try to further ensure its correctness by testing the homography against another feature point and try and check if we can find another corner. If we do, with a maximum error distance of 2 meters, we define the solution as valid. However, this implementation is far from perfect, as some solutions are encountered, passing our tests, but are incorrect. This is due to the homography being incorrect but accidentally providing an acceptable value for the corner.

Figure 2 shows the resulting debug information of an automatic feature points selection, with the detected lines in red and its intersection points in green.

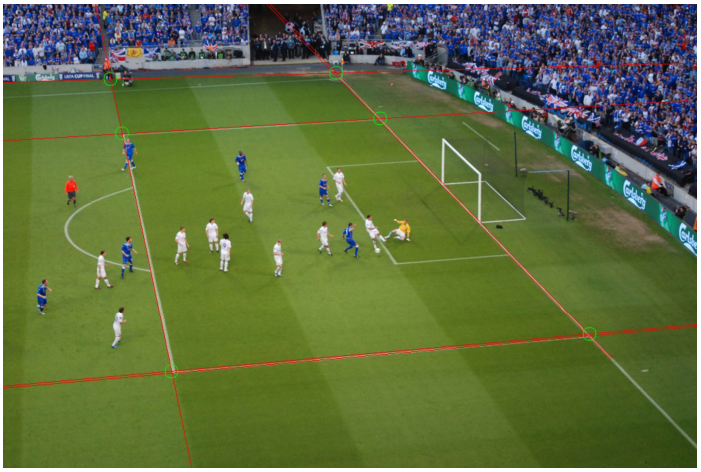

Figure 2: Image with the detected feature points and lines

3 Applications

In this section, we'll use the techniques from the previous section to develop examples of their applications in images of football games. There are two examples: (1) offside example; (2) free kick example. The first is meant to aid the referees' decision in an offside situation by drawing the offside line. The second is supposed to be a visual aid for both the viewer and referees as it's composed of two distinct parts: the line and distance to the goal line, and the circle delimiting the minimum distance to opponents.

3.1 Offside

In football, according to the International Football Association Board, a player is in offside position if any playable part of the body — head, body or feet — is in the opponent's half and is nearer to the opponents' goal line than both the ball and the second-last opponent [1]. However, a player in offside position at the moment the ball is played by a team-mate only constitutes an offence of the law when becoming involved in the active play. Therefore, there is a line, the *offside line*, where a player's position can be compared against to evaluate whether or not the player is in offside position.

We developed two programs that draw an offside line in a still shot of a play. The difference between the two implementations is the method for selecting feature points to calculate the homography. A more basic approach (described in section 2.2) relies on the user selecting the points prompted by the program, whereas a more advanced approach (described in section 2.3) uses automatic feature point detection to the same end.

Firstly, we calculate the homography between the real world and the image, and it's inverse, allowing us to transform a point's coordinates from one domain into the other. Afterwards, we request the user to select where the player is and proceed to calculate the corresponding coordinates

in the real world. There, we collect its coordinates

$$W_p = (x_p, y_p) = homography((i_p, j_p), H_{inv})$$

and determine two points — one at each end of the field — resulting in $W_1 = (x_p, 0)$ and $W_2 = P(x_p, w)$, where w is the width of the field. In our implementations, we assume the dimensions to be 105 meters in length and 68 meters in width, as the most preferred size for many professional teams' stadiums [7]. Then, we simply render a line between the two points and apply the inverse transformation process, blending the line with the field layer (retrieved as described in section 2.1) so that it stays behind the players and referees.

The two programs can lead to very similar results as seen in Figures 3 and 6. However, a slight error on the automatic approach can be seen as it continues through the field by a small margin indicating that the field's upper line — even though poorly detected — was used for the homography.

Figure 3: Offside line overlay with manual feature point selection

3.2 Free kick

In football, according to *Laws of the Game*, a free kick is a method of restarting play and awarded to the opposing team whenever a team is guilty of an offence [1]. Until the ball is in play, all opponents must maintain a minimum distance of 9.15 meters from the ball.

We've developed a solution that renders this radius from the ball, ensuring all opposing players respect the rule, and additionally, and optionally, displays the direct line between the ball and the goal line, as well as its distance in meters. Similarly to the offside example, we've implemented two approaches varying the methods of feature points gathering between manual selection (section 2.2) and automatic detection (section 2.3).

We first calculate the homography, following the same process used in the offside example. Then, we request the user to select the ball in the image. With that, we have all the necessary information for further calculations. We first calculate its coordinates in the real world using the inverse homography. Secondly, we draw the circumference with the specified radius, the line from the ball to the middle of the goal line and calculate its length. As done in the previous example, we then perform the inverse process. Using the homography we apply convert the image generated in real world coordinates to the still shot domain:

```
overlay_img = cv2.warpPerspective(overlay_rw, h,
    (im_dst.shape[1],im_dst.shape[0]),
    cv2.INTER_LANCZOS4)
```

Finally, we blend the overlay with the field layer (retrieved as described in section 2.1) so that it stays behind the players and referees, and render the text stating the distance to the goal line above all layers. The final results can be seen in Figures 4 and 7. However, the original images differ as an automatic homography was not possible in that particular example as a result of the limitations pointed previously. Furthermore, it's possible to toggle each visual mark individually (including the circumference, line and distance text).

Figure 4: Free kick overlay with manual feature point selection

4 Conclusion

In this work, we've introduced two examples of situations where computer vision and augmented reality can aid or improve the perception of football matches. An offside example displays a line representing the most offensive position without infringement of the offside rule. A free kick example displays the circumference that denotes the minimum distance to the ball for opponents during a free kick, as well as the direct line to the goal line and its distance in meters.

Both examples use common techniques such as color separation applied to the field, separating it from the people; homographies which establish relationships of transformations between two domains, linking the still shot and the real world. The former serves the purpose of enabling the rendering behind the people, directly onto the field layer, and also aids in the process of automatic feature point detection, removing some of the noise from the image. The latter, allows us to get the coordinates from one domain to the other so that calculations and precise rendering, with perspective taken into account, can be accomplished.

However, as discussed in section 2.3, the algorithms used for automatic feature point detection and matching were basic and therefore led to major limitations, with only some images working flawlessly. In terms of performance, it's difficult to evaluate it as each homography can be quickly calculated, but it might not be a good solution (empirically), therefore requiring more iterations in matching or line detection parameter tuning. As future work, we suggest: (1) automatic dominant color detection [4]; (2) improved line detection [6]; (3) improved matching between feature points and reference points.

References

[1] International Football Association Board. Laws of the game 2019/20. 2019. URL http://static-3eb8.kxcdn.com/files/document-category/062019/frRhKJNjSBAtiyt.pdf.

[2] Gerard Brand. Referee myth-busting: How many decisions do officials get right? *Skysports*, Mar 2018. URL https://www.skysports.com/football/news/11096/10808860.

[3] Christina Gough. Market size of the european professional football market from 2006/07 to 2017/18. *Statista*, Oct 2019. URL https://www.statista.com/statistics/261223.

[4] Y. Liu, Maozu Guo, and Wanyu Liu. Detection of playfield with shadow and its application to player tracking. In *2011 IEEE International Workshop on Machine Learning for Signal Processing*, pages 1–5, Sep. 2011. doi: 10.1109/MLSP.2011.6064621.

[5] Tomas Stølen, Karim Chamari, Carlo Castagna, and Ulrik Wisløff. Physiology of soccer. *Sports Medicine*, 35(6):501–536, Jun 2005. ISSN 1179-2035. doi: 10.2165/00007256-200535060-00004.

[6] Li Sun and Guizhong Liu. Field lines and players detection and recognition in soccer video. pages 1237–1240, 04 2009. doi: 10.1109/ICASSP.2009.4959814.

[7] Wikipedia. Football pitch. 2019. URL https://en.wikipedia.org/wiki/Football_pitch.

A Image Results

Figure 5: Image with the 4 feature points (in red) selected by the user

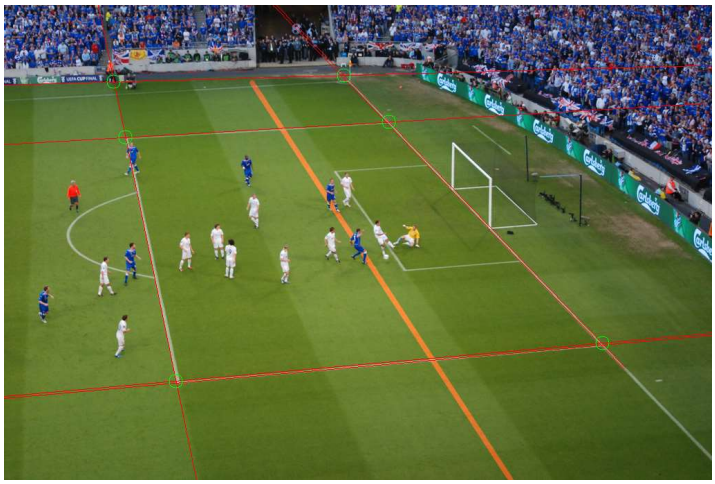

Figure 6: Offside line overlay with automatic feature point detection

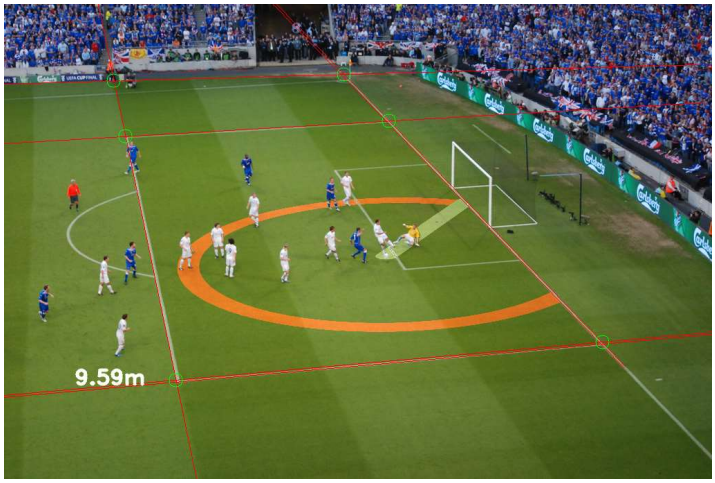

Figure 7: Free kick overlay with automatic feature point detection

B Run Instructions

B.1 Installation

1. Install Python 3 and, optionally, virtualenv
2. Copy the source code
3. If using a Python3 virtualenv, start and activate it

```
virtualenv -p python3 env
. env/bin/activate
```

4. Install Python dependencies

```
pip3 install -r dependencies.txt
```

B.2 Running

B.2.1 Offside Line with Manual Homography

- Run the manual homography offside line Python3 script using:

```
python3 offside.py
```

- This script is prepared to handle any image from the right side of the field, with all penalty area points visible.
- Click on the 4 points of the penalty area in a clockwise fashion[1]. Press Enter.
- Click on the player where you'd want to draw the line. Press Enter.

B.2.2 Free-kick with Manual Homography

- Run the manual homography free-kick Python3 script using:

```
python3 free_kick.py
```

- This script is prepared to handle any image from the left side of the field, with all goal area points visible.
- Click on the 4 points of the goal area in a clockwise fashion[1]. Press Enter.
- Click on the ball. Press Enter.

B.2.3 Offside Line with Automatic Homography

- Run the automatic homography offside line Python3 script using:

```
python3 offside_automatic.py
```

- This script should be able to handle any image from the right side of the field, with all penalty area points visible. However, this isn't guaranteed as per image parameter tuning is often needed for lines and field detection.
- Click on the player where you'd want to draw the line. Press Enter.

B.2.4 Free-kick with Automatic Homography

- Run the automatic homography free-kick Python3 script using:

```
python3 free_kick_automatic.py
```

- This script should be able to handle any image from the left side of the field, with all goal area points visible. However, this isn't guaranteed as per image parameter tuning is often needed for lines and field detection.
- Click on the ball. Press Enter.

[1] Some machines only work with counter-clockwise (mainly when running Windows)

C Source Code

C.1 offside.py

```
#!/usr/bin/env python3

import cv2
import numpy as np
import utils


if __name__ == '__main__' :

    # Read destination image
    # 2008 UEFA Cup Final (Zenit Saint Petersburg vs Rangers) - Manchester City Stadium
    # Origin: Top Right. Field Size: 105m by 68m
    im_dst = cv2.imread('../img/football1.jpg')
    clean_img = im_dst.copy()

    # Create a vector of source points.
    # Real-life Manchester City Stadium - Right Penalty Area Points
    pts_src = np.array(
        [
            [87.5, 12.84],      # Top Left
            [104,  12.84],      # Top Right
            [104,  53.16],      # Bottom Right
            [87.5, 53.16 ]      # Bottom Left
        ],dtype=float
    )

    # Get four corners of the penalty area
    print('Click on the four corners of the penalty area and then press [ENTER]')
    pts_dst = utils.get_points(im_dst, 4)

    # Calculate Homography between source and destination points
    h, status = cv2.findHomography(pts_src, pts_dst)
    h_inv = np.linalg.inv(h)

    # Get offside player point (field image)
    print('Click on the offside player and then press [ENTER]')
    player_im = utils.get_points(im_dst, 1)[0]

    # Get corresponding offside player point in real world
    player_rw = cv2.perspectiveTransform(player_im.reshape(1, 1, -1), h_inv)[0][0]

    # Get the two line points in the real world line (same x, y is the field bounds)
    line_point_1_rw = player_rw.copy()
    line_point_1_rw[1] = 0
    line_point_2_rw = player_rw.copy()
    line_point_2_rw[1] = 67

    # Get corresponding second point in the image
    line_point_1_im = cv2.perspectiveTransform(line_point_1_rw.reshape(1, 1, -1), h)[0][0]
    line_point_2_im = cv2.perspectiveTransform(line_point_2_rw.reshape(1, 1, -1), h)[0][0]

    # Draw line
    height, width, channels = clean_img.shape
    blank_image = np.zeros((height,width,3), np.uint8)
    overlay_img = blank_image
    cv2.line(overlay_img, tuple(line_point_1_im.astype(int)), tuple(line_point_2_im.astype(int)),
        (0,0,255), 5, cv2.LINE_AA)

    final = utils.blend_overlay_with_field(clean_img,overlay_img,0.5)

    # Display image.
    cv2.imshow("Image", final)
    cv2.waitKey(0)
```

C.2 free_kick.py

```
#!/usr/bin/env python3

import cv2
import numpy as np
import utils


if __name__ == '__main__' :

    quality = 100
    draw_circle = True
    draw_line = True
    draw_text = True

    # Read destination image
    # 2018 World Cup (Colombia vs Japan) - Mordovia Arena
    # Origin: Top Right. Field Size: 105m by 68m
    im_dst = cv2.imread('../img/football2.png')

    # Create a vector of source points.
    # Real-life Mordovia Arena - Left Goal Area Points
    middle_of_goal_line = np.array([0, 34], dtype=float)
    pts_src = np.array(
        [
            [0,   24.84],      # Top Left
            [5.5, 24.84],      # Top Right
            [5.5, 43.16],      # Bottom Right
            [0,   43.16 ]      # Bottom Left
        ],dtype=float
    )
    pts_src *= quality
    middle_of_goal_line *= quality

    # Get four corners of the goal area
    print('Click␣on␣the␣four␣corners␣of␣the␣goal␣area␣and␣then␣press␣[ENTER]')
    pts_dst = utils.get_points(im_dst, 4)

    # Calculate Homography between source and destination points
    h, status = cv2.findHomography(pts_src, pts_dst)
    h_inv = np.linalg.inv(h)

    # Get ball point (field image)
    print('Click␣on␣the␣ball␣and␣then␣press␣[ENTER]')
    ball_im = utils.get_points(im_dst, 1)[0]

    # Get corresponding ball point in real world
    ball_rw = cv2.perspectiveTransform(ball_im.reshape(1, 1, -1), h_inv)[0][0]

    # Draw circle
    overlay_rw = np.zeros((68*quality, 105*quality, 3), np.uint8)
    if draw_circle:
        cv2.circle(overlay_rw, tuple(ball_rw.astype(int)), int(9.15*quality), (0,0,255), 1*quality,
            lineType=cv2.LINE_AA)

    # Draw line
    if draw_line:
        cv2.line(overlay_rw, tuple(ball_rw.astype(int)), tuple(middle_of_goal_line.astype(int)),
            (128,128,128), 1*quality, cv2.LINE_AA)

    # Warp overlay
    overlay_img = cv2.warpPerspective(overlay_rw, h, (im_dst.shape[1],im_dst.shape[0]), cv2.
        INTER_LANCZOS4)

    # Draw text
    height, width, channels = im_dst.shape
    overlay_text = np.zeros((height,width,3), np.uint8)
    if draw_text:
        distance = np.linalg.norm(ball_rw - middle_of_goal_line) / quality
        text_location = (int(overlay_text.shape[1] * 0.1), int(overlay_text.shape[0] * 0.8))
```

```
        cv2.putText(overlay_text, "%.2f" % distance + 'm', text_location, cv2.FONT_HERSHEY_DUPLEX,
            1, (255,255,255), 2, cv2.LINE_AA)

    # Merge overlay
    final = utils.blend_overlay_with_field(im_dst,overlay_img,0.5)
    cv2.add(overlay_text, final, final)

    # Display image.
    cv2.imshow("Image", final)
    cv2.waitKey(0)
```

C.3 offside_automatic.py

```
import cv2
import math
import utils
import numpy as np
from itertools import permutations


def findBestHomography():
    # Loop through all the possible permutations of 4 points in the image
    all_img_pts = [np.array(i, dtype=float) for i in intersections]
    for img_pts in permutations(intersections, 4):
        # Optimization: check if polygon is convex and clockwise
        if not utils.isConvex(img_pts) or not utils.isClockwise(img_pts):
            continue

        # Loop through all the possible permutations of 4 points in the reference
        for ref_points in permutations(utils.reference_points_right, 4):
            # Optimization: check if polygon is convex and clockwise
            if not utils.isConvex(img_pts) or not utils.isClockwise(img_pts):
                continue

            # Convert to np.ndarray of np.ndarray of float64
            img_pts = [np.array(i, dtype=float) for i in img_pts]
            img_pts = np.array(img_pts)
            ref_points = np.array(ref_points)

            # Calculate homography
            h, status = cv2.findHomography(img_pts, ref_points)
            h_inv = None
            if h is None:
                continue
            try:
                h_inv = np.linalg.inv(h)
            except:
                continue

            # Check if at least one other image point matches one other reference point
            for test_img_pt in all_img_pts:
                if not test_img_pt in img_pts:
                    for test_ref_pt in utils.reference_points_right:
                        if not list(test_ref_pt) in ref_points.tolist():
                            # Apply homography to reference test point
                            test_img_in_ref = cv2.perspectiveTransform(test_img_pt.reshape(1, 1,
                                -1), h)[0][0]
                            # Check if it matches the image test point with an error of at most 2m
                            distance = np.linalg.norm(test_img_in_ref - test_ref_pt)
                            if distance < 2:
                                applyHomographyLine(img, h, h_inv)
                                return # Uncomment to loop through the possible homographies


def applyHomographyLine(img, h, h_inv):
    # Get offside player point (field image)
    print('Click␣on␣the␣offside␣player␣and␣then␣press␣[ENTER]')
    player_im = utils.get_points(img, 1)[0]

    # Get corresponding offside player point in real world
```

```
    player_rw = cv2.perspectiveTransform(player_im.reshape(1, 1, -1), h)[0][0]

    # Get the two line points in the real world line (same x, y is the field bounds)
    line_point_1_rw = player_rw.copy()
    line_point_1_rw[1] = 0
    line_point_2_rw = player_rw.copy()
    line_point_2_rw[1] = 67

    # Get corresponding second point in the image
    line_point_1_im = cv2.perspectiveTransform(line_point_1_rw.reshape(1, 1, -1), h_inv)[0][0]
    line_point_2_im = cv2.perspectiveTransform(line_point_2_rw.reshape(1, 1, -1), h_inv)[0][0]

    # Draw line
    height, width, channels = img.shape
    blank_image = np.zeros((height,width,3), np.uint8)
    cv2.line(blank_image, tuple(line_point_1_im.astype(int)), tuple(line_point_2_im.astype(int)),
        (0,0,255), 5, cv2.LINE_AA)
    img = utils.blend_overlay_with_field(img,blank_image,0.5)

    cv2.imshow("Image", img)
    cv2.waitKey(0)


if __name__ == '__main__':
    debug = True

    img = cv2.imread('../img/football1.jpg')
    field = utils.GetFieldLayer(img)
    field = cv2.GaussianBlur(field, (3, 3), cv2.BORDER_DEFAULT)
    edges = cv2.Canny(field, 100, 300)
    lines = cv2.HoughLines(edges, 1, math.radians(1.7), 150, None, 0, 0)

    lns = []

    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            a = math.cos(theta)
            b = math.sin(theta)
            x0 = a * rho
            y0 = b * rho
            pt1 = (int(x0 + 1000 * (-b)), int(y0 + 1000 * (a)))
            pt2 = (int(x0 - 1000 * (-b)), int(y0 - 1000 * (a)))
            lns.append(utils.Line(pt1, pt2))

    # Calculate intersection points
    intersections = []
    for i in range(0, len(lns)):
        for j in range(i + 1, len(lns)):
            point = lns[i].intersection(lns[j])
            if not point is None:
                intersections.append(point)
        if debug:
            cv2.line(img, lns[i].pt1, lns[i].pt2, (0, 0, 255), 1)


    # Cleanup similar points and points outside of the image
    i = 0
    while i < len(intersections):

        # Check if outside of image
        xi,yi = intersections[i]
        sy,sx,_ = img.shape
        if xi < 0 or yi < 0 or xi > sx or yi > sy:
            del intersections[i]
            continue

        # Check if similar
        j = i+1
        while j < len(intersections):
```

```
            distance = np.linalg.norm(list(x-y for x,y in zip(intersections[i],intersections[j])))
            if distance < 20:
                del intersections[j]
                continue
            j += 1
        i += 1

    # Draw points
    if debug:
        for point in intersections:
            cv2.circle(img, point, 10, (0, 255, 0))

    # Homography
    if len(intersections) < 4:
        raise Exception('Not enough points were detected for a homography')

    findBestHomography()
```

C.4 free_kick_automatic.py

```
import cv2
import math
import utils
import numpy as np
from itertools import permutations


def findBestHomography():
    # Loop through all the possible permutations of 4 points in the image
    all_img_pts = [np.array(i, dtype=float) for i in intersections]
    for img_pts in permutations(intersections, 4):
        # Optimization: check if polygon is convex and clockwise
        if not utils.isConvex(img_pts) or not utils.isClockwise(img_pts):
            continue

        # Loop through all the possible permutations of 4 points in the reference
        for ref_points in permutations(utils.reference_points_right, 4):
            # Optimization: check if polygon is convex and clockwise
            if not utils.isConvex(img_pts) or not utils.isClockwise(img_pts):
                continue

            # Convert to np.ndarray of np.ndarray of float64
            img_pts = [np.array(i, dtype=float) for i in img_pts]
            img_pts = np.array(img_pts)
            ref_points = np.array(ref_points)

            # Calculate homography
            h, status = cv2.findHomography(img_pts, ref_points)
            h_inv = None
            if h is None:
                continue
            try:
                h_inv = np.linalg.inv(h)
            except:
                continue

            # Check if at least one other image point matches one other reference point
            for test_img_pt in all_img_pts:
                if not test_img_pt in img_pts:
                    for test_ref_pt in utils.reference_points_right:
                        if not list(test_ref_pt) in ref_points.tolist():
                            # Apply homography to reference test point
                            test_img_in_ref = cv2.perspectiveTransform(test_img_pt.reshape(1, 1,
                                -1), h)[0][0]
                            # Check if it matches the image test point with an error of at most 2m
                            distance = np.linalg.norm(test_img_in_ref - test_ref_pt)
                            if distance < 2:
                                applyHomographyLine(img, img_pts, ref_points)
                                return # Uncomment to loop through the possible homographies
```

```
def applyHomographyLine(img, img_pts, ref_points):
    ref_points *= quality
    utils.middle_goal_line_right *= quality

    # Calculate Homography between source and destination points
    h, status = cv2.findHomography(ref_points, img_pts)
    h_inv = np.linalg.inv(h)

    # Get ball point (field image)
    print('Click␣on␣the␣ball␣and␣then␣press␣[ENTER]')
    ball_im = utils.get_points(img, 1)[0]

    # Get corresponding ball point in real world
    ball_rw = cv2.perspectiveTransform(ball_im.reshape(1, 1, -1), h_inv)[0][0]

    # Draw circle
    overlay_rw = np.zeros((68*quality, 105*quality, 3), np.uint8)
    if draw_circle:
        cv2.circle(overlay_rw, tuple(ball_rw.astype(int)), int(9.15*quality), (0,0,255), 1*quality,
             lineType=cv2.LINE_AA)

    # Draw line
    if draw_line:
        cv2.line(overlay_rw, tuple(ball_rw.astype(int)), tuple(utils.middle_goal_line_right.astype(
            int)), (128,128,128), 1*quality, cv2.LINE_AA)

    # Warp overlay
    overlay_img = cv2.warpPerspective(overlay_rw, h, (img.shape[1],img.shape[0]), cv2.
        INTER_LANCZOS4)

    # Draw text
    height, width, channels = img.shape
    overlay_text = np.zeros((height,width,3), np.uint8)
    if draw_text:
        distance = np.linalg.norm(ball_rw - utils.middle_goal_line_right) / quality
        text_location = (int(overlay_text.shape[1] * 0.1), int(overlay_text.shape[0] * 0.8))
        cv2.putText(overlay_text, "%.2f" % distance + 'm', text_location, cv2.FONT_HERSHEY_DUPLEX,
            1, (255,255,255), 2, cv2.LINE_AA)

    # Merge overlay
    final = utils.blend_overlay_with_field(img,overlay_img,0.5)
    cv2.add(overlay_text, final, final)

    # Display image.
    cv2.imshow("Image", final)
    cv2.waitKey(0)


if __name__ == '__main__':
    quality = 100
    draw_circle = True
    draw_line = True
    draw_text = True
    debug = True

    img = cv2.imread('../img/football1.jpg')
    field = utils.GetFieldLayer(img)
    field = cv2.GaussianBlur(field, (3, 3), cv2.BORDER_DEFAULT)
    edges = cv2.Canny(field, 100, 300)
    lines = cv2.HoughLines(edges, 1, math.radians(1.7), 150, None, 0, 0)

    lns = []

    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            a = math.cos(theta)
            b = math.sin(theta)
            x0 = a * rho
```

```
            y0 = b * rho
            pt1 = (int(x0 + 1000 * (-b)), int(y0 + 1000 * (a)))
            pt2 = (int(x0 - 1000 * (-b)), int(y0 - 1000 * (a)))
            lns.append(utils.Line(pt1, pt2))

    # Calculate intersection points
    intersections = []
    for i in range(0, len(lns)):
        for j in range(i + 1, len(lns)):
            point = lns[i].intersection(lns[j])
            if not point is None:
                intersections.append(point)
        if debug:
            cv2.line(img, lns[i].pt1, lns[i].pt2, (0, 0, 255), 1)


    # Cleanup similar points and points outside of the image
    i = 0
    while i < len(intersections):

        # Check if outside of image
        xi,yi = intersections[i]
        sy,sx,_ = img.shape
        if xi < 0 or yi < 0 or xi > sx or yi > sy:
            del intersections[i]
            continue

        # Check if similar
        j = i+1
        while j < len(intersections):
            distance = np.linalg.norm(list(x-y for x,y in zip(intersections[i],intersections[j])))
            if distance < 20:
                del intersections[j]
                continue
            j += 1
        i += 1

    # Draw points
    if debug:
        for point in intersections:
            cv2.circle(img, point, 10, (0, 255, 0))

    # Homography
    if len(intersections) < 4:
        raise Exception('Not enough points were detected for a homography')

    findBestHomography()
```

C.5 utils.py

```
import cv2
import numpy as np
import math


class Line:
    def __init__(self, pt1, pt2):
        self.pt1 = pt1
        self.pt2 = pt2
        self.dx = pt2[0] - pt1[0]
        self.dy = pt2[1] - pt1[1]
        if self.dx == 0:
            self.m = math.inf
        else:
            self.m = self.dy / self.dx
        self.b = pt1[1] - self.m * pt1[0]

    def intersection(self, other):
        if math.isinf(self.m) or math.isinf(other.m) or (self.m - other.m) == 0:
            return None
```

```
        x = (other.b - self.b) / (self.m - other.m)
        y = self.m * x + self.b
        return (int(x), int(y))


def mouse_handler(event, x, y, flags, data) :

    if event == cv2.EVENT_LBUTTONDOWN :
        if len(data['points']) < data['max_points'] :
            cv2.circle(data['im'], (x,y), 3, (0,0,255), 5, 16)
            cv2.imshow("Image", data['im'])
            data['points'].append([x,y])


def get_points(im, max_points) :

    # Set up data to send to mouse handler
    data = {}
    data['im'] = im.copy()
    data['points'] = []
    data['max_points'] = max_points

    # Set the callback function for any mouse event
    cv2.imshow("Image",im)
    cv2.setMouseCallback("Image", mouse_handler, data)
    cv2.waitKey(0)
    cv2.setMouseCallback("Image", lambda *args : None)

    # Convert array to np.array
    points = np.vstack(data['points']).astype(float)

    return points


def blend_overlay_with_field(src_img,overlay,transparency):
    hsv = cv2.cvtColor(src_img, code=cv2.COLOR_BGR2HSV)
    # green range
    lower_green = np.array([35, 60, 60])
    upper_green = np.array([65, 255, 255])
    # layer masks
    field_mask = cv2.inRange(hsv, lower_green, upper_green)
    player_mask = cv2.bitwise_not(field_mask)
    # extract layers from original image
    field_layer = cv2.bitwise_and(src_img, src_img, mask=field_mask)
    player_layer = cv2.bitwise_and(src_img, src_img, mask=player_mask)
    # creates line that is blank where the players are
    overlay_layer = cv2.bitwise_and(overlay,overlay,mask=field_mask)
    field_layer = cv2.addWeighted(field_layer,1,overlay_layer,transparency,0)
    final = field_layer + player_layer
    return final

def GetFieldLayer(src_img):
    hsv = cv2.cvtColor(src_img, code=cv2.COLOR_BGR2HSV)
    # green range
    lower_green = np.array([35, 10, 60])
    upper_green = np.array([65, 255, 255])
    # layer masks
    field_mask = cv2.inRange(hsv, lower_green, upper_green)
    player_mask = cv2.bitwise_not(field_mask)
    # extract layers from original image
    field_layer = cv2.bitwise_and(src_img, src_img, mask=field_mask)
    return field_layer


TWO_PI = 2 * math.pi
# Credits to Rory Daulton
def isConvex(polygon):
    """Return True if the polynomial defined by the sequence of 2D
    points is 'strictly convex': points are valid, side lengths non-
    zero, interior angles are strictly between zero and a straight
    angle, and the polygon does not intersect itself.
```

```
    """
    try:  # needed for any bad points or direction changes
        # Check for too few points
        if len(polygon) < 3:
            return False
        # Get starting information
        old_x, old_y = polygon[-2]
        new_x, new_y = polygon[-1]
        new_direction = math.atan2(new_y - old_y, new_x - old_x)
        angle_sum = 0.0
        # Check each point (the side ending there, its angle) and accum. angles
        for ndx, newpoint in enumerate(polygon):
            # Update point coordinates and side directions, check side length
            old_x, old_y, old_direction = new_x, new_y, new_direction
            new_x, new_y = newpoint
            new_direction = math.atan2(new_y - old_y, new_x - old_x)
            if old_x == new_x and old_y == new_y:
                return False  # repeated consecutive points
            # Calculate & check the normalized direction-change angle
            angle = new_direction - old_direction
            if angle <= -math.pi:
                angle += TWO_PI  # make it in half-open interval (-Pi, Pi]
            elif angle > math.pi:
                angle -= TWO_PI
            if ndx == 0:  # if first time through loop, initialize orientation
                if angle == 0.0:
                    return False
                orientation = 1.0 if angle > 0.0 else -1.0
            else:  # if other time through loop, check orientation is stable
                if orientation * angle <= 0.0:  # not both pos. or both neg.
                    return False
            # Accumulate the direction-change angle
            angle_sum += angle
        # Check that the total number of full turns is plus-or-minus 1
        return abs(round(angle_sum / TWO_PI)) == 1
    except (ArithmeticError, TypeError, ValueError):
        return False  # any exception means not a proper convex polygon

def signedArea(pts):
    xs,ys = map(list, zip(*pts))
    xs.append(xs[0])
    ys.append(ys[0])
    area = 0
    for i in range(len(xs) - 1):
        edge = (xs[i+1] - xs[i]) * (ys[i+1] + ys[i])
        area += edge
    return area

def isClockwise(pts):
    return signedArea(pts) > 0


# 105x68 Field, origin on the left upper corner, clock-wise order
reference_points_right = np.array([
    # Right penalty area
    [88.5, 13.84], [105, 13.84],
    [105, 54.16], [88.5, 54.16],

    # Right upper corner
    [105, 0]
], dtype=float)

middle_goal_line_right = np.array([105, 34], dtype=float)
```

D Original Images