Augmented Reality Applied to Images of Football Games

David Reis
https://sigarra.up.pt/feup/pt/fest_geral.cursos_list?pv_num_unico=201607927
João Barbosa
https://sigarra.up.pt/feup/pt/fest_geral.cursos_list?pv_num_unico=201406241
Rui Oliveira
https://sigarra.up.pt/feup/pt/fest_geral.cursos_list?pv_num_unico=201000619
Jorge Silva
https://sigarra.up.pt/feup/pt/func_geral.formview?p_codigo=208785
Filipe Rodrigues
https://sigarra.up.pt/feup/pt/func_geral.formview?p_codigo=512326

Faculty of Engineering
University of Porto
Porto, PT

Abstract

Football game broadcasters usually provide visual aids, such as offside lines or goal's distance, to provide insightful information to spectators. In order to provide these features, it is necessary to map real world coordinates with the image being shown on the screen.

In this work, we developed an application that allows a user to draw and compute different features in an input image. We also explore how we can automatically map the image to the real world and describe the main advantages and setbacks.

1 Introduction

Football matches provide many situations where a broadcaster can aid the viewers, or the officials, in interpreting certain aspects of the game. Some of these circumstances include: when we want to understand if a player is offside; or to know the distances between players, the ball, and/or the goals. In order to provide this functionality, we must be able to map the image being broadcast to the real world coordinates of the pitch. We can achieve this three-dimensional reconstruction with the use of a homography [3], from which it's possible to compute coordinates of interest and draw in a more realistic fashion, and finally return the result correctly to the original image.

The solution developed allows the user to: (i) create a nine meter circle around the free kick spot, (ii) draw an offside line, and (iii) draw an arrow from the free kick spot to the center of the goal, along with the real world distance between the two. In section 2, we discuss the various eases and difficulties in automatically finding the pitch, players and the points of interest to build an homography. Section 3 details the features in the user interface, as well as the methods used to draw on the image or calculate desired properties. Section 4 includes the conclusions of this work and possible future improvements.

2 Automatic target detection

In this section, we explain the different algorithms used to maximize the application's independence from user input, including their strengths, assumptions and which situations we can't fully automate.

2.1 Field detection

We start by detecting the football pitch in the image. This step is particularly important because: (i) it allows us to remove irrelevant information for the following detection algorithms, such as the crowd, and (ii) it gives us the boundaries of where we should draw the AR objects (for instance, the offside line should only traverse the pitch).

We used color segmentation in order to detect the football pitch, which is usually composed of a dominant green color. However, there might be some shade variations due to lighting conditions or the pattern of the field itself (strips of lighter and darker greens). As such, these variations must be captured by the segmentation algorithm.

We converted the image to the HSV color-space and filtered its content using OpenCV's *inRange* function. The Hue value for green is usually 120 in the standard 0-360 range, but since OpenCV's Hue uses 0-180 range [2], we need to convert it to 60. Consequently, we used the values (30, 50, 50) and (80, 255, 255) to filter the greens.

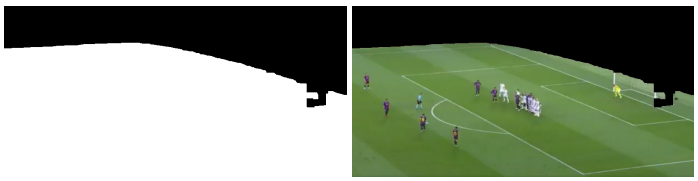
(a) (b)

Figure 1: Image segmentation by color: (a) Original image; (b) Green filter.

We then remove noise in the image using an opening morphological operation using a 11x11 rectangular structuring element. After that, we apply a closing morphological operation using a 15x15 rectangular structuring element so we can include possible pixels unintentionally removed.

Finally, we extract the field region by using contour detection. It is assumed that the football pitch is the largest green area in the image. Therefore, we compute the contour with the largest area using OpenCV's *findContours* and take every pixels inside its region as the pitch.

Our experiments show that field detection by color segmentation usually gives very accurate results. It should be noted, however, that this approach will have poorer performance in extreme weather conditions - such as when the pitch is covered in snow - or if the field is badly treated.



(a) (b)

Figure 2: Every pixel inside the largest contour is considered the field: (a) Field mask; (b) Result in original image.

2.2 Player detection

We detect players on the pitch, so that we can prevent their occlusion when drawing our desired AR objects. Using the previous computed green mask (Figure 1b) and field mask (Figure 2a), we have all the information needed to calculate the players' mask.

We assume the greens filter covers most of the pitch, while leaving the players unmasked, as in Figure 1b. Thus, we start by inverting the values of the greens mask. Since it will output false positives (such as the crowd), we restrict the values to pixels inside the pitch only. We then perform a 3x7 erosion followed by dilation, in order to remove possible noise in the resulting image. We use a larger value for the vertical component of the structuring element because we expect the players to be mainly in a vertical position (standing up).

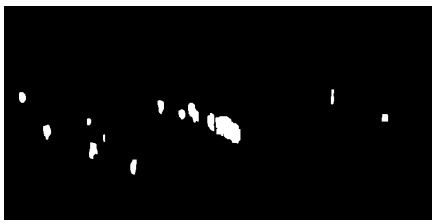


Figure 3: Players mask.

2.3 Points detection

In this section, we explain how we do the automatic detection of the points in the image which will be used to calculate the homography. Our approach consists in finding the delimiters of the goal and penalty areas and, with those lines, calculate the intersection points. Figure 4 shows all points we're interested in detecting. We decided to use these points because they usually appear in the type of images we're interested in and, more importantly, they have known, well-defined measures defined by FIFA Standards [1].

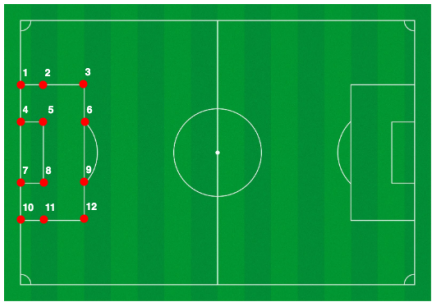


Figure 4: Feature points.

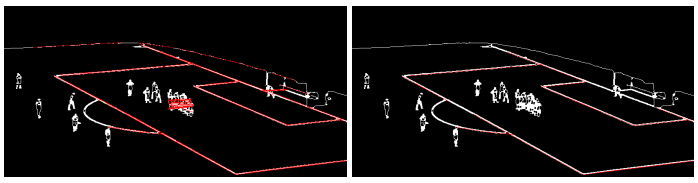
Notice in Figure 4 the points 2, 6, 9 and 11: even though they're not corners of the area, we can still use them as input points for the computation of the homography. This is specially useful if some of the lines detected are somewhat skewed in relation to the true lines of the area i.e. there are more points available to compensate for errors.

2.3.1 Detecting Lines

The first step in the algorithm is detecting the areas' lines. We applied the Canny algorithm to the pitch (using the previously computed field mask) with threshold values of 30 and 100. We also build an auxiliary image by applying a closing morphological operation using a 3x3 rectangular structuring element. Then, we apply OpenCV's *HoughLinesP* algorithm to the canny image to get lines in the image. For each detected line, we calculate two measures to make sure the detection is desirable. The measures are calculated in the auxiliary image - in our experiments, it significantly improved the results.

Firstly, we calculate the line's support. We start by creating a 3 pixel tall bounding box around the line and calculate the number of white pixels in that region. The support is then defined by the number of white pixels divided by the region's area. It is expected that lines in the field have a great number of white pixels, thus we keep those with support greater or equal to 0.9.

The second measure we calculate is the black to white ratio. As in the first measure, we create a bounding box around each line, with the difference of now being 12 pixel tall. Then, we count the number of black and white pixels in that region and calculate the black to white ratio (which is obviously defined as the number of black pixels divided by the number of white pixels). We keep only lines with a black to white ratio greater or equal to 3/2.



(a) (b)

Figure 5: Line detection using Canny and HoughLinesP: (a) Original results; (b) After filtering.

2.3.2 Grouping and Fitting Lines

OpenCV's *HoughLinesP* algorithm usually outputs several results with similar properties, representing the same area's line. So the next step in our algorithm is grouping those lines: for each result, we calculate the cartesian representation $y = mx + b$. We then group lines with similar values of m and b (tolerance of 0.1 and 10, respectively).

Afterwards, for each group, we use OpenCV's *fitLine* function, which finds the average line of that group. This allows a more robust calculation of each line of interest.

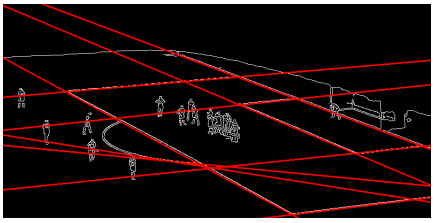


Figure 6: Final computed lines.

2.3.3 Selecting Lines

As Figure 6 shows, there might be more lines than the desired ones. The next step in our algorithm is finding the lines that represent the areas. Once again we group computed lines, but now we're only interested in grouping those with similar m - we used a tolerance of 0.3. Thus, we expect parallel lines to be grouped together, as shown in Figure 7. As we group lines together, we compute the group's average m.

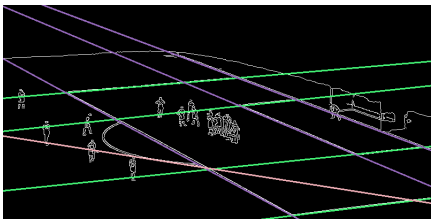


Figure 7: Grouped lines.

In a real pitch, we expect the areas' lines to be parallel and perpendicular between each other (as exemplified by the purple and green lines in Figure 7). Therefore, our next step, for each two groups, is to calculate their cross product using the respective direction vectors (the direction vector can be calculated from the line's slope).

The cross product between two parallel lines is zero. Inversely, the cross product between two perpendicular lines will be the highest value: since parallel lines are grouped together, the maximum cross product between two groups will correspond to the areas' lines.

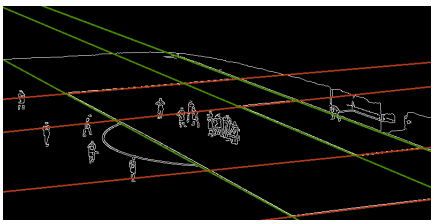


Figure 8: Final lines after filtering by cross product.

2.3.4 Calculating the Intersection Points

Now that we have the lines of the area, we can easily find the intersection points, using the standard mathematical equation $ax + c = bx + d$. We only calculate intersections between the two different groups i.e. horizontal and vertical lines.

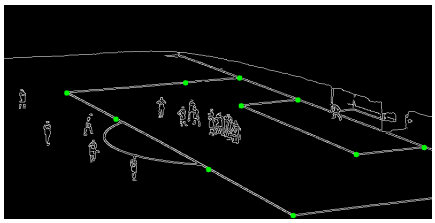


Figure 9: Intersection points.

2.3.5 Identifying Points Automatically

Ideally, we are able to automatically define the ID of each intersection point (as in Figure 4). In our application, it is only possible if we're able to find all the lines that delimit the areas - 4 horizontal and 3 vertical (see Figure 8).

If that's the case, then we only need to know the camera's direction, that is, if it is pointing right or left: if the camera is pointing left, then the IDs correspond to the ones in Figure 4; if to the right, the IDs suffer a 180º rotation (the point with ID = 1 corresponds to the lower right corner).

Let *(vx, vy)* be the direction vector for each group of lines, such that *vx* represents the variation in the x-axis and *vy* the variation in the y-axis. We start by identifying the group that corresponds to the horizontal lines, which will be the group with the largest *vx*. Afterwards, we look at the group's *vy*: if it's greater than 0, then the camera is pointing to the left; else it's pointing to the right.

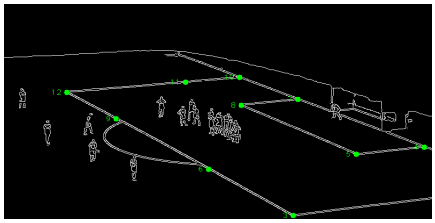


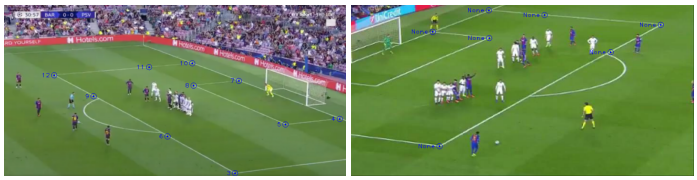
Figure 10: Labeled intersection points.

If the aforementioned conditions are not met, the user will have to input the values of each point. More details are explained in Section 3.

3 Output

When the program is run, any points automatically computed will be displayed onto the image. Here, we provide the chance for the user to confirm if the points are being identified well enough.

If it user deems the points to be miscalculated, it is possible to: drag points to their correct locations, delete others by selecting them and pressing backspace, place new ones by clicking the image, and/or change their IDs by pressing it's equivalent number (for 10, 11, or 12, the user should press Q, W or E, respectively).



(a) (b)

Figure 11: Example of first images displayed to the user after points detection: (a) with automatic identification; (b) without.

3.1 Homography Calculation

Once the user is satisfied with the points, by pressing return, the program resumes to compute the homography.

Since we know the real football pitch dimensions, to get a homography matrix, we need to feed OpenCV's *findHomography* function at least four non co-linear points in both their image coordinates (in pixels) and real coordinates (meters).

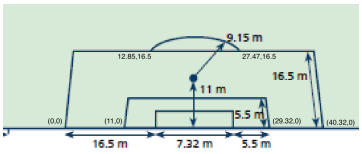


Figure 12: Dimensions of the penalty box and some real point coordinates.

3.2 Free Kick Distance to Goal

After the homography is obtained, the system is ready to draw the visual aids. To be able to draw an arrow from the ball to the middle of the goal, we must first: (i) get the ball real world coordinates from the image coordinates, which are captured when the user clicks the image; then, (ii) since the real world coordinates of the center of the goal are known a priori, we compute the real world distance between the ball and the center of the goal; we then (iv) draw an arrow on a blank image with the real coordinates to then (v) apply a transformation to the arrow's image; finally (vi) add it to the original image.

3.2.1 Ball and Goal's Center Coordinates and in-between Distance

To get ball coordinates, we get the image coordinates where the user clicked with the OpenCV's *EVENT_LBUTTONDOWN* mouse event, and then we transform those into real coordinates with *perspectiveTransform* function and the homography.

The goal's center has well-defined real world coordinates, and since we draw the arrow on real world coordinates, it is not necessary to calculate the image coordinates of the goal's center.

The distance between them is calculated by applying the Euclidean distance to their real coordinates, giving us the total distance in meters.

3.2.2 Drawing the Arrow onto the Original Image

To draw the arrow in the original picture, we first create an empty one and draw the arrow on it with the real coordinates using OpenCV's *arrowedLine* function, so we can then apply the inverted homography with the *warpedPerspective*. After that, we give it some colour with *cvtColor* function and use the *GaussianBlur* function with size 5 so it looks smoother.

Now the arrow is ready to be added to the original image, but first, we apply the *bitwise_not* function onto the players and then we apply the *bitwise_and* function onto the image with the previous mask. This way, once we place the transformed arrow with the *addWeighted* function, the players won't be under the arrow.

Figure 13: Example of the result of drawing the arrow onto the image.

3.3 Minimum Player Wall Distance to Free Kick Spot

To represent the minimum player wall distance to the ball on a free kick situation, we do a very similar process to the previous subsection, but instead of drawing an arrow, we draw a circle and a circumference with OpenCV's *circle* function.

Figure 14: Example of the result of drawing the circle onto the image.

3.4 Offside Line Assistance

The offside line is also has a very similar process to both previous features, since we needed to use the homography to compute how a parallel line should be displayed on a particular point chosen by the user.

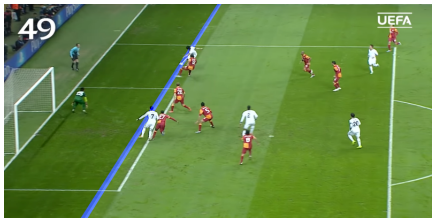


Figure 15: Example of the result of drawing the offside line onto the image.

4 Conclusions and Future Work

In this work, we developed an application that automatically detects multiple objects of interest in a football game image, including the pitch, players and area corners (to produce an homography). Furthermore, we allow the user to add, move or delete detected points in order to acquire better accuracy. Finally, the application is able to draw the offside line, the free kick arrow with respective distance, and also the minimum player wall distance.

Our belief, from developing this work, is that the process of mapping between image and real world coordinates in order to draw on the image is a relatively simple one, after having the homography. We consider that automatically detecting the homography is a difficult task because, even when analyzing specific events in a football match, there are many different situations, locations, and angles to take into consideration. This makes it hard to always be sure which lines on the pitch are being tracked (even if being tracked at all).

We believe that Canny edge detection followed by Hough transform provide satisfactory results, given that appropriate heuristics are used to filter undesired results. We find that the results of this application could be improved by applying a better grouping algorithm before applying the cross product (described in section 2.3.3). In our experience, the cross product does a very good job at finding the most orthogonal group of lines, which usually correspond to the areas, but some of these lines are sometimes lost in the aforementioned grouping.

Finally, we also believe that detecting the field and players by segmenting the color green is a very easy and fast method that provides good results.

References

[1] Laws of the game 2018/19. https://www.fifa.com/about-fifa/what-we-do/education-and-technical/referees/laws-of-the-game/. Accessed: 22-12-2019.

[2] Changing colorspaces. https://docs.opencv.org/trunk/df/d9d/tutorial_py_colorspaces.html, . Accessed: 18-12-2019.

[3] Homography. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga4abc2ece9fab9398f2e560d53c8c9780, . Accessed: 22-12-2019.