UNIVERSITY OF BUCHAREST

The Faculty of Mathematics and Computer Science

DISSERTATION THESIS

An AI for football using computer vision

Coordinator :

Conf. dr. Marius Popescu

Student :

Paduraru Miruna Gabriela

Bucharest 2018

Table of Contents

I. Introduction

I.1 Objective

Machine learning and understanding of human actions is a complex, diverse and challenging area that has received much attention within the past years.

Intelligent visual surveillance has got more research attention and funding due to increased global security concerns and an ever increasing need for effective monitoring of public places such as airports, railway stations, shopping malls, crowded sports arenas, military installations, etc., or for use in smart healthcare facilities such as daily activity monitoring and fall detection in old people's homes.

Modeling human behaviors and activity patterns for recognition or detection of special event has attracted significant research interest in recent years.

I.2 Motivation

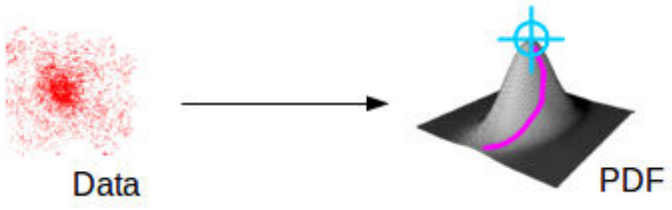
Imagine how would be if you could detect the human behaviors in real football matches or in FIFA game: when do they shoot or pass or tackle or when they keep running.

Video processing has found many applications in team sports analysis, statistics collection and video archiving. Rapid developments of imaging technology and increasing video processing power have been intensively improving the development of sports video analysis tools. Those tools include the extraction of player technique, tactics and furthermore the decision making about the game. Therefore sport videos provide an excellent means of observing and analyzing the games.

In ball games more complex rules are applied to determine the winner. Therefore annotating video indexing becomes a more difficult task which requires the position of the ball as well as the players. During a typical game the attention of the audience as well as the players is always focused on the ball. The position of the ball determines the movement of the players which has direct effect on the game events and their outcome. On the other hand, the possession of the ball provides the information on which team has better control of the game that also conclude in the decision of a team applying their tactics in a better way.

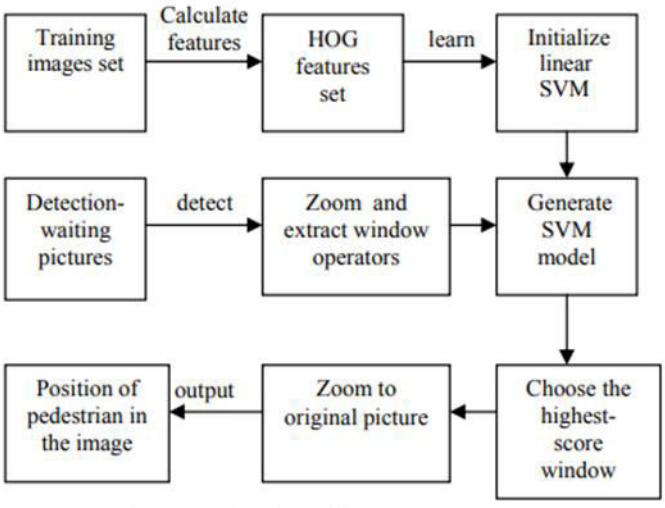
I.3 Tracking algorithms try

1. Mean Shift Tracking – is a non-parametric feature space analysis technique, a so-called mode seeking. It is a procedure for locating the maxima of a density function given discrete data sampled from that function. In a sense, it is using a non-parametric density gradient estimation. It is useful for detecting the modes of this density.



Disadvantage of this method is inappropriate window size can cause modes to be merged, or generate additional "shallows" modes. In that case we need to use adaptive window size. Tried algorithm was from OpenCV. It contains mean-shift implementation via cvMeanShift Method.

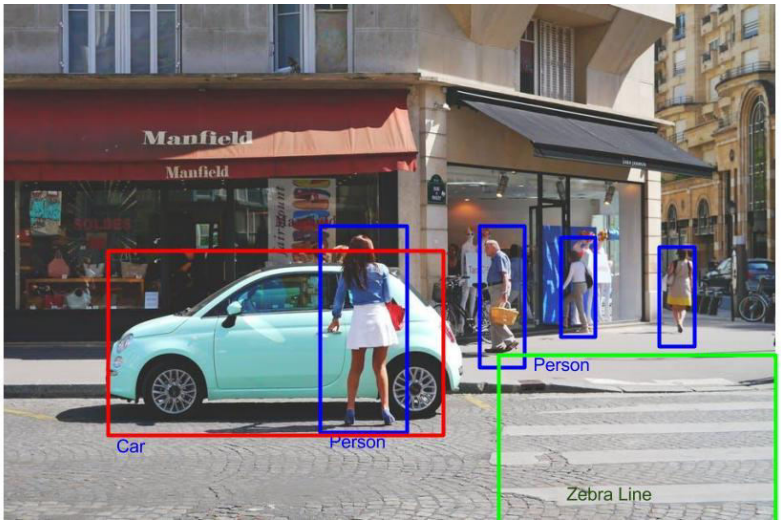
2. The histogram of oriented gradients (HOG) - is a feature descriptor used in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.



The disadvantage is that the final descriptor vector grows larger, thus taking more time to extract and to train using a given classifier. Also, another disadvantage is detection of small objects because the target window should be zoom in more and so the time is increasing more and more noise is added to the result.

3. Single Shot Multibox Detector

The SSD approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections.



Disadvantage for this method is processing time that is 7 FPS, very slow for video processing.

I.4 Documentation structure

Documentation contains an *OpenCV and technologies used* chapter that explain what is OpenCV, HSV vs RGB space difference and presents all functions used in project from OpenCV.

Another chapter presents application *layers* in this order: *Computer Vision Layer*, *Decision making Layer*, *Input Sending Layer*.

And the last chapter presents *instructions to run and debug* application made.

II. OpenCV and technologies used

II.1 What is OpenCV?

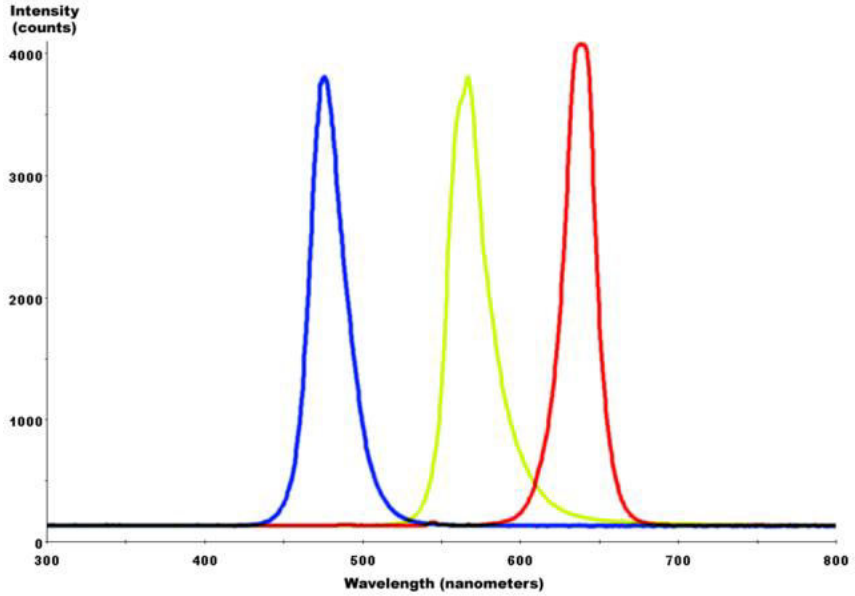
OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. The library has more optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc.

II.2 HSV vs RGB space

Why do we use the HSV color space so often in vision and image processing?

RGB is an additive color model. It means that different proportions of Red, Blue and Green light can be used to produce any color. The RGB color model was created specifically for display purpose (display screens, projectors etc.).

A key concept one should understand is that RGB is simply an additive combination of narrow band waves. Consider the following figure:



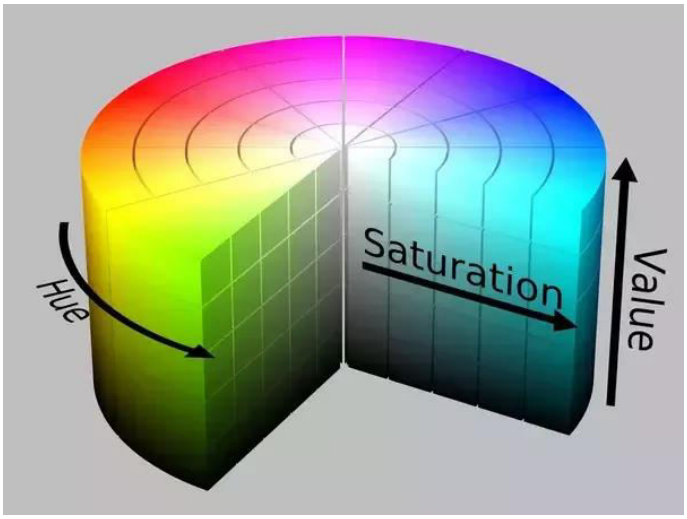
Any combination of the above given narrow band waves can produce a different color. It can be simulated any color using any combination of spectra, but when it is chosen to use combinations of the above shown narrow band **RGB** spectra, it means to work on the **RGB** color space.

HSV

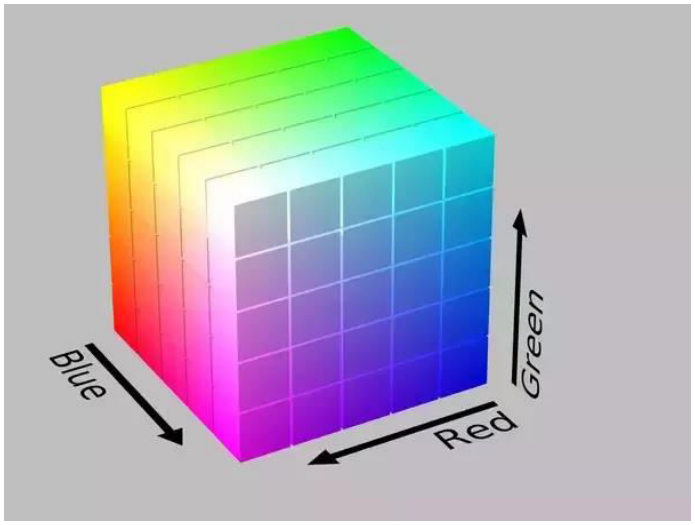
In computer vision is often wanted to separate color components from intensity for various reasons, such as robustness to lighting changes, or removing shadows.

This is very useful in many applications. For example, if is desired to do histogram equalization of a color image, probably it is wanted to do that only on the intensity component, and leave the color components alone. Otherwise it will get very strange colors.

This space transforms the **RGB** space into a more convenient representation.



VS



The "Whiteness" / "Lightness" is a function of R, G and B. When we view it in the cube, there is a separate dimension known as Value (V) dedicated towards “Whiteness”. In the **RGB** cube, the Saturation or "colorfulness" is given by the distance of your color from the 3D diagonal. In the **HSV** model, the Saturation is directly given as another dimension.
HSV is named as such for three values: *hue*, *saturation*, and *value*. This color space describes colors (**hue** or tint) in terms of their shade (**saturation** or amount of gray) and their **brightness value**.

Hue is the color portion of the color model, and is expressed as a number from 0 to 360 degrees. **Saturation** is the amount of gray in the color, from 0 to 100 percent. A faded effect can be had from reducing the saturation toward zero to introduce more gray. **Value (or Brightness)** works in conjunction with saturation and describes the brightness or intensity of the color, from 0-100 percent, where 0 is completely black and 100 is the brightest and reveals the most color.
In this application **cv::cvtColor()** was used to convert from BGR to HSV color space for all images used.

```
// Transform image to HSV format
cv::cvtColor(inputImage, inputImage, CV_BGR2HSV);
```

II.2 OpenCV functions used in project

1) Probabilistic Hough Transform is a popular technique to detect any shape, if you can represent that shape in mathematical form. Probabilistic Hough Transform is an optimization of Hough Transform, it doesn’t take all the points into consideration, instead take only a random subset of points and that is sufficient for line detection.

How it works?

A line can be represented as y = m*x+c or in parametric form, as $\rho = x\cos\theta + y\sin\theta$ where ρ is the perpendicular distance from origin to the line, and θ is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise.
Any line can be represented in these two terms, (ρ, θ). So first it creates a 2D array or accumulator (to hold values of two parameters) and it is set to 0 initially. Let rows denote the ρ and columns denote the θ. Size of array depends on the accuracy you need. Suppose you want the accuracy of angles to be 1 degree, you need 180 columns. For ρ, the maximum distance possible is the diagonal length of the image. So taking one pixel accuracy, number of rows can be diagonal length of the image.

Consider a 100x100 image with a horizontal line at the middle. Take the first point of the line. You know its (x,y) values. Now in the line equation, put the values $\theta = 0,1,2,....,180$ and check the ρ you get. For every (ρ, θ) pair, you increment value by one in our accumulator in its corresponding (ρ, θ) cells. So now in accumulator, the cell (50,90) = 1 along with some other cells.

Now take the second point on the line. Do the same as above. Increment the the values in the cells corresponding to (ρ, θ) you got. This time, the cell (50,90) = 2. What you actually do is voting the (ρ, θ) values. You continue this process for every point on the line. At each point, the cell (50,90) will be incremented or voted up, while other cells may or may not be voted up. This way, at the end, the cell (50,90) will have maximum votes. So if you search the accumulator for maximum votes, you get the value (50,90) which says, there is a line in this image at distance 50 from origin and at angle 90 degrees.

First parameter of function is Input image should be a binary image, second parameter is output vector of lines. Third and fourth parameters are ρ and θ accuracies respectively. Fifth argument is the threshold, which means minimum vote it should get for it to be considered as a line. The sixth parameter is minimum length of line (line segments shorter than this are rejected) and the last parameter is maximum allowed gap between line segments to treat them as single line.

In this project the **Probabilistic Hough Transform** was used to find white edges inside pitch.

```
// Find the white edges inside pitch
cv::HoughLinesP(hsv_input_image, output.m_outPitchLines, 1, CV_PI / 180, hough_threshold,
hough_minLineLength, hough_maxLineGap);
```

2) Hough Circle Transform - works in a roughly analogous way to the Hough Line Transform explained above. In the line detection case, a line was defined by two parameters (ρ, θ), in the circle case, there are three parameters : C (ρ, θ, r), where (ρ, θ) define the center position and r is the radius.

In the project hough circle transform was used to find the ball coordinates.

```
cv::HoughCircles(ballImgGray, circles, cv::HOUGH_GRADIENT, 1, 50, cannyThreshold,
GlobalParameters::MIN_VOTES_TO_CONSIDER_BALL_CENTER, minRadiusForCircle,
maxRadiusForCicle);
```

ballImgGray – input image (grayscale)

circles – a vector that stores sets of 3 values: (ρ, θ, r) for each detected circle

`cv::HOUGH_GRADIENT` - define the detection method

dp = 1 - the inverse ratio of resolution

min_dist - 50 : minimum distance between detected centers

cannyThreshold – upper threshold for the internal Canny edge detector

`GlobalParameters::MIN_VOTES_TO_CONSIDER_BALL_CENTER` – threshold for center detection

minRadiusForCircle – minimum radius to be detected

maxRadiusForCircle – maximum radius to be detected

3) Polygon approximation with cv::approxPolyDP (Douglas-Peucker algorithm)

If we are engaged in shape analysis, it is common to approximate a contour representing a polygon with another contour having fewer vertices.

cv::approxPolyDP is an implementation for approximate a contour representing a polygon from OpenCV. This functions acts on one polygon at a time, which is given in input curve. The output of it will be placed on the approximate curve output array.

```
void cv::approxPolyDP(
  cv::InputArray  curve,       // Array or vector of 2-dimensional points
  cv::OutputArray approxCurve, // Result, type is same as 'curve'
  double          epsilon,     // Max distance from 'curve' to 'approxCurve'
  bool            closed       // If true, assume link from last to first vertex
);
```

The parameter *epsilon* is the accuracy of approximation you require. The meaning of the epsilon parameter is that this is the largest deviation you will allow between the original polygon and the final approximated polygon. *Closed*, the last parameter, indicated whether the sequence of points indicated by curve should be considered a closed polygon. If set to true, the curve will be assumed to be closed (i.e., the last point is to be considered connected to the first point).

In the project the **cv::approxPolyDP** was used to detect the controlled player markers.

```
// Approximate the contour poly and see if we can get a triangle out of it
const Contour& selectedContour = contoursInSubImage[subContourIndex];
Contour postProcessedSubContour(selectedContour.size());
cv::approxPolyDP(selectedContour, postProcessedSubContour, 0.07f *
cv::arcLength(selectedContour, true), true);
```

`cv::arcLength( InputArray curve, bool closed );` - this function is used to computes a curve length or a closed contour perimeter, *curve* – input vector of 2D Points, *closed* – flag indicating whether the curve is closed or not.

4) Thresholding Operations using cv::inRange

OpenCV's "inRange" function checks if **low < pixelvalue < high**, and if it is, then the pixel passes the test and is turned white, else it is turned black. **cv::inRange** function is useful when thresholding for certain colors, as it is more than a simply high pass filter.

In the project **cv::inRange** function was used to make a mask to use in finding the ball (who is chosen to be white in this prototype).

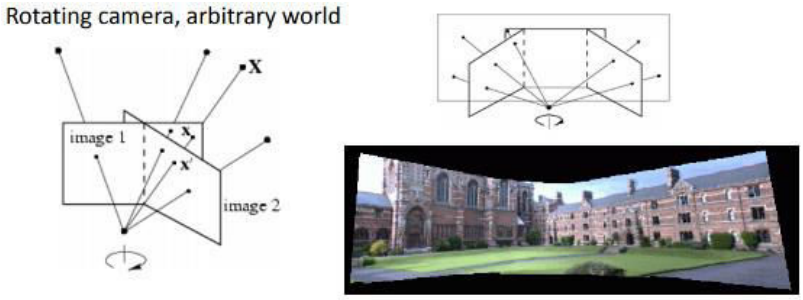
```
cv::Scalar lower_white(minWhite[0], minWhite[1], minWhite[2]);
cv::Scalar upper_white(maxWhite[0], maxWhite[1], maxWhite[2]);
cv::inRange(hsvInputImage, lower_white, upper_white, src2_hsv_mask);
```

5) Homograph using cv::findHomography()

If it is considered a point x = (u,v,1) in one image and x'=(u',v',1) in another image; a homograph is a 3 by 3 matrix M. When applied to every pixel the new image is a warped version of the original image.

Two images are related by a homography if and only if:

- ➔ Both images are viewing the same plane from a different angle
- ➔ Both images are taken from the same camera but from a different angle



cv::findHomography is a function from OpenCV to find the transform between matched keypoints.

```
CV_EXPORTS_W Mat findHomography( InputArray srcPoints, InputArray dstPoints,
                                int method = 0, double ransacReprojThreshold = 3,
                                OutputArray mask=noArray(), const int maxIters = 2000,
                                const double confidence = 0.995);
```

srcPoints - Coordinates of the points in the original plane

dstPoints - Coordinates of the points in the target plane

method - Method used to compute a homography matrix. The following methods are possible:

- "0" - a regular method using all the points, i.e., the least squares method
- "RANSAC" - RANSAC-based robust method
- "LMEDS" - Least-Median robust method
- "RHO" - PROSAC-based robust method

ransacReprojThreshold - Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC and RHO methods only)

mask - Optional output mask set by a robust method (RANSAC or LMEDS)

maxIters - The maximum number of RANSAC iterations

confidence - Confidence level, between 0 and 1

In the project **cv::findHomography** function was used to get the homograph matrix - a conversion between image coordinates to real pitch coordinates.

```
m_homographMatrix = cv::findHomography(imageKeypoints, realStadiumKeypoints, CV_RANSAC);
```

6) cv::imshow – this function displays an image in the specified window

```
void imshow(const String& winname, InputArray mat);
```

winname - Name of the window

mat - Image to be shown

7) cv::bitwise_not – calculates per-element bit-wise inversion of the input array

```
void bitwise_not(InputArray src, OutputArray dst, InputArray mask = noArray());
```

src - input array

dst - output array that has the same size and type as the input array

mask - optional operation mask, 8-bit single channel array, that specifies elements of the output array to be changed

8) **cv::findContours** – the function retrieves contours from the binary image

```
void findContours( InputOutputArray image, OutputArrayOfArrays contours,
                   OutputArray hierarchy, int mode,
                   int method, Point offset = Point());
```

image - source

contours - Detected contours. Each contour is stored as a vector of points

hierarchy - Optional output vector, containing information about the image topology

mode - Contour retrieval mode

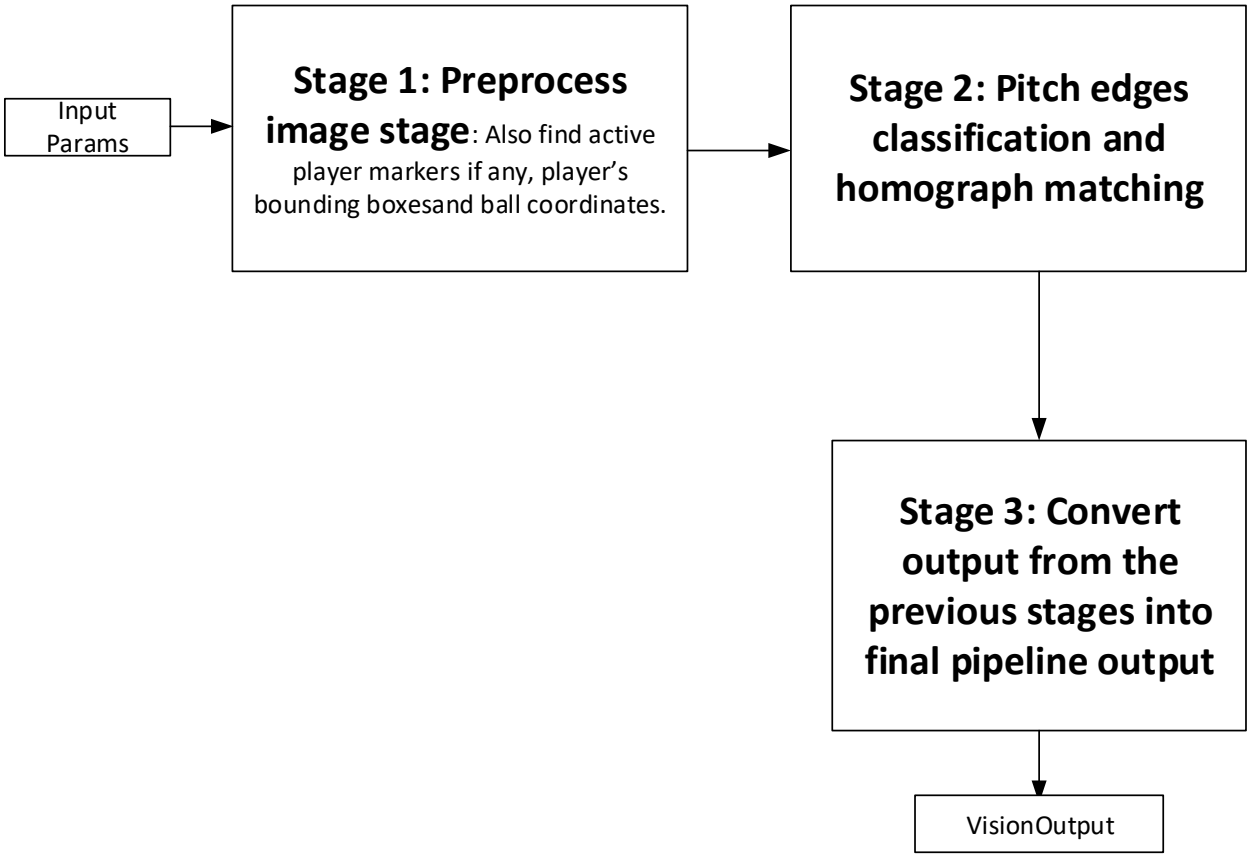
method - Contour approximation method

offset - Optional offset by which every contour point is shifted

III. Layers

III.1 Computer Vision Layer

The computer vision layer is a pipeline process. The input *params* contain the captured input image that must be processed. *The first stage* preprocesses the image and also finding active player markers if any, player's bounding boxes and ball coordinates, while *the second stage* processes the white pitch edges from the field, classifies them then computes the homograph matrix. The final output (*VisionOutput*), is obtained by converting the coordinates in the image space to the real world coordinates.



A. Stage 1

This has two parts. *The first one* does some image preprocessing, identifies pitch lines and removes them from the input image, detects the ball coordinates if any. *The second one* is responsible mainly for finding players' bounding boxes. The output of the first one is used as input for the second one, and both contribute for the final output of Stage 1. Figures **Fig. A11** and **Fig. A12** show these two processes. Note that in both diagrams, a link from one box to another represents an image transform stage. The state of the image in the flow is defined by the text in the boxes.

A1. The first part of Stage 1

Fig. A1: Stage1, first processing step transform flow.

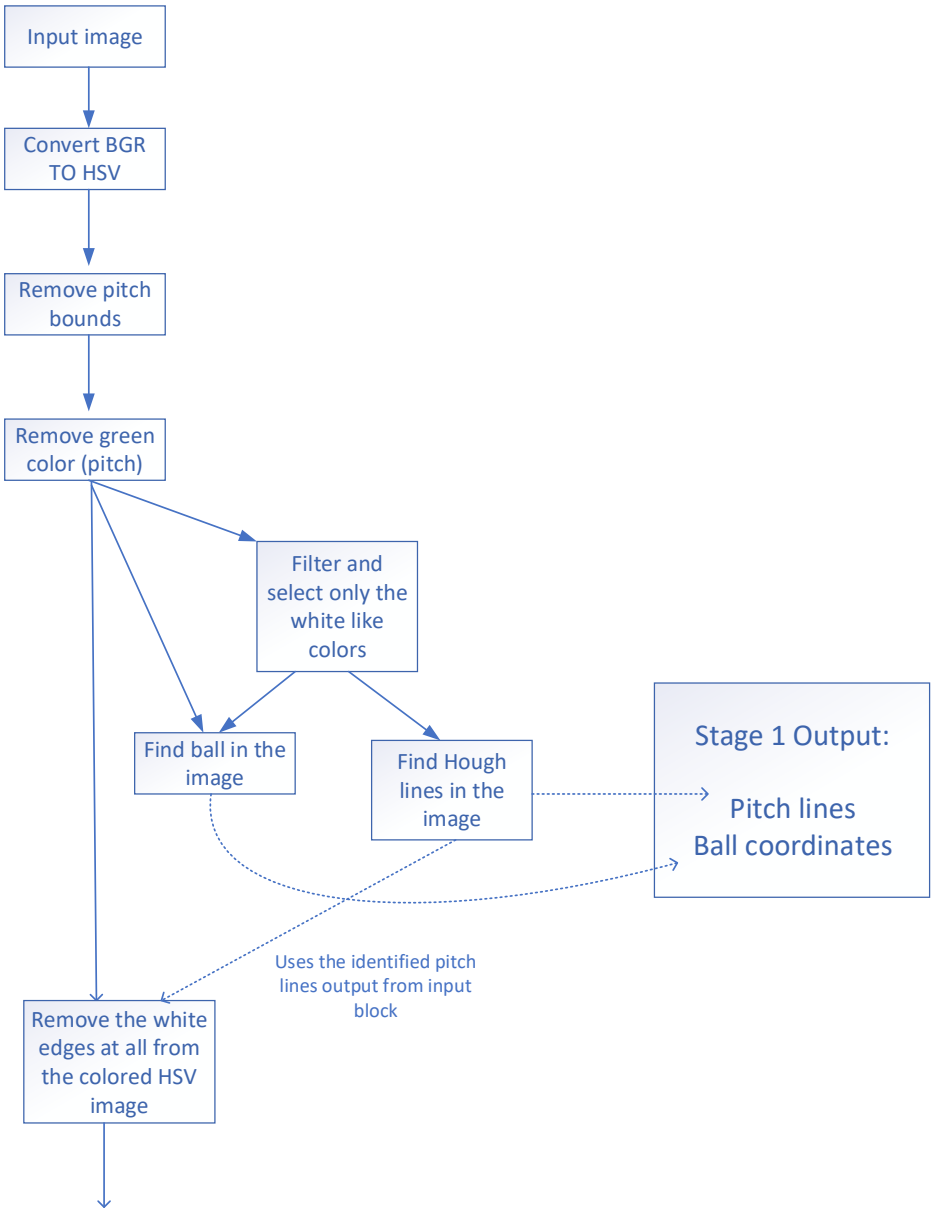


Fig. A11 shows the data flow transformations from the input image sent as parameters for the first part of Stage 1, to the output being used for the second part of the same stage.

The first step is to transform the input image given in the RGB format to the HSV space. HSV is better suited for colors classification because it can separate the low vs high light conditions better**. Figure HSV** shows how a sample image looks like after transform.

Fig. HSV space

The next step is to remove the useless boundaries, like crowds and everything else that is not part of he field. To do so, we draw scanlines on both columns and rows, left-right and up-down to identify the continuous group of pixels that are not part of the field.

For each row R

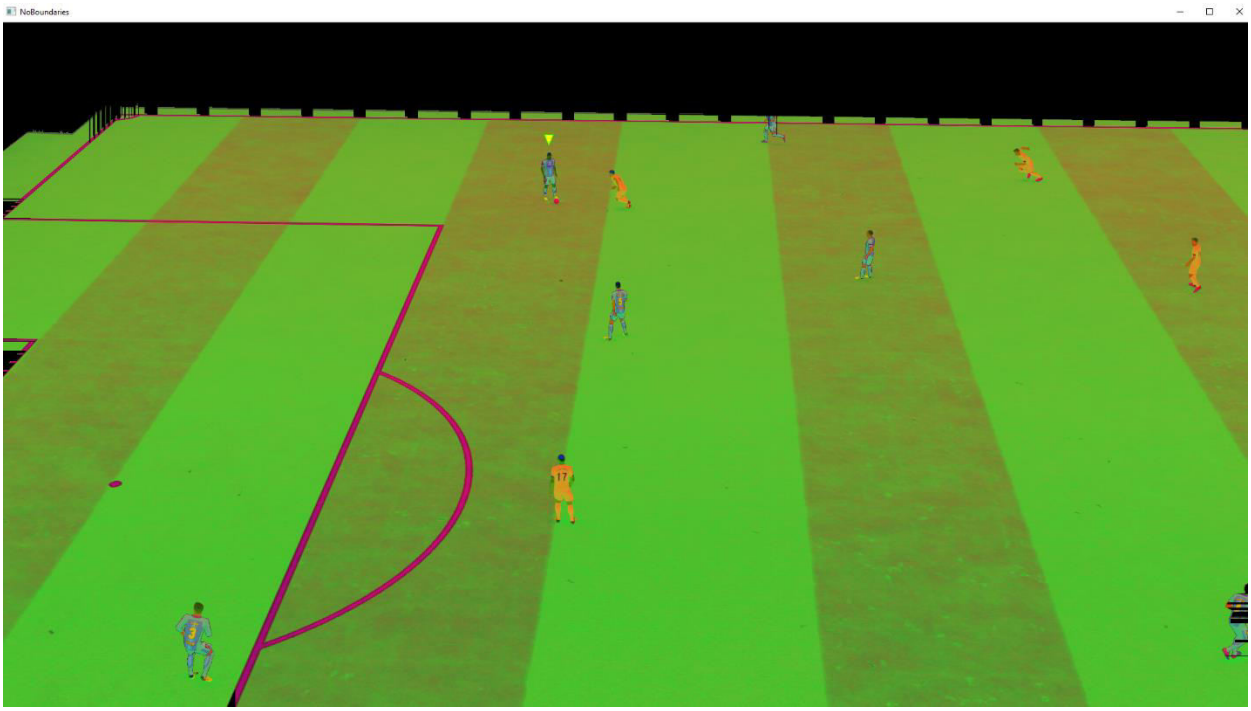
Cmin = search first pixel that is in field left to right

Cmax = search first pixel that is in field right to left.

Keep only pixels from [**Cmin**, **Cmax**], make others black.

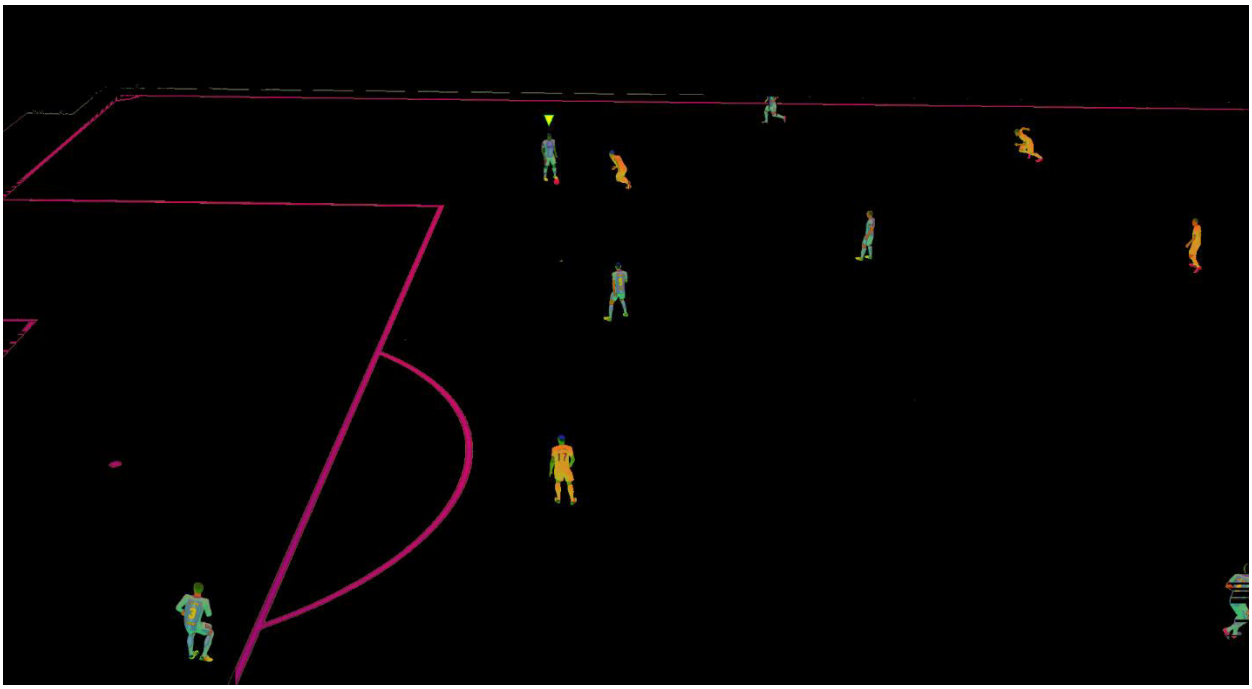
A similar code is used for columns cutting. A pixel is in the field if the previously consecutive iterating pixels counting shows more than a fixed threshold of green pixels, or white ones (possibly starting pitch edge pixels). The result is depicted in **Fig. NoBoundaries.**

Fig. noBoundaries



The next step is to remove the green color at all from the image. We do this by settings pixels on full black color (zero) if the pixel's **Sat Hue Val** values are in the green range. In code, there is a lookup table with ranges for base color clusters (white, black, red, yellow, blue, green, orange). The result can be observed in **Fig. No green**.

Fig. No green

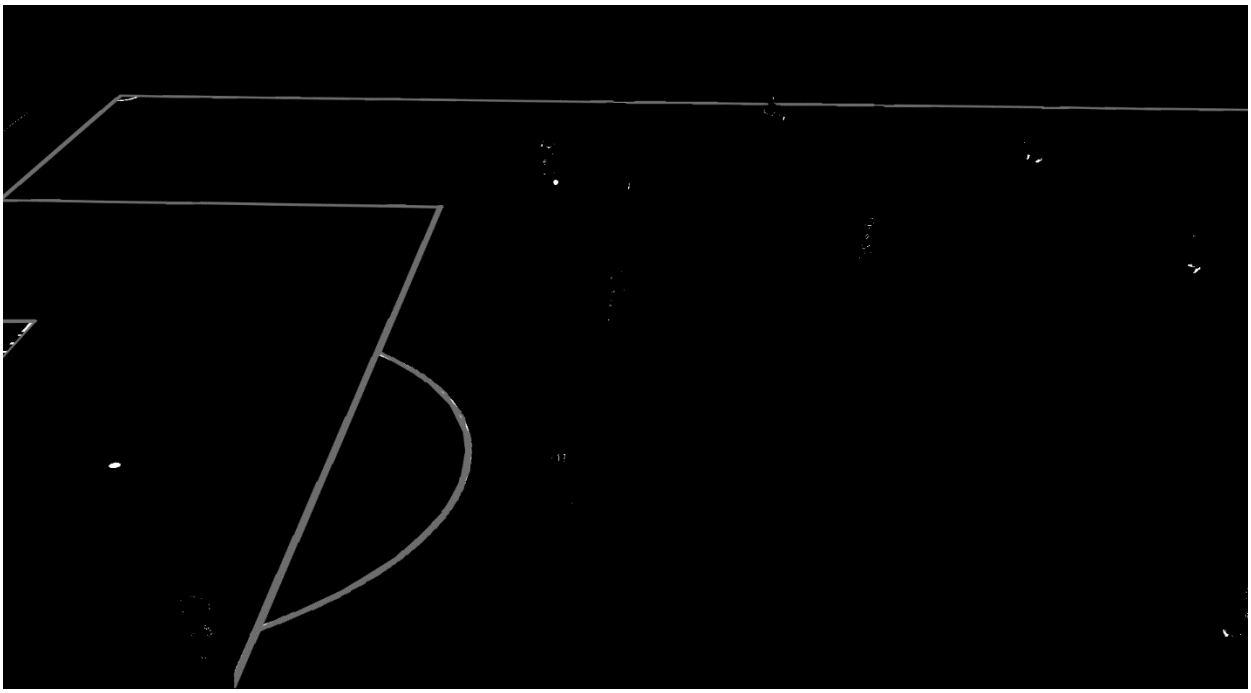


The next step, as shown in **Fig. HSV Mask**, is to create an image by filtering out the other colors than white from the input one, from the previous step. Note that the transformation flow does a fork at this point (also keeping the image in HSV format from previous steps).

Fig. HSV mask

Using this input image we use **Probabilistic Hough Transform** to find the pitch edges and store them in the final output of **Stage 1 (Fig. Hough lines)**.

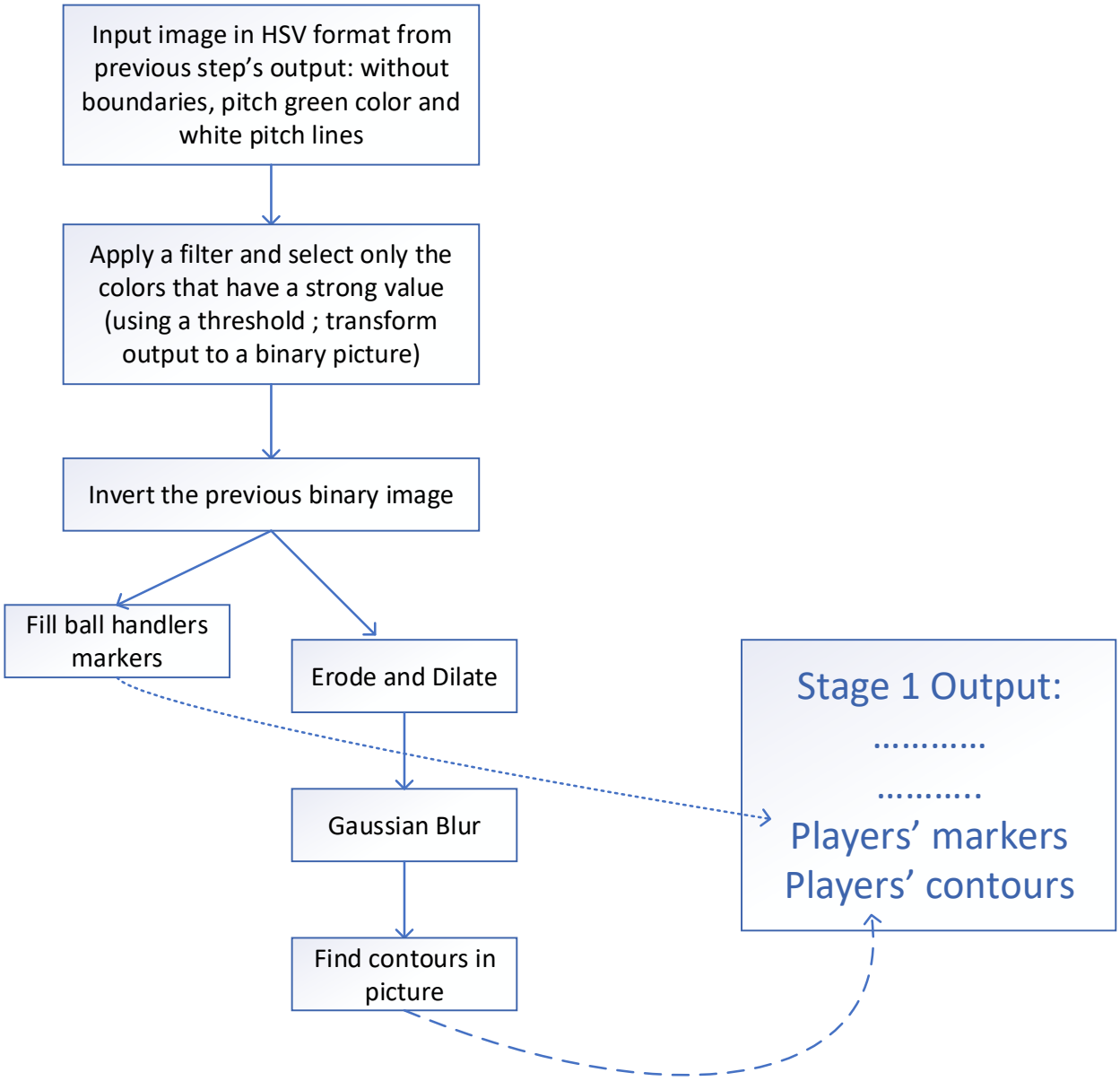
Fig. Hough lines



Before leaving this part, the pitch edges are removed from the image in HSV format. The result, which also consist as the second part of **Stage 2**, is depicted in **Fig. noLines**.

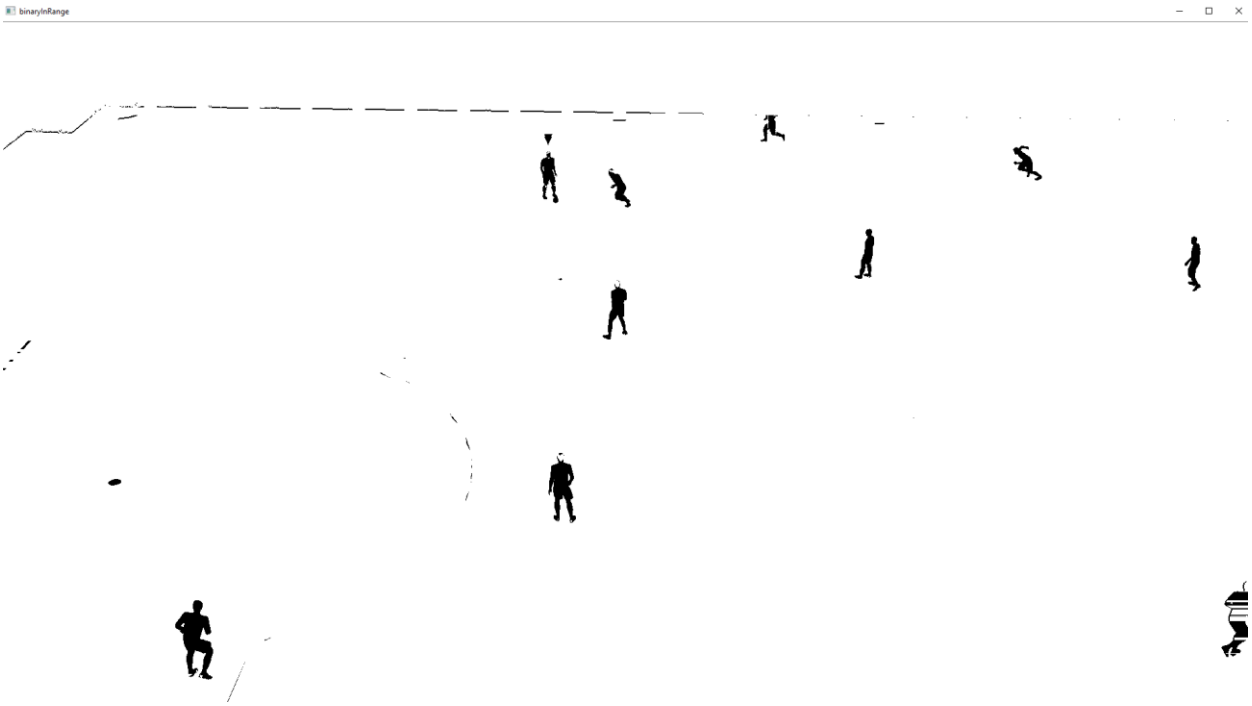
Fig. noLines

Fig. A12: The transformations flow in the second part of **Stage 1**

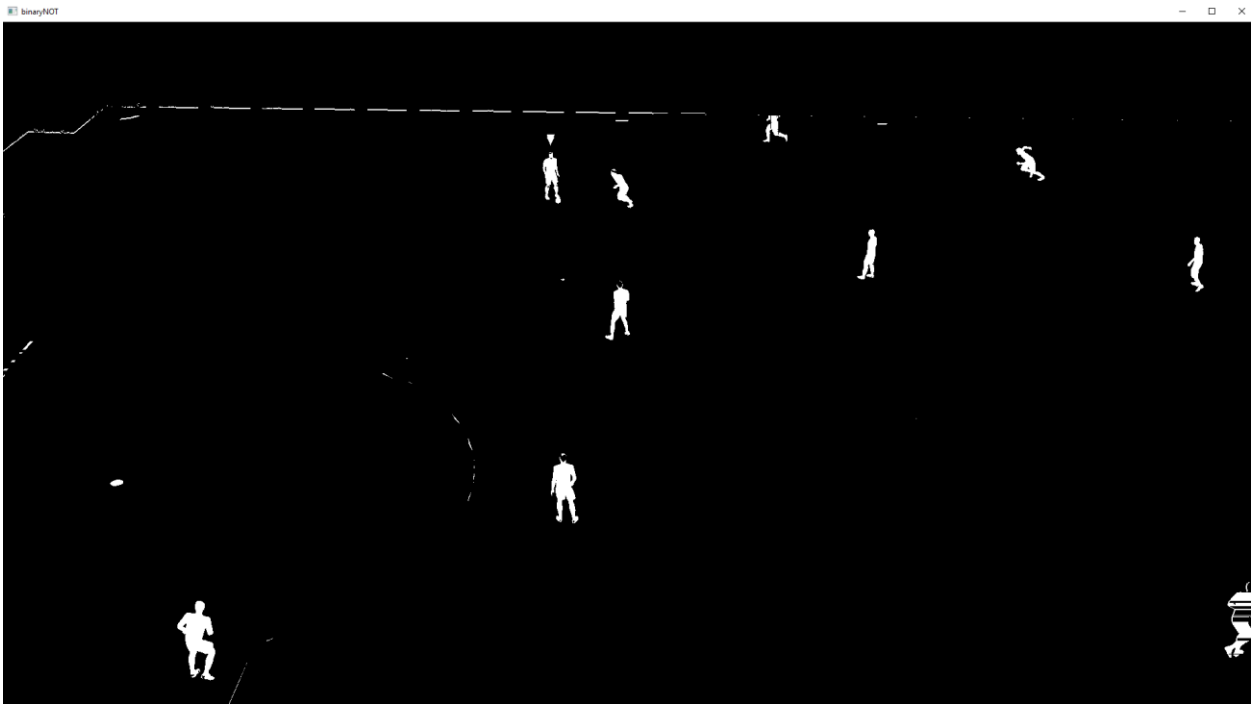


The first transformation step is to convert the input image to a binary one using a threshold given **cv::inRange**, which results in a binary image like in **Fig. Binary**. The purpose is to eliminate all colors that are not strong enough (i.e. noise reduction).

Fig. Binary



Then, a **cv::bitwise_not** is applied such that the colors are inverted (**Fig. notBinary**).



The next left side in the transformation flow (i.e. finding players markers) is left as a special topic at the end of this section, se we'll focus on the right branch for now.

The next two steps are doing **Erosion** and **Dilation** operation with rectangular masks of (4,4) respectively (7,7). (**Fig. After Erosion**, **Fig. After Dilation**).

Fig. After Erosion

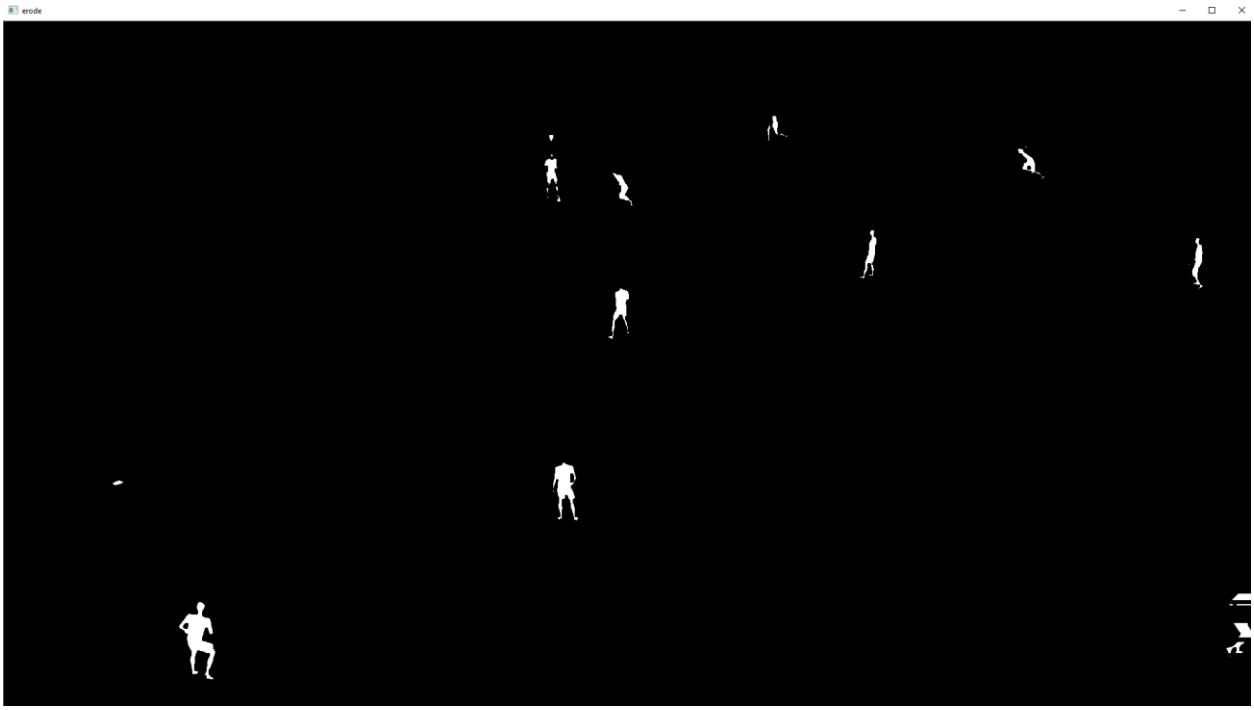
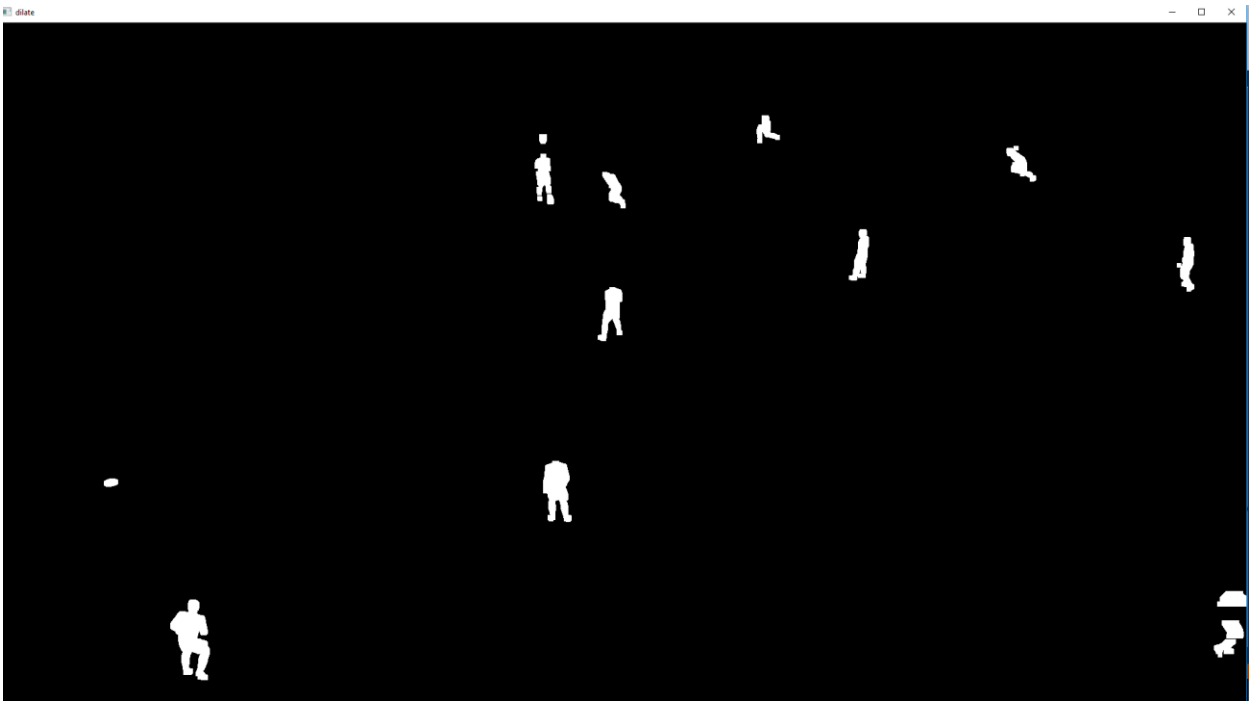
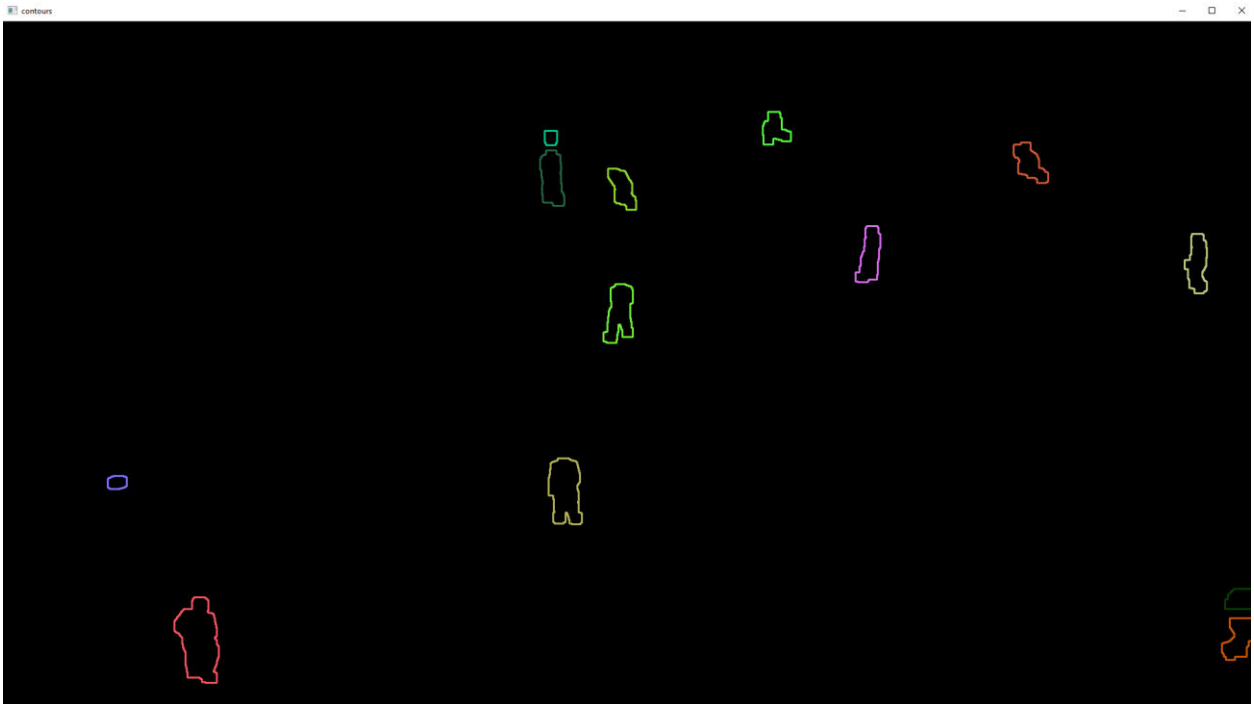


Fig. After Dilation



A Gaussian Blur is then used to smooth the pixels and eliminate even more noise. Finally using **cv::findContours** we find the contours on the image and it looks like a contour usually represent a player's boundary (**Fig. Contours**).

Fig. Contours



Note that the player markers finding is specific to FIFA game and not happening in a real football match. Our layers below Computer Vision one can work correctly in both versions, but if the player marker is found then there is a slightly advantage in recognizing who is the ball handler. In FIFA, a red triangle will appear above the ball handler's of the home team and a blue triangle for the away team's ball handler.

The operation is based both on the initial input HSV colored image sent as input and on the binary figure from **Fig. notBinary**. A contour finding method using **cv::contour** is applied then (TODO: One thing to note is that we do this operation twice, so performance wise, we should gather the operation under the same code and do it only once). These contours resulted are polygonal shapes and are then analyzed for length using **cv::arclength**, which computes the length of each polygon. If this is above a threshold then it is probably not a valid one. If not, the next elimination step is to check if we can approximate the polygonal contour to a triangle using **cv::approxPolyDP**. If still passes the test, perform a color histogram in the colored image, in the same area where we found the poly in the binary image. If it is mostly red or blue color, then assign the marker to the corresponding team, otherwise it means that that is a false contour for a player marker.

```
AllContours [] = Find Contours in the binary image
For each Poly in AllContours
    Length = cv::arclength(Poly)
    If Length > T  // too big to be a marker
        continue
    PolyApprox = cv::approxPolyDP(Length*factor); // Try to simplify the polygon's points
    If PolyApprox.size() > 3   // Not a triangle ? can't be marker
        Continue
    Histogram H (ColoredHSVImg, PolyApprox);
    If H.MainColorPercent  > T
        If H.MainColor == 'Red'
            Stage1.output.AddPlayerMarker(PolyApprox, HOME);
        Else if H.MainColor == 'Blue'
            Stage2.output.AddPlayerMarker(PolyApprox, AWAY);
```

The ball finding algorithm uses the input images **Fig. HSV Mask** and **Fig. NoGreen**, Main idea is to find the contours in HSV Mask image and search within them the ball. We use two search sizes for contours lengths in identifying the false positive contours. First we try a small one and if not found, use a bigger threshold. This is done for optimization purposes since. The second one is needed since the ball is sometimes attached to a player's foot and this could

mean that ball can hide within a bigger contour. Most of the time it won't so the first size layer will be enough. Rejecting so many individual contours to check is a real optimization.

On each of these contours we run a Hough circle detection algorithm. If found, we sort them by sizes and eliminate some other false positive results using radius heuristics and histograms of colors using the **Fig. NoGreen** image input – which is in HSV space and still contains some colors. If the histogram doesn't show more than a threshold of white color then it isn't the ball. For example, this eliminates the players' heads that were found in different contours than their body (however this rarely happens).

How do we find the ball?

Step 1: The ball is known to be mostly white (at least in this prototype)

Dilate its pixel to obtain a regular contour - fill the black / orange markers on the ball

Step 2: Segment mask given and find all the contours with **cv::findContours** function

For all contours

Get the bounding box of the contour with **cv::boundingRect** function

Check the contour size

Check if contour is in the target rectangle

See if this is a circle in the initial image with **cv::HoughCircles** function

For all circles found

Check the area around ball – we expect a white color

Check if not outside image

Push the circles in the output

B. Stage 2

This stage receives uses as input the lines obtained from Stage 1's pitch lines (obtained through Hough line finding probabilistic method). Overall the pseudocode below defines the steps done in this stage:

B1. Classify segments in two sets: horizontal and vertical, using slope thresholds and filtering out segments that are not in the dominant slope bins. Output: HorizonalSegments, VericalSegments sets.

B2. Cluster segments that are part of the same pitch feature (e.g. endlines, sidelines, box, small box, middle line). Run this step for both HorizontalSegments and VerticalSegments sets.

B3. Select the most visible clusters that we can trust that could mean something useful.

B4. Classify clusters found at point B3 and find the concrete pitch features

B5. Use features found at point 4 to compute a homograph matrix

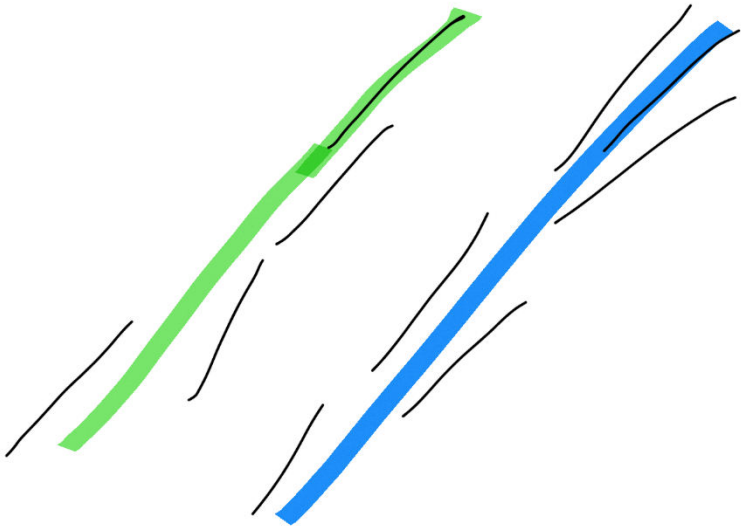
The text below discusses these steps in more details:

B1. The purpose of this step is to classify the pitch segments found in the previous step in two sets: Horizontal and Vertical segments. In the first part we classify them by simply comparing the slope of the segment with a threshold. If higher than that, the it is a vertical segment (e.g. 45 deg). The second part puts the segment in bins according to the slopes found. Only the segments corresponding to the dominant slopes are kept, the others are discarded. Note that dominants slopes finding is based on the sum of lengths of all segments in each bin. Lengthiest bin's segments are chosen for both sets and the rest is discarded.

```
For each segment S in InputSegments
    If slope(S) > T
        VerticalSegments.add(S)
    Else
        HorizonalSegments.add(S)
// Computes the histograms of slopes and find the longest bin cluster
HistSlopesHorizontal = Histogram of slopes in HorizonalSegments
HistSlopesVertical = Histogram of slopes in VerticalSegments
DominantBinHorizonal = HistSlopesHorizonal.argmax(total dimension of segments in
bin);
DominantBinVertical = HistSlopesVertical.argmax(total dimension of segments in bin);

// Keep only the segments in the dominant cluster
HorizonalSegments = HistSlopesHorizonal[DominantBinHorizonal]
VerticalSegments = HistSlopesVertical[DominantBinVertical]
```

B2. This step groups segments that are part of the same feature in the same clusters (similarity with connected components). The heuristic used is that two segments are in the same connected component if their slope don't differ too much and if the start/ end points seems to be on the same direction. The purpose is to eliminate the noise in the output of Hough line detection algorithm, which might see discontinuities in the white edges in the pitch. An example can be found in Fig. SegmentsDiscontinuities.

Fig. SegmentsDiscontinuities



The pseudocode of this process is depicted below. A representative segment for each cluster is chosen to simplify computations later. When multiple segments are belonging to a cluster, we extend the base segment using projections of new segments' endpoints onto it and extending the endpoints of the representative segment using the projection results. The pseudocode below is applied for both horizontal and vertical segments.

```
// Slope of a segment
   slope(S)= ((S.end0.Y – S.end1.Y) / (S.end0.X – S.end1.X))
                  // if diff on X is 0 dx+= eps

   // Distance from one segment to another
   // Dist(S,D) = min(dist(S.start, D), dist(S.end, D))

   Comp[] = -1 // The index of the cluster where each segment belongs too
   Dirs[] = undefined{ slope, point }   // The direction – slope – and starting point of each
cluster
```

```
NextComp= -1    // Current filling cluster index
RepresentativeSegment[] = -1   // The representative segment of each segment

for i in [0, Segments.size()]:
    If Comp[i] == -1 // Not yet in a component ?
        Dirs[i] = slope(Si), Si.start
        NextComp++
        Comp[i]= NextComp
        RepresentatigeSegment[i] = Si

       // See which other segments are in the same cluster
        for j  in [i + 1, Segments.size()]:
            If Comp[j] != -1
                continue
            If OnSameDir(Dirs[i], Sj) // Compare slopes
                 Comp[j]=NextComp
                 Extend(RepresentativeSegment[NextComp], Sj)
```

Build HorizontalClusters and VerticalClusters sets based on Comp array (i.e. if they are in the same cluster put them in a single entry in the two sets).

Figure Fig. Clusterization shows with a debug sample the graphical result of this step.

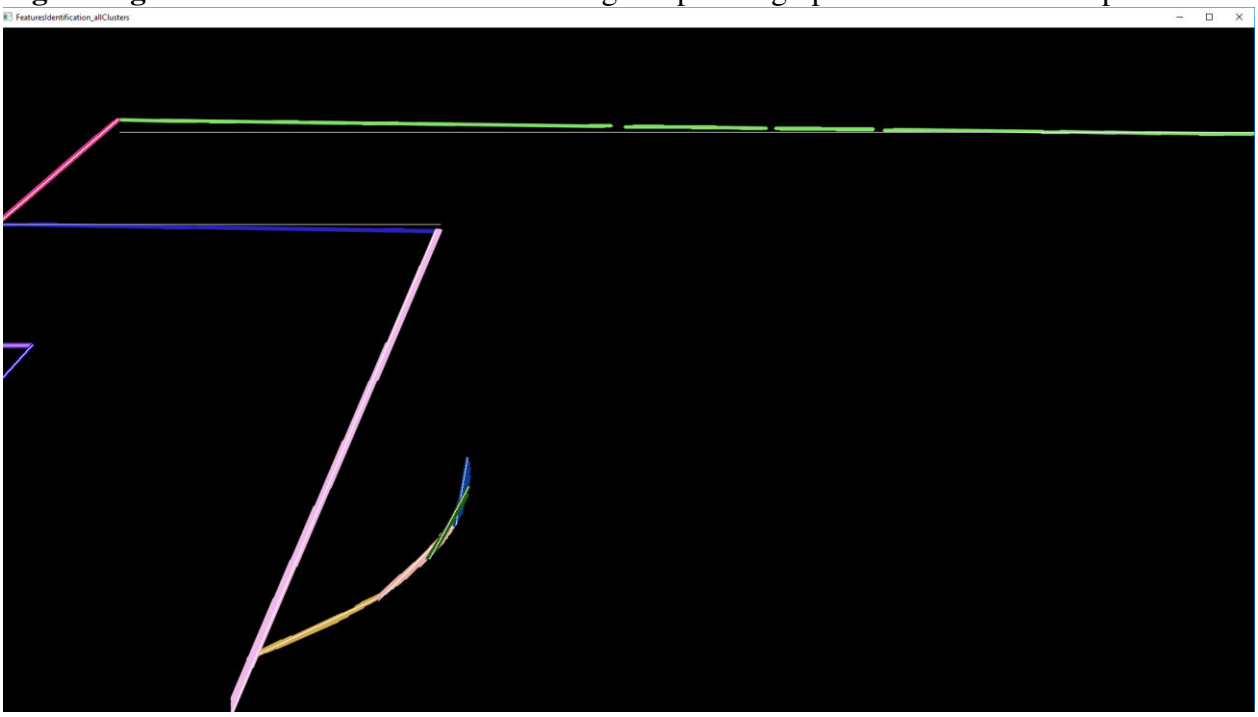
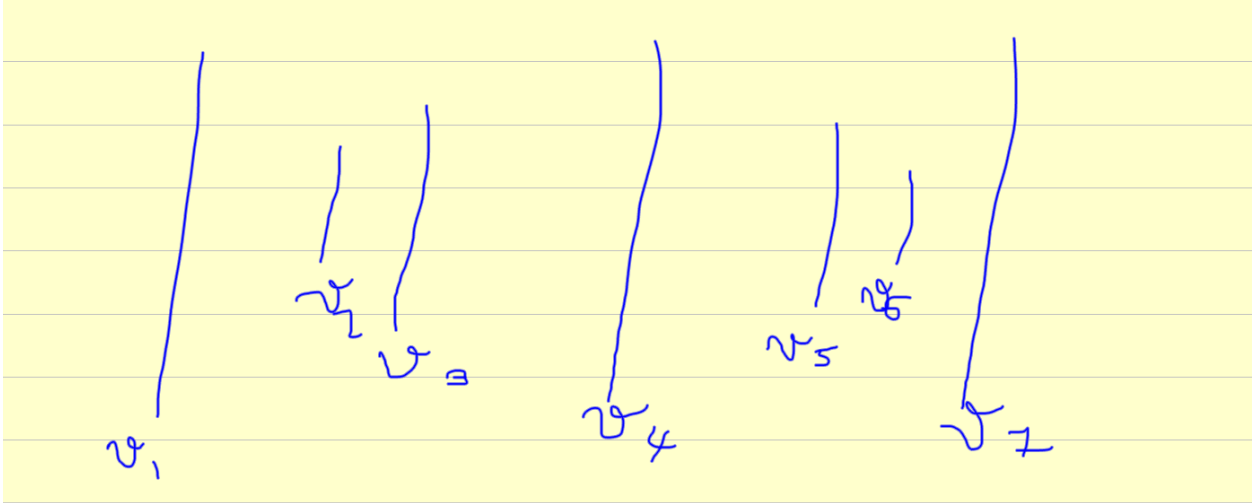


Fig. Clusterization shows the clusters with different colors. For example, check the green sideline. It is composed with several smaller segments as identified by Hough lines run from previous steps, but we unify them all under the same segment. The gray line over it shows the representative segment of this cluster.

B3. The purpose of this step is to select only the clusters that are representative in terms of finding the feature points.

Clusters are sorted using two separate keys: by length and by order (horizontal by Y axis, while vertical ones by X axis). On both horizontal and vertical sides, we get the longest cluster first then select another two (or less if they don't exist) that are parallel to the longest one.

For example, in the picture below there are 7 vertical clusters that represents endlines (v1-v7), small box (v2-v6), big box (v3-b5) and midline (v4).



Below is the pseudocode for finding the observable clusters set for vertical side. The code is analogous for horizontal side.

```
CMax = select the longest representative visible cluster in VerticalClusters
For each cluster C in Cluster different than CMax
    If C is parallel with CMax
        VisibleVerticalClusters.add(C)
```

B4. The Clusters classification purpose is to classify the clusters found in the previous steps into real pitch edges, and in the end to classify the intersection of these pitch edges into feature points. A collection of feature points from the pitch are marked in the **Fig. PitchFeaturePoints**.

Fig. PitchFeaturePoints (shows the feature point types in the middle and boxes; they are similar for the Home team).

Some of the details are omitted in the to simplify the presentation and reader's understanding. For instance, details on which side (left or right) is a feature, can be further read in the source code.

Step 1: try to find pitch endlines among the visible horizonal clusters.

```
sidelines = { }
for the first two longest clusters HC in VisibleHorizontalClusters set
   if HC not long enough
      continue

   RS = representive segment of HC
   if one of the two RS's endpoints are in the safe part of the input image considering height ( i.e. Y coordinate is in the image and not in the first or last 3%)
      if HC is in the upper part of image
        HC.feature = SIDELINE_UP
      else
        HC.feature = SIDELINE_DOWN

      sidelines.add(HC)
```

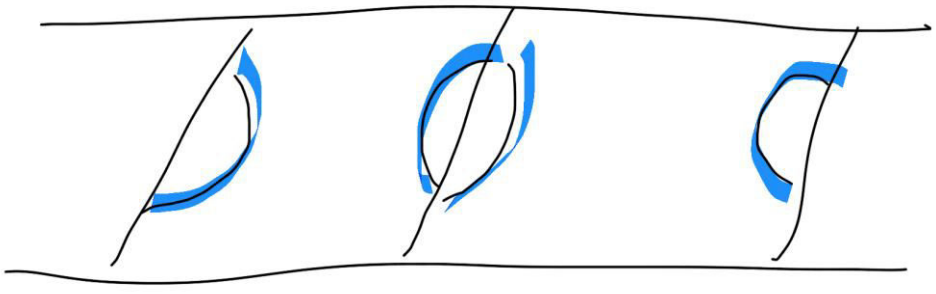
Step 2 : find endlines and midline

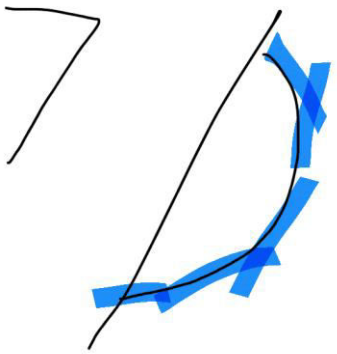
```
endlines = { }
for each VisibleVerticalCluster VC
```

```
if VC doesn't intersects any of the sidelines
    continue

R = intersection between VC a sideline S
if R is not an endpoint of S
  // this is the midline. Assuming that we can't identify other features so exit
    VC.feature = MIDLINE
    if R is in the upper part:
    output.addKepoint(MIDLINE_UP)
  else
output.addKeypoint(MIDLINE_DOWN)
    return
  else
    VC.feature = ENDLINE;
    endlines.add(VC)

  if R is in the upper part:
    output.addKepoint(ENDLINE_UP)
  else
output.addKeypoint(ENDLINE_DOWN)

    If endlines is empty
      return
    // Not supported – usually camera sees only one sideline. Otherwise it means that it is a
    top view camera and is not safe to process in this state
    if endlines.size() > 1
      return;
    isLeftSided = true if endlines[0] is in left side of the image else false
```

***Step 3*:** eliminate the false vertical clusters that are part of the ellipses around the big box or middle line ; the main observation is that the ellipses clusters are between the big box vertical cluster and the other goal in the view. See **Fig. ElipsesAroundLines** to understand where the ellipses are and see **Fig. AfterElipsesElimination** to see the changes that were made to the initial point **Fig. Clusterization**.

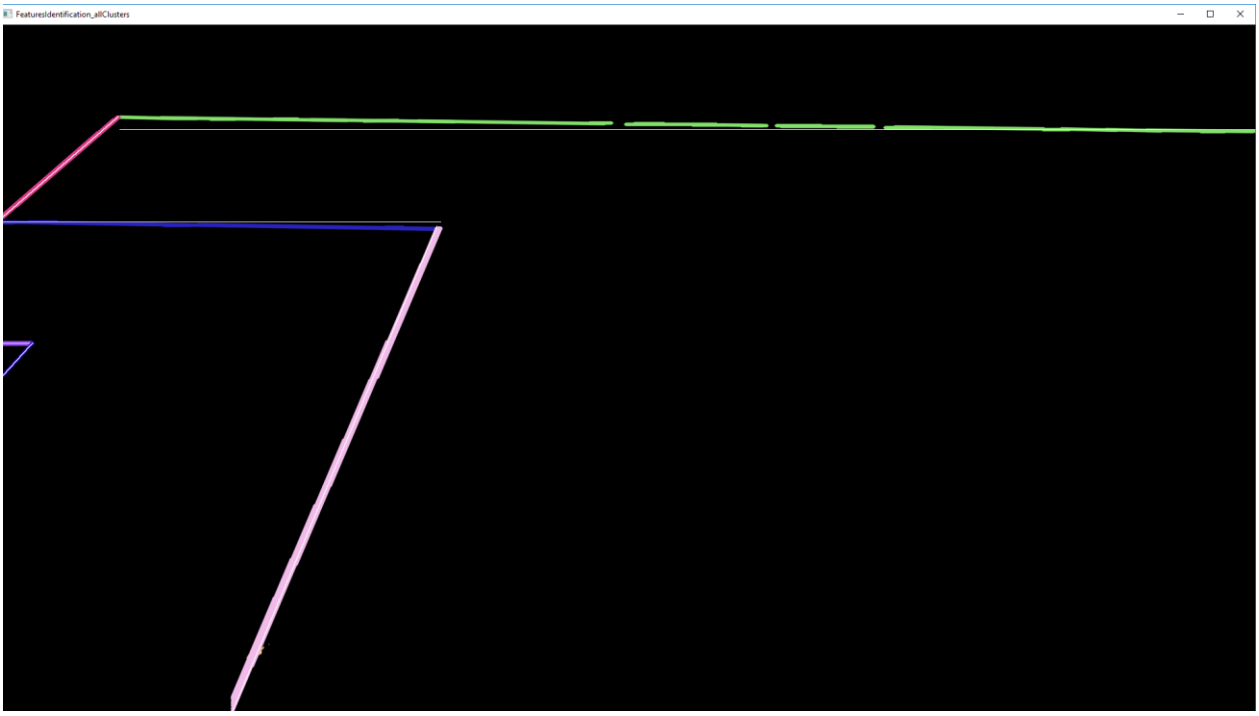
Fig. ElipsesAroundLines: The 'noise' that we want to eliminate in this step.





```
// Find and assume the big box cluster
// It is the longest vertical cluster which is not sideline
bigboxCluster_vertical = nil
bigboxVerticalLimit = nil
For each VisibleVerticalCluster VC ordered by length
   if VC is endline or VC is not parallel with endline
     continue

   R = VC.representativeSegment
   bigboxCluster_vertical = VC

   if isLeftSided
      bigboxVerticalLimit = min(R.start.X, R.end.X) + safeOffsetX
   else
      bigboxVerticalLimit = max(R.start.X, R.end.X) - safeOffsetX

for each VisibleVerticalCluster VC
   If (isLeftSided AND VC.minX > bigboxVerticalLimit )
       OR
      (isLeftSided == false AND VC.minX < bigboxVerticalLimit)
         Remove VC from VisibleVerticalCluster

   smallBoxCluster_vertical = longest vertical cluster between endlines[0] and
bigboxCluster_vertical
```

Fig. AfterElipsesElimination: shows the changes made to the initial starting point **Fig. Clusterization** by eliminating ellipses using Step 3.



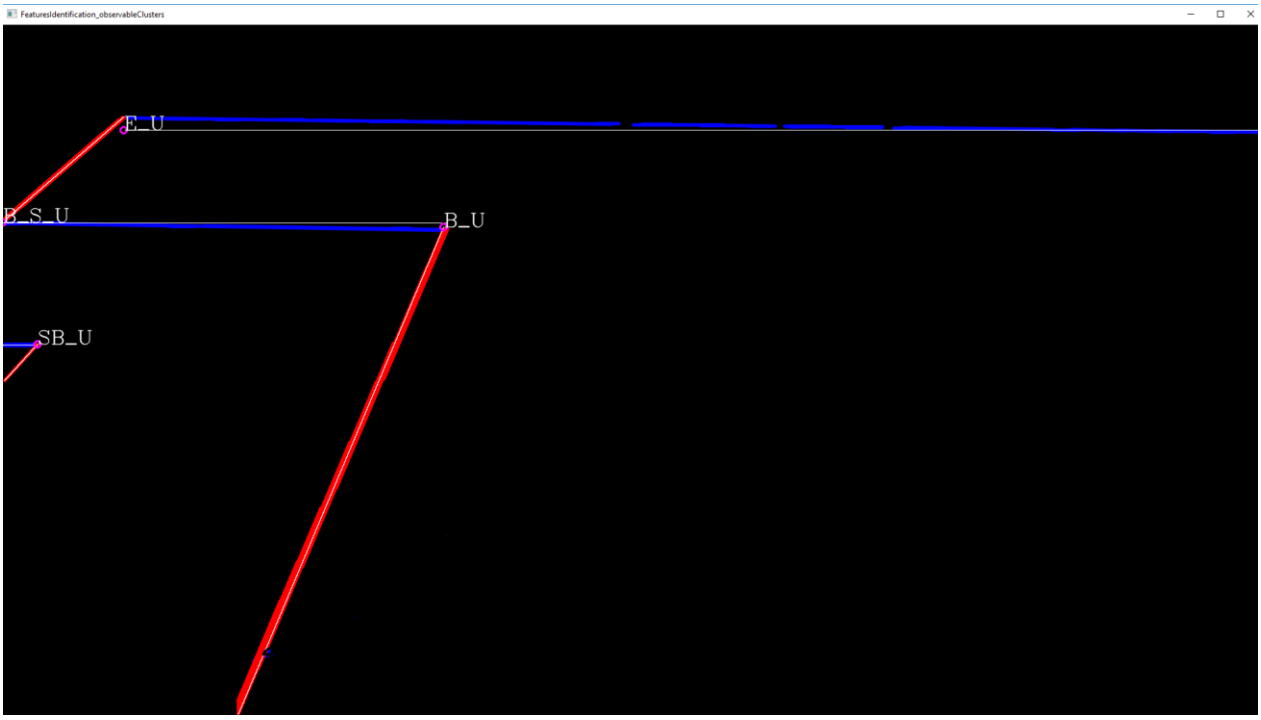
Step 4: similar with Step 3, find bigboxCluster_horizontal[2] and smallBoxCluster_horizontal[2]. Note that there can be zero, one or wo of these.

Step 5: check intersections between bigboxCluster_horizontal[2] and smallBoxCluster_horizontal[2] with endline[0]. Add the intersection points as keypoints features to the output named: BOX_ON_SIDELINE_UP / BOX_ON_SIDELINE_DOWN, respectively SMALLBOX_ON_SIDELINE_UP / SMALLBOX_ON_SIDELINE_DOWN. The "_UP" and "_DOWN" prefixes are identified by checking if the point appears in the upper or lower part of the image on height. Check also Fig. PitchFeaturePoints for understanding better where these points are.

Step 6: check intersections between the big box horizontal and vertical endpoints, and small boxes one, i.e. the pair bigboxCluster_horizontal[2] and bigboxCluster_vertical, and smallboxCluster_horizontal[2] and smallboxCluster_vertical. These keypoints are: BOX_UP / BOX_DOWN and SMALLBOX_UP / SMALLBOX_DOWN. Check also Fig. PitchFeaturePoints for understanding better where these points are.

Using the same debug sample as above, the results until these step can be seen in **Fig. FeaturePoints**.

Fig. FeaturePoints: Some observable feature keypoints in the image – BOX_ON_SIDELINE_UP (B_S_U), ENDLINE_UP (E_U), BOX_UP(B_U), SMALL_BOX_UP (SB_U).



B5. Homograph matrix**.** The last step of this stage is to compute a homograph between the identified pitch features in the image, and the ones of a real stadium. A real stadium picture is given with real coordinates such as the one above this text in **Fig. PitchFeaturePoints** (the data structure that maps between pitch features and coordinates in the real pitch is done at initialization time). If there are more than 4 feature points, a homograph matrix can be computed using **cv::findhomograph** function. The resulted matrix will serve for converting between image and a real football pitch coordinates in the next stage.

For example, using the features found in **Fig. FeaturesPoints**, a homograph matrix can be found and using it every actor position on the pitch in image space, can be then converted to a real football stadium coordinates simply by multiplying the matrix with the image space coordinates.

C. Stage 3

In this stage, the final output is computed in the VisionOutput data structure. If the previous layer provided the coordinates of the players, ball and pitch features in the image space and homograph matrix if one could be built, this layer is more like a post computation process that converts from image to real world stadium coordinates and provides some additional members that the next layer (Decision Making for example) could be interested in.

III.2 Decision Making Layer

Decision Making Layer is the layer which handles input from the *computer vision layer* and decides what kind of action to execute.

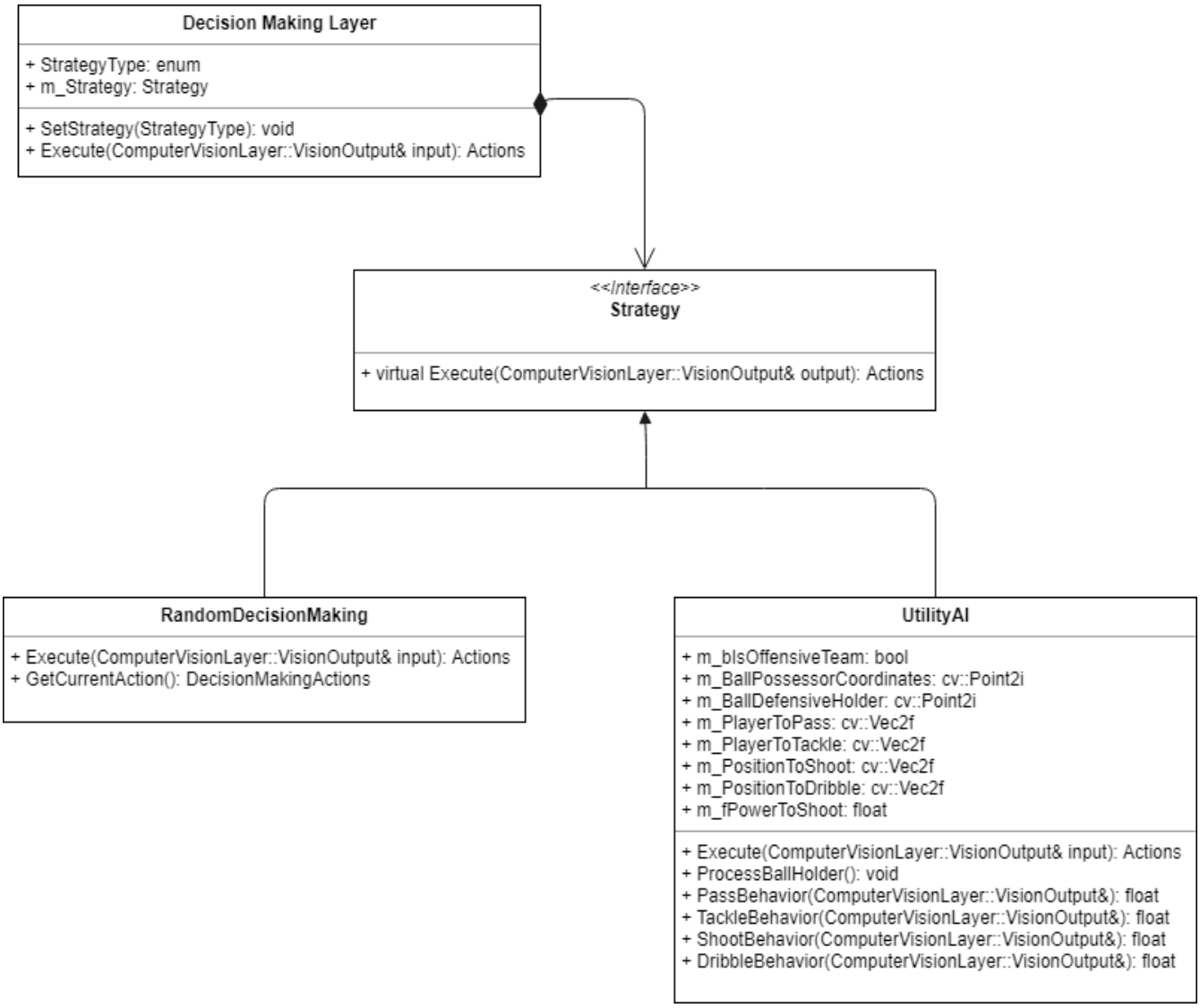
Here is found the AI part of application who decide depending on what is seen, like: position of controlled player, positions of teammates / away team, ball position, lines on the ground (goal line, the lines from small boxes or large ones etc.), the action which should be done like:

- Dribble - if player is in offensive state – he has the ball
- Shot - when current player sees the gate or features of the gate
- Chip shot - is a shot in which the ball is kicked from underneath with accuracy but with less than maximum force, to launch it high into the air in order either to pass it over the heads of opponents
- Driven shot - is used to generate speed and power on the ball
- Stand tackle - tackles are used to regain possession of the ball for your team; standing tackle is one where you remain on your feet
- Slide tackle - the sliding tackle is when you are off your feet
- Pass - when the player gives the ball to a teammate
- Sprint - when the player increases its speed

Decision Making Layer class uses the Strategy pattern that is behavioral software design pattern that enables selecting an algorithm at runtime.

```
class DecisionMakingLayer
{
public:
    enum StrategyType
    {
        Random,
        AIBehavior,
    };

    DecisionMakingLayer() { m_Strategy = NULL; }
    void SetStrategy(int _type);
    Actions Execute(ComputerVisionLayer::VisionOutput& input);

private:
    Strategy * m_Strategy;
};
```

Two types of strategies was implemented, random one – who chooses an action random to be executed and another strategy that uses the output from computer vision layer.



Decision Making Layer is composed of a Strategy. Instead of implementing a behavior the Decision Making Layer delegates it to Strategy. The decision making layer would be the class that would require changing strategies. Strategy is implemented as interface so that we can change the type of strategy without affecting our decision making class.

Types of actions are represented by this enum (actions described above):

```
enum DecisionMakingActions
{
	DRIBBLE,
	SHOOT,
	SHOOT_CHIP,
	SHOOT_DRIVEN,
	PASS,
	TACKLE_SLIDE,
	TACKLE_STAND,
	SPRINT,
	NO_ACTION,
	COUNT
};
```

About Random Decision Making it has a function *GetCurrentAction()* that return randomly an action that should be executed later by **Input Sending Layer**. *Execute()* function, in both type of strategies, fills a structure, *Actions* with what it need. This structure looks like this:

```
struct Actions
{
	float	fStickAngle;
	bool	bDribble;
	bool	bShoot;
	bool	bShootChip;
	bool	bShootDriven;
	bool	bPass;
	bool	bTackleSlide;
	bool	bTackleStand;
	bool	bSprint;
	bool	bNoAction;

	float  fPower;


	void reset()
	{
		fStickAngle = 0.0f;
		bDribble = false;
		bShoot = false;
		bShootChip = false;
		bShootDriven = false;
		bPass = false;
		bTackleSlide = false;
		bTackleStand = false;
		bSprint = false;
		bNoAction = false;

		fPower = 0.0f;
	}

	Actions()
	{
		reset();
	}
};
```

The actions are registered as boolean, the stick angle (fStickAngle) meaning moving direction of player (to update the stick of xbox controller) and also the power (fPower) for shot case, to keep a few frames pressed the button until the power is over.

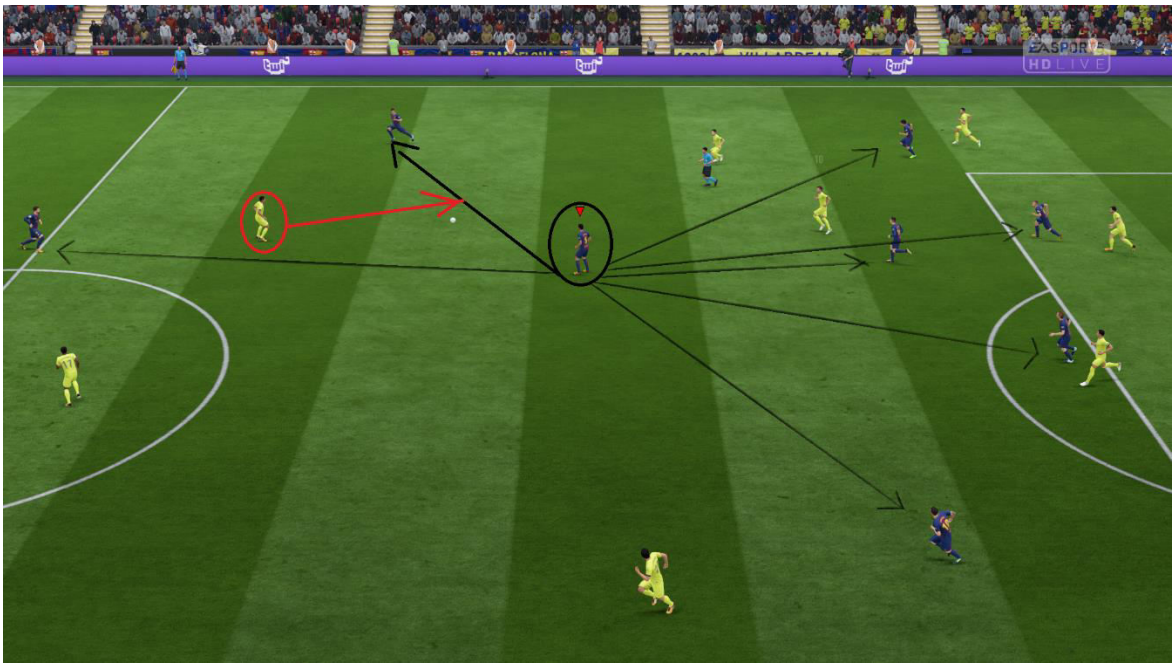
AIBehavior has four types of behaviors that referred to actions described above : *Pass Behavior*, *Shot Behavior*, *Tackle Behavior* and *Dribble Behavior*.

Pass Behavior – In first the check if is in offensive team is done. If it is true (m_bIsOffensiveTeam) means that the home team has the ball and the controlled player is able to pass to another teammate.

The main idea behind the pass behavior is to see which player in the home team we can pass so that the probability of the opposing team will reach the ball while moving is small as possible.

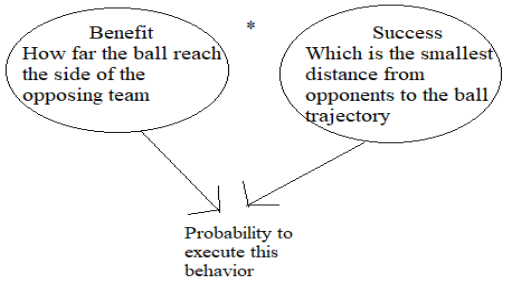
For example in the bellow image the player with marker has more possibilities to pass to a teammate. The probability of giving pass to any of them is calculated accordingly:

- take the distance from the player to each of the teammates - *benefit*
- for each distance taken, the distance from the opposing team's players to the trajectory of the ball is calculated – *success*



The **benefit** is the distance from the player to the teammate to which I want to pass so if the distance is bigger, the benefit is bigger (of course the direction must be towards the opposing gate).

The **success** is distance from the closest opponent player to the ball trajectory, a shorter distance, lower success.



An interpolation function is used to calculate the probability. If distance between the player controlled and home teammates is in a certain range for benefit calculation is mapped the minimum distance with the least probability and the highest distance with 1.0f. In same manner for success calculation.

Tackle Behavior

In this case the player controlled should be in defensive state (away team have the ball). To perform a tackle the player in defensive should be at a relatively small distance from ball possessor.

There are 2 types of tackle:

-> *stand tackle* – when the player remains on his feet

-> *sliding tackle* – he is no longer on his feet

The sliding tackle is choose when the opponent has dribbled the ball too far from his feet. A slide tackle will allow the player to quickly lunge in and make a clean tackle without any physical contact with the player.

So, when the ball is far from the opponent feet (defined distance) slide tackle will be chosen, otherwise a stand tackle will be performed.

For example in this case (image below) the player can perform a sliding tackle because the ball is relative far from the opponent feet.

Similar, the probability is calculated by benefit * success formula.

The **benefit** is represented by the angle between the player controlled and the ball possessor.

The **success** is represented by the distance between the player controlled and the ball possessor. The distance should be as small as possible to have a greater success.

Shot Behavior

If the player is in offensive state and he sees the opponent gate or some gate opponent features, he can perform a shot, a normal one or chip shot or driven shot depending on some characteristics.

A chip shot is chosen when the goalkeeper is further away from the gate and the player controlled is in the same situation. He should use less than maximum force because the ball should launch high into the air to pass it over the heads of opponents. When defending a chip, defenders has more time than other shooting method as ball stay long time in the air. So if the player sees the gate and he is further away from it, a chip shot will be chosen.

A driven shot is chosen when the ball possessor is closest to the opponent gate and he is to the left or right of the gate. A driven shot generate more power, the ball possessor should use the maximum power for this type of shot.

The gate is visible, so the player is able to shot, the position of the opponent gate is fixed and known.

There are the case when the gate is not visible and in this case should be sought some characteristics, small boxes or large one from the opponent gate.

Check their visibility in the following order:

1. Small box up and small box down
2. Box up and box down
3. Small box up and box up
4. Small box down and box down

Let's note with:

B_up = box up

B_down = box down

b_up = small box up

b_down = small box down

For example in the image below **small box up** corner (2) and **box up** (1) corner are visible. In this case the player can shoot at the estimated position of gate : in the direction **of box up** to **small box up**, this means : **P = (B_up + (b_up – B_up) * 2)**.

Similar when the player sees the small box down and box down :

P = (B_down + (b_down – B_down) * 2)

Another example, when the **box up** corner (2) and **box down** corner (1) are visible the gate point estimated should be the middle of these 2 points plus some pixels on x axis :

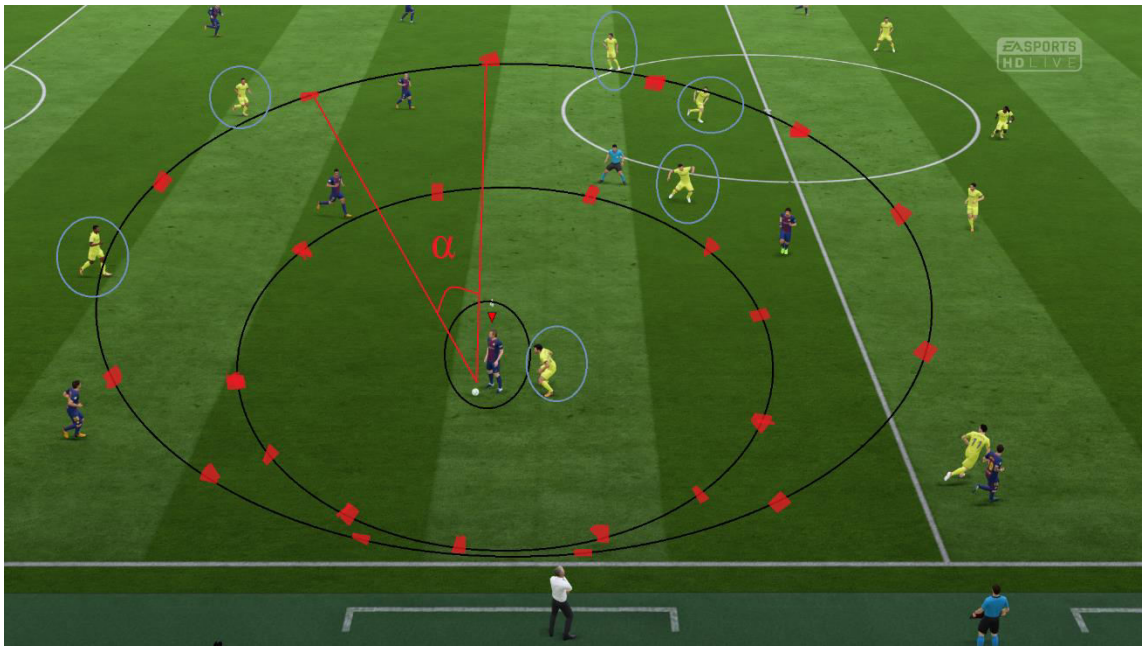
P = (B_up + B_down) / 2 + Vec2(fieldSideOfAttack * SOME_PIXELS, 0);

Similar for small **box up** corner and small **box down** corner:

P = (b_up + b_down) / 2 + Vec2(fieldSideOfAttack * SOME_PIXELS, 0);

Dribble Behavior

If the player controlled is in offensive team he could dribble (moving with the ball). Choose a fixed angle and a fixed distance to find some points to which the player can go. We calculate the vectors at the starting point (where the player is located) to these chosen points.



Then all the positions of the players in the opposing team were taken to see who had the greater chance (shorter distance) to reach the player's trajectory (vectors calculated in the previous step). For diversity, two types of distances were chosen like in the image above.

Also the optimal point was chosen by calculating the probability by the **benefit * success** formula. In this case the **benefit** is the distance to dribble (two types of distance were chosen); distance is bigger => the benefit is bigger. **Success** is the minimum distance from an opposing team player to the controlled player's trajectory.

Similar to the other behaviors an interpolation function is used to calculate the probability.

III.3 Input Sending Layer

Input Sending Layer uses virtual joystick - vJoy project that is centered around a virtual joystick driver. The class that implemented the functionality for this layer looks like this:

```
class InputSendingLayerImpl
{
public:
    InputSendingLayerImpl();

    JOYSTICK_POSITION_V2 GetJoystickPositionData()
    {
        return m_iReport;
    }

    void ProcessActions(Actions& _actionsToDo);

    void init();

    void PrintVJoyStatus(UINT _devID);

    void DecimalToHexa(int _value, LONG &_outHexNum);
    void AngleToHexaValues(float _angle, LONG &_xValue, LONG &_yValue);

    void SendPositionDataToVJoyDevice(UINT _devID, JOYSTICK_POSITION_V2 &_iReport);

private:
    JOYSTICK_POSITION_V2 m_iReport; // The structure that holds the full position data
    Actions m_ActionsProcessed;
};
```

Once the driver is installed, a virtual joystick device will automatically be installed. The joystick position are pushed to the device through a well defined interface.

Any external user-mode software (applications or service) will be able to place position data on the virtual device.

Interface of feeder application with vJoy Device is done through interface DLL (vJoyInterface.dll).

Driver information functions

The following functions return general information about the installed driver:

VJOYINTERFACE_API SHORT __cdecl GetvJoyVersion(void);

Get the vJoy driver Version Number if found or 0 if not found.

Current version is 0x200

VJOYINTERFACE_API BOOL __cdecl vJoyEnabled(void);

Returns TRUE if vJoy Driver installed and there is at least one enabled vJoy device.

VJOYINTERFACE_API PVOID __cdecl GetvJoyProductString(void);
Returns pointer to vJoy product string.
Returns NULL if error occurs.

VJOYINTERFACE_API PVOID __cdecl GetvJoyManufacturerString(void);
Returns pointer to vJoy manufacturer string.
Returns NULL if error occurs.

VJOYINTERFACE_API PVOID __cdecl GetvJoySerialNumberString(void);
Returns pointer to vJoy Serial Number string.
Returns NULL if error occurs.

vJoy Device Properties

The following functions return configuration information about a specific vJoy Device.
Every device (VJD) has a unique Report ID (rID).
Report ID value may be 1 to 16.

VJOYINTERFACE_API int __cdecl GetVJDButtonNumber(UINT rID);
Get the number of buttons defined in the specified vJoy Device
Possible numbers of buttons is 0 to 32

VJOYINTERFACE_API int __cdecl GetVJDDiscPovNumber(UINT rID);
Get the number of descrete-type POV hats defined in the vJoy Device
Possible numbers of descrete-type POV are 0 or 1
A descrete-type POV is a POV that has 5 states: North, West, South, East and neutral.

VJOYINTERFACE_API int __cdecl GetVJDContPovNumber(UINT rID);
Get the number of continuous-type POV hats defined in the specified vJoy Device
Possible numbers of continuous-type POV are 0 or 1

VJOYINTERFACE_API BOOL __cdecl GetVJDAxisExist(UINT rID, UINT Axis);

Test if given axis defined in the specified vJoy Device

Axis values are documented in HID documents and in file Public.h

Write access to vJoy Device

The following functions are used to Update the position data of a specific vJoy Device (VJD) or to get its Status.

An application may Acquire and Relinquish ownership of one or more vJoy Devices. To Update the positions of the vJoy Device, the application should own it.

Ownership can be Relinquished either by the owner-application or by removal or disabling of the driver.

Status can be either:

1. VJD_STAT_OWN: The vJoy Device is owned by this application.
2. VJD_STAT_FREE: The vJoy Device is NOT owned by any application (including this one).
3. VJD_STAT_BUSY: The vJoy Device is owned by another application. It cannot be acquired by this application.
4. VJD_STAT_MISS: The vJoy Device is missing. It either does not exist or the driver is down.
5. VJD_STAT_UNKN: Unknown state

VJOYINTERFACE_API BOOL __cdecl AcquireVJD(UINT rID);

Acquire the specified vJoy Device.

If successful returnes TRUE and the status of the specified vJoy Device switches to VJD_STAT_OWN. Otherwise FALSE.

Application must call RelinquishVJD() when the specified vJoy Device is no longer required.

Additional calls to this function are ignored.

VJOYINTERFACE_API VOID __cdecl RelinquishVJD(UINT rID);

Relinquish the specified vJoy Device. If was owned by this application the status of the specified vJoy Device switches to VJD_STAT_FREE.

After calling this function the specified vJoy Device can be Acquired by any application.

Calles to this function are ignored if the specified vJoy Device is not owned by this application.

VJOYINTERFACE_API BOOL __cdecl UpdateVJD(UINT rID, PVOID pData);
Update the position data of the specified vJoy Device.
The position data is in a structure pointed at by the second parameter.
For this function to be successful, the specified vJoy Device must be owned by this application.
If successful returnes TRUE. Otherwise FALSE.

VJOYINTERFACE_API enum VjdStat __cdecl GetVJDStatus(UINT rID);
Get the status of the specified vJoy Device.

How is it used in this project?

All access to vJoy driver and to the vJoy devices is done through vJoy interface functions that are implemented in file vJoyInterface.dll.

It is advisable to base your feeder on the supplied example and make the needed changes. Here are the five basic steps followed by this layer to send the input to FIFA game:

Test Driver:	Check that the driver is installed and enabled. Obtain information about the driver. An installed driver implies at least one vJoy device. Test if driver matches interface DLL file
Test Virtual Device(s):	Get information regarding one or more devices. Read information about a specific device capabilities: Axes, buttons and POV hat switches.
Device acquisition:	Obtain status of a vJoy device. Acquire the device if the device status is owned or is free.
Updating:	Inject position data to a device (as long as the device is owned by the feeder). Position data includes the position of the axes, state of the buttons and state of the POV hat switches.
Relinquishing the device:	The device is owned by the feeder and cannot be fed by another application until relinquished.

Feeding the vJoy device

Test vJoy Driver:

Before you start feeding, check if the vJoy driver is installed and check that it is what you expected:

```
// Get the driver attributes (Vendor ID, Product ID, Version Number)
if (!vJoyEnabled())
{
	_tprintf("Function vJoyEnabled Failed - make sure that vJoy is installed and
	enabled\n");
	goto Exit;
}
else
{
	wprintf(L"Vendor: %s\nProduct :%s\nVersion Number:%s\n",
	static_cast<TCHAR *> (GetvJoyManufacturerString()), static_cast<TCHAR
	*>(GetvJoyProductString()), static_cast<TCHAR
	*>(GetvJoySerialNumberString()));
};
```

Test vJoy Virtual Devices

Check which devices are installed and what their state is:

```
// Get the status of the vJoy device before trying to acquire it
VjdStat status = GetVJDStatus(DevID);

switch (status)
{
case VJD_STAT_OWN:
	_tprintf("vJoy device %d is already owned by this feeder\n", DevID);
	break;
case VJD_STAT_FREE:
	_tprintf("vJoy device %d is free\n", DevID);
	break;
case VJD_STAT_BUSY:
	_tprintf("vJoy device %d is already owned by another feeder\nCannot continue\n",
	DevID);
	return -3;
case VJD_STAT_MISS:
	_tprintf("vJoy device %d is not installed or disabled\nCannot continue\n",
	DevID);
	return -4;
default:
	_tprintf("vJoy device %d general error\nCannot continue\n", DevID);
	return -1;
	};
```

Acquire the vJoy Device: Until now the feeder just made inquiries about the system and about the vJoy device status. In order to change the position of the vJoy device you need to Acquire it (if it is not already owned):

```
// Acquire the vJoy device
if (!AcquireVJD(DevID))
{
	_tprintf("Failed to acquire vJoy device number %d.\n", DevID);
	goto Exit;
}
else
{	_tprintf("Acquired device number %d - OK\n", DevID);

}
```

Feed vJoy Device: Feed the vJoy device with position data. Reset the device once then send the position data for every control (axis, button, POV) at a time.

In Input Sending Layer there is a function *ProcessActions(Actions& _actionsToDo)* that receive an input structure like the one explained in **Decision Making Layer** section.

Remember this structure is used to know what type of action should be execute, like: move direction updated by fStickAngle, if the player is in dribbling, or if he should pass, or shoot, or to attack the opponent through a tackle.

This function is called after is received the actions structure from **Decision Making Layer**, the layer that decide what type of action should be executed in the current capture. The values received by vJoy device are in hexadecimal.

```
void InputSendingLayerImpl::ProcessActions(Actions& _actionsToDo)
{

	// The structure that holds the full position data

	JOYSTICK_POSITION_V2 iReport = GetJoystickPositionData();

	// Always update the stick angle

	m_ActionsProcessed.fStickAngle = _actionsToDo.fStickAngle;
	if (_actionsToDo.bDribble)
	{
		AngleToHexaValues(m_ActionsProcessed.fStickAngle, iReport.wAxisX, iReport.wAxisY);
		m_ActionsProcessed.bDribble = true;
	}
	else
	{
		iReport.wAxisX = NEUTRAL_AXIS_HEX_VALUE;
		iReport.wAxisY = NEUTRAL_AXIS_HEX_VALUE;
		iReport.wAxisZ = NEUTRAL_AXIS_HEX_VALUE;
		m_ActionsProcessed.bDribble = false;
	}
	if (_actionsToDo.bShoot)
	{
```

```
                iReport.lButtons = 2;
                m_ActionsProcessed.bShoot = true;
        }
        else
        {
                iReport.lButtons = 0;
                m_ActionsProcessed.bShoot = false;
        }

        if (_actionsToDo.bPass)
        {
                iReport.lButtons = 4;
                m_ActionsProcessed.bPass = true;
        }
        else
        {
                iReport.lButtons = 0;
                m_ActionsProcessed.bPass = false;
        }

        if (_actionsToDo.bTackleSlide)
        {
                m_ActionsProcessed.bTackleSlide = true;
                iReport.lButtons = 8;
        }
        else
        {
                iReport.lButtons = 0;
                m_ActionsProcessed.bTackleSlide = false;
        }
        // And all other actions from Actions structure in the same manner.

        // Send position data to vJoy device
        PVOID pPositionMessage = (PVOID)(&_iReport);
        if (!UpdateVJD(_devID, pPositionMessage))
        {
                AcquireVJD(_devID);
        }
}
```

IV. Instructions to run and debug

How to run in final mode

➔ Compile in RELEASE mode

➔ Ensure that IS_SINGLE_FRAME_CAPTURE_ENABLED is NOT enabled. Also make sure that IS_DEBUG_WINDOW_ENABLED is NOT enabled.

➔ For best quality use a Barcelona vs Villareal match, on Anfiled stadium and default ball.

(Try to disable the timer, names of players and others)

How to run in final mode with screenshot debug

➔ Same as above but if you enable IS_DEBUG_WINDOW_ENABLED you can press **ALT + A** at any time and it will stop the application and get you the debugging frame in that moment, as in the picture below, where you can view computer vision output text, markers and input text.

How to debug custom frames

➔ Set both IS_SINGLE_FRAME_CAPTURE_ENABLED and IS_DEBUG_WINDOW_ENABLED and build preferably on DEBUG (event if the time is 10x times longer to execute)

➔ Search for the following variable and change it to the screenshot you want to debug

```
const char* GlobalParameters::m_debugImgPath = "BarcaVillareal/fifa18_2.png";
```

➔ Take a screenshot and use it to debug. This is the most easiest way to test quickly.

➔ Search for variables named **isLocalDebuggingEnabled**. Enable them locally by setting true instead of false. But enable them as close to your problem to show visually the inputs you get from debug windows. If you enable at the top you'll likely see too many windows opened.

References

1. **Learning OpenCV 3:** Computer Vision in C++ with the OpenCV Library

Adrian Kaehler, Gary Bradski

O'Reilly Media

2. **Article : Adaptive Field Detection and Localization in Robot Soccer**

Yongbo Qian, Daniel D. Lee

GRASP Lab, University of Pennsylvania, Philadelphia PA 19104, USA

3. **Article : Feature Detection and Localization for the RoboCup Soccer SPL**

Amogh Gudi, Patrick de Kok, Georgios K. Methenitis, Nikolaas Steenbergen

University of Amsterdam

4. **Article : Go! – A Multi-paradigm Programming Language for Implementing Multi-threaded Agents**

K.L. Clark∗ Dept. of Computing Imperial College London, UK

Dept. of Computing Imperial College London, F.G. McCabe Fujistu Labs of America Sunnyvale CA, USA

5. **Article : Histograms of Oriented Gradients for Human Detection**

Navneet Dalal and Bill Triggs

INRIA Rhone-Alps, 655 avenue de l'Europe, Montbonnot 38334, France

6. **Article : People Detection in Color and Infrared Video using HOG and Linear SVM**

Pablo Tribaldos, Juan Serrano-Cuerda, Marıa T. Lopez , Antonio Fernandez-Caballero, and Roberto J. Lopez-Sastre

Instituto de Investigacion en Informatica de Albacete

Universidad de Castilla-La Mancha, Departamento de Sistemas Informaticos

Universidad de Alcala, Dpto. de Teorıa de la senal y Comunicaciones

7. https://opencv.org/
8. https://github.com/opencv/opencv
9. https://github.com/shauleiz/vJoy
10. http://vjoystick.sourceforge.net/site/index.php/download-a-install/download