POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

3D Trajectory Reconstruction in Table Tennis

Image Analysis and Computer Vision - Automation and Control Engeneering

Authors: **Matteo Bartoli** Mat.945886
Giovanni Buzzao Mat.928918

Advisor: Prof. Vincenzo Caglioti
Academic Year: 2021-22

Table of contents

1 INTRODUCTION

Table tennis is a demanding sport where the judges have to pay attention to many complex rules in a short period of time.

A computer system in this field, for instance, can help to better decide unclear plays, feeding the judges with data and measurements, or can be used by couches to analyze the players performances during training.

However, we must not think that for a computerized system the challenge is simpler, indeed there are a bunch of problems to consider, such as:

- **High object motion**: The high speed of the ball may cause it to be blurred and distorted in shaped.
- **Multiple moving objects**: Apart from the ball, the players, the rackets and background elements in the scene exhibit motion.
- **Uneven lighting**: Light sources on the ceiling tends to cause the ball to be brighter at the top than at the bottom.
- **Occlusion**: The ball can be blocked by the player, by the rackets, or even disappear from the field of view in the upper part of the video.
- **Merging**: Low contrast from the ball and the background may cause difficulties in distinguishing both.
- **Object Confusion**: Objects in the background with similar color (e.g. human skin) and shapes (e.g. a bald person head) may be confused with the ball.
- **Small size**: Size of the ball is only a fraction of the frame, which randers histogram-based detection unsuitable.
- **Time constraint**: latency from detecting to tracking the ball must be minimised.

2 TWO-VIEW GEOMETRY

To better understand what are the bases for creating a system of this type, we must understand how a computer is able to analyze a 3D scene, and for doing it we need summarize the theory behind.

The mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane done by a single camera can be described by the pinhole model. However due to the 3D to 2D conversion, we lose the depth of the image. Therefore, to find the depth in this kind of projects at least two cameras are needed.

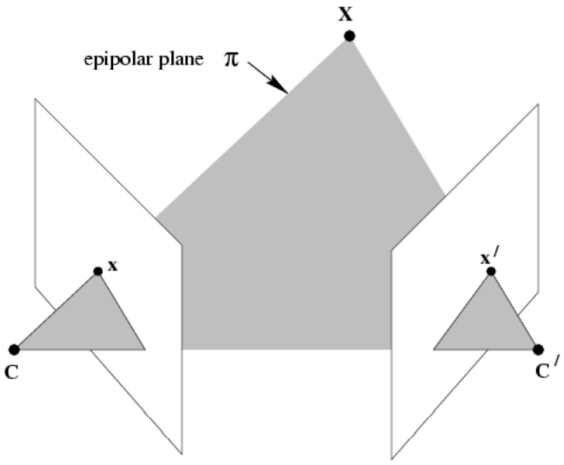
The intrinsic projective geometry of two views is called Epipolar Geometry. It depends only on the internal parameters of the camera and their relative pose while being independent from the scene structure.

The main component of epipolar geometry is the 3×3 matrix F called the fundamental matrix and satisfies the following property:

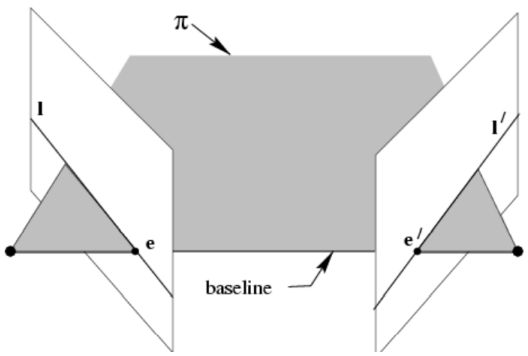
$$xFx' = 0$$

Where x and x' are the image coordinates of the real-world point X in the first and second view.

The points x, x', the camera centers c, c', and X are coplanar and constitute the plane π called the epipolar plane. The epipolar plane intersects the image plane in a line called the epipolar line.



For each point x in one image, there exists a corresponding epipolar line l' in the other image. Any point x' in the second image matching the point x must lie in the epipolar line l'.



The general approach to perform the 3D reconstruction from two views consists of 3 steps:

1. Compute the Fundamental matrix:

   Find a set of corresponding points $x \leftrightarrow x'$ between the two camera images and use the property $xFx' = 0$ to find linear equations. Note that 8 points are needed to solve the system linearly, or more using least square.

   It is important to note that some ambiguity still exists if the fundamental matrix is not computed uniquely. This approach should be supplied by 3D knowledge of the scene. If the fundamental matrix is computed uniquely the image can be reconstructed only to a projective ambiguity.

Image information provided	View relations and projective objects	3-space objects	reconstruction ambiguity
point correspondences	F		projective
point correspondences including vanishing points	F, H_∞	π_∞	affine
Points correspondences and internal camera calibration	F, H_∞ ω, ω'	π_∞ Q_∞	metric

   But adding the plane at infinity and the image of the absolute conic we can achieve a metric reconstruction.

2. Compute the camera matrices from the fundamental matrix

   Take a tentative pair of cameras compatible with F_{12}:

$$P_1 = [I \mid 0] \quad P_2 = [[e_2]_\times F_{12} + e_2 v^T \mid \lambda e_2]$$

   Where e_2 is the LNS of F_{12}, λ and v are any nonzero scalar and vector

3. Compute 3D point for each pair of corresponding points that is done by triangulating the pairs of viewing rays associated through cameras P1 and P2

3 SIX-POINTS CALIBRATION

To overcome the ambiguity problem finding the Fundamental matrix we decide to do the 3D reconstruction starting from calibrated images.

We try the 6-points calibration method because the calibration with vanishing points can't be done due to the insufficient number of planes therefore not all needed independent constrains can be found.

Therefore, this kind of calibration allow to find, in addition to the intrinsic parameters of the camera, also the extrinsic parameters of the camera.

The intrinsic parameters depend only on the type of camera because tell how your camera handles pixel. They are expressed together in the intrinsic matrix K.

$$K = \begin{bmatrix} s_x & \alpha & t_u \\ 0 & s_y & t_v \\ 0 & 0 & 1 \end{bmatrix}$$

Where t_u and t_v are the coordinates of the principal point, s_x and s_y are the focal lengths and α is the skew factor representing the distortion.

The extrinsic parameters depend on the camera position and orientation (called camera pose) with respect to a reference frame taken in world coordinates. They are a rotation matrix *R* and a translation vector *t.*

All those parameters are contained in the Projection Matrix $\mathrm{P} = \mathrm{K}[\mathrm{R}\ \mathrm{t}]$

The method used by the 6-points calibration to find P is pretty intuitive, knowing that, as definition, the matrix P relate the world points to the image points $\mathrm{x} = P\mathrm{X}$, is sufficient to find enough equations to constrain all elements of P.

As shown, each point correspondence in homogeneous coordinates gives two equations.

$$\begin{bmatrix} \lambda x \\ \lambda y \\ \lambda \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{aligned} \lambda x &= P_{11}X + P_{12}Y + P_{13}Z + P_{14} \\ \lambda y &= P_{21}X + P_{22}Y + P_{23}Z + P_{24} \\ \lambda &= P_{31}X + P_{32}Y + P_{33}Z + P_{34} \end{aligned}$$

$$(P_{31}X + P_{32}Y + P_{33}Z + P_{34})x = P_{11}X + P_{12}Y + P_{13}Z + P_{14}$$
$$(P_{31}X + P_{32}Y + P_{33}Z + P_{34})y = P_{21}X + P_{22}Y + P_{23}Z + P_{24}$$

The P matrix is a 4x3 matrix so we have to find 12 equations that are given by 6 non-coplanar points. All the equation's coefficients can be putted in a matrix A to solve the linear system $Ap = 0$ with SVD.

```
%    p11       p12       p13       p14       p21       p22       p23       p24      p31             p32             p33             p34
A = [-X(1,1), -X(2,1), -X(3,1), -X(4,1), 0,        0,        0,        0,       X(1,1)*u(1,1),  X(2,1)*u(1,1),  X(3,1)*u(1,1),  X(4,1)*u(1,1);
     0,       0,       0,       0,       -X(1,1), -X(2,1), -X(3,1), -X(4,1),  X(1,1)*u(2,1),  X(2,1)*u(2,1),  X(3,1)*u(2,1),  X(4,1)*u(2,1);
     -X(1,2), -X(2,2), -X(3,2), -X(4,2), 0,        0,        0,        0,       X(1,2)*u(1,2),  X(2,2)*u(1,2),  X(3,2)*u(1,2),  X(4,2)*u(1,2);
     0,       0,       0,       0,       -X(1,2), -X(2,2), -X(3,2), -X(4,2),  X(1,2)*u(2,2),  X(2,2)*u(2,2),  X(3,2)*u(2,2),  X(4,2)*u(2,2);
     -X(1,3), -X(2,3), -X(3,3), -X(4,3), 0,        0,        0,        0,       X(1,3)*u(1,3),  X(2,3)*u(1,3),  X(3,3)*u(1,3),  X(4,3)*u(1,3);
     0,       0,       0,       0,       -X(1,3), -X(2,3), -X(3,3), -X(4,3),  X(1,3)*u(2,3),  X(2,3)*u(2,3),  X(3,3)*u(2,3),  X(4,3)*u(2,3);
     -X(1,4), -X(2,4), -X(3,4), -X(4,4), 0,        0,        0,        0,       X(1,4)*u(1,4),  X(2,4)*u(1,4),  X(3,4)*u(1,4),  X(4,4)*u(1,4);
     0,       0,       0,       0,       -X(1,4), -X(2,4), -X(3,4), -X(4,4),  X(1,4)*u(2,4),  X(2,4)*u(2,4),  X(3,4)*u(2,4),  X(4,4)*u(2,4);
     -X(1,5), -X(2,5), -X(3,5), -X(4,5), 0,        0,        0,        0,       X(1,5)*u(1,5),  X(2,5)*u(1,5),  X(3,5)*u(1,5),  X(4,5)*u(1,5);
     0,       0,       0,       0,       -X(1,5), -X(2,5), -X(3,5), -X(4,5),  X(1,5)*u(2,5),  X(2,5)*u(2,5),  X(3,5)*u(2,5),  X(4,5)*u(2,5);
     -X(1,6), -X(2,6), -X(3,6), -X(4,6), 0,        0,        0,        0,       X(1,6)*u(1,6),  X(2,6)*u(1,6),  X(3,6)*u(1,6),  X(4,6)*u(1,6);
     0,       0,       0,       0,       -X(1,6), -X(2,6), -X(3,6), -X(4,6),  X(1,6)*u(2,6),  X(2,6)*u(2,6),  X(3,6)*u(2,6),  X(4,6)*u(2,6)];

[~,~,v] = svd(A);
sol = v(:,end);   % sol = (p11 p12 p13 p14 p21 p22 p23 p24 p31 p32 p33 p34)
```

Our proposed code is in the function `calibration6Pnt`

To extrapolate K, R and t from P we used the "QR" decomposition that decomposes an nxn matrix into the product of a rotation matrix and an upper triangular matrix so we need to pass to the function the inverse of [KR]

```
[R,K] = qr(inv(P(:,1:3)))
```

```
t = K\P(:,4)
```

Having all intrinsic and extrinsic parameters is possible to do the 3D reconstruction by the triangulation Alghorithm.

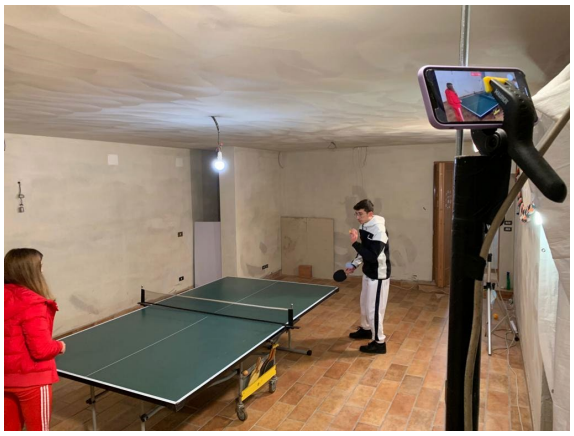
Despite the validity of this method in our setup it's hard to find 6 non coplanar points with known coordinates so at the end we decide to use the calibration App provided by Matlab, for also improving the final result and robustness.

4 SETUP

We mounted two identical Iphone 11s on supports at the same height to keep the cameras fixed during all the recordings and to give the right angle to the images.

The cameras have a frame rate of 30 fps and a resolution of 1920x1080 pixel.

We light up the scene with spotlights from behind the cameras in order to make the scene as clear as possible.



5 TABLE DETECTION

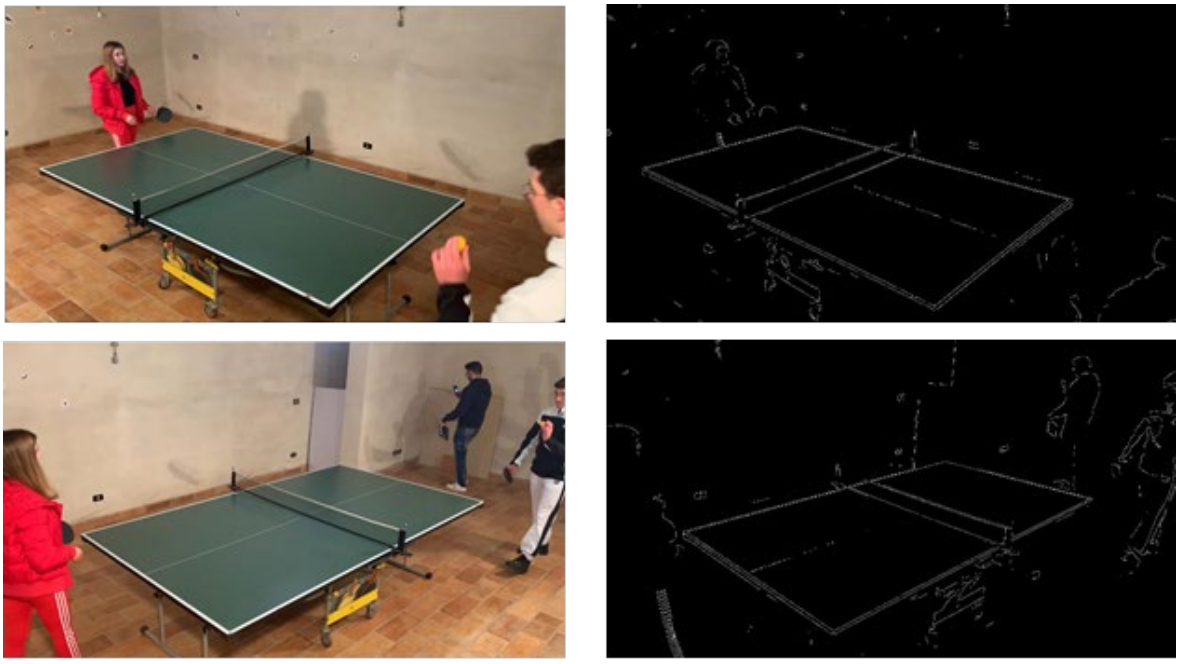
The table detection is done completely automatically using different algorithms to extrapolate from a frame of the video the lines equations that best represent the sides of the table. Therefore, their intersections, which are the vertices of the table, are needed to compute the position and the orientation of our cameras in order to reconstruct the 3D trajectory of the ball, since we have chosen to start from already calibrated cameras.

In addition this our proposed algorithm is tunable, with the main parameters taken out from subfunctions allowing quicker customization of the system on different setups.

The steps to achieve this result can be summarized as:

5.1 LINES DETECTION

Before detecting lines, we need to use Canny algorithm for computing the edges. Being the sides well distinguishable we tuned the intensity threshold and the standard deviation of the Gaussian filter (sigma) to have a relevant noise reduction and background removal while avoiding an intense distortion.



Then with Hough Peaks and Hough Lines algorithms we found the set of lines. A fine tuning was needed to have a high number of lines, avoiding the shortest.

As output it gives a struct containing the informations of the selected lines, such as their endpoints, orientation and length.

5.2 LINES SELECTION

This step is done using a neighborhood's color-based selection algorithm with tunable ROI. It takes as input the lines computed before and decide for each one if is on the table side or not relying on the color of neighbor's pixels. As output gives a struct similar to the one before but containing only the selected lines.

The main steps followed by this algorithm can be summarized as:

It's to notice that it does multiple passes for each line, changing some detection strategies, to don't lose data discarding the line at the first negative attempt.

5.2.1 ROI selection

First of all, we discard the lines having the endpoints below a given y-threshold. This removes some background noises, especially in large environments.

The y-threshold is passed from the main with the variable PAR.yTh, to allow a user-friendly tuning.

5.2.2 Midpoint and Keypoints

This process is done with a separate custom function

[keyPnt,m] = computeKeyPnt(x,theta,PAR)

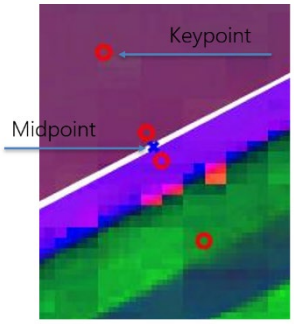
The inputs are the endpoints of a line and its orientation in degree. As Output gives the 4 key points coordinates relying on the specified parameters and the mid point.

The keypoints are stored in a vector always in the same position, so was necessary to consider also the orientation of the line.

We need two reference distances that are passed as detection parameters from the main.

PAR.farTh is fixed. It didn't give detection issues.

PAR.nearTh is dynamic: if the line is not selected in this pass it is decreased and the all evaluation start again. This is done to tackle both large and small lines (see last step)




```
% Far superior  (3)       o    |Distance for far points [PAR.farTh]
%                               |
% Near superior (1)       o    |   |Distance for near points [PAR.nearTh]
% Line  ------------------x-------------------------
% Near inferior (2)       o
%                             Points are taken on the orthogonal direction
% Far inferior  (4)       o   starting from the midpoint of the segment.
```

5.2.3 Clolour picking

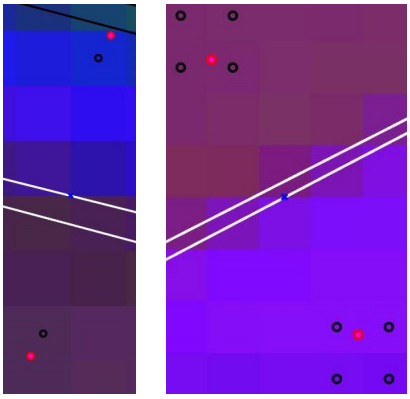
There are some considerations to do on this step, that hide more difficulties then appear.

- The keypoints are computed geometrically so they doesn't correspond to specific pixels.
- Some pixels can have disturbances, such as salt and pepper grain or color and light reflections from near objects or players
- Averaging colors can be risky because we are evaluating lines in their orthogonal direction (few pixel, especially in low resolution images)

We try 2 methods for picking the right color:

1. Choosing the color of the pixel that best approximate the position of the key point. This is implemented in the function keyColor = computeColor(keyPnt,Ihsv,PAR)

2. Computing the coordinates of the four neighbor pixels and averaging their color. This is implemented in the function keyColor = computeAveragedColor(keyPnt,Ihsv,PAR)



We notice that for large images is better to averaging the color to improve accuracy, instead in low resolution images is better to take just one sample because there is the risk to exceed the border of the white line.

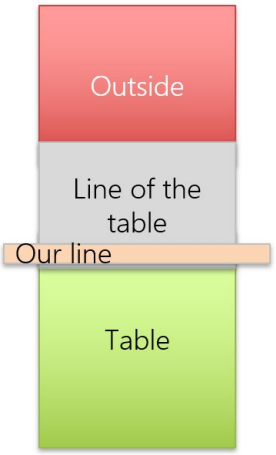
Is the user that can choose the best method for its setup and resolution specifying his choice in the parameter PAR.pickingMode.

5.2.4 Keypoints colour evaluation

This evaluation is done in the function

chosen = areGoodIntKeyPnt(keyPnt,Ihsv,PAR)

As input it takes the 4 key points and some detection parameters to define the colours, and give a boolean value as output, true if the keypoints are well coloured, false otherwise.

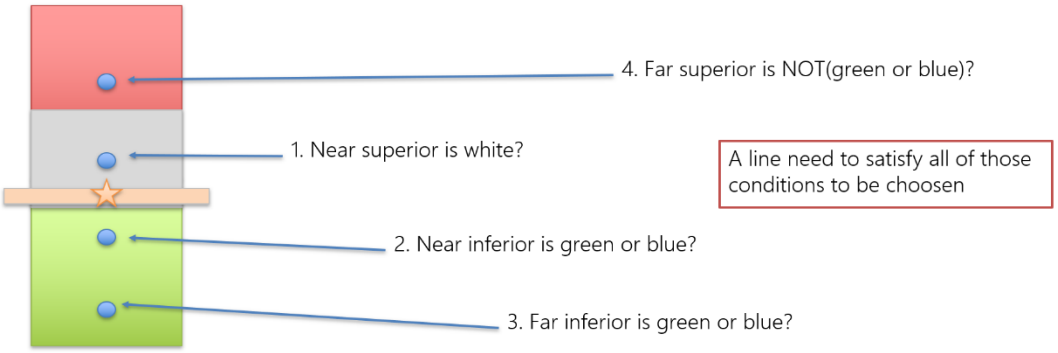


We are going to select just the internal lines, so the lines closer to the green or blue part of the table. This because if we consider the external lines, we don't have sufficient robust constrains for all keypoints.

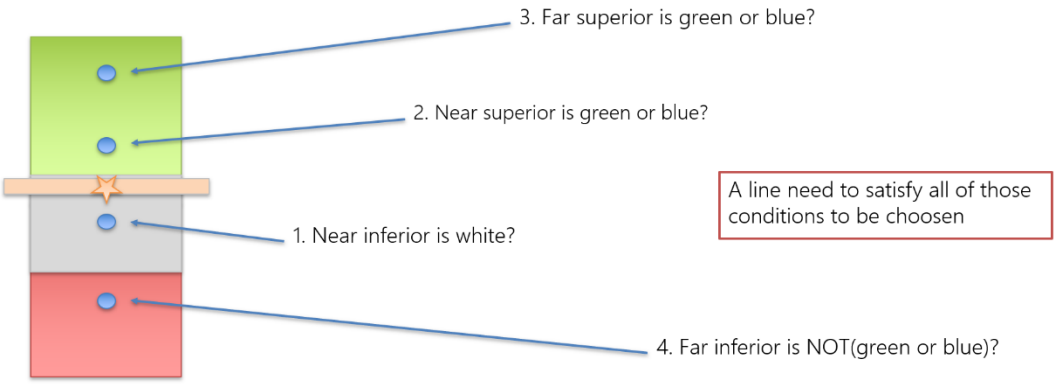
This practically exclude the line width from our playable area, but knowing its dimension a correction is done during the 3D reconstruction

There are two cases in which the color of the keypoints is right depending on which side of the table the line is.

First Case:



Second Case:



The colours in the HSV space are detected relying on some detection parameters passed as inputs:

```
PAR.HRange      = 0.40 ; %[0-1] H range for the same color
PAR.S4White     = 0.32 ; %[0-1] Max Saturation for White
PAR.V4White     = 0.65 ; %[0-1] Min Brightness for White
PAR.V4Color     = 0.20 ; %[0-1] Min Brightness for Colors
```

There are specific functions for each colour that give as output a boolean value if the specific colour is detected. Functions use the parameters as thresholds like:

White: r = color(2)<PAR.S4White && color(3)>PAR.V4White;

Green: r = isColor(color,PAR) && color(1)<1/3+PAR.HRange/2 && color(1)>1/3-PAR.HRange/2;

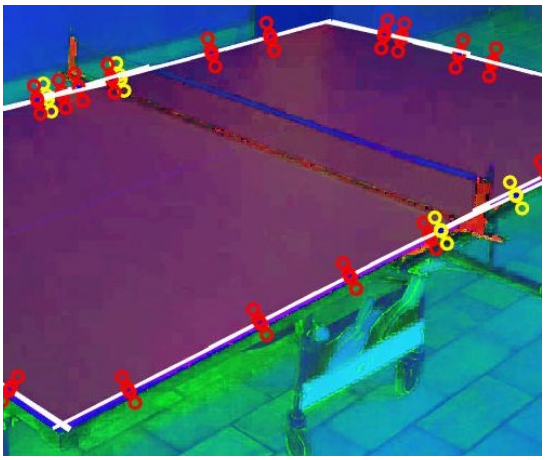
Blue: r = isColor(color,PAR) && color(1)<2/3+PAR.HRange/2 && color(1)>2/3-PAR.HRange/2;

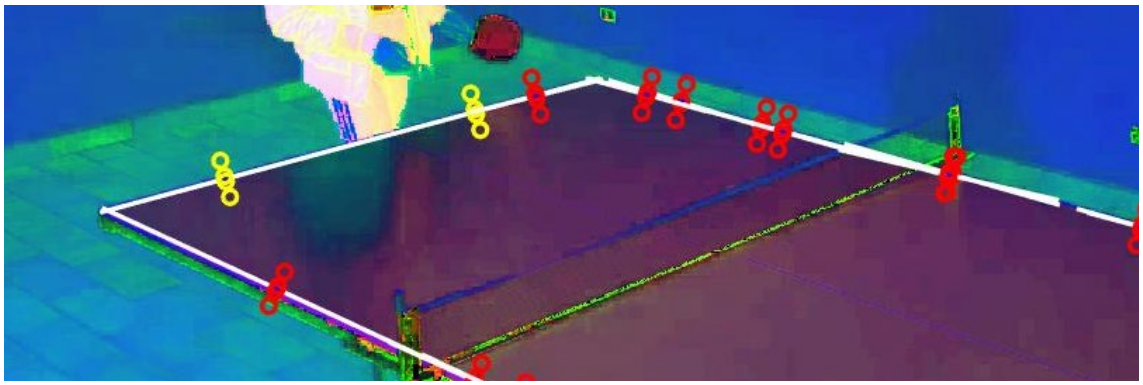
5.2.5 Line Revaluation

If a line doesn't pass the selection at the fist attempt doesn't mean that is to discard yet, indeed could be that the keypoints were taken in a bad spot. Is better to further inspection the line changing the key points.

This is done with 2 different strategies:

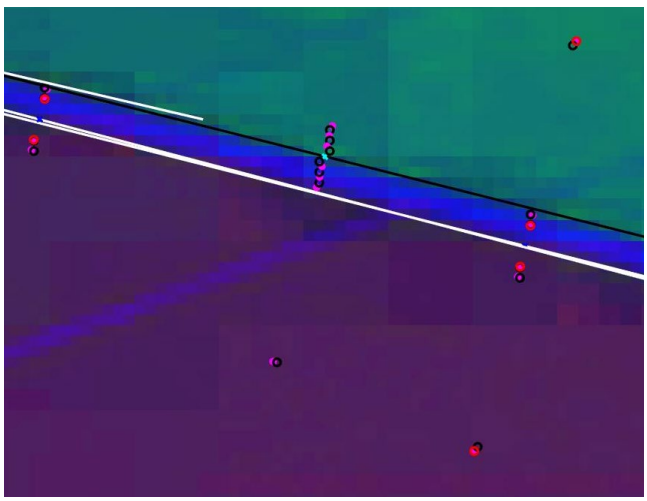
1. Try to divide the line in two parts taking its midpoint as an endpoint for both parts; then take the new keypoints from the new midpoint of each part. This helps in case there are colour reflections on the table that change colour in that spot, also is robust to the noise produced by the net in the middle of the sides.
   The evaluation process is started again with those new keypoits.
   The overall line is selected if both its parts are selected





2. Reducing the nearTh: as said before is likely that for thin and far lines the initial threshold is too big and the key points exceed the white area of the line.

   All keypoints are recomputed end revaluated with the new threshold.



The process ends when all lines are evaluated, and for each line the evaluation process end when:

- The line is chosen
- The nearTh is zero and the two-midpoints evaluation is done -> Line discarded

This following image report a piece of what is printed by the algorithm chosenig the full visualization in the PAR.showSteps.

Note that the line is chosen after the reduction of nearTh.

```
Evaluating Line #102, theta =  1.3439
   nearTh = 3
   Try with one middle point
      start[ 647  547], end[ 838  503], mid[ 742  525] :
         Near Superior [0.062 0.212 0.647] is NOT White
         Near Inferior [0.333 0.099 0.396] is NOT White
   Try with two middle points
      start[ 647  547], end[ 742  525], mid[ 695  536] :
         Near Superior [0.067 0.420 0.561] is NOT White
         Near Inferior [0.467 0.050 0.392] is NOT White
   nearTh = 2
   Try with one middle point
      start[ 647  547], end[ 838  503], mid[ 742  525] :
         Near Superior [0.133 0.048 0.820] is White
         Near Inferior [0.350 0.114 0.345] is a Color (is Green)
         Far Inferior  [0.319 0.120 0.392] is a Color (is Green)
         Far Superior  [0.068 0.472 0.631] is a Color (is NOT Green) is a Color (is NOT Blue)
      --> CHOSEN by 1
```

5.3 SIDES RECOGNITION

The objective of this function is to cluster the endpoints of the selected lines. In each cluster there are the points belonging to same possible side (they will fitted in the last step)

The algorithm as input takes Selected lines and gives as output a struct containing the clusters of points representing the sides

Note: In this step we don't know how much clusters we end with, it could be more then 4 if some lines not belonging to a side is wrongly selected in previous step. (For example the upper line of the net if its keypoints fall in the net texture)

The main procedure can be sintetize as

1. Takes the first line
   1.1. New side creation
   1.2. Store its endpoints and its direction as an approximative reference direction for the side
   1.3. Invalidate the line (to consider it only once)

2. Search the lines with similar orientation within range (+- PAR.thetaTh)
   2.1. Store its endpoints into the same cluster
   2.2. Invalidate the line (to consider it only once)

3. If there isn't another similar oriented line, takes the first valid line and repeat the process from 1

4. The Algorithm ends when all lines are invalidated

5.4 Sides Sorting and Purging

The objective of this Algorithm is to select only the 4 clusters that represent the 4 sides of the table discarding the false positive. In addiction it sorts the sides by parallelism.

This algorithm takes as input the clusters of points representing the possible sides and gives as output the sorted clusters of points of the 4 sides

The procedure consist in 3 steps:

1. Compute the center of each cluster by averaging the coordinates of the points, and save it as a new field of the cluster.
2. Sort the clusters by their theta reference computed in the previous step
3. Purge the unwanted clusters knowing the table geometry

Now having clusters ordered by their theta reference, means that the first two are referred to the two parallel sides having the lower direction

Therefore, if there aren't outliers, the 3rd cluster has a theta reference greater then the first two by a large amount (specified by the parameter rho), otherwise the 3rd cluster is oriented similarly to the first two.

In case of ambiguity only the clusters with the minimum and the maximum y of their center are taken as valid from all similarly oriented clusters.

At the end we have 4 clusters, 2 for each direction, respectively with the minimum and maximum values of y-coordinate of their centers.

5.5 SIDES FITTING

The objective of this section is finally to fit the side's clusters and find the table's vertices.

Each cluster is fitted with a line, so it uses a polynomial fitting with a first order polynomial. The fitting returns the vector of parameters [A C] of the explicit line eqn.

$y = Ax + C$

Then, after putting the lines in the form [a b c], it computes the table's vertices in homogeneous coordinates using the cross product between those lines

The result is a vector containing the coordinates of the four vertices ordered clockwise with the same starting vertex in both views (for doing vertices sorting an information of which camera belong the analyzed frame needs to be passed as input

6 BALL TRACKING

6.1 GENERAL STRUCTURE

For this algorithm we use and re-adapt the algorithm in the paper [7].

These are the general overview and the pseudo-code of our algorithm.

Algorithm 1 Ball Tracking Algorithm

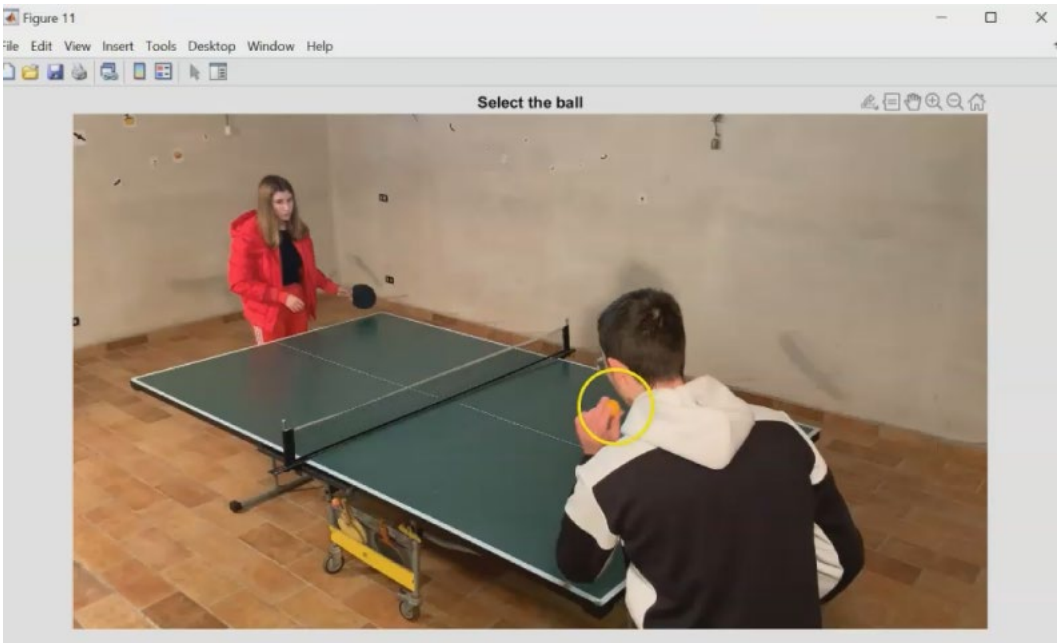
while Video has frame **do**
  frame ← Read video frame
  if First Frame **then**
    Positions ← User input initial ball position
    Thresholds ← Define initial color thresholds
    ROI ← Crop ROI based on frame and initial position
    Frame ← Read video frame
  else if Ball found before **then**
    ROI ← Crop ROI based on previous found position
  else ▷ In case the position was not found previously
    ROI ← Adaptive control of the ROI
  end if
  ROI ← Blurr ROI ▷ To smooth the edges we apply a blurring filter in the ROI
  CandidateBalls ← First Pass(ROI) ▷ First pass has a coarse threshold
  if *CandidateBalls* were found **then**
    for each *CandidateBall* **do**
      ROI ← Add the ROI of the *CandidateBall*
    end for
    CandidateBalls Second Pass(ROI) ▷ Second pass has a relaxed threshold
    Objects ← Object Evaluation
    Positions ← Add *Object* with smallest error from *Objects*
  end if
end while
Trajectory ← Calculate 2D Trajectory

The Inputs of this algorithm are:

- DX Video frames
- SX Video frames
- Ball Diameter (12 pixels)
- ROI Dimension (20*Ball diameter)
- Parameter if ball not found (K = 3)
- First Threshold
- Second Threshold
- Maximum number of iteration (5)

Firstly the user is requested to select the ball from the first frame of each video; from this user selection (both on DX and SX first frame) the algorithm:

- Compute the starting Ball position
- Define the ball color (hsv)
- Define the ROI square: define the first ROI as a square centered in (X,Y), the dimension of this square depends on the previously chosen parameters (ROI Dimension). This should be a trade-off between reliability and computational cost.



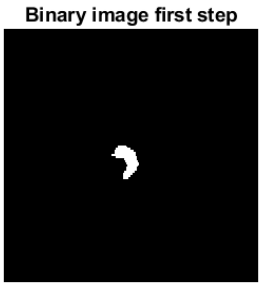
6.2 ROI AND FIRST PASS THRESHOLDING

Algorithm 2 First Pass Thresholding

while *CandidateBall* not found AND $i < MaxIterations$ **do**
  BinaryROI ← Color threshold the ROI ▷ returns a binary image with the white pixels representing the color of the ball
  if Only one object is found in the *BinaryROI* **then**
    CandidateBall ← set as found
    BestFirstThreshold ← set current thresholds as best
  else if More then one object found in the *BinaryROI* **then**
    if Less objects were found than in the previous iteration **then**
      BestNumberOfObjects ← *NumberOfObjects*
      BestFirstThreshold ← set current thresholds as best
    end if
    FirstThreshold ← Increase the Threshold
  else
    FirstThreshold ← Decrease the Threshold
  end if
  i ← increase the iteration number
end while
FirstThreshold ← *BestFirstThreshold*
return FirstThreshold, BinaryROI

Considering only the ROI, the next step is the First Pass Thresholding.

Cutting the image and considering only the ROI Square, the system extrapolates a binary image where all pixels with a color difference smaller than the threshold (first threshold) are turned white and all the pixels with a color difference greater than the threshold are turned black.



Then, using the function 'bwconncomp' we compute the connected components of our binary image.

1)If the number of object is exactly one, we found the candidate ball, and so we set the current Threshold as the best First Threshold.

2)If we have more than one object and If the number of object is smaller than the previous iteration than we are going in the right direction, so we update the threshold and proceed decreasing the threshold (if it remains inside the minimum set value =4)

first_threshold=first_threshold(100-m)/100 (m is a parameter set to 5)*

After that we repeat the process if the number of iterations is under the maximum allowed.

3)If no object was found we are going in the wrong direction and we have to increase the threshold:

first_threshold=first_threshold(100+m)/100 (m is a parameter set to 5)*

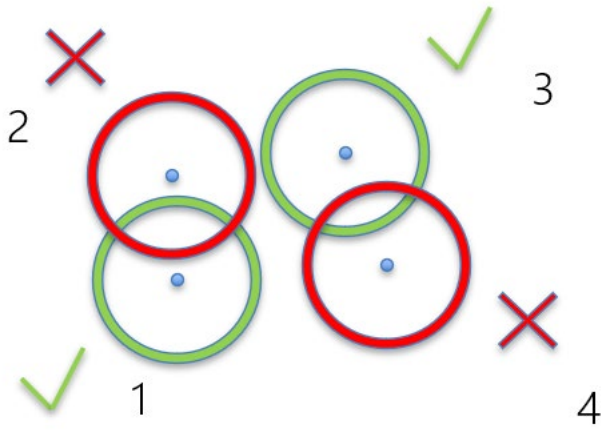
When the process is finished, so when the maximum number of iterations allowed is reached or only one candidate ball is found; we save the Threshold as the best threshold found, and with that threshold we re-compute the binary image of the ROI.

6.3 CENTROIDS AND CANDIDATE BALLS DEFINITION

Firstly, we perform the computation of the centroids of the connected components of the ROI.

Then we have to define the candidate Ball, starting from the centroid and the (known) dimension of the ball we save the first centroid and all centroids that are far enough from the other saved, that means that the distance between centroids is greater than the ball diameter.

All other centroids that doesn't satisfy this rule are discarded.



6.4 SECOND PASS THRESHOLDING

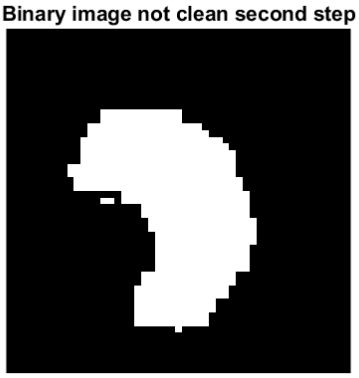
Algorithm 3 Second Pass Thresholding

```
for each ROI do
    SecondThreshold ← restore initial thresholds ▷ Each ROI cannot
be affected by the other, so they all start with the same thresholds
    while ObjectArea not OK AND i < MaxIterations do
        BinaryROI ← Color threshold the ROI
        for each of the objects in the BinaryROI do
            Error ← Calculate the percentage error between the areas
of the object and the ball
            if Error is within threshold then
                ObjectArea ← OK
                BestSecondThreshold ← set current threshold
            else if Object area is greater than the ball area then
                if smallest percentage error so far then
                    BestPercentageError ← Error
                    BestSecondThreshold ← set current threshold
                end if
                SecondThreshold ← Increase the Threshold
            else
                if smallest percentage error so far then
                    BestPercentageError ← Error
                    BestSecondThreshold ← set current threshold
                end if
                SecondThreshold ← Decrease the Threshold
            end if
        end for
        i ← increase the iteration number
    end while
    BinaryROI ← recalculates the best ROI found
    Binaries ← Add the best BinaryROI of each object
    Thresholds ← Add the best BestSecondThreshold of each object
end for
return Threshold, Binaries
```

For each candidate ball the process re-start from the second threshold, and so the binary image is re-computed.

The next step is to compute the area of the candidate object, compare it with the theoretical area of the ball and compute the percentage error between areas:

fun=(ball_area-area_candidate)/ball_area



- If the result ‘fun’ is lower than the module of the Error Threshold, this means that we found the ball thus we save the error and the second threshold

- If the result ‘fun’ is greater than the Error Threshold, we have to decrease the threshold and repeat the process with the new second threshold:

  second_threshold=second_threshold(100-g)/100*

- If the result ‘fun’ is lower than the Error Threshold and outside the allowed range, we have to increase the threshold and repeat the process with the new second threshold:

  second_threshold=second_threshold(100+g)/100*

The process ends when the candidate object area is within the acceptable range or when the maximum number of iteration is reached.

The output is the binary image of the candidate object with the relative second Threshold.

6.5 FIRST STAGE EVALUATION

Firstly we 'clean' the binary image by removing all the small objects around the principal one.

Then next step is to evaluate our final candidate object through a Two-Stages Evaluation.

The first step is eliminatory and checks if it has rounded upper contour (RUC), if the location is consistent with the predicted location (T), if it exhibits motion at its center (Mc) w.r.t. the previous position and if the position of the centroid of the predicted object had some motion w.r.t. to the previous position of the ball (Mp).

The parameters RUC, T, M are formally defined in the table 1., each indicator gives as result 0 if is out of the acceptable threshold and 1 if it is.

For the indicators we introduce the next predicted position: this function applies linear extrapolation in order to predict the position of the ball in the next frame; the inputs are the previous position, the actual position of the ball and the frame (to check if the predicted position is inside).

Rounded Upper Contour (RUC)	$RUC = \begin{cases} 1, & \text{if } E_{RUC} < t_{RUC} \\ 0, & \text{if } E_{RUC} \geq t_{RUC} \end{cases}$ where t_{RUC} is a preset threshold and E_{RUC} is an objective (error) function defined as: $E_{RUC} = \frac{\sum \lvert \frac{d_i - r}{r} \rvert}{N}$ and $d_i = \sqrt{(x_i - x_c)^2 + (y_i - y_c)^2}$ where $(x_i y_i)$ is a pixel in the upper contour, i the pixel index, N the number of pixels, r the radius and (x_c, y_c) the centre obtained by solving the equation of a circle for (x_i, y_i).
Position (T)	$T = \begin{cases} 1, & \text{if } L_{diff} < t_T \\ 0, & \text{if } L_{diff} \geq t_T \end{cases}$ Where L_{diff} is the Euclidean distance between the OOI actual and predicted positions and t_T is a threshold. The predicted location is the linear extrapolation of OOI locations in previous frames.
Motion (M)	$M = \begin{cases} 1, & \text{if } OC_{diff} < t_M \\ 0, & \text{if } OC_{diff} \geq t_M \end{cases}$ Where OC_{diff} is the Euclidean distance between the candidate ball and the OOI in the previous frame and t_M is a threshold.

6.6 SECOND STAGE EVALUATION

- If less than 2 indicators are positives the candidate object is discarded.

- If at least 2 indicators are positives we have a good probability that the candidate object is really the ball, and we proceed by computing the estimation error considering Eruc, area, width, height, perimeter and roundness.

Area (A)	Number of pixels in the object.
Maximum width (W)	Horizontal distance between left and rightmost pixels of the object.
Maximum height (H)	Vertical distance between top and bottom most pixels of the object.
Perimeter (P)	Length of object boundary
Roundness (R)	Given by: $R = \frac{4\Pi A}{P^2}$

$$E = \frac{E_{RUC} + w_A \frac{|A_c - A_b|}{A_b} + w_W \frac{|W_c - W_b|}{W_b} + w_H \frac{|H_c - H_b|}{H_b} + w_P \frac{|P_c - P_b|}{P_b} + w_R \frac{|R_c - R_b|}{R_b}}{(n_p + 1)n_c}$$

- Between all the candidate object the algorithm chooses the one with the lower error.

At the end of this step:

If the ball is found:

- Save the position of its center inside the trajectory
- Reset the ROI dimension at the initial value (20*ball_diameter)
- Set the ROI position in the next predicted position
- Update the ball color
- Go to the next Frame

If the ball is not found:

- Save a null value Nan inside the trajectory
- Increase the ROI based on the initial setted parameters K
  *(NewROI = k*currentROI)*
- Go to the next frame

All these Steps are done for each frame, at the end our result is the trajectory vector with all the results for each frame (position or Nan).

7 2D TRAJECTORY ESTIMATION

7.1 GENERAL STRUCTURE

Our algorithm's purpose is to be robust in front of :

- Tricky situation with occlusions
- Not synchronized DX and SX Videos
- Hypothetical not well-working Ball tracking algorithm

We have to estimate and synchronize 2D Trajectories from the (not synchronized) output of the ball tracking algorithm.

For each frame the Ball Detection function returns:

- The coordinates in pixels of the estimated position (both for DX and SX cameras)
- Nan (null value) if in the number of iteration the function can't find the ball position

So we start from a vector of positions where each position is associated with the (relative) time of the frame at which that position was estimated.

7.2 CONTINUOUS 2D TRAJECTORY COMPUTATION

This step is done in order to decrease possible de-synchronization errors coming from cameras.

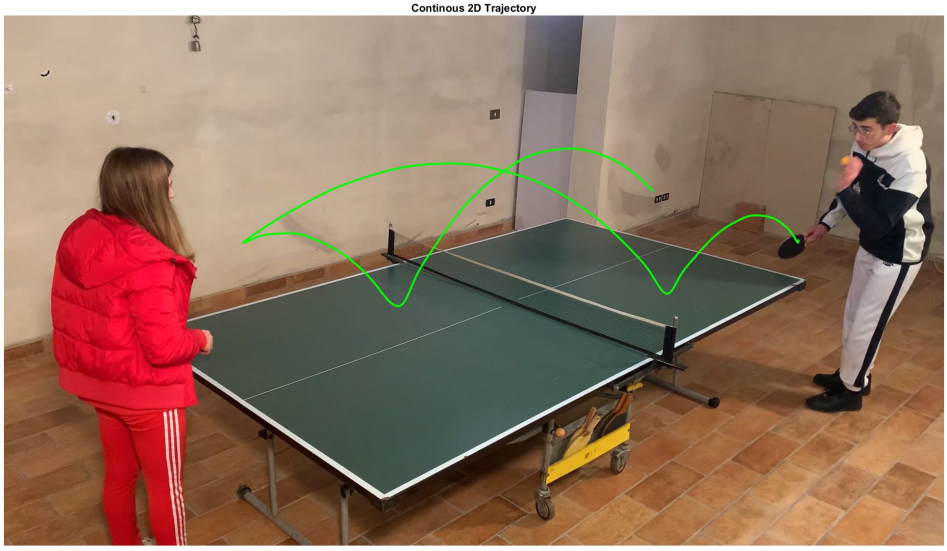
Using the 'Spline' interpolation algorithm we interpolate the trajectories considering all positions associated with their frame's time, starting from the first not-null value and ending with the last not-null value of each trajectory.

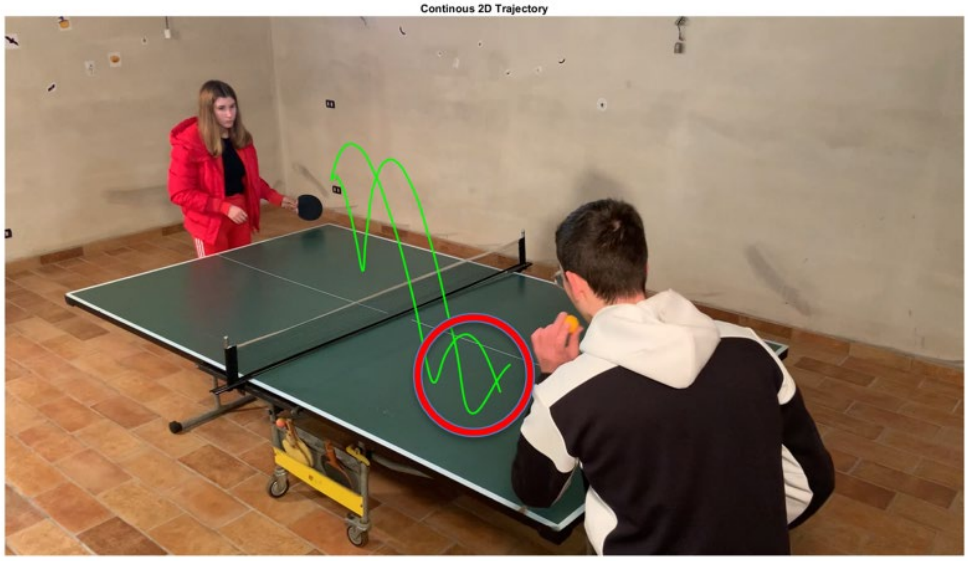
From the interpolated trajectory we extrapolate positions at regular frequency.

This time interval must be a trade-off between accuracy and required computational power, we choose 10*frame rate.

In Table Tennis the speed of the ball is high (over 100 km/h) and so the frame rate of the camera is particularly important in order to achieve an optimal accuracy, however we can interpolate the trajectory as well as possible.

Note that the DX and SX trajectories could be different due to the ball tracking algorithm, or due to something that can occlude the camera during the game, they must be synchronized.





7.3 Bounce and Shot Detection

Starting from the interpolated trajectory, we analyze each point of the trajectory with the bounce detection algorithm and the Racket Shot algorithm in order to analyze if it could be an important Synchronization point.

7.3.1 Bounce Detection Algorithm

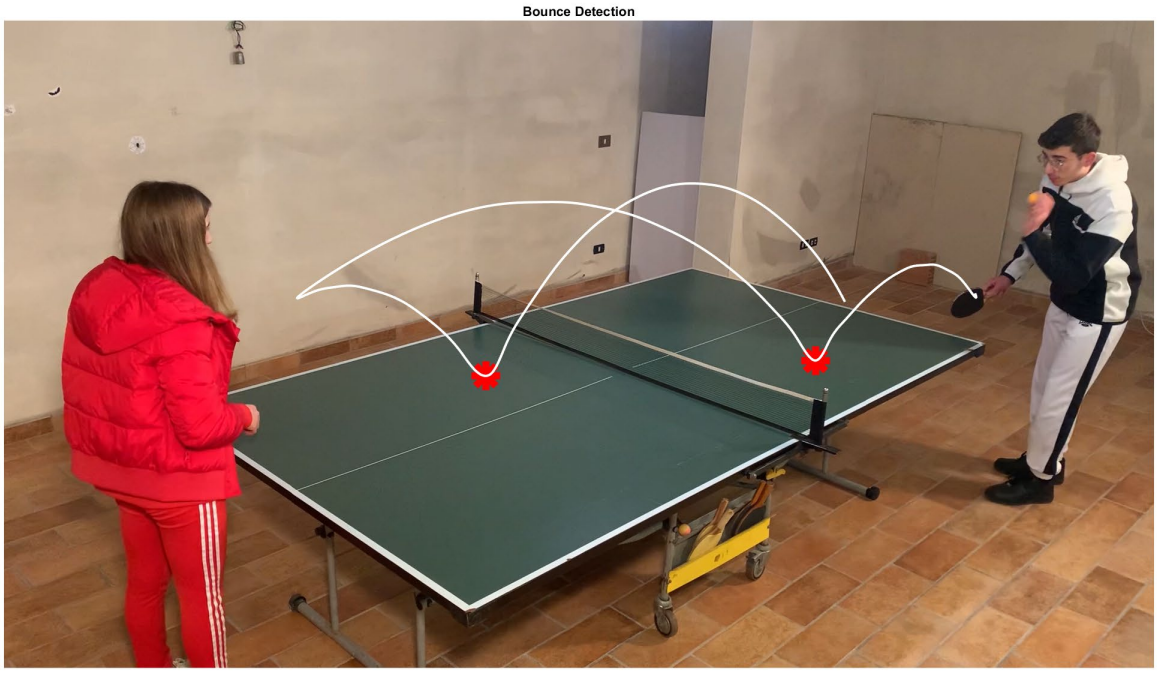
For each point P(i) of the trajectory we consider the four previous points P(i-1), P(i-2), P(i-3) and P(i-4) and the four consecutive points P(i+1), P(i+2), P(i+3) and P(i+4).

If :

- All the four previous points have a decreasing Y coordinate
- All the four consecutive points have an increasing Y coordinate
- The direction of the X coordinate remains the same

Then the analyzed points is a bounce point !

Note that the two trajectories could have different lengths because of ball tracking.



7.3.2 Shot Detection Algorithm

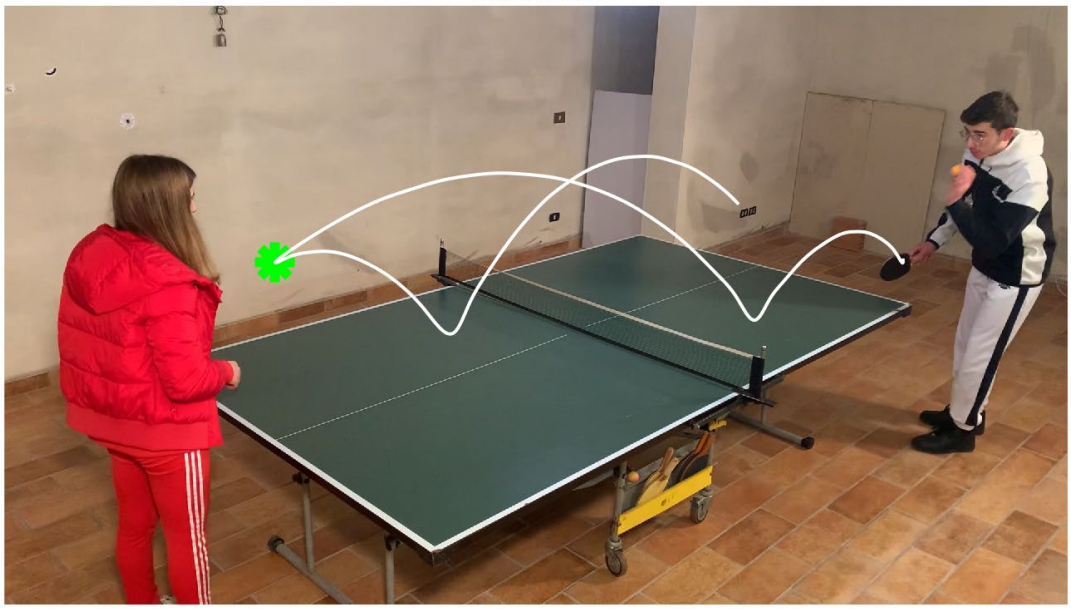
For each point P(i) of the trajectory we consider the four previous points P(i-1), P(i-2), P(i-3) and P(i-4) and the four consecutive points P(i+1), P(i+2), P(i+3) and P(i+4).

If :

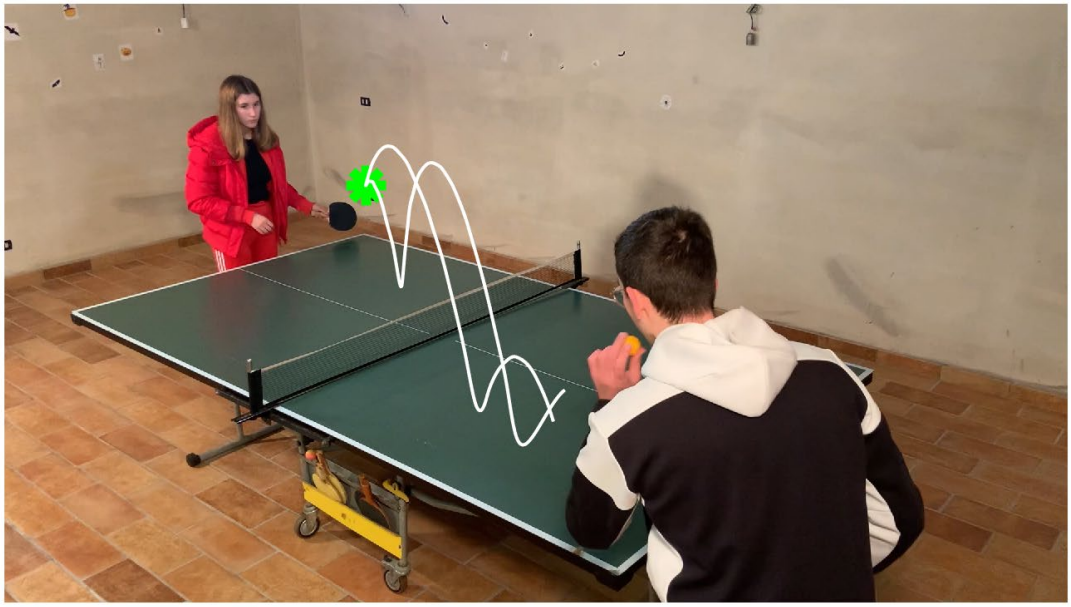
- All the four previous points have a decreasing/increasing X coordinate
- All the four consecutive points have an increasing/decreasing (opposite) X coordinate

Then the analyzed points is a Shot point !

Shot Detection



Shot Detection



7.4 TRAJECTORIES SYNCHRONIZATION

7.4.1 Method 1: Synchronization in Time

If we have already synchronized cameras all the trajectory points of DX and SX cameras are already matched in time, but our trajectories are results of the ball tracking function, that could not detect the ball immediately.

We want our DX and SX trajectories to start at the exact same point and to finish at the exact end point.

- We consider the first point of each trajectory and the time of point with the higher time (so the point of the trajectory that starts after) becomes the time of the first points of our final trajectory.
- We consider the last point of each trajectory and the time of point with the lower time (so the point of the trajectory that ends earlier) becomes the time of the last points of our final trajectory.

7.4.2 Method 2: Synchronization Bounce-based

If the cameras are not synchronized, we can synchronize the two videos considering the last bounce.

We know at-priori that the bounce instant between DX and SX trajectory must be the same.

Step 1: computation of Delta-time, so the time difference between the last bounce instant of the DX trajectory and the last bounce instant of the SX trajectory.

Step 2: Re-scaling of the time of all the points of the latest trajectory (according to the previous step) . The new time for each point of the trajectory is Tnew = Told – DeltaTime.

Step 3: Once SX and DX trajectories have points synchronized in time, the final step is to perform the Synchronization in time.

7.4.3 Method 3: Synchronization Shot-based

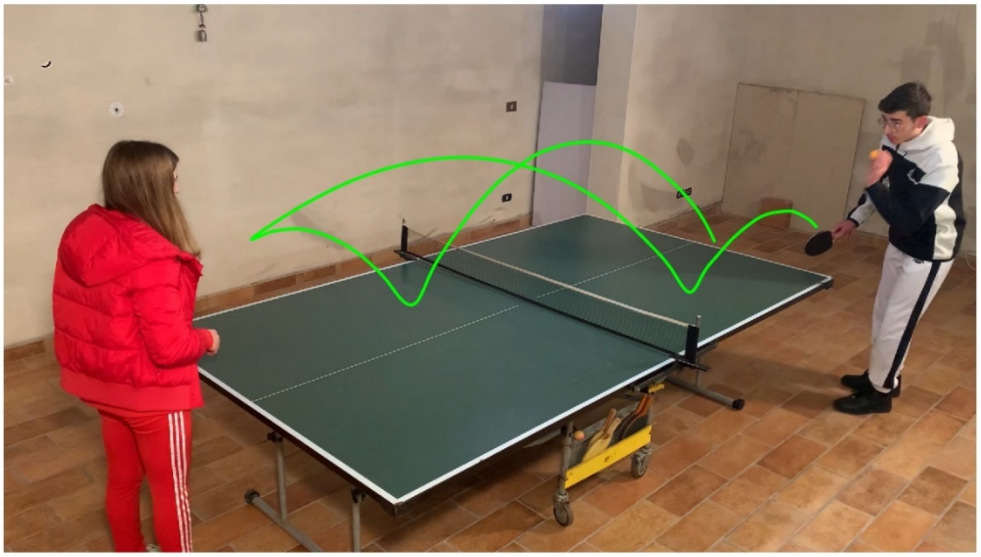
If the cameras are not synchronized and for some reasons we can't perform the bounce based synchronization (for example because we are considering a short trajectory, or because the ball bounce on the edge of the table and the algorithm can't recognize the bounce), we can synchronize the two videos considering the last shot.

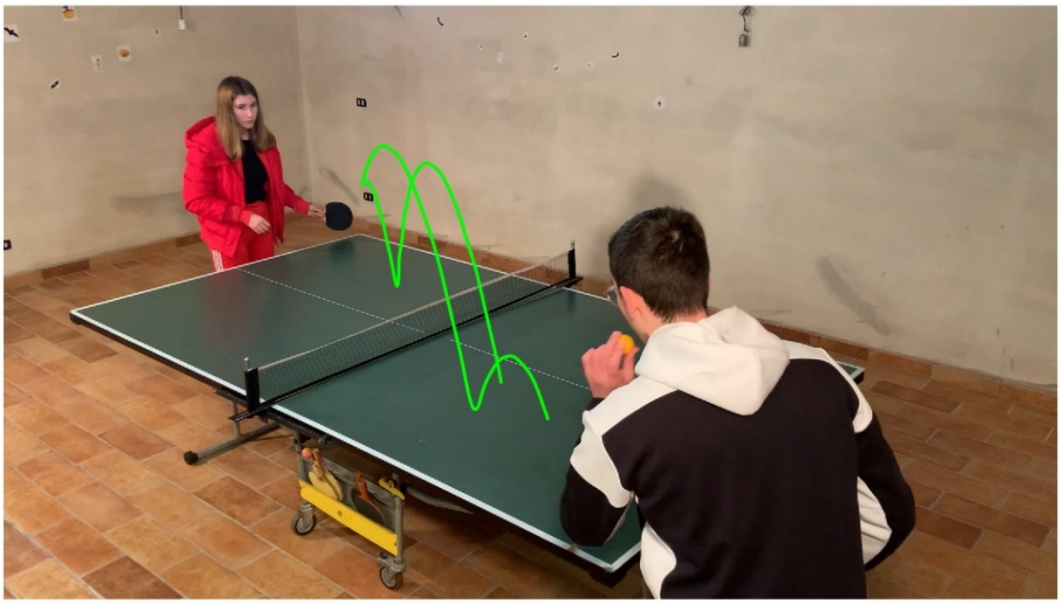
We know at-priori that the Shot instant between DX and SX trajectory must be the same.

Step 1: computation of Shot-Delta-time, so the time difference between the last shot instant of the DX trajectory and the last shot instant of the SX trajectory.

Step 2: Re-scaling of the time of all the points of the latest trajectory (according to the previous step) . The new time for each point of the trajectory is Tnew = Told – ShotDeltaTime.

Step 3: Once SX and DX trajectories have points synchronized in time, the final step is to perform the Synchronization in time.





OUTPUT:

The final result of these procedures is that the final DX and SX trajectories are representing exactly the same scene from the first to the last point and they are fully synchronized in-time.

Now we can consider these trajectories to reconstruct the final 3D Trajectory.

8 3D RECONSTRUCTION

We use a checkboard calibration because the results are much more reliable and because in this way we can determine uniquely the final 3D estimation from two calibrated cameras.

It consists of capturing different views of the checkerboard pattern (20 for each camera) and through the Matlab script calculate the calibration matrix and other camera parameters.

The use of a checkered board to calibrate the cameras provides a more accurate calibration (plus removal of distortion if needed), especially that the accuracy is required for this kind of applications.

8.1.1 Calibration:

DX Calibration:

- Intrinsic Matrix K

1710.028	0	905.8785
0	1707.25024	508.0093431
0	0	1

From K:

- Focal length = [1710.0285 , 1707.25]
- Principal point = [905.58 , 508.009]

SX Calibration is the same being the two I Phones identical:

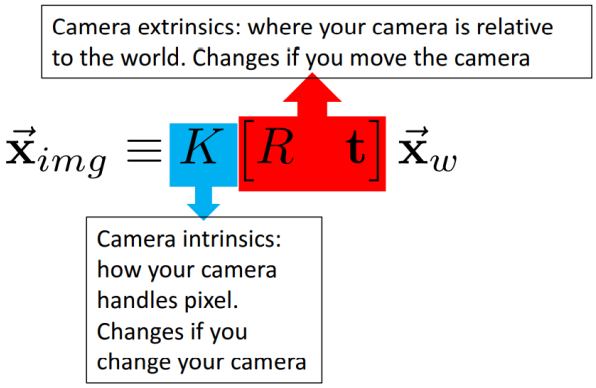
- Intrinsic Matrix K

1710.028	0	905.8785
0	1707.25024	508.0093431
0	0	1

From K:

- Focal length = [1710.0285 , 1707.25]
- Principal point = [905.58 , 508.009]

The next step is to extrapolate the extrinsics from camera intrinsics parameters, table vertices in pixel coordinates (computed in lines detection) and correspondent World Vertices points, that we arbitrarily choose based on official regular dimensions of professional Ping Pong tables in Tournaments.



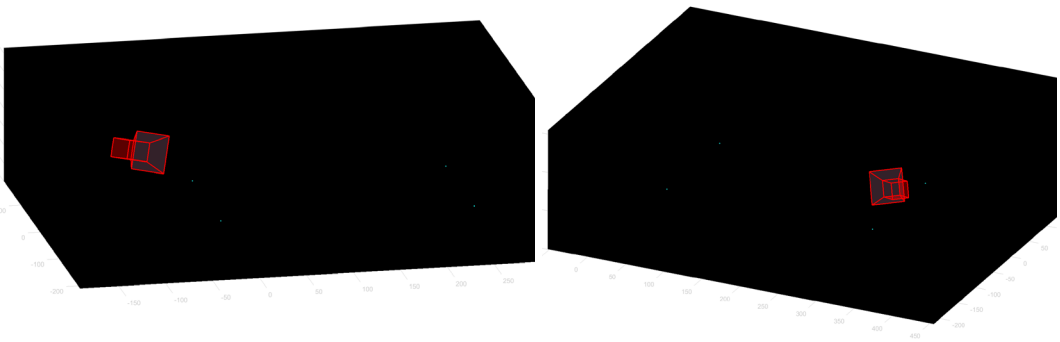
The outputs are:

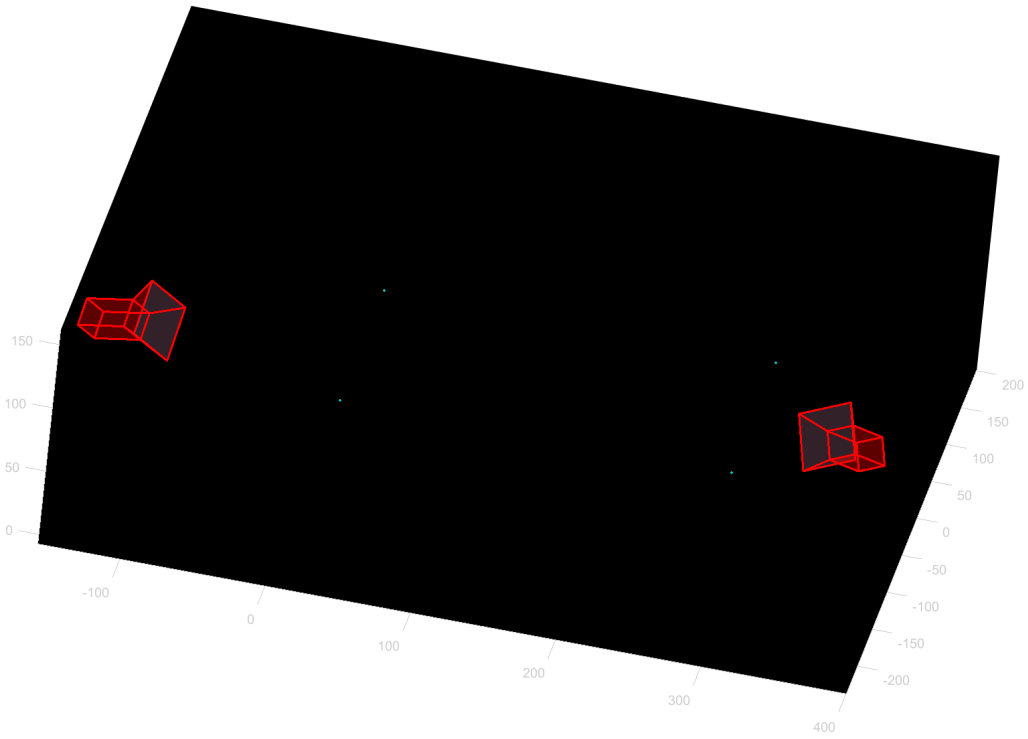
- Rotation Matrix: 3-by-3 matrix describing rotation from the world coordinates to the camera-based coordinates
- Translation Vector: 3-by-1 vector describing translation from the world coordinates to the camera-based coordinates

The next step is to easily compute from the Rotation Matrix and the translation vector, the 3-by-3 camera orientation matrix and the 1-by-3 camera location vector in world coordinates.

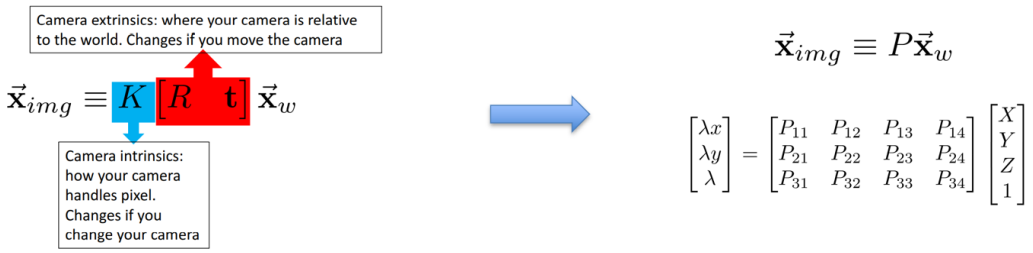
Camera orientation = rotationMatrix'

*location = -translationVector * rotationMatrix'*





In order to perform the final Triangulation we need the 3 by 4 Projection Matrix P of each camera which projects a 3D world point in homogeneous coordinates into the image.



Usually, in order to compute the Intrinsic Matrix and perform the Calibration, for each point we know in the world coordinates (X, Y, Z) and in image coordinates (x, y), this provides two equations of this type:

$$(P_{31}X + P_{32}Y + P_{33}Z + P_{34})x = P_{11}X + P_{12}Y + P_{13}Z + P_{14}$$
$$(P_{31}X + P_{32}Y + P_{33}Z + P_{34})y = P_{21}X + P_{22}Y + P_{23}Z + P_{24}$$

And so through at least 6 non-coplanar points we can obtain the necessary constraints to obtain the Projection matrix.

In our case we do not have 6 non-coplanar and known clear points in the image, and so we compute firstly the calibration matrix through the checkboard calibration, and then:

P = K[R t]*

Where:

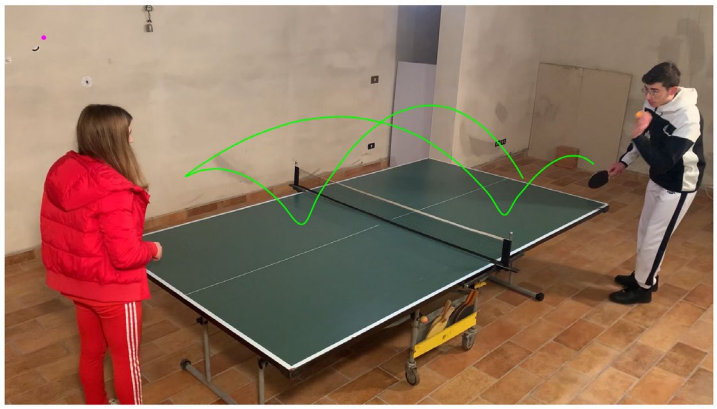
- P is the 3X4 Projection matrix
- K is the 3X3 Intrinsic matrix
- R is the 3X3 Rotation matrix
- t is the 3X1 Translation Vector

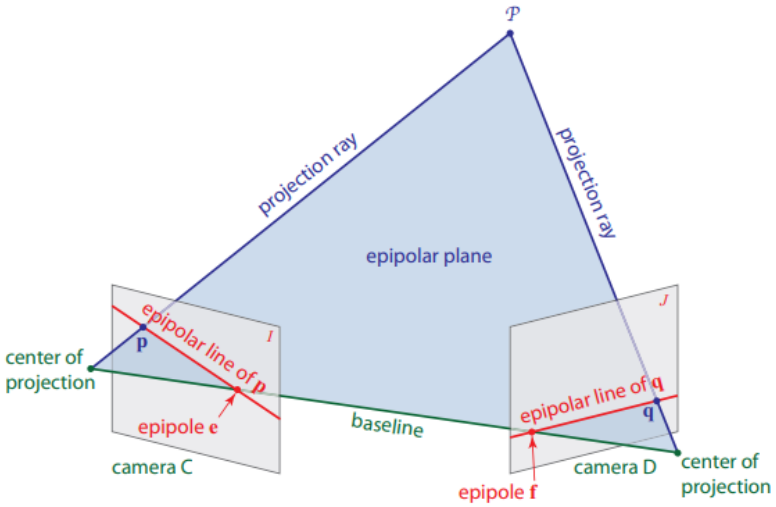
Our triangulation is feasible and unique

$(P_1, P_2) \mapsto F$ unique

INPUT :

- DX Final 2D Trajectory
- SX Final 2D Trajectory
- DX Projection Matrix P1
- SX Projection Matrix P2





$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \longleftarrow \vec{\mathbf{x}}_{img}^{(1)} \equiv P^{(1)}\vec{\mathbf{x}}_w \qquad \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \longleftarrow \vec{\mathbf{x}}_{img}^{(2)} \equiv P^{(2)}\vec{\mathbf{x}}_w$$

One image gives two equations:

$$X(P_{31}^{(1)}x_1 - P_{11}^{(1)}) + Y(P_{32}^{(1)}x_1 - P_{12}^{(1)}) + Z(P_{33}^{(1)}x_1 - P_{13}^{(1)}) + (P_{34}^{(1)}x_1 - P_{14}^{(1)}) = 0$$

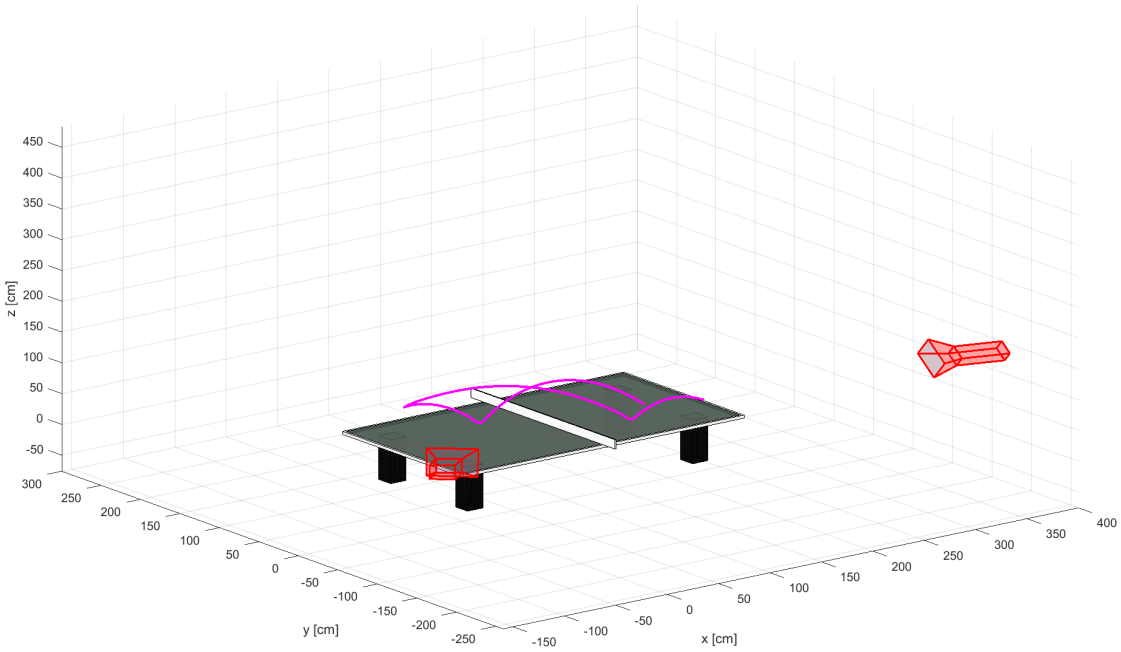
$$X(P_{31}^{(1)}y_1 - P_{21}^{(1)}) + Y(P_{32}^{(1)}y_1 - P_{22}^{(1)}) + Z(P_{33}^{(1)}y_1 - P_{23}^{(1)}) + (P_{34}^{(1)}y_1 - P_{24}^{(1)}) = 0$$

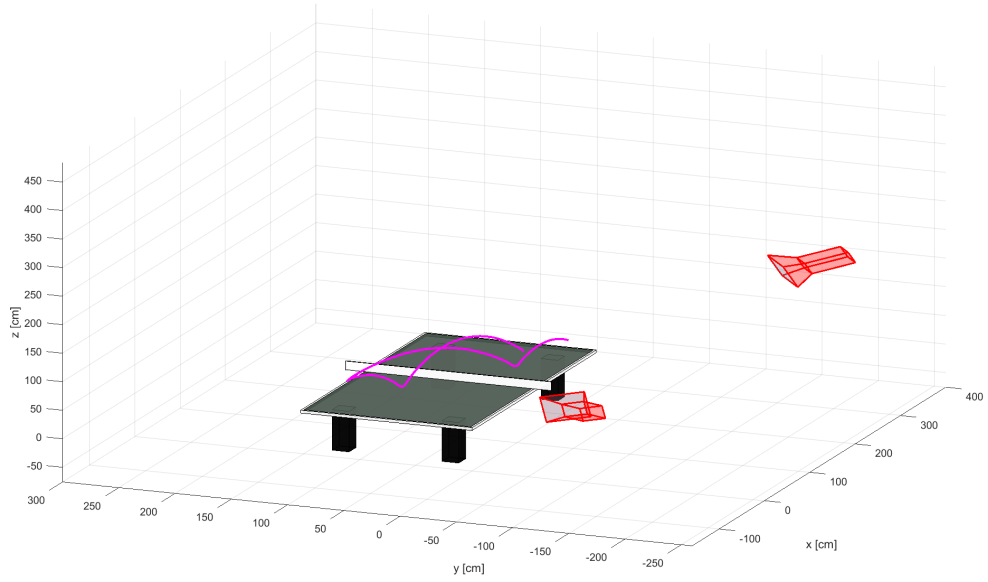
$$x_1 = \frac{P_{11}^{(1)}X + P_{12}^{(1)}Y + P_{13}^{(1)}Z + P_{14}^{(1)}}{P_{31}^{(1)}X + P_{32}^{(1)}Y + P_{33}^{(1)}Z + P_{34}^{(1)}}$$

$$y_1 = \frac{P_{21}^{(1)}X + P_{22}^{(1)}Y + P_{23}^{(1)}Z + P_{24}^{(1)}}{P_{31}^{(1)}X + P_{32}^{(1)}Y + P_{33}^{(1)}Z + P_{34}^{(1)}}$$

With the two images we can solve linear equations to get 3D point location!!

The final Output is the 3-D ball Trajectory in world coordinate system!

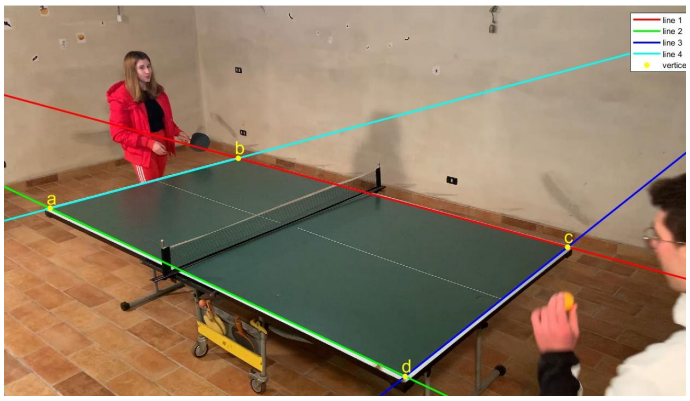

z [cm]
450
400
350
300
250
200
150
100
50
0
-50
y [cm]
300
250
200
150
100
50
0
-50
-100
-150
-200
-250
x [cm]
-150
-100
-50
0
50
100
150
200
250
300
350
400


z [cm]
450
400
350
300
250
200
150
100
50
0
-50
y [cm]
300
250
200
150
100
50
0
-50
-100
-150
-200
-250
x [cm]
-100
0
100
200
300
400

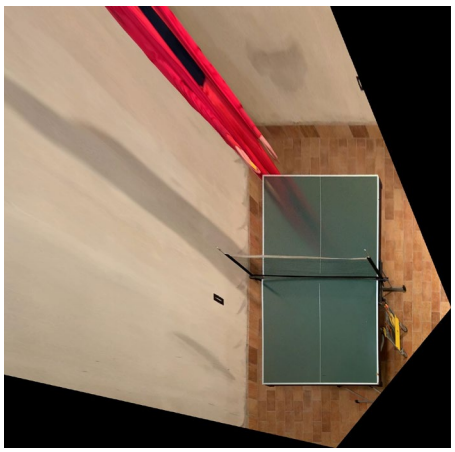
9 CONCLUSIONS & FUTURE DEVELOPMENTS

9.1 TABLE DETECTION

Our Table Detection algorithm is fully automatic without need of human operation. It is robust and precise, furthermore we develop it in such a way to make it capable of recognizing both green and blue official tables, so it can be used in all official matches.



Starting from the knowledge learned in class we rectify the first frame in order to see if even in the rectified image the table is perfectly rectangular. Using the known size of the table, we fitted a homography that takes the table plan to the image plane. It was used then to rectify the image, in the rectified image the table is almost perfect.



9.2 Ball Tracking

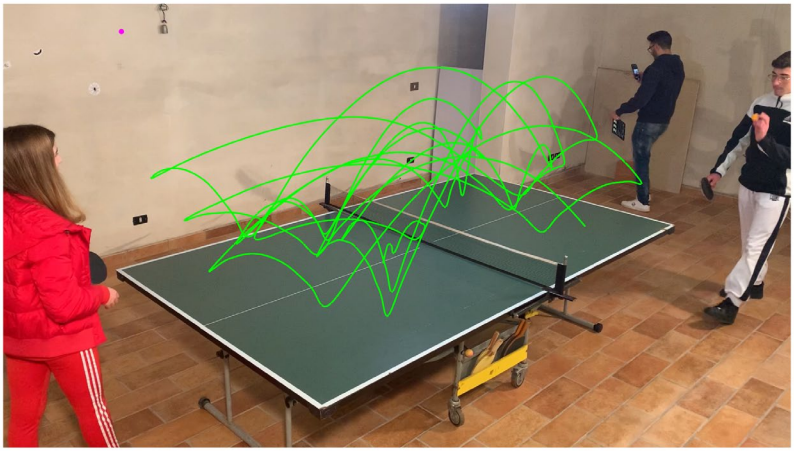
We use two identical Iphones that records at 30 fps; this detection rate is quite low compared to the others in papers, so this makes the situation challenging for our algorithm, but although this, our algorithm performs the ball tracking with really accurate performances.

Sometimes in the first frames it finds difficulties to find the ball, but once that it finds it the algorithm proceeds in an optimal way

One way to improve this algorithm is to improve the optimization of memory management and computation time (Matlab is not optimal in terms of computation time); firstly improving it (for example) in C++, then an interesting way could be to introduce machine learning and to train a neural network in order to make it capable of capture the ball position.

9.3 2D Trajectory estimation

The trajectories can be fully synchronized with our 3 methods, and if one of these doesn't work (due to occlusion or due to an hypothetic non well working ball Tracking algorithm) we can try with the other two methods.

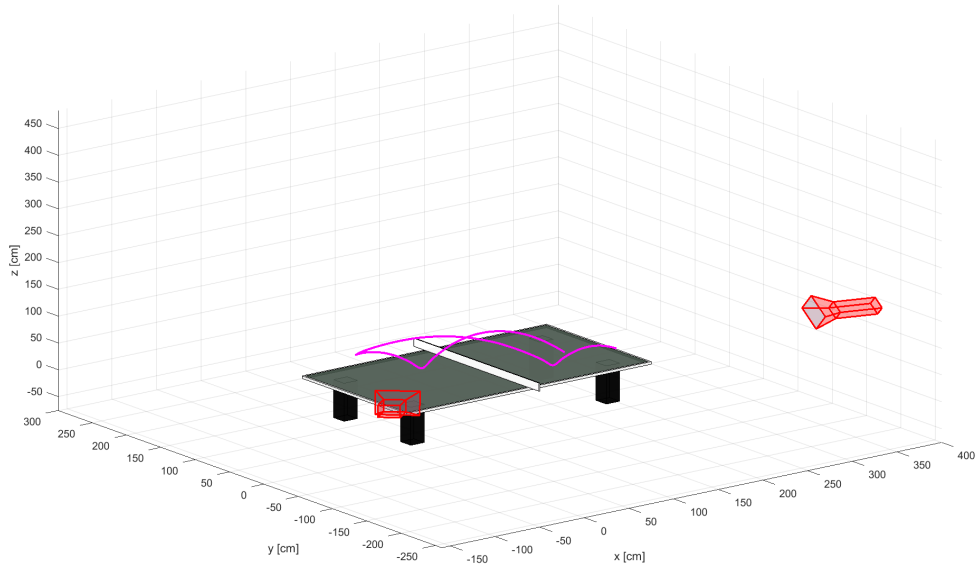


Increasing the fps would also increase the accuracy of these algorithm, and Spline interpolation would be more precise with a higher number of points.

With more than 2 cameras this procedure will change a bit but it would be more robust, being the cameras more than one even if one of them is occluded by the Player.

9.4 3D RECONSTRUCTION

We are proud of our results, they are robust and precise. Our system has in the main code many explained setting parameters that the user can change at varying of the situation, the needs and the environment.



Increasing the image quality would also increase our accuracy, we worked with images 1920x1080 pixels. Increasing the detection rate would significantly increase our algorithm performances, we worked at only 30 fps.

9.5 OTHER FUTURE DEVELOPMENTS

An important rule is about the service: a simple but important step could be to write an algorithm able to understand when the ball trajectory is part of the service, and extrapolate the height of the ball in order to be capable to understand if the service is regular or not.

Including more than 2 cameras will improve significantly the robustness of the whole system. As discussed before, a multiple camera system with 3 or more cameras would be able to reconstruct the 3D trajectory of the ball in each situation.

This project was long and demanding but it was really fun, the Computer Vision field is very interesting and we are happy to have had the opportunity to do such a project, we are really satisfied and we would like to continue to deepen this field even after graduation.

Matteo Bartoli 945886

Giovanni Buzzao 928918

10 REFERENCES

[1] B. CAPRILE AND V. TORRE. USING VANISHING POINTS FOR CAMERA CALIBRA- TION. INTERNATIONAL JOURNAL OF COMPUTER VISION, 4(2):127–139, MAR 1990.

[2] R. I. HARTLEY AND A. ZISSERMAN. MULTIPLE VIEW GEOMETRY IN COMPUTER VISION. CAMBRIDGE UNIVERSITY PRESS, ISBN: 0521540518, SECOND EDI- TION, 2004.

[3] J. KANNALA AND S. S. BRANDT. A GENERIC CAMERA MODEL AND CALIBRATION METHOD FOR CONVENTIONAL, WIDE-ANGLE, AND FISH-EYE LENSES. IEEE TRANS- ACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 28(8):1335– 1340, AUG 2006.

[4] D. LIEBOWITZ AND A. ZISSERMAN. METRIC RECTIFICATION FOR PERSPEC- TIVE IMAGES OF PLANES. IN PROCEEDINGS. 1998 IEEE COMPUTER SOCI- ETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CAT. NO.98CB36231), PAGES 482–488, JUNE 1998.

[5] Q.-T. LUONG AND O.D. FAUGERAS. SELF-CALIBRATION OF A MOVING CAMERA FROM POINT CORRESPONDENCES AND FUNDAMENTAL MATRICES. INTERNATIONAL JOURNAL OF COMPUTER VISION, 22(3):261–289, MAR 1997.

[6] SHO TAMAKI AND HIDEO SAITO. RECONSTRUCTION OF 3D TRAJECTORIES FOR PERFORMANCE ANALYSIS IN TABLE TENNIS. PAGES 1019–1026, 2013.

[7] PATRICK WONG AND LAURENCE DOOLEY. TRACKING TABLE TENNIS BALLS IN REAL MATCH SCENES FOR UMPIRING APPLICATIONS. BRITISH JOURNAL OF MATHEMATICS & COMPUTER SCIENCE, 1:228–241, OCTOBER 2011.

[8] ZHENGYOU ZHANG. A FLEXIBLE NEW TECHNIQUE FOR CAMERA CALIBRATION. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, 22, DECEMBER 2000.

[9] L. ZHAO, C. WU, AND S. LIU. A SELF-CALIBRATION METHOD BASED ON TWO PAIRS OF ORTHOGONAL PARALLEL LINES. IN 2012 FOURTH INTERNATIONAL CON- FERENCE ON INTELLIGENT NETWORKING AND COLLABORATIVE SYSTEMS, PAGES 373–376, SEPT 2012.