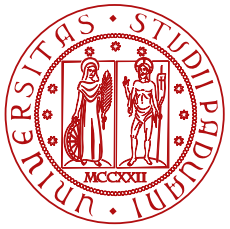


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Final Project in Computer Vision

Billiard Table Vision System

Candidates

Manuel Lovo 2122856

Alberto Bressan 2125514

Martina Boscolo Bacheto 2109755

University of Padova

Academic Year
2023/2024

Contents

1

Introduction to the project

The goal of this project is to create a computer vision system capable of analyzing **video footage of various events in the "Eight Ball" billiard game**. Eight Ball is a call shot game played with a white cue ball and fifteen object balls, numbered 1 to 15. One player must pocket balls numbered 1 to 7 (solid colors), while the other player aims to pocket balls numbered 9 to 15 (stripes). The black 8-ball has a special role and must be pocketed last. The player who pockets all their assigned balls and then the 8-ball wins the game.

The computer vision system should provide high-level information about the match status (e.g., ball positions) for each video frame. This information should be displayed as a 2D top-view minimap.

In detail, for each frame of the input video, the system should:

1. Recognize and locate all the balls within the playing field, distinguishing them by category:
   - White "cue ball"
   - Black "8-ball"
   - Solid colored balls
   - Striped balls
2. Detect all main lines (boundaries) of the playing field.
3. Segment the area inside the detected playing field boundaries into the following categories:
   - White "cue ball"
   - Black "8-ball"

- Solid colored balls
- Striped balls
- Playing field

4. Display the current game state in a 2D top-view visualization map, updating it each frame with current ball positions and the trajectory of any moving balls.

It's then required the computation of some metrics, such that **mAP for ball localization** and **mIoU for segmentation masks**, using ground truths provided in the dataset that we will discuss in the next section.
The system to be developed should be robust under various conditions. Specifically, it must accurately recognize all the main elements (playing field and balls) despite different camera perspectives and table colors. Additionally, it should neglect any individuals (e.g., players, referees, spectators) around the billiard table.

1.1 Dataset

The dataset provided contains a folder for each **video footage**, for a total of 10 videos.
For each folder, there is:

- **Main video** in .mp4 format
- **First frame and last frame** in .png format
- **Ground truths bounding boxes**
- **Ground truths segmentation masks**

The ground truths have been important not only for computing the metrics, but also for **parallelizing the work**.
In fact, by using them it's possible to subdivide tasks and then merge all together, with the tradeoff that error propagates through different tasks.

2

Work Pipeline

In the development of the project, four main tasks have been identified:

- **Field detection**
- **Ball segmentation**
- **Ball classification**
- **Development of a 2D tracking system**

The correct pipeline to follow is:

1. Identify the playing field. This accomplishes the first task and aids the subsequent task by reducing the Region of Interest (ROI) to only the field, which contains the balls.
2. Detect the balls, obtaining circles that enclose the balls. This step is then used for the classification of the balls by focusing solely on the content within the circles.
3. Classify the balls by considering only the regions provided by the previous step.
4. Develop the tracker, which uses the bounding boxes derived from the second step and the ball classifications obtained in the third step.

These tasks have to be achieved for different input videos, taken from different perspectives and different types of billiard table.
From the beginning of the project, we understood the **importance of parallelizing the work**.
Due to the dataset provided, which contains the ground truth for bounding boxes, one component of the group was able to work on the first two tasks, one

on the third and one on the last task.
At the end, after merging all the parts, it was necessarily to compute **different types of metrics** for quantifying the goodness of the work.

2.1 Field Segmentation

The first task was to identify the main boundaries of the playing field.
More precisely, to define the playing field, we can use the lines belonging to the upper parts of the top rails of the table. These are the parts where the cloth covering the playing field surface is attached to the wooden structure of the table.
As mentioned above, the dataset provided contains different videos, and the system we have developed **must be robust to all perspectives and conditions**.
However, an important simplification can be leveraged: in a single video, the perspective does not change. Therefore, once **we have identified the field for the first frame, this can be used for the subsequent frames**.
In Figure 2.1 the input given to our system, or rather the different first frames of the different videos.

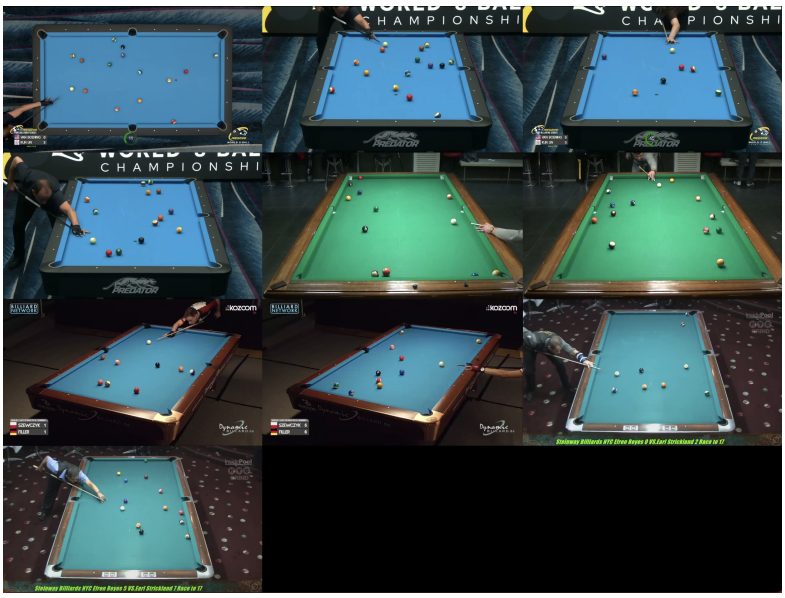


Figure 2.1: Input to the system

It is important to consider that this task is difficult due to the high noise around the table and the different colors of the playing field. Therefore, finding the

correct methods is not quite easy. Our idea is based on a **masking technique** in the **HSV space**.

HSV is a color space based on three values: Hue, Saturation, and Value. In tasks like segmentation or object detection, HSV can yield better results than the classic RGB.

More specifically, we have found a **range of colors in HSV**, using some sliding techniques, **which correctly masks all the different colors of the table**. This method works very well, since it identifies the boundary of the field and eliminates all the surrounding noise, as we can see from Figure 2.2

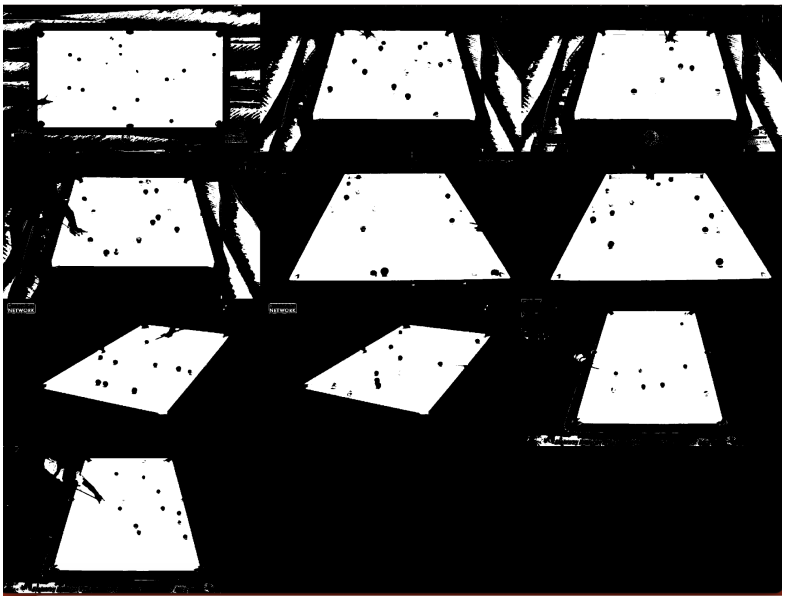


Figure 2.2: HSV masks obtained from Figure 2.1

N.B.: If the dataset contains a video with a red playing field, our method might not work correctly, and it would be difficult to find a range of colors that works well for all.

We have chosen this method because the requirement is that the system must provide the correct output for videos in the dataset.

An **alternative implementation could involve a mask in RGB or other color spaces based on the color of the table**, similar to the method developed for ball segmentation.

Then, before applying the function of OpenCV for obtaining lines with HoughTransform, it was needed to do some **postprocessing**; in this order, the following operations are done:

1. **erosion**

2. **dilation**
3. **smoothing** (high kernel)

After all these steps we have obtained binary images where contours are well defined, and, these, will be the input for a **Canny edge detector**.
After the application of a Canny Edge detector (function provided in OpenCV), the resulting images are the following:

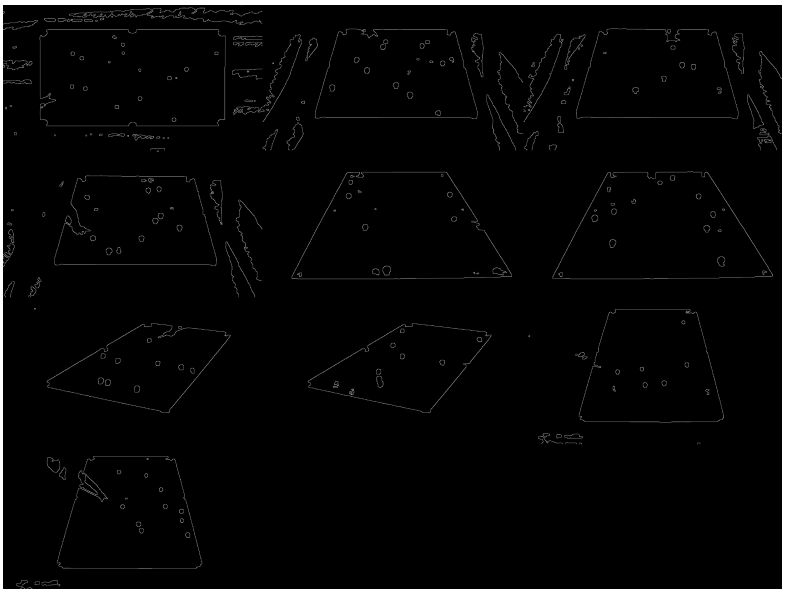


Figure 2.3: Edges obtained after the application of Canny detector

As we can see, the edges found are in a **desirable shape (single point, thin)**; however, as we have learned from theory, edges lack the concept of **lines**, which are defined by a sequence of adjacent points forming a geometric equation.

The Hough Transform is helpful in this context as it identifies lines in images. We have used the HoughLines function, which identifies lines based on the subdivision of the ρ, θ space and the minimum number of points required to consider them a line.

In our function for finding lines, **we have considered only the first four most prominent lines** since we need to define the boundary of a playing field that is ideally a rectangle.

Thus, the aim of the previous step is to create masks and follow edges so that the most highlighted edges form the playing field boundary.
The method implemented then finds the **intersection points** between lines and, by using another function, orders the points in **clockwise** order for using them in further processes and for drawing boundary of the field (identified by lines

connecting two consecutive points).
The final result can be seen in Figure 2.4

Figure 2.4: Lines defined by the whole process

Then, as last step, we have extracted the region of interest contained inside each playing field, that will be the input of the second task, or rather balls detection/segmentation.

2.1.1 Considerations and possible Enhancements

It's mandatory to do some considerations on the method used and on the final result.
The method, for our dataset, works well, finding precise field boundaries for all images and it's computationally not expensive.
The main criticism found was in the setting of different parameters of different functions; for example, a very small change in the quantization of the parameter space could lead to different prominent lines found. However, after a consistent search using sliding implemented in C++, the parameters used find correctly all fields.
As said before, the method could be improved by combining it with a mask based on the median color extracted from a small ROI inside the table, in order to handle all different types of table colors.

2.2 Balls detection

The second principal task required by the system is to detect balls inside the playing field. More precisely, the request is the computation of a segmentation mask of the playing field for each frame. However, to obtain this, we first need to define the balls.

Ideally, our system needs to be robust to **occlusion**, e.g., when a ball is covered by another ball; to **illumination changes**, e.g., recognizing balls in regions with low intensity levels; and to **different scales**, e.g., due to perspective, some balls in an image appear smaller than others.
The input of this part of the process is the **ROI extracted from the previous step, so the area inside the playing field**, as we can see from Figure 2.6.
By removing some noise, less false positive are found.

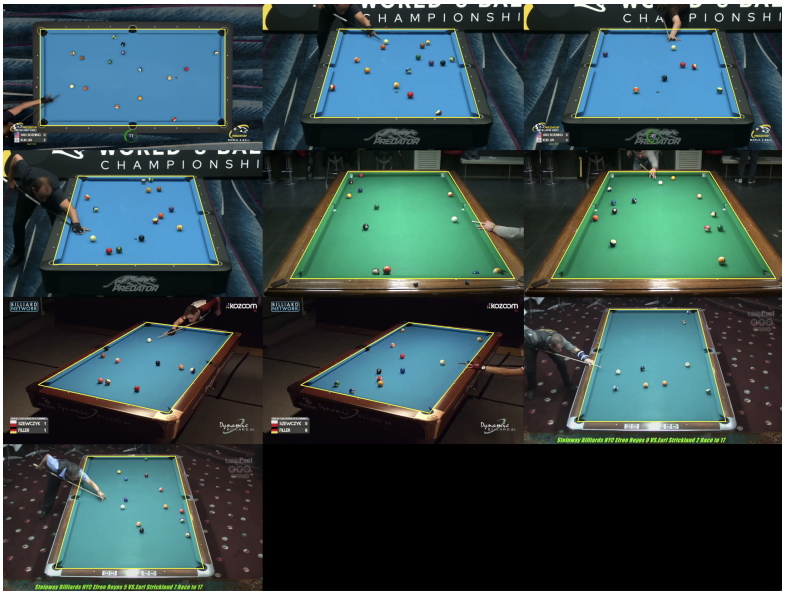


Figure 2.5: ROI extracted inside the playing field

We believe it is difficult to detect all the balls in each image in a single pass. Therefore, we have implemented an overall method that applies the same idea multiple times with different parameters to find all the balls, accepting some false positives as a trade-off.

Once all the balls are detected, the main goal is to remove all the **false positives** and to **maintain a single circle for each ball**, as each ball can be detected by different processes.

The method used for creating a mask where balls differ from the background

is based on **masking in RGB** using the color of the table. To compute the color of the table, we extracted a square ROI centered in the image (where we analyzed that there are no balls in each image) and took the **median RGB color of this ROI**. We chose the median color since the ROI could contain a piece of a ball, which would influence the value.

Then, after extracting the color of the table, the method works as follows: if the color of a pixel differs from the color of the table within a threshold, it is considered a "table pixel."

Additionally, to try to remove shadows or reflections in the regions near the balls, the method considers points slightly different from black and white as "not ball." However, it is more difficult to choose the correct threshold for this functionality, so it has been fixed to a precise value for each use of the method.

Thus, the unique parameter to tune for this method is **the threshold** that decides if a pixel is similar to the table or not.

By using different values of this threshold, we obtain different masks, allowing the detection of all balls.
A lower threshold works correctly for masking almost all balls but, due to illumination changes, some images had some problems, as we can see from Figure 2.6.

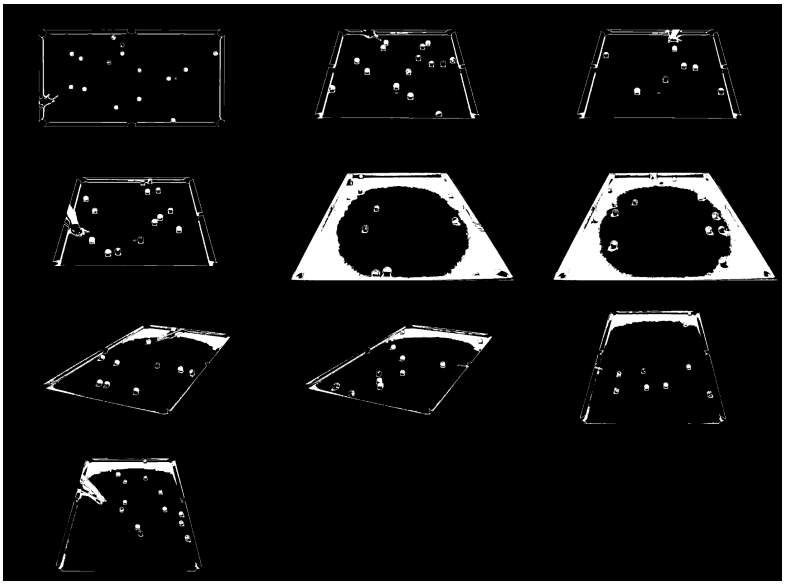


Figure 2.6: Masks in RGB with threshold equal to 49

It is easy to notice that in some images there are regions where different illuminations cause portions of the table to have a slightly different color, which is not

recognized as the table.

Therefore, as mentioned before, we have used different thresholds, precisely five thresholds, to identify all balls.

The most difficult balls to detect were positioned close to the upper boundary, where the region is very dark. To find them, we used a higher threshold which, as a downside, resulted in some false positives by decomposing parts of the image, as shown in Figure 2.7.

Figure 2.7: Masks in RGB with threshold equal to 129

Moreover, additional to these 5 different masks, we have used a method that masks precisely black balls, since in some cases were difficult to find them in very very dark regions.
Then, these masks were postprocessed using principally a combination of **dilation and smoothing**, but, in some cases, also **median blur** and **erosion** were used.
Despite the fact that the **HoughCircles** function performs a Canny edge detection on the image to identify circles, we have preferred to use our own Canny function, where both the lower and upper thresholds can be tuned.

We then give our masks as input to the OpenCV function HoughCircles, using different parameters for each different process mask.

The most important parameters that can be tuned in HoughCircles are the maximum and minimum radius of the circles and the threshold for center detection. By adjusting these parameters, we have obtained the results shown in

Figure 2.8, where for the first frame of all videos, all balls are found, and for the last frame of all videos, only one ball is missing (very occluded as we will see later).

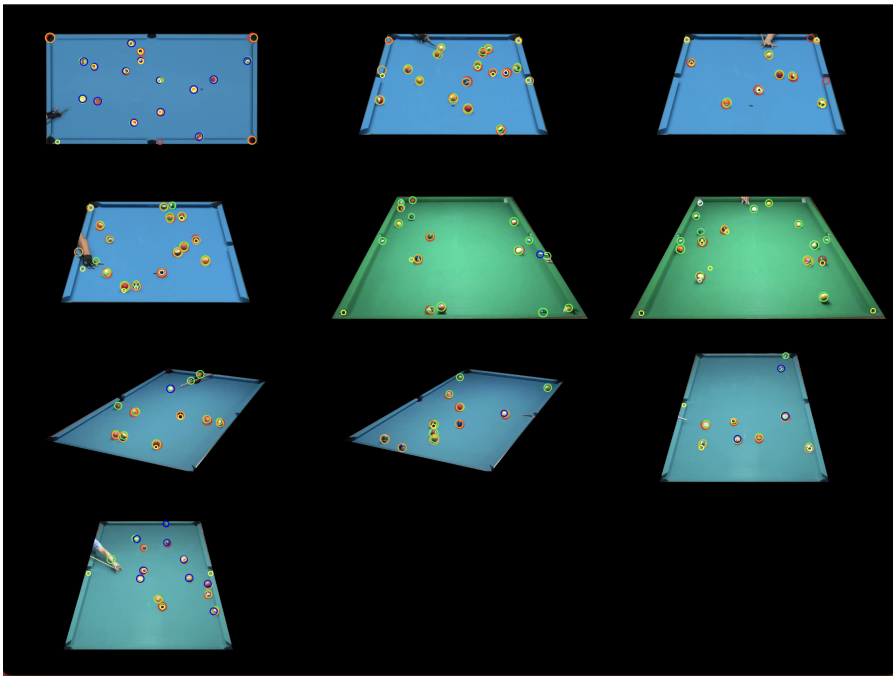


Figure 2.8: All circles detected

As we can see, all the balls are **correctly found**; however, there are some false positives, mostly near the holes, around the table, and near the hands and arms of the player.

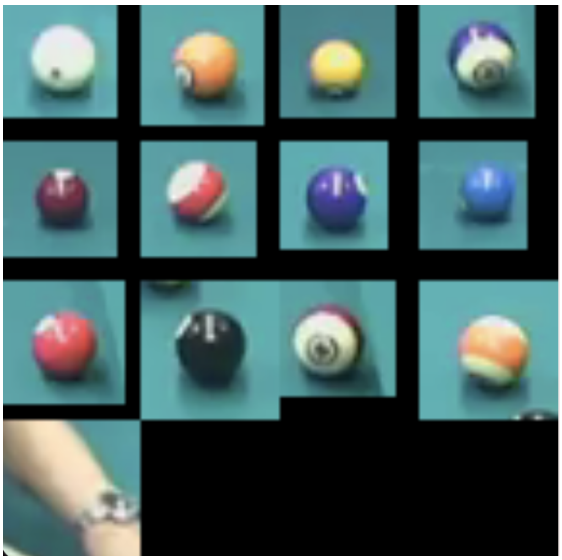
At this point, the aim is to **remove these false positives and maintain only one circle for each ball**.

We have implemented different methods for refining these circles:

- **Remove circles near holes**: As part of the task of detecting the playing field, we collected the vertices of the table. This information is used for removing some false positives. T
  he idea is simple: if the circle center is **near a hole within a threshold**, the circle is removed.
- **Filter circles by color**: As seen in the image, some circles are found at the boundary or inside the playing field. If the color inside the circle is too similar (within a threshold) to the color of the table, the circle is removed.
- **Remove circles near lines**: If the previous method does not remove the circle, this method ensures that all circles found too close to the boundary field are removed.
  However, some circles were not removed due to perspective issues (e.g.,

centers that are closer in the real world but appear further apart in the image), requiring a higher threshold and thus also removing true positives.

- **Remove circles using the ROI of the circle**: The most critical part was encountered when removing false positives near hands, arms, etc.
  To address this, we developed a method that takes a bounding box around the circle, performs some operations, and then re-applies HoughCircles on the ROI. In this way, we removed all false positives (potentially losing some true positives that are found by other masks).
  In Figure how looks extracted ROIs.



At this point, we had only correct circles, but more than one per ball.

So we implemented two main methods that work as follows:

- The first method looks for the **sum of differences between the pixels color inside the circle and the color of the table**. If two circles are too close (within a threshold), they are probably referring to the same ball, so we maintain only the circle with the greatest sum of differences.
  This is because if a circle is not well-centered and captures a portion of the table, the sum is lower compared to a circle well-defined around the ball.

- After the first filtering, the second method looks for circles that are too close (using the same reasoning as above) and maintains only the circle with the larger radius.
  This is because, due to different processes, we have noticed that sometimes a circle is too small compared to the correct ball, so we keep the larger one.

So, at the end we have obtained all circles encapsulating balls, from which it's easy to obtain also **bounding boxes**, required by the system.
In Figure 2.9 it's shown the result for the first frame of the videos, where all the balls are found.

Figure 2.9: Circles refined for first frames

However, in the last frame, the program misses one single ball, that is the one represented in Figure 2.10. As we can see, this ball was very occluded by the ball standing in the front of it and, due to this fact, a single bigger ball is found.

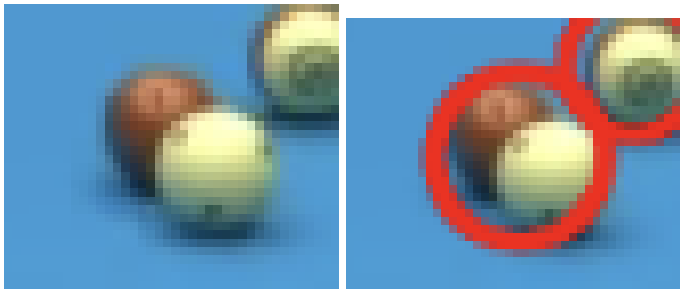


Figure 2.10: Single false negative in the detection

2.2.1 Considerations and possible Enhancements

It's important to do some considerations on the work done.
The method implemented works quite well, succeeding on identifying all balls inn first frames and missing only one in last frames.
However, this is a qualitative evaluation and it would be better to have a quantitative evaluation.

Using the IoU metric, equal to the one that we will use after in the computation of metrics, we have obtained how much **false positives, false negatives, true positives** we have in our system, fixing the threshold of IoU $\alpha = 0.5$.
For first frames, there were a total of **118 balls** and our system correctly identifies **118 balls** (no false negatives), with **114 true positives** and **4 false positives**.
So, using **precision and recall** on the total we have:

- $Precision \approx 97\%$
- $Recall \approx 97$

Instead, for last frames, there were a total **112 balls** and our system correctly identify **111 balls**, meaning **1 false negative**, with **106 true positives** and **5 false positives**.

- $Precision \approx 95\%$
- $Recall \approx 95\%$

In our opinion, the detection of all balls in the images is influenced by the application of dilation, median filtering, and other techniques. While these methods are effective for identifying the balls, they may also inadvertently enlarge or reduce their sizes.

Moreover, it is important to consider that errors can propagate through the system. For instance, if a ball is falsely detected as a positive, this creates an inconsistency in the Intersection over Union (IoU). During classification, this inconsistency may lead to an overlap area composed only of colored pixels, even if the ball is actually striped.
However, our system can be obviously improved and in the following some possible ideas for enhance it:

- Compute the color of the table using the mean of different ROI in the image
- Use different space of color like HSV, CieLAB etc. for improving the masking
- Apply histogram equalization or gamma transform for trying to improve the images before the masking

2.3 Ball classification

The task of ball classification consists in assigning to each ball detected in the previous tasks a label based on its characteristics, more precisely 4 classes should be used:

- CLASS 1: White ball
- CLASS 2: Black ball
- CLASS 3: Solid balls
- CLASS 4: Striped balls.

Initially we subdivided the problem in two subproblems, firstly we focused on classifying the striped and solid balls, while the detection of the white and black ball was done separately.

2.3.1 Solid/Striped classification

The main idea behind the classification of solid and striped balls was to focus on the color components of the bounding boxes and on the presence of edges.

For any computation we only considered the circle inscribed in each bounding box, building a circular mask centered in the bounding box and with diameter equal to the minimum between the width and length of the box, which approximates well the area occupied by the ball, considering how the bounding boxes were constructed.

We considered three parameters:

- **White ratio**: ratio between the the number of white pixels in the region of interest and the total number of pixels. The computation of the white pixels was done on the HSV image setting thresolds for the saturation and value parameters. This parameters were chosen by inspection, using trackers depending on the level of white we wanted to consider.
- **Dark ratio**: ratio between the number of dark pixels in the region of interest and the total number of pixels. The computation of the dark pixels was done on the HSV image setting thresolds for the value parameter.
- **Canny edges**: number of edges on a Canny image of the region of interest, it was computed by counting the non-black pixels in the area, after applying the Canny algorithm for edge detection.

To do the actual classification we used the above mentioned parameters, iteratively setting different parameters and using adeguate threshold values. The main objective was to correctly classify all the solid balls first.

We focused more on getting an accurate classification of the solid balls first, because the striped ones are more complex and can have very different features depending on their position. By having an accurate and solid classification of the solid balls, it is sufficient to consider all other balls as striped without having to deal with all possible positions and color combinations of the striped balls.

We divided the classification in rounds, at each round we used the different parameters and chose the best thresolds, that would correctly classify the highest number of balls, while maintaining a large enough margin. The main ideas behind the choice of parameters and thresholds were that solid balls would have a close to zero **white ratio**, a possibly high **dark ratio** and a small number of **edges**, on the other hand the striped balls would have ratios in the middle range for both white and dark pixels, and a larger number of edges given by the stripes. The actual choices were made by inspection using trackers on the different frames given.

In some rounds we also applied pre-processing method to the image such as **Gaussian smoothing** and **median filtering**.

These methods were used to remove noise, and to try to limit the inferences of shadows and reflections on the balls.

In the end to obtain this first classification we used ten rounds, setting 12 total thresholds, six of which on white pixels ratio, two on dark pixels ratio and four on Canny edges count.

2.3.2 White/Black ball classification

Since we only consider "valid" games, both the black and white ball should be in the field for the classification. For each ball we computed the white ratio, using a high thresholds, hence considering only the brightest pixels, and the dark ratio, using a low threshold to select the black pixels. We then evaluated the highest value for both ratios and labeled the white ball with the first and the black ball with the second.

Figure 2.11: Final classification of balls for all first frames of the provided videos. Red: Striped balls, Blue: Solid balls, Black: Black ball, White: White ball

2.3.3 Considerations

The main challenges we encountered for the classification are the following:

Shadows Since the lighting in the image is always coming from the top it generates shadows on the lower half of the ball, this does not generate problems if the point of view is high enough, but can arise difficulties when it is lower as shown in Figure 2.12.

In striped balls the shadows determine the white stripes to be detected as dark pixels as in Figure 2.13, hence the white pixels ratio is not an adeguate parameter for the classification, but by using the Canny edges we were able to correctly classify all these balls.

In solid balls the shadows don't arise problems for dark color balls (e.g green ball, violet ball) but can for lighter balls, where the edge determined by the shadow can be detected as a stripe, as the ball in Figure 2.14. If the color of the

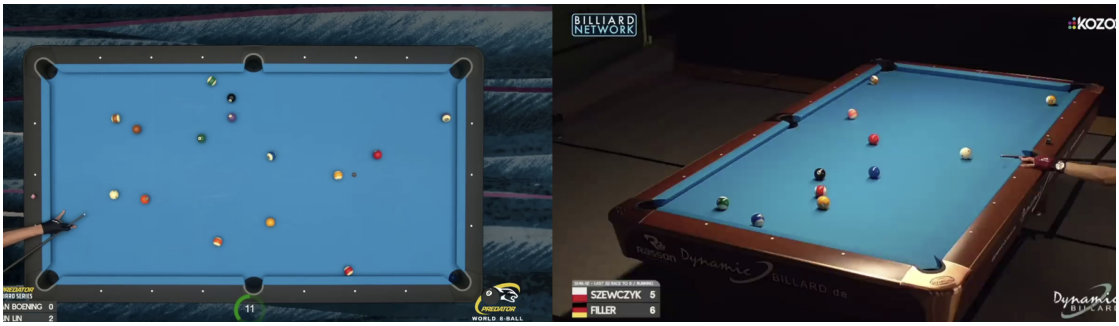


Figure 2.12: Top view vs angled viewd

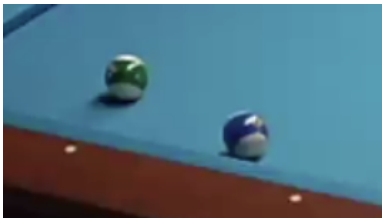


Figure 2.13: Striped balls with low shadows

ball in the non-shadowed area is light enough and the thresholds for white ratio computation are not set accordingly the ratio could be in the range of striped balls.

We were able to classify these balls by fine tuning the threhsolds for white ratio.

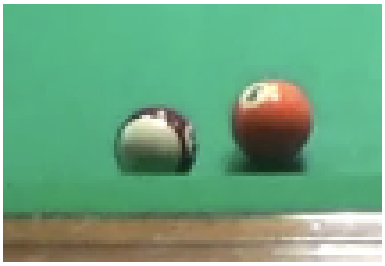


Figure 2.14: Solid balls with low shadows

Light reflexes The lighting coming from the top is also responsible for the presence of light reflexes on the balls that get mistaken for white areas.

This generates problems especially in solid balls (e.g. Figure 2.15) where the white pixel ratio increases and can lead to misclassification.

This problem can be solved by applying erosion or median filters to the image to mitigate the presence of the areas, by using the dark ratio, in case of dark color

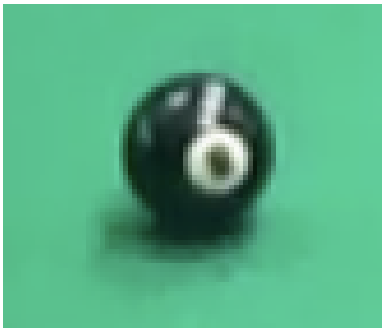


Figure 2.15: Black ball with white reflex of light on top

balls, and by fine tuning the thresholds for white ratio for the light color balls as above.

Illumination variance The different illumination conditions of the scenes made the building a robust classification difficult. This was especially seen when applying effective thresholds found in darker scenes to brighter scenes, as the two in 2.16. We observed that the lighing difference caused the balls to have ratios in completely different ranges and made the thresholds ineffective.

Initially we found a possible solution by applying histogram equalizations to the images and we were able to obtain uniform ratios for the different images. Later on we fine tuned the parameters and thresholds such that the equalization was not needed anymore.

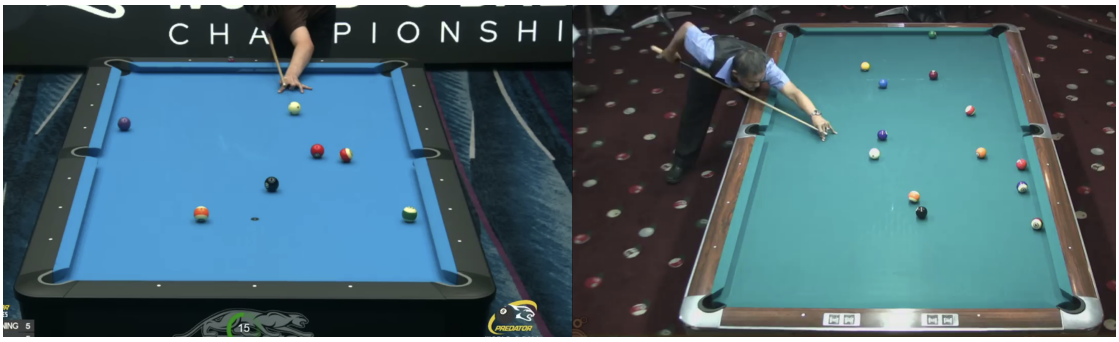


Figure 2.16: Normal lighting conditions vs bright lighting

Colors We observed that in the last rounds of classification the balls that remained unclassified were the ones ranging in the dark yellow-light orange range. This occured because in both grayscale and hsv images this range is close to the white one, as seen in Figure 2.17 It becomes difficult to distinguish striped

balls from solid ones, we therefore relied on the edges detection to correctly classify these cases.

Figure 2.17: Orange balls in BGR and grayscale

2.3.4 Results

As mentioned before, to be able to parallelize the different tasks, the classification task was implemented using the ground truth bounding boxes provided to us. We were able to correctly classify all the balls as can be seen in Figure 2.18, when applying the classification method to the balls segmented by our program we lost some accuracy due to the non-ideality of the bounding boxes.

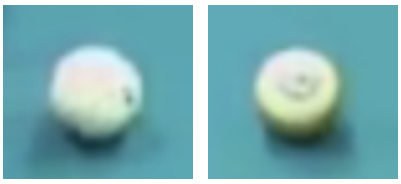
Figure 2.18: Perfectly classified balls using ideal bounding boxes

We corrected the parameters of the methods to be effective for our program and were able to obtain the following accuracies in the first frames provided:

- 95% correctly classified striped/solid balls
- 100% correctly classified black balls
- 95% correctly classified white balls.

We were not able to correctly classify the white ball in the last frame of clip 1 from game 4, shown in Figure 2.19. We can see that the placement of the ball in fig **??** b makes it almost indistinguishable from the white ball Figure **??** a for the methods we described above.

Figure 2.19: Last frame Clip 1 Game 4



(a) (b)

Figure 2.20: Actual white ball (a) and misclassified one (b)

2.4 2D TOP-VIEW VISUALIZATION

This task consist in the analysis of videos from some billiard matches. In particular, the task aims to produce, given a video, the representation of the current state of the game in a 2D top-view visualization map, to be updated at each new frame with the current ball positions, showing also the trajectory of each ball that is moving. The end result is a new video, where the 2D minimap is superimposed on the bottom-left corner and gets updated real times while the balls are moving.

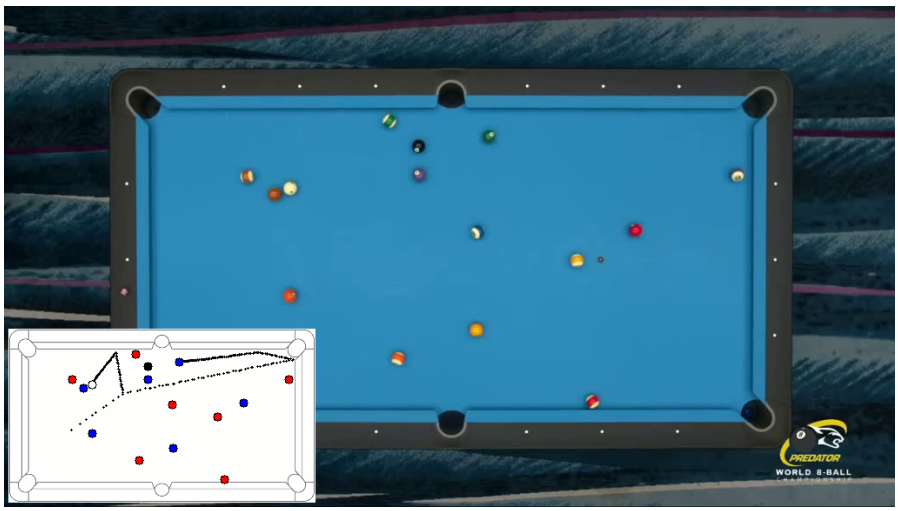


Figure 2.21: Expected result, frame extracted from a video

To achieve this, OpenCV Tracker Class was exploited. Given the requirement that the code must compile on the Virtual Lab, particular attention was posed to just using tools available in OpenCV 4.8, which is the version of the VL. Several tracker were tested. Some of them required deep learning external libraries, so they were discarded. Others, like the cv::TrackerKCF and the cv::TrackerMIL were not able to follow the balls at all. In figure 2.22 it's possible to see a test with the TrackerKCF: the balls are too fast for the tracker system, which loses them, leaving the bounding boxes in the initial position, without ability to recover.

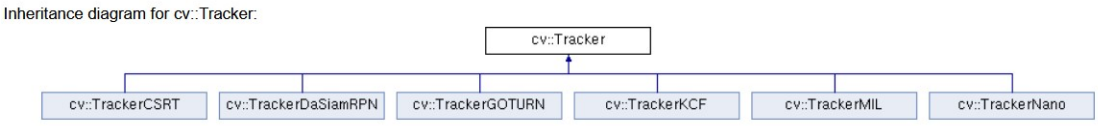


Figure 2.22: cv::Tracker Class for OpenCV 4.8

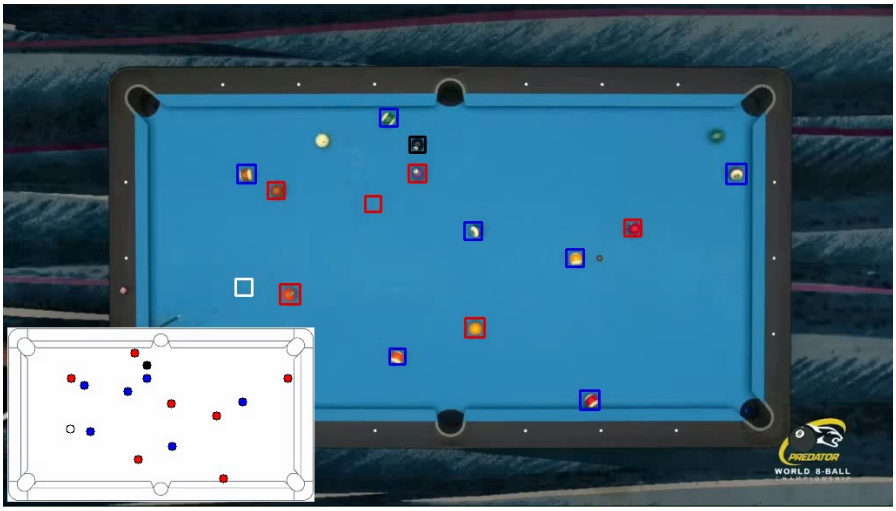


Figure 2.23: cv::TrackerKCF for OpenCV 4.8

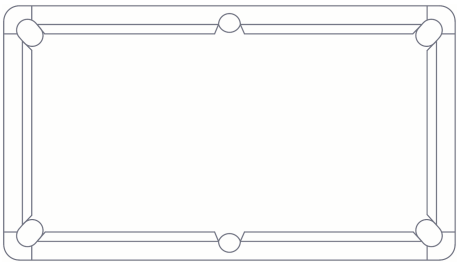
Eventually, cv::TrackerCSRT (Channel and Spatial Reliability Tracker) was chosen, given the fact that was the only one able to follow the fast moving balls, although requiring longer time for computation.

2.4.1 Tracking system

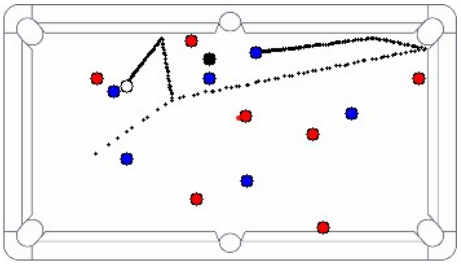
The tracking system developed works initializing for each bounding box computed in the previous tasks, for the initial frame, a tracker. The bounding box represent the region of interest containing the object that the tracker attempts to follow. At each frame each tracker gets updated. The task executed with the cv::TrackerCSRT is computationally heavy, and the production of the output video from an input video of approximately 6 seconds is of about 90 seconds. Considering this is not a real time tracking system, but more a "post game" analysis system, we preferred to give more importance to the accuracy of the analysis than the efficiency of the tracker. Other approaches different from the use of OpenCV Tracker were tested, like the use of difference among consecutive frames to detect moving objects. While this method proved to be faster, it was less precise and more susceptible to noise. The technique struggled particularly with distinguishing between relevant motion (such as the ball) and irrelevant movements, like players moving on the field.

2.4.2 2D MINIMAP

For the representation of the game in 2D topview, we decided to use a similar picture to the one in the project delivery. Using a plain white rectangle would have been easier, not requiring to extract the exact points of the intersections of the edges where the pockets are positioned, but with few line of code, we were able to extract those points and use them for the destination of the matrix projection. The 2D billiard table is positioned horizontally in all the videos, leaving 5 pixels from the bottom and from the left side of the window. This is all mainly for aesthetics reasons. It is important to notice that not all the tables have such big holes in the corners, so sometime, for some videos it may seams like the ball is in the hole, but it's actually really near the corner.



2D billiard table empty



2D billiard table with balls

The main task for the realization of this step was the computation of the projection matrix for applying the prospective transformation, that is, mapping four points from the 3D vision of the original table to the 2D map. This is achieved with OpenCV getPerspectiveTransform function, that taking in input the set of the four original table vertex (orientated according to a criteria explained in the next section) and the four destination points computed previously, calculates a 3x3 transformation matrix that maps the source points to the destination points. This transformation matrix can then be used to map each required point from the original image, like a ball, to the 2D map. In particular, for each frame of the video, the center of the bounding box/ROI that the tracker is tracking , which correspond to the center of the ball, get projected in the 2D table, following the same color code used in the previous tasks:

- White: White ball
- Black: Black ball
- Blue: Solid balls

- Red: Striped balls.

A black contour is added to all the balls with the main purpose of showing the white ball.

For drawing the path, the center of the bounding box of each tracked object get collected to a vector frame per frame, and, using the previously calculated matrix, transformed for the 2D representation. Then, a point is drawn for each point saved, and the accumulation of the points drawn represent the path of the tracked objects. At the end, the complete path of all the balls is displayed. It's important to notice that when the ball is fast, points are more sparse, while when the ball is slow, points get near, giving also the information of the speed of the balls! We chose not to eliminate balls in the holes, thinking that would be helpful to see which kind of ball has been pocketed, although for classification purposes this may lead to worse metric computation (position of the bounding boxes in the last frame of the video are used to compute the metrics in the metrics computation task). This could be avoided implementing a function for eliminating the balls near to the vertices of the table under a certain threshold.

2.4.3 Table orientation

In the videos of the dataset, tables are orientated along different directions: sometime horizontally, sometime vertically. It's important to establish a rigorous way for understanding the orientation of the table so that to project correctly the balls on the 2D map.

After some attempts, no general mathematical rule was found for determine which are the longer sides of the table working only with the length of the sides (with different viewpoints) or its angles. So the approach used was to detect the "lateral" holes to determine which are the longer sides, and use this information to rearrange the vertex such that they always start forming a short side if reading them clockwise. Using the assumption that the orientation of the table does not changes during a video, the computation of the holes is made on the last frame of the video, so to avoid hands of the player or the billiard cue.

The process to find holes can be summed up in this points:

- Drawing a shortened line between each pair of vertices on a mask image.
- Dilating this line to create a region of interest around the edge.
- Applying thresholding to the masked region of the original image.

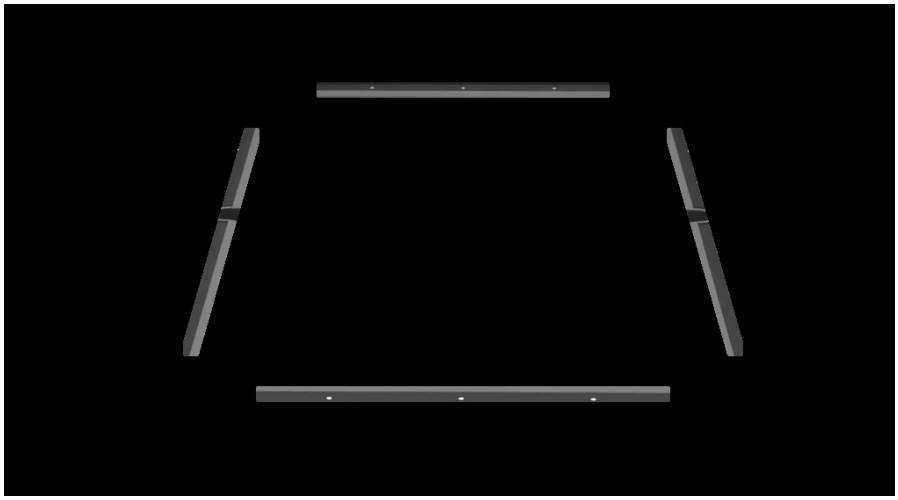


Figure 2.24: Table contours

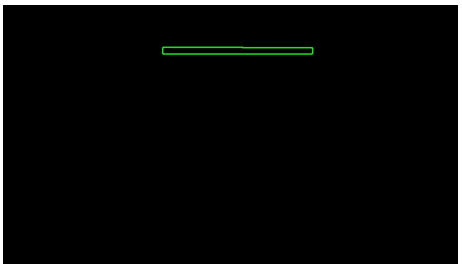
- Counting the contours in this processed region, where multiple contours indicate the presence of a hole.

It's important to notice that not always for each image are found exactly 2 sides with two connected components and the other two with just one, but the process it's still quite robust because it needs only some combinations to work (i.e. to find two sides with just one connected component, while the others may have 3, for example if there is an hand). Moreover, it has been made the assumption that if for a table no holes are found, the table is suppose to be oriented vertically, given the fact that this way it's easier that the lateral holes are hidden (while for an horizontal orientation at least one hole should be clearly visible).

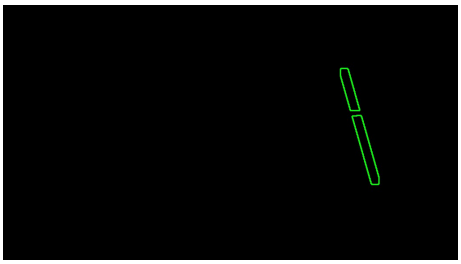
2.4.4 Results and Considerations

The tracking system task was a more "mechanical" task with respect to the segmentation and classifications one, mainly requiring the application of some OpenCV functions. It is however important to notice that some time had to be dedicate in the understanding of the correct management of the videos and the choice and application of those functions.

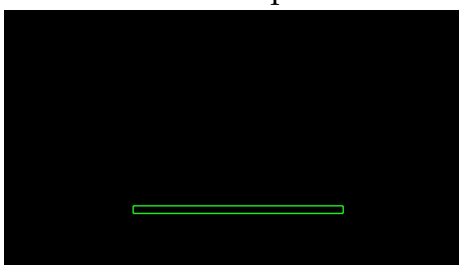
In conclusion, the the tracking system was able to correctly track and map all the moving balls for the entire duration of the videos, except in one video (Game 2 Clip 1) where, after the clash between two balls, the two tracker kept following the same ball, losing the other one. More details on the result in the section "Results".



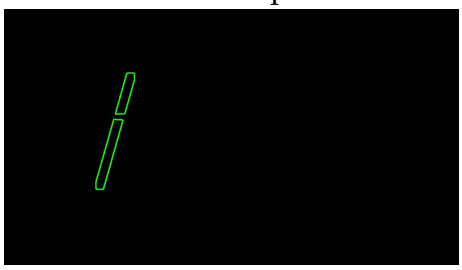
Connected components: 1



Connected components: 2



Connected components: 1



Connected components: 2

2.5 Metrics

When we develop a system, in order to know quantitatively how good it is, we can compute some metrics.
More precisely, the metrics required by the project delivery are **mean Average Precision** and **mean Intersection over Union**. We will analyze each request and we'll explain how it's computed by our system.

2.5.1 mean Average Precision

When we talk about **mean Average Precision (mAP)**, we are referring to the mean of the **Average Precision (AP)** values computed across all classes.

- **Precision** is the number of **true positives** divided by the total number of **predictions**.
- **Recall** is the number of **true positives** divided by the total number of **ground truths**.

Average Precision (AP) can be confusing because it represents the **area under the precision-recall curve**. For each image and each class, we need to compute the **AP** and then take the mean to get the **mAP**.

In our case, we need to compute **mAP for ball localization**, which measures how accurately our system localizes the ball compared to the correct ball positions provided in the dataset.

We use **bounding boxes** for this task, and first, we need to determine when a bounding box is a **true positive** or a **false positive**.

- A ball is a **true positive** when the **Intersection over Union (IoU)** with the corresponding **ground truth** bounding box is greater than a threshold (0.5 in our case) and the predicted ID matches the ground truth ID.
- **IoU** is calculated as the overlap area between two bounding boxes divided by the total area of both bounding boxes minus the overlap area.

Once we have determined **true positives** and **false positives**, we focus on a single class. For each predicted bounding box of that class, we compute **recall** and **precision** values. Then, we use the **PASCAL VOC 11 Point Interpolation Method** to compute the **AP**.
In our case we have developed three functions for the computation of this metrics and, merging them together, this is the principle of work.

We take the ground bounding boxes and the detected bounding boxes. For detected each ball we create a triple of elements such that the **highest IoU** between ground balls, the **ID of the ball** and the **true ID of the relative ground ball**.
Then, we sort our vector of triples for IoU values since the computation of the AP depends on the order and we order in this way to create a sort of **confidence score**: from the theoretical point of view, if a bounding box is not accurate (small IoU), there are more possibilities of miss-classifying it.
Then, for each class of the detected balls (so we look for the second element of the triple), we create a vector of tuple of the type (1,0)/(0,1) depending if it's a true positive or a false positive (criterion explained above).
At the end, for each of this vector of tuple, so for each class, we compute AP using the PASCAL VOC 11 Point Interpolation Method implemented by a function (that receives as input also the number of ground truth for that class, in order to compute recall values) and, at the end, we take the mean among all APs.

2.5.2 mean Intersection over Union

The second metrics that we need to compute is the **mean Intersection over Union** on **segmentation masks**.
More precisely, we have to create, based on our system results, a mask in grayscale with these grayscale values for each pixel belonging to these classes:

- 0. Background
- 1. white "cue ball"
- 2. black "8-ball"
- 3. ball with solid color
- 4. ball with stripes
- 5. playing field (table)

Then, we have to compare our masks with the ground truth masks provided in the dataset and we have to compute the **Intersection over Union**.
The meaning of this IoU is slightly different respect the case for object detection: we need to count the number of pixels with the same class for pair of images, for each class. Then we compute the division between **"intersection pixels"** and **"union pixels"** for each class and, at the end, we take the mean among classes.

In our system we have developed a method for computing this metrics that is very easy: we take a couple of images, one ground and one detected by the program and, iterated for each class, we look to all pixels and if the pixels in both image has the ID of the class of interest, we increment intersection, otherwise if it belongs to the class of interest in at least one image, we increment union.
At the end, we take the division and then we compute the mean among classes.

3

Code Structure

3.1 Execution of the code and file organization

We followed this organization in folders for file management:

- **src**: containing the cpp files.
- **include**: containing the h files.
- **Input_videos**: containing the input video.
- **Output_videos**: where processed videos get saved.
- **First_frames_images**: containing the first frame of each video;
- **First_frames_detected_bboxes**: bounding boxed computed for the first frame of the videos get saved in a txt file inside this folder following the same format to the ones of the ground truth.
- **First_frames_truth_bboxes**: containig txt files for the ground truth of the bounding boxes for the first frame.
- **First_frames_truthmasks**: containing the ground thruth masks for first frames.
- **Last_frames_detected_bboxes**: bounding boxed computed for the last frame of the videos get saved in a txt file following the same format to the ones of the ground truth.
- **Last_frame_2D_maps**: last frame of the video with 2D representation get saved here.
- **Last_frames_images**: containing the last frame of each video.

- **Last_frames_truthmasks**: containing the ground truth masks for first frames.
- **Table_vertices**: containing txt file with the vertices of the table computed in the segmentation task.

The program needs Input_videos, First_frames_images and Last_frames_images to be populated for the segmentation, classification and tracking system tasks to work. For the metrics computation, also the ground truth folders needs to be populated.

CMakeLists.txt is provided in the main folder of the project. Four executable are defined:

- **complete**: executes all project.
- **segmentation_and_classification**: execute segmentation and classification tasks.
- **tracking**: executes tracking system (only after segmentation_and_classification execution).
- **metrics**: executes metrics computation (only after tracking execution).

3.2 Libraries

We developed the following libraries for the relative tasks:

- Utilities.cpp
- Field_detection.cpp
- Balls_segmentation.cpp
- Balls_classification.cpp
- table_orientation.cpp
- tracker.cpp

As choice for the first three tasks (field segmentation, balls detection and classification) we have worked on vector of images, for the clarity of the code.

3.2.1 Utilities.cpp

In this library there are those utilities function, that can be useful in all processes in the project.

- **combinedImages**: This function, given a vector of images as input, create a single image containing all these images, ready to be displayed all together. It creates a sort of grid and insert each image in a cell of the grid. It has been useful since was important to see all images in the same moment.
- **vectorDilation**: This function, given a vector of images as input, applies to each image the dilation function of OpenCV, ensuring that the size of the kernel is mindful (not odd).
- **vectorErosion**: This function, given a vector of images as input, applies to each image the erosion function of OpenCV, ensuring that the size of the kernel is mindful (not odd).
- **vectorSmoothing**: This function, given a vector of images as input, applies to each image the GaussianBlur function of OpenCV, ensuring that the size of the kernel is mindful (not odd).
- **vectorMedianBlur**: This function, given a vector of images as input, applies to each image the medianBlur function of OpenCV, ensuring that the size of the kernel is mindful (not odd).
- **vectorCanny**: This function, given a vector of images as input, applies to each image the Canny function of OpenCV, ensuring that the vector of images given as input is in grayscale.
- **drawCircles**: This function has been used for drawing circles on an image passed by reference. This is very important since circles derives from different processes and it's useful to draw different circles in the same image for refining purposes.
- **readImages**: This function has been used for reading images from a vector of file paths, with the possibility of decide if read them and obtain a BGR image or a grayscale image (this difference is due to the fact that final masks are in grayscale.) The choice is performed by using a boolean value.

3.2.2 Field_detection.cpp

In this library there are those functions that have been used for the detection of the field boundary.
As we have done for Utilities.cpp, functions are based for working on vector of images.

- **HSVsegmentation**: Computes masks in HSV space based on a range of colors for a vector of images. The function converts each image to HSV space and creates a mask where the colors within the specified range are white, and the rest are black.
  This has been used for finding the main field.

- **computeIntersect**: Computes the intersection point between two lines represented by their rho and theta values in the Hough transform space. Returns the intersection point if the lines are not parallel.
  This has been used for obtaining the vertices of our rectangle defining the playing field.

- **orderPoint**: Orders a vector of points in a clockwise direction around the centroid of the set of points. The function calculates the centroid and sorts the points based on their angles with respect to the centroid.
  This function has been used for ordering the vertices of the rectangle since then we need them in this order, for filling the polygon and for other functionalities.

- **vectorHoughLines**: Detects lines in a vector of images using the HoughLines function, computes intersections between the strongest lines, given a limit value of lines, and keeps points within image bounds, returning them as output.
  The function orders these points, draws lines on the output images, and creates masks inside the detected field.

3.2.3 Balls_segmentation.cpp

In this library there are those functions that have been used for the detection of balls.

- **computeMedianColor**: Computes the median color of a neighborhood centered at a specified point in an image.
  The function iterates over pixels within an area defined by the neighborhood size, around the center provided as input, collecting the BGR values and determining their median.
  This approach helps avoid the influence of small pieces of balls that might be present in the table portion.

- **computeMedianColorOfCenteredROI**: Computes the median color of a region of interest (ROI) centered in an image. This function uses the `computeMedianColor` function, providing the image center as the point parameter and the user-defined ROI size.
  We have used this for computing the color of the table for the mask.

- **vectorMaskRGB**: Segments the table from the balls in a set of images.
  Pixels are masked if their color is within a threshold distance from the table color, provided as input, black (shadows or holes), or white (holes

or other features). The function returns a vector of masks where, ideally, only balls are identified.

- **houghCircles**: Applies the Hough Circle Transform to an image to detect circles.
  The function converts the image to grayscale if necessary and uses the specified parameters for the HoughCircles function, returning a vector of detected circles.
- **calculateDistance**: Calculates the Euclidean distance between two points. The function computes the squared differences for the x and y coordinates, sums them, and takes the square root.
- **filterCircles_nearHoles**: Removes circles that are too close to specified hole points. In input is given a raw set of circles as output from HoughCircles.
  The function computes the distance between each circle and the hole points, removing circles within a specified threshold distance.
- **calculateColorDifference**: Calculates the Euclidean distance between two colors in BGR space given as input.
  The function computes the squared differences for each color channel, sums them, and takes the square root.
- **getColorDifferenceSum**: Calculates the sum of the differences between the colors inside a circle and the median color of the table. The function creates a mask for the circle, iterates over the pixels inside the circle, and sums the color differences using the `calculateColorDifference` function.
- **filterCircles_replicas_1**: Refines two sets of circles by removing duplicates found by different processes.
  The function maintains only one circle from overlapping pairs, selecting the circle with the higher sum of color differences from the table's median color, since, ideally, it's the one that has less portion of table inside the circle.
- **filterCirclesByColor**: Filters out circles with colors similar to the color of the table. The function calculates the median color within each circle, compares it to the table's median color, and removes circles within a specified color difference threshold.
  This has been used for removing those circles that were found in the table or in portions of boundaries of the field.
- **refineCircles_replicas_2**: Refines two sets of circles by removing smaller circles when they overlap with larger ones.
  The function compares the distances between circles in the two sets and keeps only the larger circle when they are close to each other.
  This has been used since circles derives from different processes and, due to this fact, it may happen that two circles are found for the same ball.
  This has been used after `filterCircles_replicas_1`.
- **closestPointOnLineSegment**: Computes the closest point on a line segment to a given point. The function projects the point onto the line segment and clamps the projection factor to the segment's endpoints if necessary.

- **pointsNearLines**: Refines circles that are close to the table's boundary. The function computes the minimum distance from each circle's center to the nearest edge of the table and removes circles within a specified distance threshold.
- **refineCircles_ROI**: Refines detected circles by applying the Hough Circle Transform to regions of interest (ROIs) around each circle. The function adjusts the circles' positions relative to the original image after detection in the ROIs.
- **maskBlack**: Creates masks for images to detect regions with very dark pixels. The function sets mask pixels to zero if their color values are within a specified threshold for black, helping to detect black balls in regions with changing illumination.

3.2.4 Balls_classification.cpp

In this library there are the function used for the balls classification task.

- **applyMedianFilter**:

  This function takes in input an image and a bounding box, and applies a median filter in the area of the image covered by the bounding box.

- **countCannyEdges**: This function takes in input an image, a bounding box and two integer values. The function computes the circle inscribed in the bounding box and uses it as mask to apply the Canny edge detection to the input image. The two input integers are used as lower and upper thresholds for the Canny function.

- **whiteRatio** This function takes in input an image, a bounding box and two integer values. After converting the input image from BGR to HSV, the function computes the circle inscribed in the bounding box and sets two thresholds for the saturation and value parameters. The function counts the number of pixels in the ROI that have values above the thresholds and computes the ratio between this number and the total number of pixels in the ROI. While using six parameters (two for hue, two for saturation, two for value) only the upper saturation and the lower value thresholds are tunable, since they are the ones used to identify white pixels.

- **darkRatio** This function takes in input an image, a bounding box and an integer values. After converting the input image from BGR to HSV, the function computes the circle inscribed in the bounding box and sets a threshold value parameter. The function counts the number of pixels in the ROI that have value below the treshold and computes the ratio between this number and the total number of pixels in the ROI. While using six parameters (two for hue, two for saturation, two for value) only the upper value threshold is tunable, since it is the one used to identify darker pixels.

- **smoothWhiteRatio** This function takes in input an image a bounding box and three integers. It is used to apply Gaussian smoothing and the median filter to the image before applying the whiteRatio function. The three integer are used to tune the parameters of the Gaussian filter (kernel size) and whiteRatio (upper saturation and lower value thresholds).

- **classifyBall** This function takes in input the image and a bounding box and is the one responsible for the actual classification of solid and striped balls. It computes ratios and values using the other functions of the library and iteratively applies various thresholds to in the bounding box area of the image. The function terminates when an if-else condition is met, and the class of the considered ball is returned ("3" if solid, "4" if striped).

3.2.5 Classes.cpp / Classes.h

In this header/cpp we have defined useful **structure and classes** and **methods** related to classes for our project.
The choice of structures respect to classes in some cases is due to the fact that for these objects there were not needed to have methods
More specifically, we have defined:

- **structure Circle** with x,y (center)and radius as parameter.

- **structure Bounding Boxes** with x,y (top left corner), width, height as parameter

- **class Ball** with x,y,width,height as a BoundingBox and ID as parameter. Moreover this class has also some methods:

  - **validateID**: Validates the ball ID. If the ID is not between 0 and 5, it prints an error message and sets the ID to 0.
  - **display**: Displays the ball's information, including its ID, position, and size.
  - **getInfo**: Returns the ball's information as a string, including its ID and position.
  - **computeArea**: Computes and returns the area of the ball based on its width and height.

- **createBoundingBoxes**: Creates bounding boxes from Circle objects. For each circle, it scales the radius by a given factor and calculates the bounding box coordinates and dimensions. Returns a vector of vectors of BoundingBox objects.

- **createBallsFromBoundingBoxes**: Merges the work of classification and bounding box detection, creating Ball objects from vectors of BoundingBoxes and classification IDs. For each image, it creates a vector of Ball objects with dimensions taken from the bounding boxes and IDs from the classifier. Returns a vector of vectors of Ball objects.
- **readBallsFromTextFiles**: Reads text files containing ground truth data and creates Ball objects. Each file contains lines with the position, size, and ID of a ball. Returns a vector of vectors of Ball objects, one for each image.
- **showBoundingShapes**: This function takes as input a vector of vector of ball and a boolean and depending on the boolean, draws in the images given as input circles or bounding boxes, with the color depending on the ID of the ball.
- **createFinalMasks**: Creates final masks for images by drawing bounding boxes and circles based on Ball objects and table boundary points. The masks assign specific gray values to different types of pixels for computing mIoU. Takes a vector of images, a vector of vectors of Ball objects, and a vector of vectors of table boundary points, and modifies the images in place by drawing the masks.
- **transformMasksToColor**: Converts grayscale masks to color images for better visualization. Each pixel value in the grayscale image is mapped to a specific color. Takes a vector of grayscale images and returns a vector of color images.

3.2.6 TABLE_ORIENTATION.CPP

This library contains methods that have been used detecting the orientation of the table.

- **calculateDistance**: Calculates the euclidean distance between two points.
- **getPaths**: Retrieves file paths matching a pattern in a specified folder. Returns a vector of file paths matching the given pattern.
- **readVerticesFromFile**: Reads vertex coordinates from a file.
- **processImageLine**: Processes a line segment of the table edge in the image. Computes a shorter line for avoid holes in the angles, then dilates the line and applies a medianBlur for smoothing the image. Finally, applies a threshold to better detect the holes. Returns the processed image.
- **countComponents**: Counts the number of separated components in the image.

- **orderVerticesShortFirst**: Reorders the vertices based on the detected table orientation. Order should be such the first point and the second one form a short side, the others are ordered consequently clockwise.
- **writeVerticesToFile**: Writes the ordered vertices to a file. This is used mainly for debug reasons.
- **processImage**: Processes a single image, detecting table lateral holes and ordering vertices. In particular, if vertices are detected such that the first and the second vertex form a long side, all the vertices are shifted by one position such that now first and second vertex form a shord side.
- **tableOrientation**: To execute all the steps above. Returns a vector of vectors containing the vertices of the table for each video oriented clockwise with first and second vertices forming a short edge.

3.2.7 TRACKER.CPP

This library contains methods that have been used for implementing the tracking system. This file imports and uses Class.h. and orientedTable.h

- **drawTransformedBalls**: Draws a ball in the 2D map after computation of the prospective transformation of the original ball.
- **drawTransformedPath**: Draws the trajectory of the ball in the 2D map starting from the center of the ball in the original image and computing the prospective transformation. A point on the 2D representation is left in correspondece of the center of the ball for each frame.
- **readBoundingBoxes**: Reads bounding box information from a file.
- **initializeVideoCapture**: Initializes video capture from a file.
- **computePerspectiveMatrix**: Computes a perspective transformation matrix. For this function hardcoded points are selected for the destination, being the one that correspond to the vertices of the billiard table in the selected image to be used as 2D map.
- **initializeOutputVideo**: Initializes a VideoWriter for output.
- **initializeTrackers**: Initializes a trackers for each ball in the first frame.For each ball a TrackerCSRT is created. Each tracker maps a specific ROI, that is actually a bounding box of a ball.
- **loadAndResizeTableMap**: Loads and resizes the table map image. In particular, the desired height is set to be 200 pixels, while the width is such that the original ratio of the image is maintained.
- **processSingleFrame**: Processes all frames in the video.

- **writeLastFrameBB**: Writes the bounding box information of the last frame to a file and saves the last frame image. This is used then for ball classification of the last frame of each video and for the computation of the metrics.
- **processFrames**: Processes all frames in the video.

3.2.8 METRICS.CPP

This library contains methods that have been used for computing metrics and result for our problem.
More specifically, in the project delivery is required the computation of **mAP for ball localization** and **mIoU for balls and playing field segmentation**.

- **calculateIoU**: This function has been used for computing the IoU of two bounding boxes, given as input two Ball objects.
  This has been used then in the computation of mAP for ball localization.
- **checkDetection**: This function has been used for checking if a ball is a true positive or a false positive based on the IoU.
  In input it takes the value of the IoU and the threshold for determining if it's a TP or a FP and then return a bool true based on the result.
- **calculateIoUVectors**: This function has been used for computing, for each image, a vector of triple, corresponding to all balls of the image.
  More specifically, the triple is composed by the max IoU between the current bounding box and the truth bounding boxes, the ID of the current boudning box and the ID of the "correspective" true bounding box.
  This is why we need to compute AP for each class.
- **calculateAP**: This function has been used for computing AP given a vector of tuple (TP, FP).
  The idea is to obtain, for each class of each image, a vector of tuple of this type:
  - $(1, 0)$ if it is a TP for that class, so $IoU > 0.5$ AND $ID = ID\_True$
  - $(0, 1)$ if it is a FP for that class

  Then the function computes the AP with 11 point interpolation Pascal using as input this vector and the number of ground truths for class.
- **calculateAPs**: This function has been used for computing finally the mAP for each image.
  For each image we have a vector of triples as explained above, and from this we obtain the vector of tuple (TP,FP) for each class.
  However, before all, we sort our vector in decreasing order based on IoU.
  That's because, since the final value is afflicted by the order, we put before

what is in theory more probably to be correctly classified, so bounding boxes well centered.
At the end, the final value of mAP is obtained.

- **computeIoUForClass**: This function has been used for computing the IoU for a single class in a single pair of segmentation masks. More specifically, it counts the number of pixels in common between the ground truth and the predicted masks, and the number of pixels in the union of the masks. Then, by using these values, we compute the IoU by dividing these two quantities.

- **computeMeanIoU**: This function has been used for computing the mean IoU for all classes in a single pair of segmentation masks. It simply applies the computeIoUForClass function for each class and take the mean between the IoU of different classes.

- **vectormIoU**: This function applies computeMeanIoU for each couple of images contained in a two vector of images and, for each image, returns the mIoU.

3.2.9 Considerations

As mentioned before, in the aim of working efficiently some tasks have been parallelized. This may have lead, together with the requirement that each file should belong to one member of the team, to the development of some conceptually similar functions that differs among them for only few details. Also, for allowing the independence of the tasks, some functions write the output in a file, such that to be read in other independent tasks, such as the metrics computation.

3.3 Main

We have developed **three different main** for three different implementations. More specifically, we have developed one main for **field segmentation and balls detection and classification**, one for **tracking and 2D minimap** and one for computing **metrics**. This has been really useful for testing the tasks separately (although a main can be executed only if the previous task has been computed). To execute all the tasks at once, a fourth main has been created, containing the same content of the other three main, organized in three functions.

- **Main_field_balls.cpp**: In this main there is the development of segmentation of the playing field, and the balls detection and classification.
  We have used libraries described above with the approach described in the Section 2. At the end we saved our detected bounding boxes in files that are then used in the main to compute the metrics.

- **main_tracking.cpp**: In this main there is the development of the ball tracker and the 2D minimap. Moreover, it produces also the classification of the last frame for each video, providing in output a .txt file containing all the bounding boxes with relative class id. Works by reading the videos, the initial bounding boxes and the table vertices from the relative folders.

- **Main_metrics.cpp**: In this main there is the development of the computation of metrics.
  More specifically, we took as input files containing detected bounding boxes and truth bounding boxes and, by using libraries described above, we computed mAP and mIoU.

- **main.cpp**: contains main_field_balls, main_tracking, main_metrics functions (the three previous main merged together).

4

Results

In the following part we report **the results obtained on the first frames provided by the project delivery and the last frames, obtained at the end of tracking, done on the first frames bounding boxes obtained by our program, in order to include also the goodness of the tracker**.
For each video, we will report **quantitative and qualitative** results, or rather **mAP, mIoU, segmentation mask and object detection bounding boxes** for first and last frame.
As we will see then, the metrics for last frames are slightly worst, due to this reasons:

- When a ball is pocked, the tracker maintain the ball in the table
- The tracker slightly moves the position of the bounding boxes also for the balls that did not actually move.

4.1 Segmentation masks and metrics computation

Game 1 Clip 1

First frame

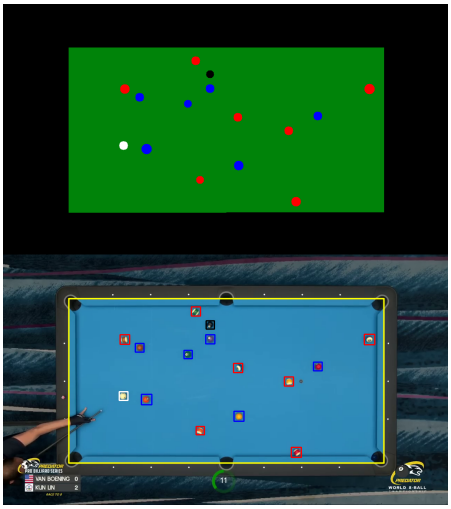


Figure 4.1: Qualitative results on first frame of the Game 1 Clip 1

mAP: 1
mIoU: 0.796942
Last frame

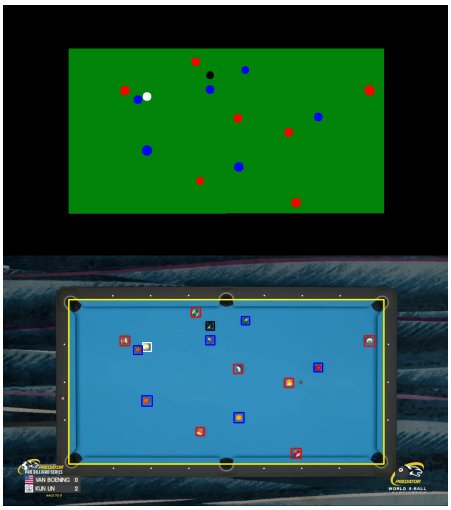


Figure 4.2: Qualitative results on last frame of the Game 1 Clip 1

mAP: 0.954545
mIoU: 0.78474

Game 1 Clip 2

First frame

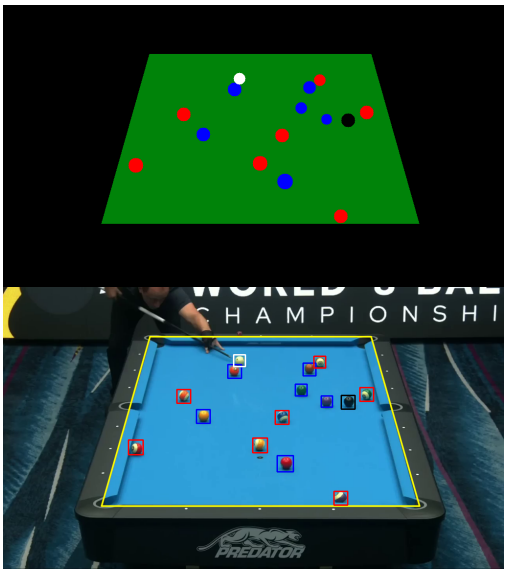


Figure 4.3: Qualitative results on first frame of the Game 1 Clip 2

mAP: 1
mIoU: 0.781846
Last frame

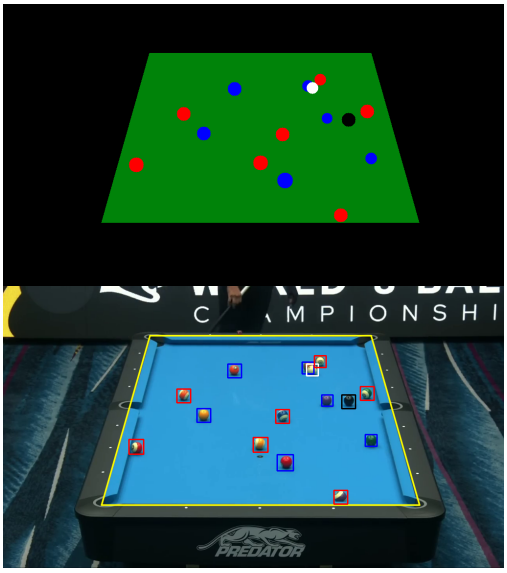


Figure 4.4: Qualitative results on last frame of the Game 1 Clip 2

mAP: 1
mIoU: 0.812064

Game 1 Clip 3

First frame

Figure 4.5: Qualitative results on first frame of the Game 1 Clip 3

mAP: 1
mIoU: 0.834732
Last frame

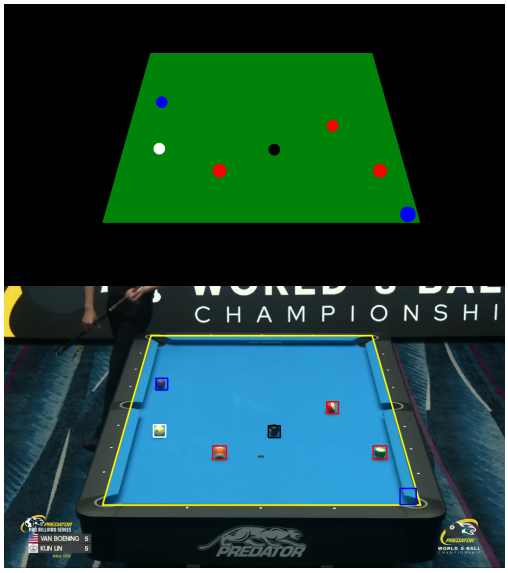


Figure 4.6: Qualitative results on last frame of the Game 1 Clip 3

mAP: 0.886364
mIoU: 0.776863

Fourth video

First frame

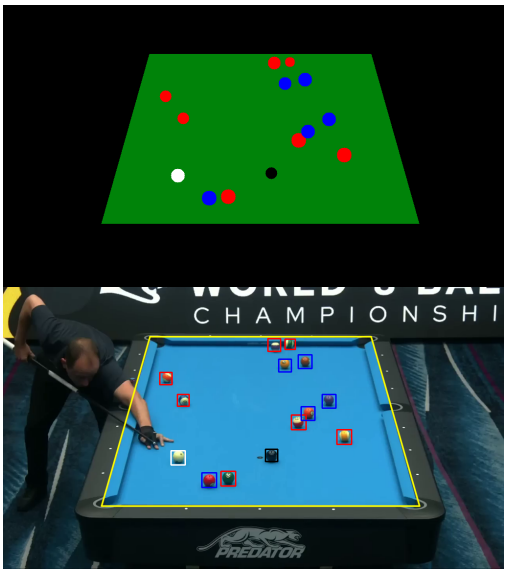


Figure 4.7: Qualitative results on first frame of the Game 1 Clip 4

mAP: 0.931818
mIoU: 0.768384
Last frame

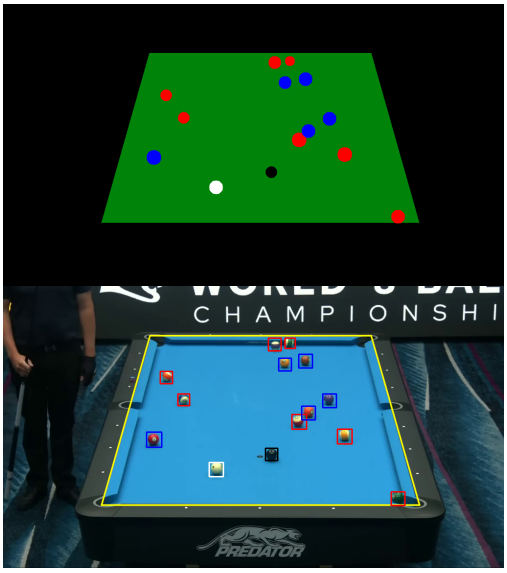


Figure 4.8: Qualitative results on last frame of the Game 2 Clip 4

mAP: 0.931818
mIoU: 0.777112

Game 2 Clip 1

First frame

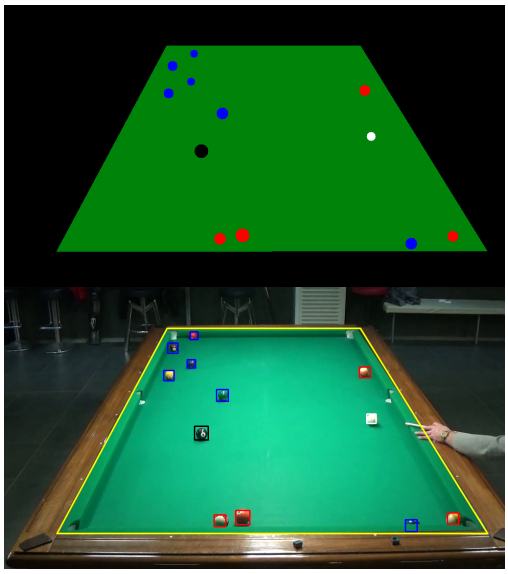


Figure 4.9: Qualitative results on first frame of the Game 2 Clip 1

mAP: 0.892045
mIoU: 0.692788
Last frame

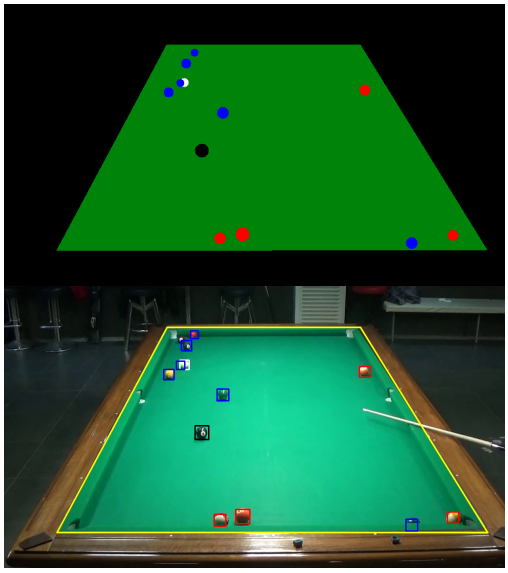


Figure 4.10: Qualitative results on last frame of the Game 2 Clip 1

mAP: 0.755682
mIoU: 0.681246

Sixth video

First frame

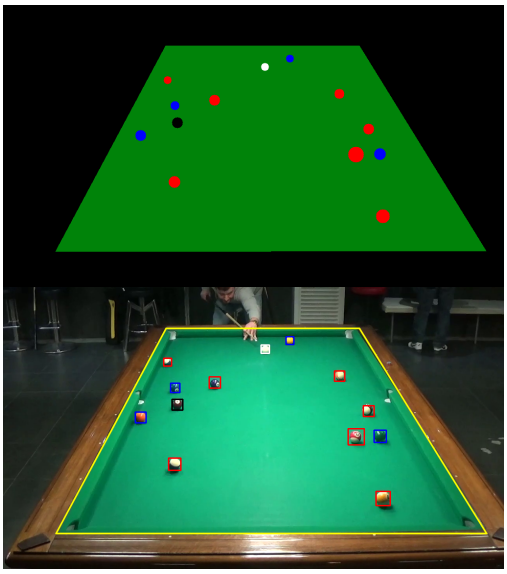


Figure 4.11: Qualitative results on first frame of the Game 2 Clip 2

mAP: 0.954545
mIoU: 0.804568
Last frame

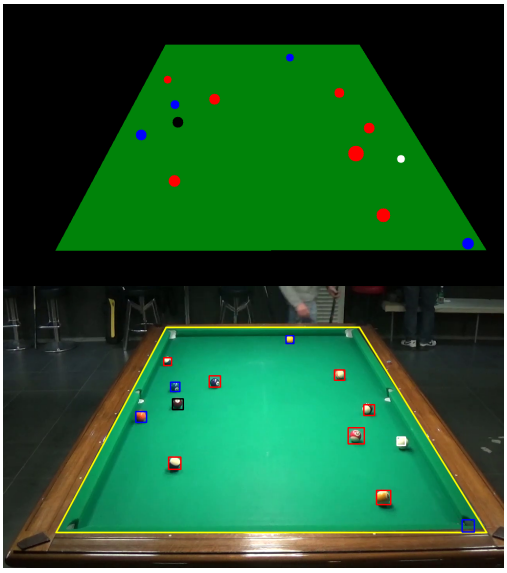


Figure 4.12: Qualitative results on last frame of the Game 2 Clip 2

mAP: 0.425325
mIoU: 0.563605

Game 3 Clip 1

First frame

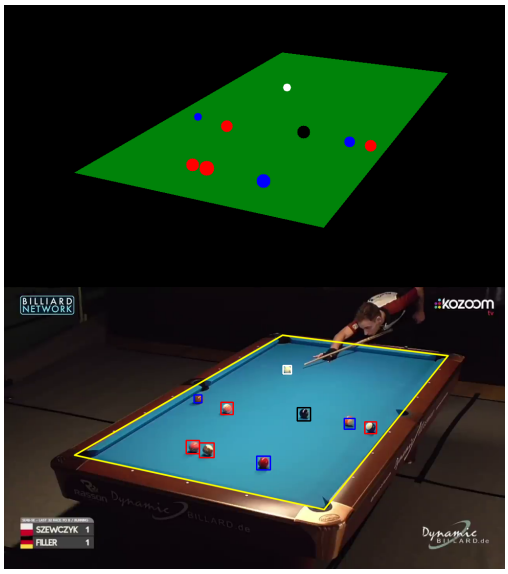


Figure 4.13: Qualitative results on first frame of the Game 3 Clip 1

mAP: 0.829545
mIoU: 0.696031
Last frame

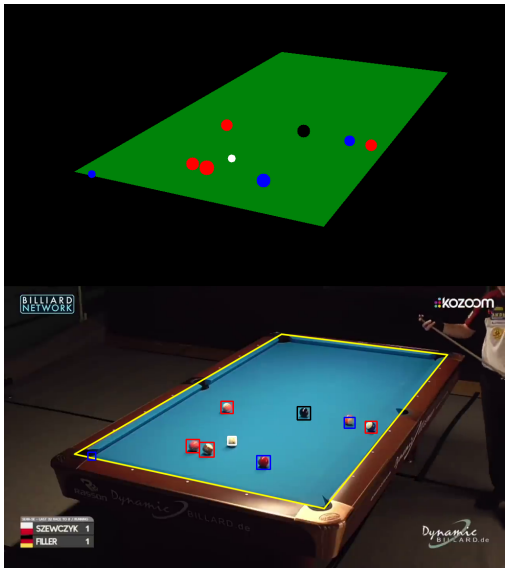


Figure 4.14: Qualitative results on last frame of the Game 3 Clip 1

mAP: 0.676136
mIoU: 0.630167

Game 3 Clip 2

First frame

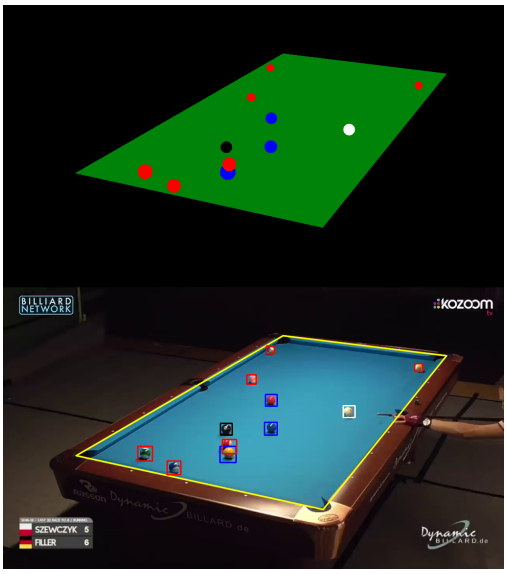


Figure 4.15: Qualitative results on first frame of the Game 3 Clip 2

mAP: 0.954545
mIoU: 0.797972
Last frame

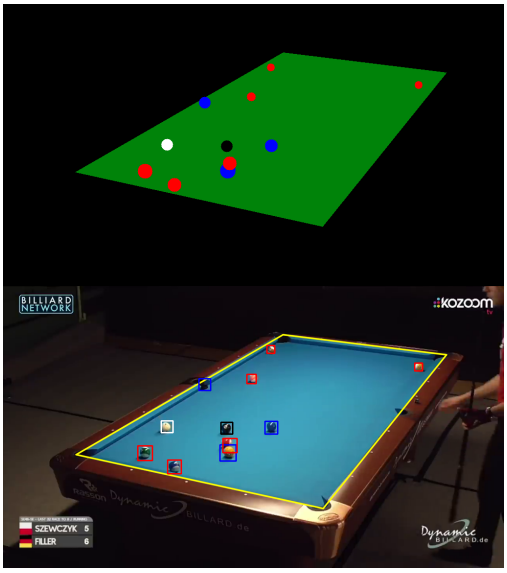


Figure 4.16: Qualitative results on last frame of the Game 3 Clip 2

mAP: 0.795455
mIoU: 0.74323

Game 4 Clip 1

First frame

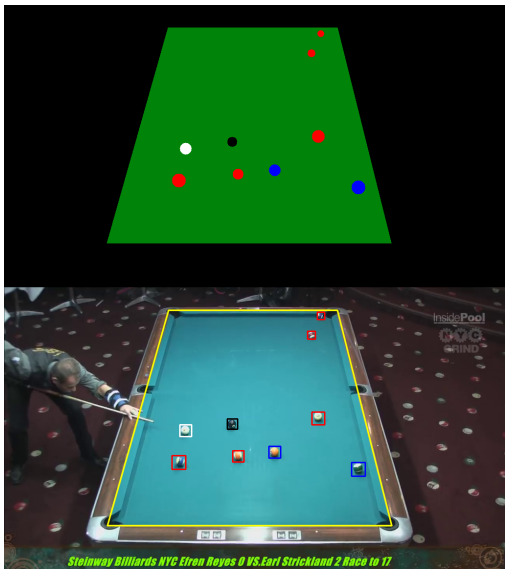


Figure 4.17: Qualitative results on first frame of the Game 4 Clip 1

mAP: 0.954545
mIoU: 0.720027
Last frame

Figure 4.18: Qualitative results on last frame of the Game 4 Clip 1

mAP: 0.590909
mIoU: 0.666562

Game 4 Clip 2

First frame

Figure 4.19: Qualitative results on first frame of the Game 4 Clip 2

mAP: 0.931818
mIoU: 0.757483
Last frame

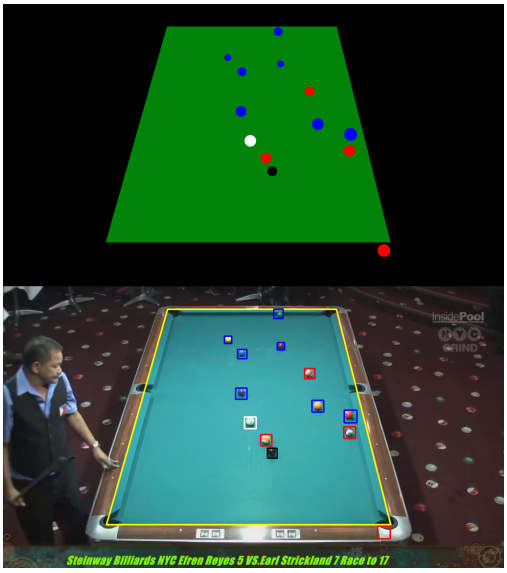


Figure 4.20: Qualitative results on last frame of the Game 4 Clip 2

mAP: 0.840909
mIoU: 0.720551

4.2 Last frames of the tracking system with the 2D map

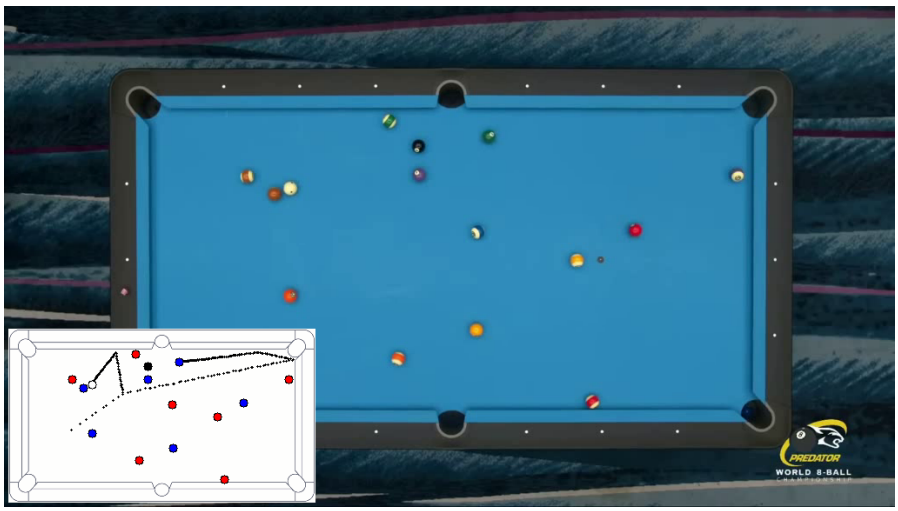


Figure 4.21: Last frame: Game 1 Clip 1

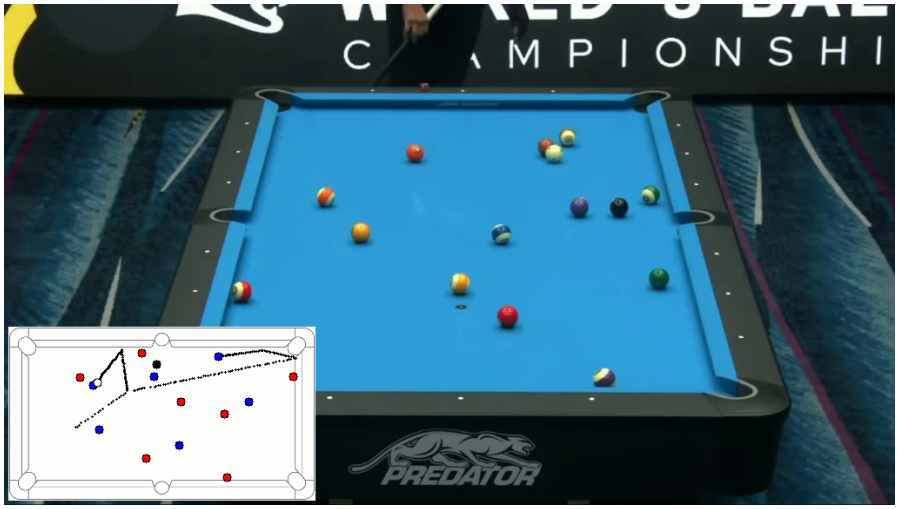


Figure 4.22: Last frame: Game 1 Clip 2

Note: from the 2D representation it's easy to notice that Game 1 Clip 1 and Game 1 Clip 2 are actually the same game but from two different points of view.

Note: here the white ball is hidden behind a blue ball, although still partially visible. Also, this is the only case in which the tracking system gets fooled in this dataset. From the 2D map seems that a ball moved alone, the reality is that a solid ball got lost by the tracker after contact with the white ball.

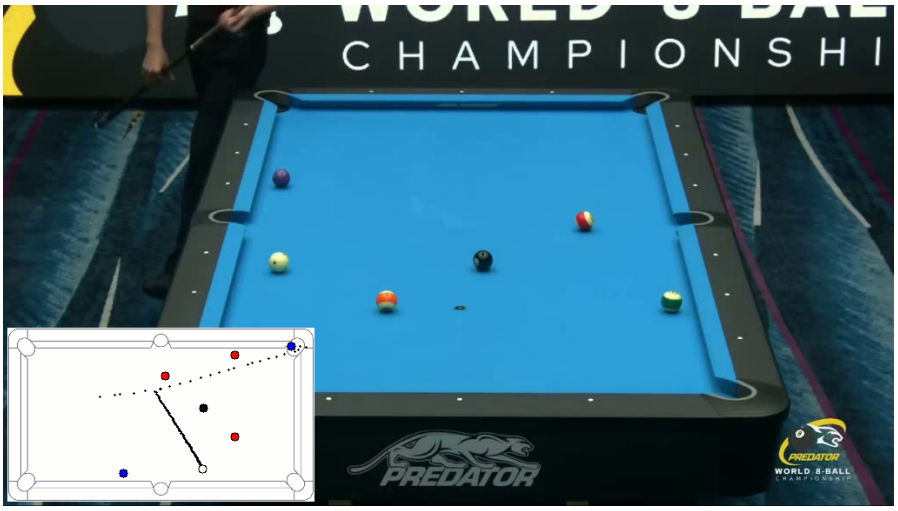


Figure 4.23: Last frame: Game 1 Clip 3

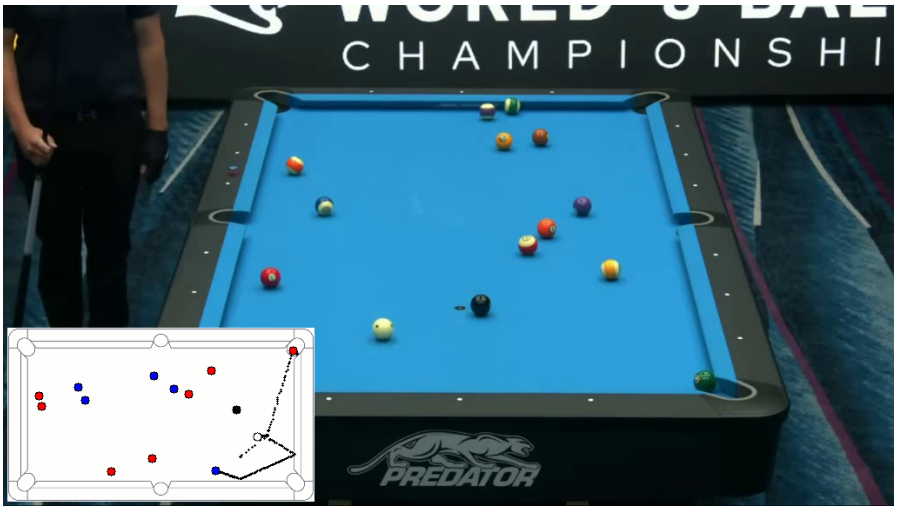


Figure 4.24: Last frame: Game 1 Clip 4

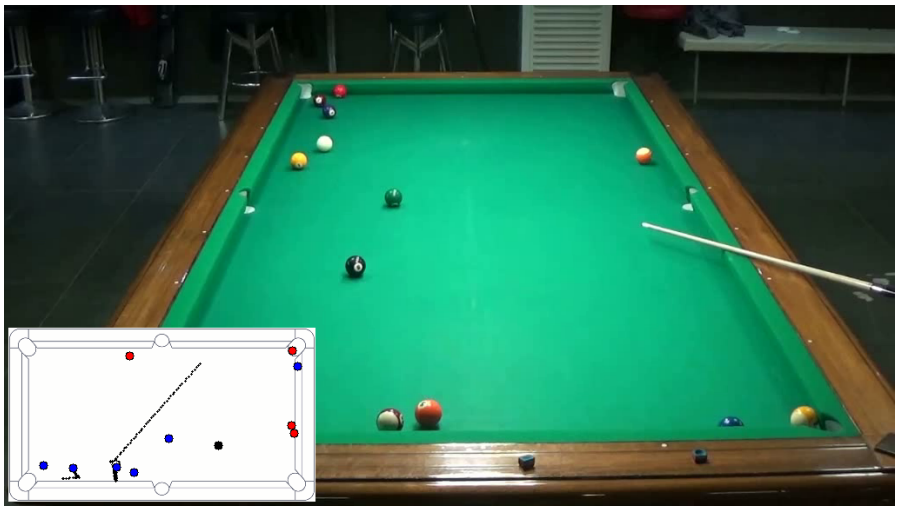


Figure 4.25: Last frame: Game 2 Clip 1

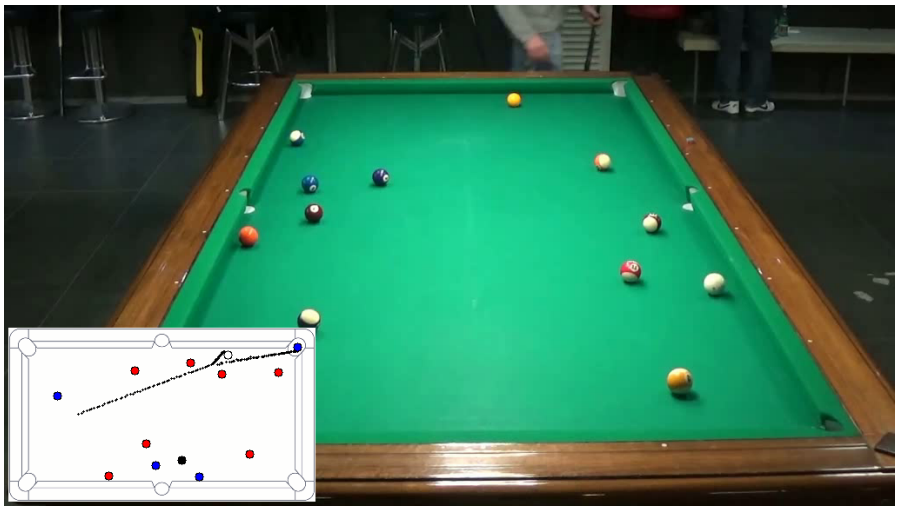


Figure 4.26: Last frame: Game 2 Clip 2

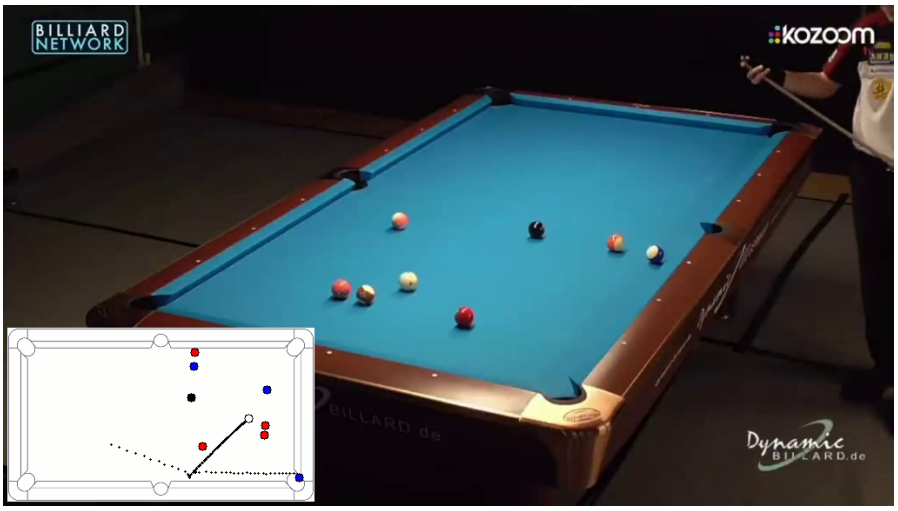


Figure 4.27: Last frame: Game 3 Clip 1

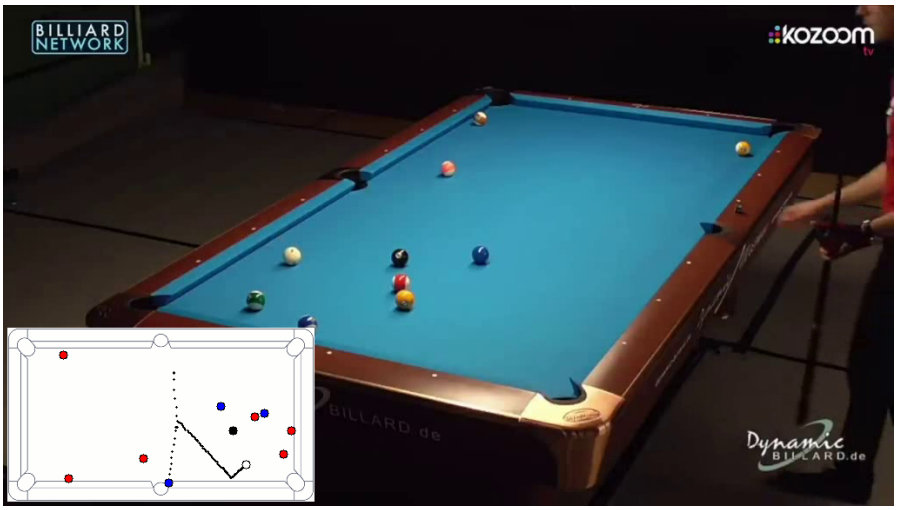


Figure 4.28: Last frame: Game 3 Clip 2

Figure 4.29: Last frame: Game 4 Clip 1

Figure 4.30: Last frame: Game 4 Clip 2

5

Contribution of each member

In this section are reported contributions of each member of the group, with related hours of working.

Lovo Manuel, 150 working hours

- Detection of the playing field and implementation of the related library
- Detection of balls and implementation of the related library
- Implementation of the library Classes.cpp and Metrics.cpp
- Implementation of the main for metrics and the general main
- Section "Introduction", "Approach" for his developments, "Code Structure" for his developments, "Results" of the report
- Code ordering, fixing and merging

Alberto Bressan, 70 working hours

- Classification of balls and implementation of the related library
- Section "Approach" and "Code Structure" for his developments
- Merging of code and minor fixing

Martina Boscolo Bacheto, 120 working hours

- Implementation of the tracking system (reading and saving videos, use of the tracker, 2D game representation)
- Orientation of the billiard table
- Testing on the Virtual Lab environment (all tasks and metrics)
- CMakeLists.txt and partial files organization
- Minor code fixes and merging with the other tasks
- Minor contribution (some of the ideas) for field segmentation.
- Sections of the report related to the previous tasks, in particular: "2D top-view visualization", related files in "Code Structure", "Execution of the code and file organization", 2D maps in "Results"