# Sport video analysis for billiard matches

Computer Vision Project - June/July 2024

Information Engineering Department (DEI) - University of Padua

Artico Giovanni
giovanni.artico@studenti.unipd.it

Giacomin Marco
marco.giacomin.4@studenti.unipd.it

Toffolon Mattia
mattia.toffolon@studenti.unipd.it

## Contents

# 1 Introduction

In this first section, the main purpose of the project and how the developed system has been evaluted will be presented.

## 1.1 Problem specifics

The aim of this project is to develop a computer vision system for analyzing video footage of various "Eight Ball" billiard game events by providing high-level information about the status of the match for each video frame in the form of a 2D top-view minimap.
More in detail, the system should be able to:

- Recognize and localize all the balls inside the playing field, distinguishing them based on their category (1-the white "cue ball", 2-the black "8-ball", 3-balls with solid colors, 4-balls with stripes);
- Detect all the main lines (boundaries) of the playing field;
- Segment the area inside the playing field boundaries detected in point 2 into the following categories: 1-the white "cue ball", 2-the black "8-ball", 3-balls with solid colors, 4-balls with stripes, 5-playing field;
- Represent 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 and the trajectory of each ball that is moving;
- Create a modified version of each video in which the generated 2D maps are overlayed on a corner of the respective frames.

The available dataset consists of 10 folders each one corresponding to a different game event and containing:

- a `.mp4` file corresponding to the event video;
- a `.png` file corresponding to the first frame of the video;
- a `.png` file corresponding to the last frame of the video.

## 1.2 Performance evaluation

To evaluate the system performance, the metrics that were adopted are:

- mean Average Precision (mAP);
- mean Intersection over Union (mIoU).

Further insights on how these values were computed and used can be found in Section 7.

## 1.3 Sub-task identification

To develop the system, the problem was divided into the following sub-tasks:

- Table segmentation;
- Table sides recognition;
- Balls localization and segmentatation;
- Balls identification;
- Balls tracking;
- Map rendering;
- System performance evaluation.

## 1.4 Used material

If not specified otherwise in the relative section, the developed algorithms always operate exclusively on the given `.mp4` files and eventual data generated by the program itself. The only additional external material used was the empty minimap of the billiard table which was necessary for the requested video rendering.

## 2 Table Segmentation

The following assumptions were made for the segmentation of the table:

- the table is of uniform color and it is a prevalent part of the image;
- the table is centered (or close to being centered) in the image.

The reason for these is going to be evident when the full algorithm for the table segmentation is explained.

The first thing that had to be found was a rough segmentation of the table. Finding the lines with only the Canny-transform of the full image was problematic, as the strongest lines in the image do not correspond to only the table fabric, but also the table itself and other things in the image. The initial approach was to use Otsu's thresholding. Since the table is of uniform color, it was assumed that the table would have been thresholded the same way throughout. Althought, this proved unfruitful because the obtained segmentation was too rough (Figure 1). The second approach involved K-means clustering with Kmeans++ initialization, in particular exploiting the adaptability of the algorithm to the use of spatial features to improve the segmentation, as we assumed the table being a continuous entity. Out of the K clusters obtained, the most centered one was picked as shown in figure 2a.

The segmentation obtained was fairly precise but contained multiple disconnected components. However, we can assume the table being the biggest one and pick it. Additionally, before doing this, the morphological operation *Opening* is applied in order to remove weakly connected objects. The K parameter was empirically chosen as 3, since 2 lead to undersegmentation of the image and 4 to oversegmentation. The resulting segmentation is precise in most cases, but given the nature of Kmeans, on some images the algorithm fails because it clusterizes some unrelated parts of the table with the fabric, due, for example, to lighting effects. For this reason further processing was required: given the obtained rough mask of the table, we now obtain the mean color of the image in the pixels where the mask is non-zero. After this, we use a color thresholding by distance as shown in figure 2b from the mean and the same way we did before to obtain the largest connected component. Thresholding on the hue channel was the most robust option (instead of using the distance between BGR colors). In particular, a low distance of 5 was chosen, but different values close to 5 showed similar results.

After having obtained this result, the morphological operation *Close* was applied in order to remove some noise from the mask, like balls and other objects. Afterwards the *Hough Transform* algorithm is used to find the lines, with a threshold in order to remove similar ones. The vertices of the table were found using the equations from wikipedia. It must noted that the last frame is used for this task, as most of the time it doesn't contain the player in the table since it's the end of his turn. The segmentation found is technically not precise since we count as part of table player, holes and billiard balls. The latter can be easily solved by simply removing the areas of the balls that are detected. As for the other two cases, they were ignored for a compromise on performance, as in some cases the change in illumination is severe, causing some parts of the table to have extreme differences in color, for example in the internal parts of the table. This isse was solved through the *Hough Transform*. It should also be considered that the morphological operations used to isolate the connected component of the table, also worsen the general table outlines. For this reason the output of the line detection is preferred, as it yields more consistent and stable results as shown in figure 2c.

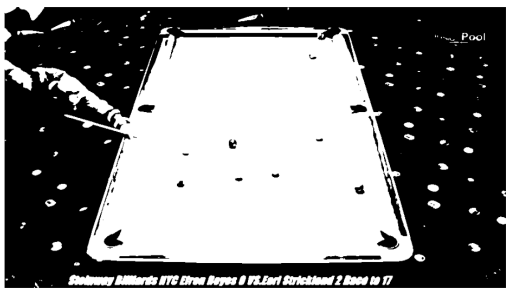
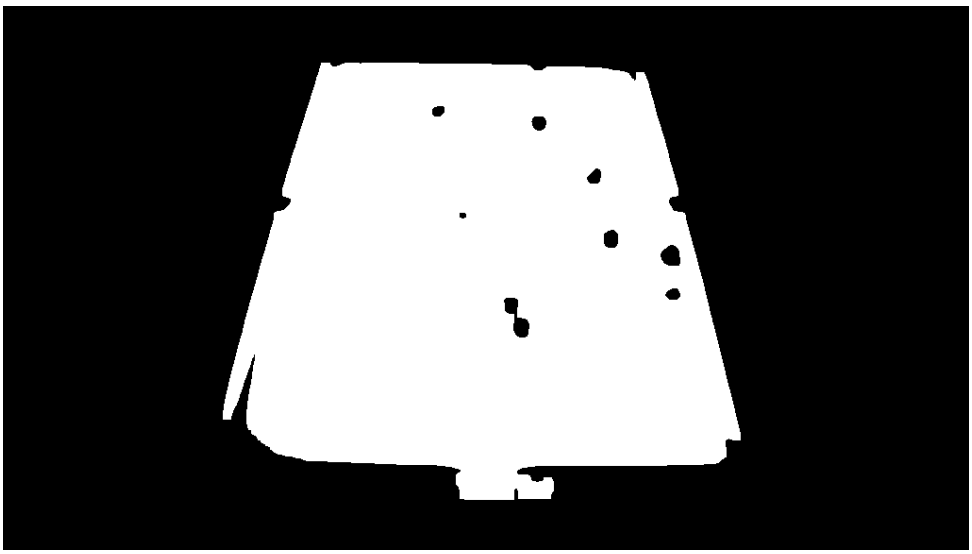
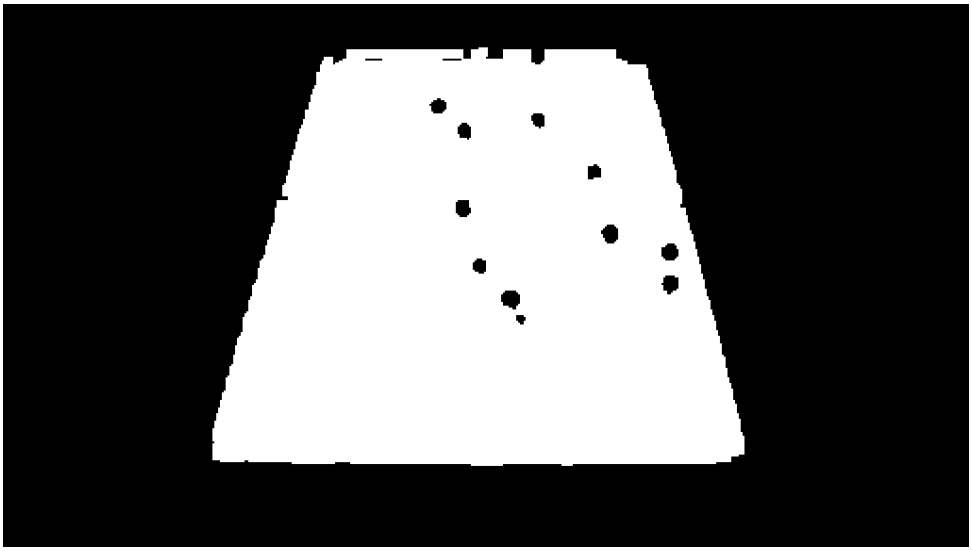


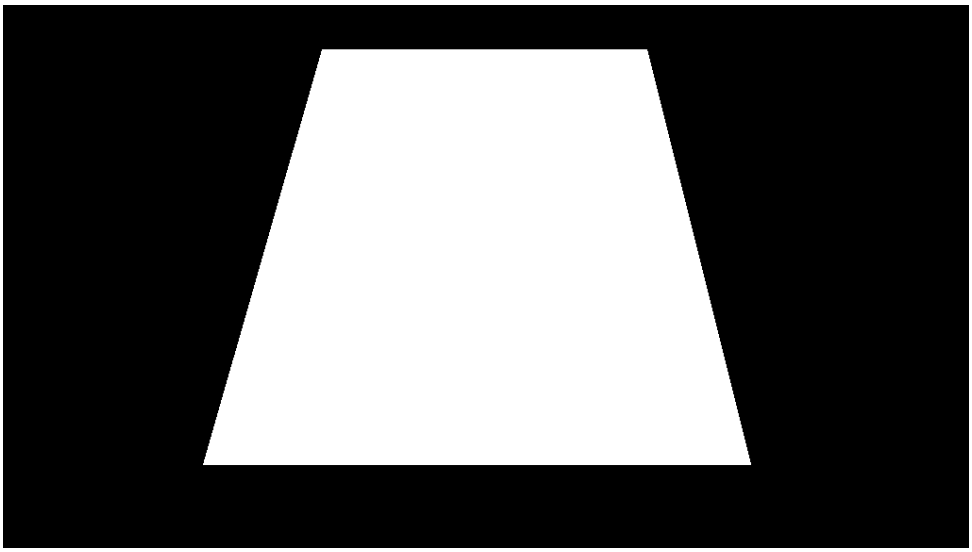

Figure 1: Example of Otsu Segmentation applied to game4_clip2



(a) Initial Kmeans clustering result applied to game4_clip2



(b) Color clustering result



(c) Mask obtained from filling the area contained in the lines found

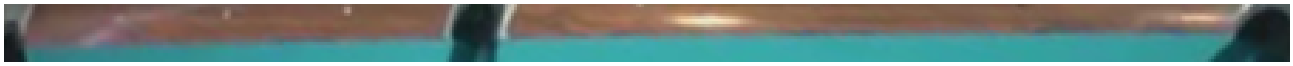
Figure 2: Stages of the segmentation applied to game4_clip2

(a) Processed rectangle of a long side

(b) Processed rectangle of a short side

Figure 3: Output of the processing of the sides

(a) extreme illumination change in the center of the side



(b) not centered hole

Figure 4: Examples of difficulties in side recognition

# 3 Table Side Recognition

The points of the table quadrangle are ordered from the upper-left-most point in clockwise order, but this doesn't give any information on how the sides are ordered. It is unfortunately impossible (or out of my mathematical capabilities) to find any meaningful geometrical relation between the sides lengths given the perspective deformations in some cases. The only feature we could exploit was the precence of holes in the middle of the longer sides. The first operation was to obtain the rotated rectangles containing the sides of the pool table, with a small width in order to avoid keeping in the images unneeded information.

Multiple approaches were tested to distinguish the two types of sides. The first was to use features (such as *Harris Corners* or *SIFT*), as intuitively they should appear on the holes and not on uniform sides since they should be classified as edges. This didn't work, as the features and corners detected were spread throughout the considered sides most of the as shown in figure 5.

The next approach was to try *Template Matching* between different segments of the image, more specifically as the center contains a hole which should make the match significantly worse than the left and right segments. This did not work for two reasons:

- because of the perspective, the hole can appear in a part of the segment of the image that is not the central one;
- because of the illumination and small imprecisions on the table segmentation the matching would fail even on the segments which should result similar.

Both of these are shown in figure 4.
Ultimately, the way to recognize the sides was the following:

1. apply the Canny tranform on the initial image;
2. extract the rectangles from the image;
3. rotate the rectangles so that they are all horizontal;
4. apply sobel on the x-axis shown in image 3;
5. sum over the contributions on the middle two quarters (to avoid considering the corner holes) of the rectangles normalized over the number of pixels contained in each;
6. determine the longer sides by the highest contributions.

This works because of the nature of the edges on the sides: on shorter sides there are mostly parallel lines, instead on longer ones there are strong edges orthogonal to the direction of the sides given by the holes.

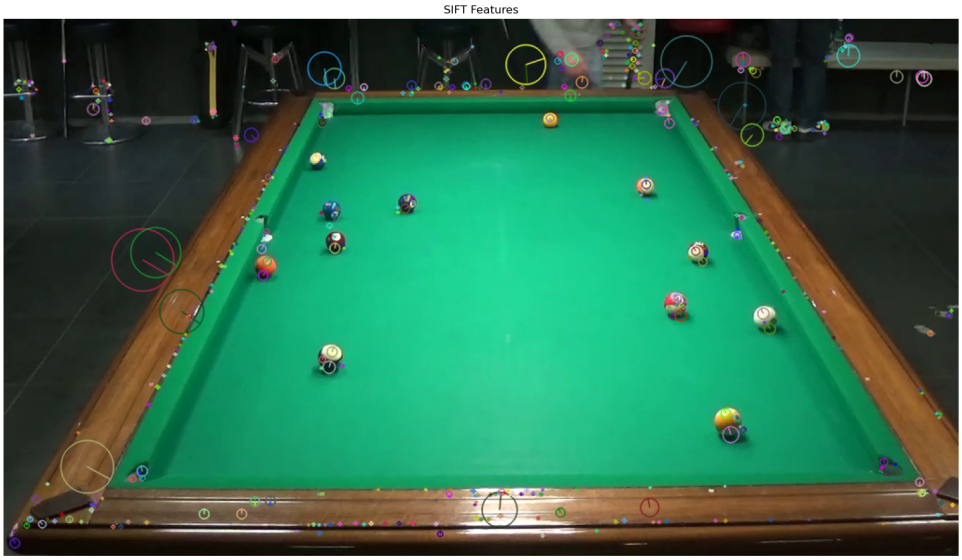



Figure 5: Sift features in a sample image

# 4 Balls Localization and Segmentation

To solve the ball localization and segmentation sub-tasks the following algorithm was implemented. The latter goes through the following three major steps:

1. Region proposals generation
2. False positive filtering
3. Bounding boxes refinement

Before starting with the actual algorithm presentation, a few consideration must be made. Exclusively for this project part, the used first video frame is the one contained in the given "frames" folder, which is different from the one taken directly from the video using OpenCV due to compression factors. This choice was made because better system performance relative to ball localization and segmentation were obtained using the former.

Furthermore, this section only accurately covers the Ball Localization task solution, showing how the final set of bounding boxes is obtained. The adopted solution for the Balls Segmentation task, simply consists in creating a mask obtained by fitting a circle inside the found bounding boxes. As it will explained, the boxes were constrained to be squared, so this operation was trivial.

## 4.1 Region proposals generation

This first step of the algorithm aims to generate regions of interest that will very likely contain a ball. The idea behind this step is to find circular regions in the image and create for each one of them a bounding box to contain it. To do this, the Hough Transform algorithm applied to circles was adopted (*HoughCircles()* in OpenCV).

Although, the algorithm wasn't applied directly on the grayscale version of the image since this would have lead to a great number of false positives and false negatives. The false positive origin from circles in the image coming from objects in the table different from the balls. False negatives instead are caused by some balls having almost exactly the same grayscale pixel intensity value of the table cloth, making `HoughCircles` unable to detect them. For this algorithm step, the aim is to nullify the number of false negatives, ensuring that every ball is covered by bounding box. The removal of false positives from the set of found boxes is covered by the second step of the algorithm.

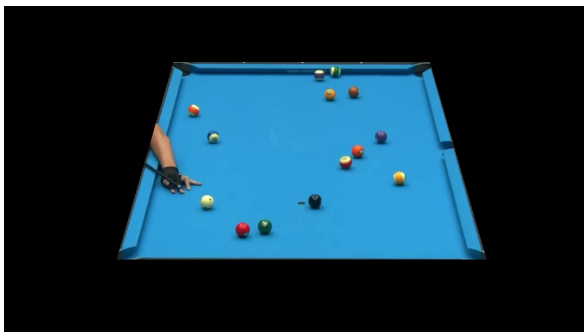
A pre-processing that highlights the balls in the table was needed in order to ease the detection via `HoughCircles`. The first try involved `K-means` clustering in the color space. Ideally, with parameter `K` big enough, table and balls should have been clustered in different sets given the differences in color. Although, given the great number of table cloth color variations caused by major illumination differences between table areas, a great number of found centroids belonged to the table instead of the balls, leading to many balls being clustered into table regions. Given this result, the second try involved associating each pixel with the closest center (Euclidean distance) in the color space given a pre-defined sets of centers. The latters were manually defined as ideal ball colors and estimated table cloth color (this estimation is explained later in this section). In this way it was possibile to correctly clusterize all table pixels but with the inconvenience of including some balls to the table cluster, since the estimated cloth color was closer to the pixel intensity that the respective ideal ball color. This problem is caused mostly by the fact that the illumination condition of the balls makes the ball much darker than it actually is. To counteract this issue, additional centers were added to the set exploiting the "value" channel of the HSV enconding the generate darker variations of the ball colors. Despite the higher quality of the newly found clustering, a few false negatives were still generated. The cause of this issue was still the extreme color similary between some balls and table cloth. More specifically, if a dark green ball is located in a poorly lit region of a green table, the ball became undetectable. Same reasoning for the blue case.

The best image pre-processing found with respect to the aim of this algorithm step, proved to be the following. Firstly, the mask obtained through the table segmentation was used to isolate the table in the image in order to constrain HoughCircles to detect circles only inside the table (Figure 6).

Secondly, to ease to detection of circles relative to balls in the table, the following local operation applied to the image pixels proved to be very effective. Since the table area is mostly empty, the pixel intensity of the table was evaluated as the mean intensity of non-zero value pixels in the masked image. After this, each non-zero pixel was set to the absolute value of the difference between the pixel intensity and the evaluated table one. In this way, all table pixels were set at a value close to zero and instead the balls at a higher intensity across at least one of the three channels due to differences from the table. This results in an alternative version of the image in which the table is darker and the transition between table and balls is much more marked (Figure 7).

After this, the image was converted to grayscale and HoughCircles was used on it with a general range of parameters (Method: `Hough_Gradient`, Radius range: 5-15, Distance between circles: `img.rows`/32, Accumulator threshold: 12). Working only with the BGR version of the image proved to be unsufficient to find all


(a) Original image



(b) Masked image

Figure 6: Removal of regions outside of the table one (game1_clip4)

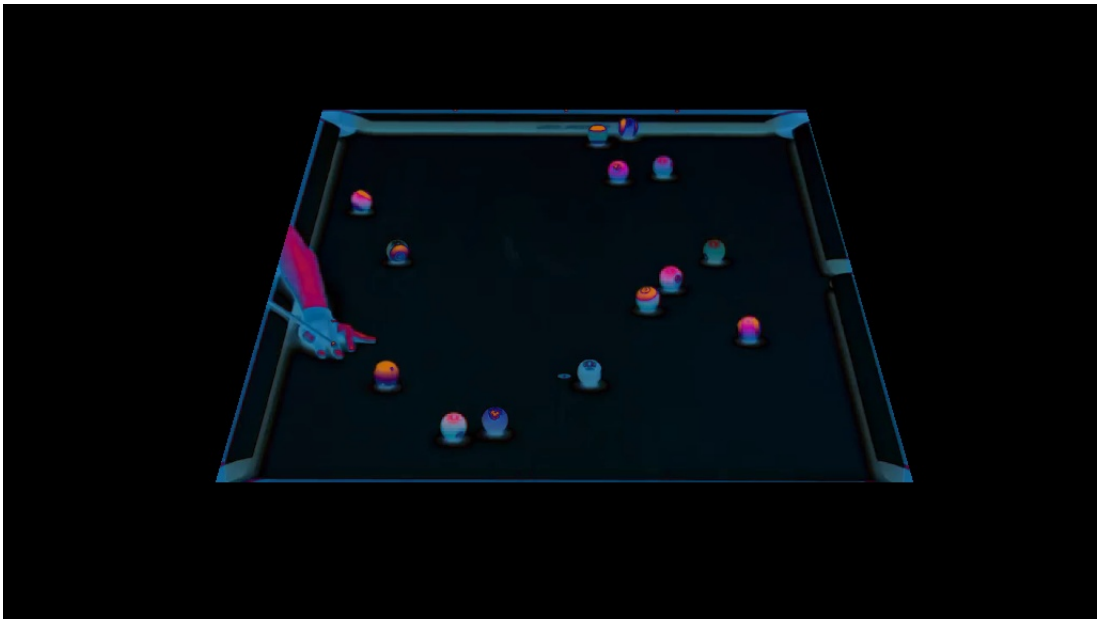


Figure 7: Custom local operation is applied to non-zero value pixels (game1_clip4)

balls because of the poor attention to illumination changes of the table color that made impossibile to detect some balls. To solve this issue the previous steps are performed a second time on the HSV version of the image in order to focus more on the colors brightness (value channel). Figure 8 illustrates an example of these two detections.

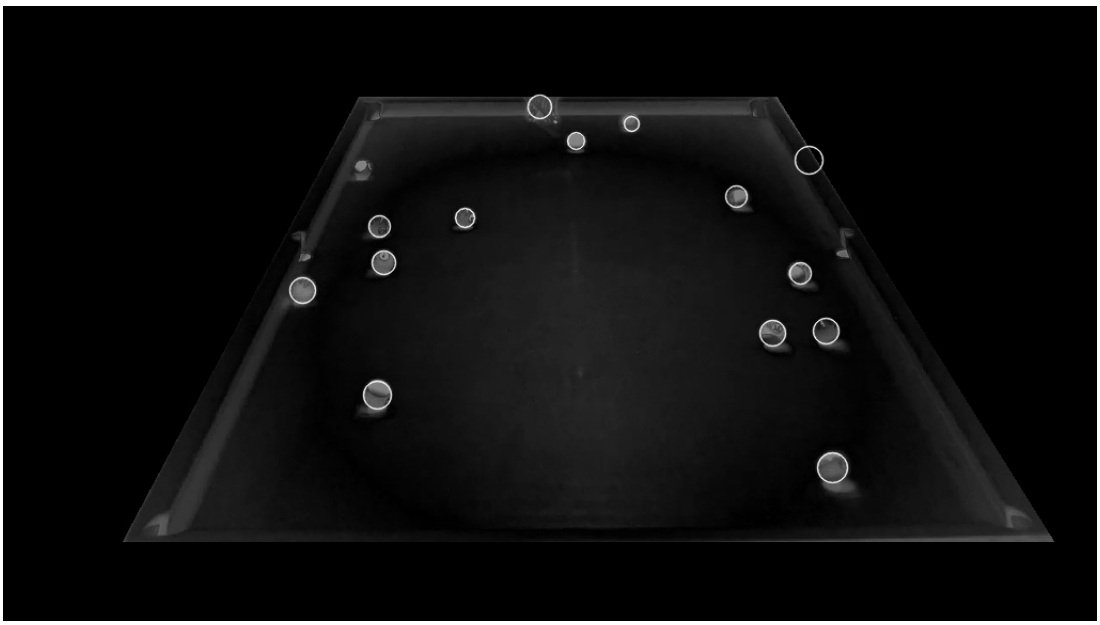
Many balls were detected in both iterations so a smart way to merge the detected circles was necessary. The final vector of circles is generated by adding all the ones found from the BGR version of the image, that proved to represent more accurately the ball contours, and every circles obtained from the HSV version that do not overlap with the ones in the first set. Then, an additional check to remove possible duplicate (overlapping) circles in the merged set is performed. Figure 9 shows the output of such merge on the previous example.

Circles were then converted into bounding boxes by creating a Rect object for each circle with abscissa and ordinate equal to the circle ones minus the circle radius, width and height equal to double the circle radius (diameter). Through this procedure, for each image a set of bounding boxes is generated, among which exactly one bounding box per ball is contained. Figure 10 shows an example of the results of the procedure upto this point.

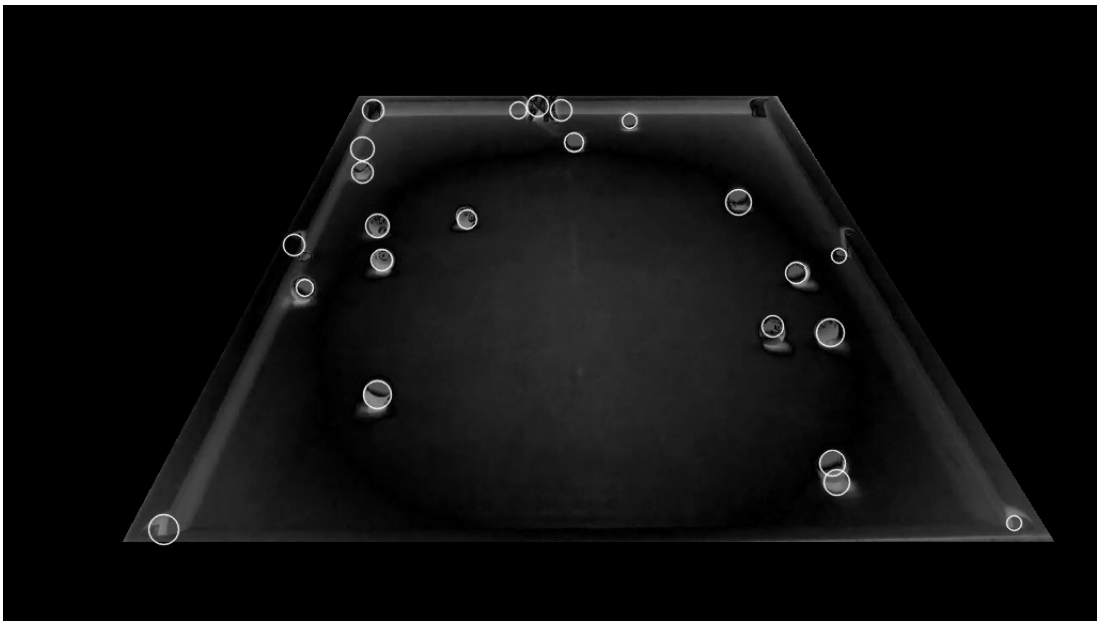
## 4.2 False positive filtering

As it can be noted from Figure 10, to be able to propose regions that cover the whole set of balls in the table, many false positive are generated. This happens since there are many circular regions in the table that do not come from any ball.

The first attempt at filtering such false positives resulted in the following algorithm. Given the fact that the balls are always inside the table, the surrounding area of the detected circles (identified as the circular ring of width double the circle diameter and center equal to the circle one) is supposed to have mean intensity similiar to the table cloth one and the mean intensity of the area contained in the circle to be different from it.



(a) starting with the BGR version



(b) starting with the HSV version

Figure 8: Circles detected via HoughCircles on the pre-processed image (game2_clip1)

By estimating the table color and setting proper distance thresholds in the color space, the implemented filter was able to correctly detect most false positives with a few exceptions. If the player has his arm particulary stretched on the table, circles detected on his hand were detected as true positives and in case of balls touching the table borders with width almost imperceptible due to perspective effects, the relative circles were detected as false positives. These errors were caused by the wrong assumptions initially made.

Because of this, other properties of false and true positives were necessary to be identified in order to create an efficient filter. The found ones are the following.

False positives can belong to either one of the following two classes: circles relative to holes and circles belonging to the player hand and arm if present inside the table region. Additionaly, as it will be later better explained, false positives relative to holes are isolated whereas ones relative to the player are always grouped together. Using these informations, the following algorithm to filter the false positives was created and implemented.

The filter is composed by two sub-filters placed in cascade to detect the false positives. The first one operates in the following manner. Given the facts that false positive relative to holes can be found near the longest table borders, and false positives generated by the player are never isolated (distant from other bounding boxes) since many circles around the arm and hand are always detected, the filter defines an area inside the projection of the

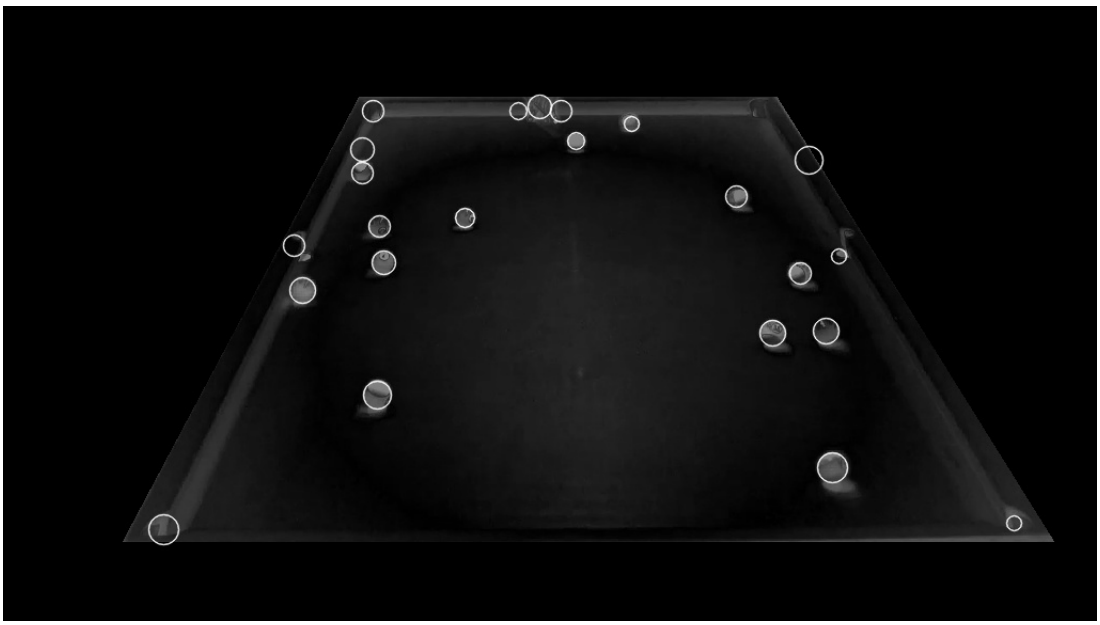


Figure 9: Final set of circles found in the image (game2_clip1)


Figure 10: Results of the region proposals generation step (game1_clip4)

table that excludes the longest table borders ("safe area") and filters all isolated (non overlapping) bounding boxes which projected center is inside of such area. To assure that a bounding box is truly isolated, just for this step all boxes width were expanded by a factor 1.5 so that close but naturally non-overlapping boxes will now overlap. Figure 11 shows an example of projected table image on which the "safe area" has been drawn. Note that such rectangle was set at 98% of the image width and 85% of its height. These dimensions proved to be effective for the purpose. Figure 12 instead shows the result of the bounding boxes expansion applied just for this first filter step. As it can be noticed, close boxes now overlap, having a better understanding of how much one is isolated.

The second and final filter exploits the following property. By dividing the image into connected components, one can notice how found regions belonging to holes, table borders and player arm always touch the borders of the table projection. This information was exploited to implement the following procedure. The image is converted in HSV format and the Canny algorithm is applied to the image in order to detect edges. Then, the projection of the image is obtained using the trasformation matrix returned by the *getTransformation()* function. This image is then given to the *connectedComponentsWithStats()* method (using Spaghetti labeling algorithm) to obtain the labeled regions along with useful statistic such the extremal top, bottom, left and right positions of the region. In this way, a new image is obtained from the transformed canny one by setting at

Figure 11: Projected table with "safe area" (game1_clip4)

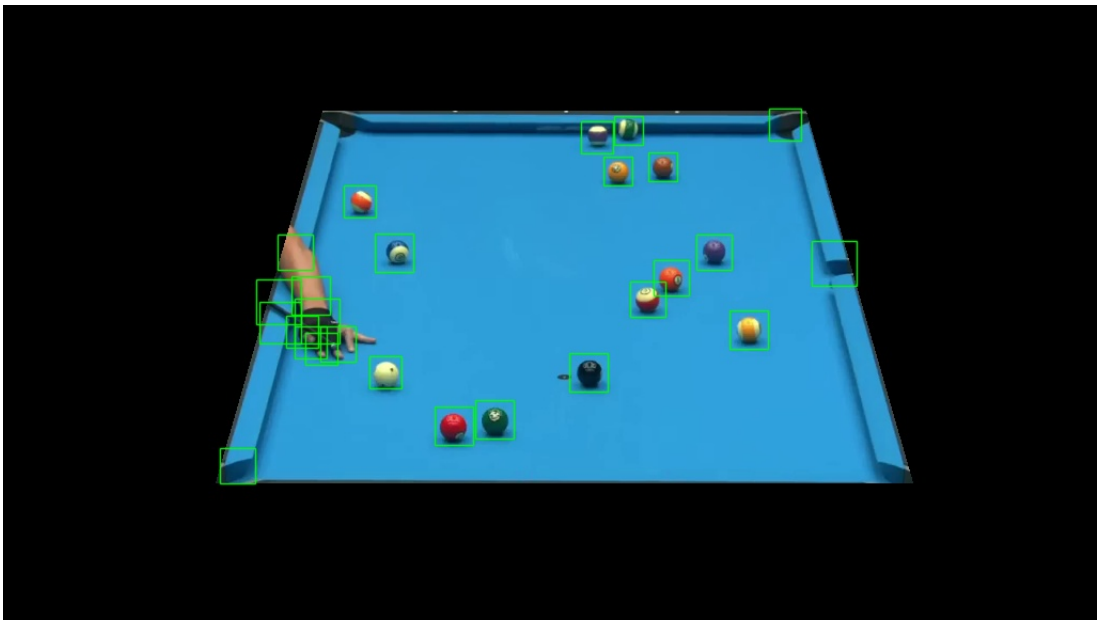


Figure 12: Expanded bounding boxes in the image (game1_clip4)

zero all pixels belonging to regions that touch the image borders or are labeled as background (label 0). At this point, the percentage of non-zero pixels in each patch of this last image defined by the bounding boxes is computed. Following our assumptions, all boxes relative to false positives will now contain a low amount of non-zero pixels. A threshold set at 24% proved to be effective. Figure 13 shows the image resulting from the processing done via *Canny* and *connectedComponentsWithStats* on which the previously identified bounding boxes are drawn.

Although, this method isn't perfect since it can produce false negatives. In fact, in case of balls close to the borders, their area counted as connected to a component of the table border, therefore the respective pixels in the image were set to zero. To fix this kind of errors, a second round of this procedure is performed but on a new version of the image obtained by running two iterations of the Opening morphological operator with an elliptic structuring element of size $(3, 2)$. The idea behind this additional step lies in the assumption that table border and balls regions if connected are weakly connected components. To ensure that no false positive is filtered as true in this last additional step, the percentages computed during the first iteration are saved into a vector and used during the second in the following manner. If the relative region is no longer touching the image border and so in the second run the percentage has greatly increased (at least +30%), then the bounding box is detected as true positive and added the set of true positives previously computed. Figure 14 shows an

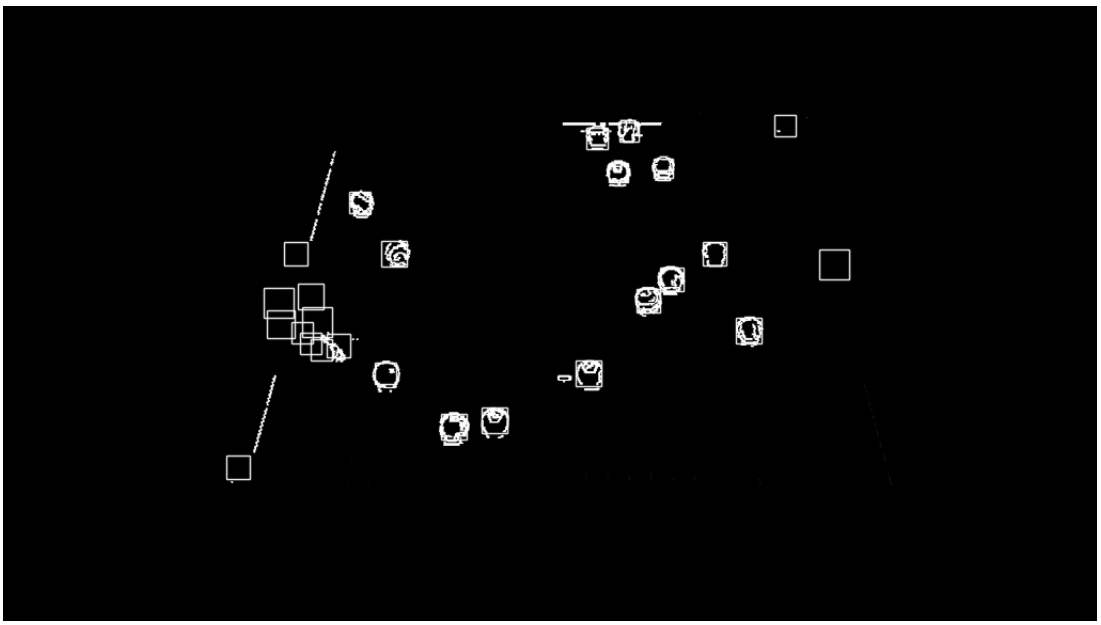


Figure 13: Results of the canny image processing with bounding boxes (game1_clip4)

example of this image processing in which when using the Opening operator a ball is disconnected from the table border and therefore becomes detectable as positive (see the top-left border).

Using this filter, only the regions (bounding boxes) relative to the balls were kept, ensuring a number of false positives and false negatives equal to zero across the whole provided dataset. Figure 15 shows an example of the final results of the complete false positives filtering step of the algorithm.

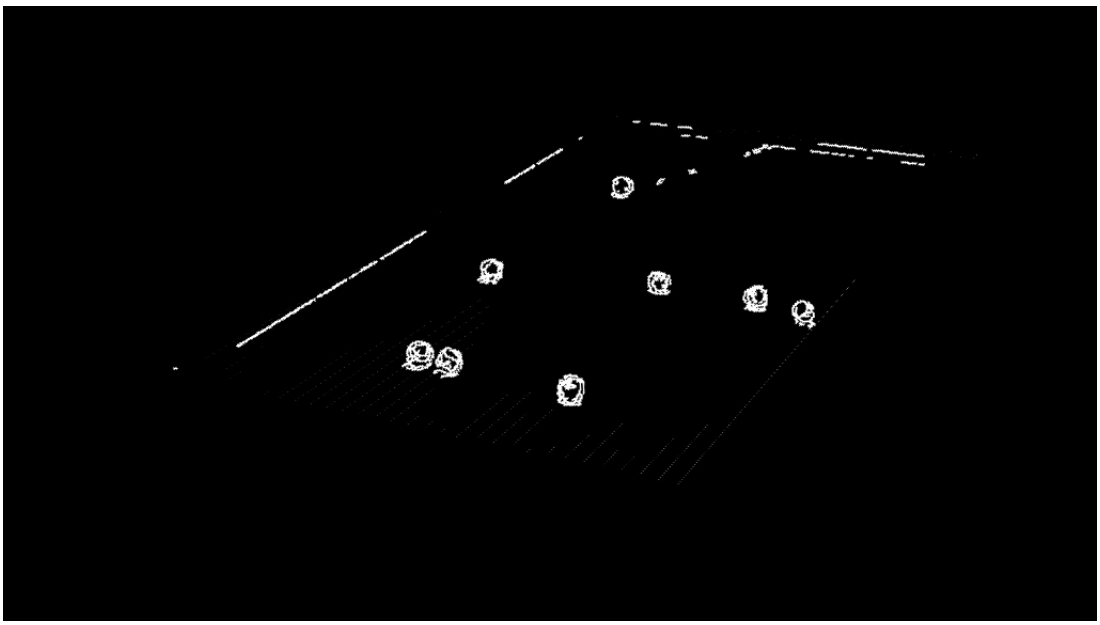
### 4.3 Bounding boxes refinement

As suggested by the previous section titles, the found bounding boxes can only be considered as region proposals, since the circles found through *HoughCircles* in the first step of the detection are not always accurate. This happens because the algorithm was used with a general range of parameters that is supposed to work on every possibile image. To obtain the final set of bounding boxes, such regions must be refined. To do so, we relied once again on *HoughCircles*. To ensure greater precision on the ball circle detection the following operations were carried. Since in some cases the obtained bounding boxes appeared to be sensibly bigger or smaller than the actual ball shape, to have some reference box dimensions the average bounding box width was computed. For each box center, a squared mask with dimensions equal to the mean box width multiplied by a scaling coefficient (1.1) is applied to the image in such point to focus on the surrounding area around of the ball. The scaling is necessary to ensure that the whole ball appears in the mask image, since the previously computed bounding boxes are often really tight.

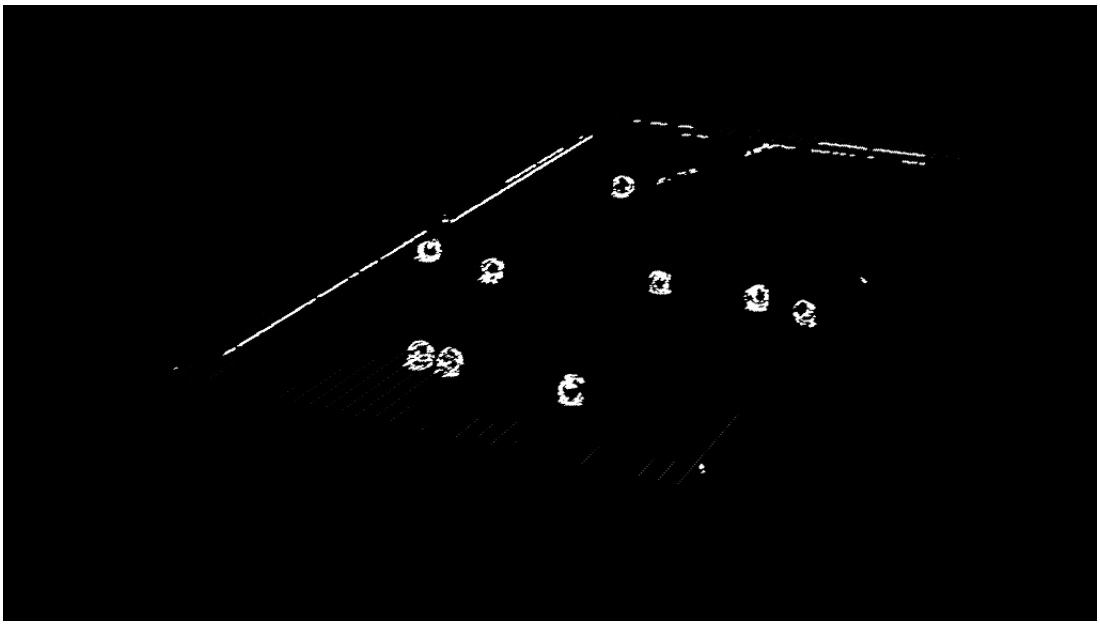
After doing so, *GrabCut* algorithm (*grabCut()* in OpenCV) is used to perform foreground extraction in order to obtain an approximate segmentation of the ball in the masked image in HSV format. Then, *HoughCircles* is used on the obtained image with the same set of parameters used before with the exception of the accumulator threshold now set at 10, and the radius range that now goes from a third to 70% of the mean bounding box width. The choice on this parameters was made to ensure the detection of circles with true ball dimensions in the specific image, which were estimated through the mean width computation.

This estimation is valid since at most a few boxes have dimensions sensibly different from the true ones. This information is exploited also to ensure that this refinement does not worsen the quality of already precise bounding boxes. More precisely, only the bounding boxes for which the detected circle has absolute value of difference between radius and previous box width halfs greater than 4 are updated. The update is performed by substituting the previous box with the found circle coverted to bounding box in the manner explained before. Figure 16 shows an example in which a ball was not precisely detected in the first step and the respective bounding box is refined by the procedure explained in this section.

It must be noted that sometimes the *GrabCut* algorithm fails to return an accurate ball segmentation since it sometimes considers table borders, holes and shadow of the ball as part of the latter. In such cases, *HoughCircles* won't find any circles and therefore the relative bounding box won't be updated. For the same reason, this algorithm is adopted only in this last step and it is not used as backbone of the whole ball localization and segmentation procedure.



(a) without Opening (untouched)



(b) with Opening (modified)

Figure 14: Differences in the results of the image processing (game3_clip1)

As briefly mentioned before, the found bounding boxes are very tight. This is a consequence of the chosen convertion method from found circles and boxes. This proved to be a nice property to have to solve the balls indentification task, since less background is included in the boxes. Although, the latters are not always perfect with respect to the localization task (bounding box precision), as some ball pixels may be left out. For these reasons the following decision has been made. Just for the balls indentification step the original (tight) bounding boxes were used, instead to initialize the vector of *Ball* structs an augmented version of them was used (same box center but with box width increased by 2).


Figure 15: Results of the false positives filtering step (game1_clip4)

(a) original box (b) GrabCut output (c) refined box

Figure 16: Example of bounding box refinement procedure (game1_clip2)

# 5 Balls Identification

The task of ball identification is as such: given a precise window inside an image that is assumed to contain a ball of the game of pool, we want to identify which type it belongs to (whether cueball, 8-ball, striped or solid color). We assume we don't need to identify the precise number of the ball outside of the number 8 one.

The main insight into this task is the fact that the most identifying element of a ball is its amount of white paint. A cueball is completely covered in it, a striped ball is covered about halfway and a solid or 8-ball has only a small white circle around its number. If we can find a way to reliably count the proportion of white pixels in the ball area and a corresponding expected range associated with each class the remaining task will only be one of distinguishing the 8-ball from the other solid color balls.

The main challenges of this sub-task are:

1. ensuring robustness to illumination changes
2. coping with specific ball color (some of them might be of a brightness approaching white)

A typical input image will look like this:

Figure 17: Grayscale image of a cueball

Figure 18: Grayscale image of a solid ball

My first approach was to attempt Otsu thresholding segmentation on a grayscale version of the ball image, via the built-in `THRESH_OTSU` option in the `threshold` function. Unfortunately this produced a very inconsistent separation between white and colored pixels which only sometimes corresponded to a real white region on the image, especially for images of solid colored balls.

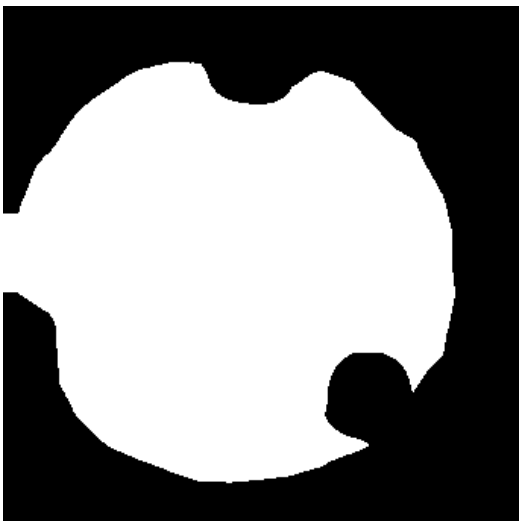


Figure 19: Otsu segmentation of a cueball

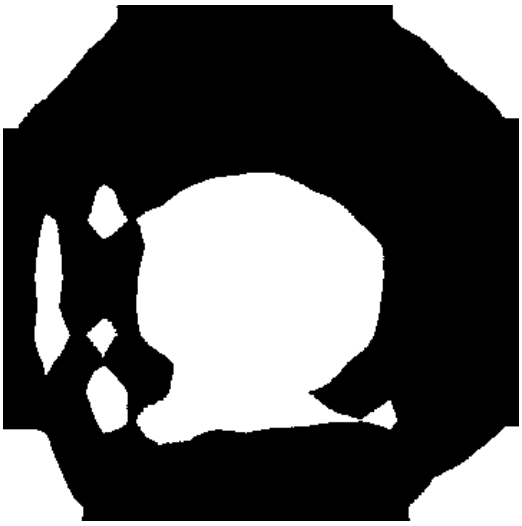


Figure 20: Failed otsu segmentation of a solid ball

My second attempt involved color-space clustering via `kmeans`. The underlying intuition is as such: a well-cropped window of a ball should contain roughly three important areas of uniform color: the table cloth and the colored and white parts on the surface of the ball. This means that we could theoretically employ a clustering algorithm on the color space of the pixels to separate these areas and count how many respective pixels appear per each type in the image. Unfortunately, this type of segmentatation performed even worse than the previous one due to illumination effects that cast pixels of one class into another.

A possible way to counteract this would be to factor in radial distance from the estimated ball center to the feature space of the clustering, which would at least help differentiate the pixels belonging to the ball from the ones belonging to the cloth. However, I didn't attempt this because I found a method that proved to be significantly more performand than the previous two:

The third approach consists in a simple thresholding and white pixel counting, with the difference that the image is first set to zero outside of the inscripted ellipse (the approximate location of the ball) to avoid false positives in the surrounding area and the grayscale intensity of the remaining pixels is stretched to occupy the full 0-255 range. A relative threshold (chosen on the basis of the example data) is then applied and the ball type is decided based on the percentage of white pixels remaining.

Unfortunately, this method produces some incorrect classifications and cannot distinguish between a solid color ball and an 8-ball. While the first problem would require a more sophisticate approach (possibly a machine learning one if sufficient performance is required), the second is solvable by building a higher level classification method: the function `classifyBalls` takes in the whole image, plus a vector of windows that correspond to the predicted balls locations. The function isolates the brightest and darkest windows, which get labeled respectively as the cueball and the 8-ball. The remaining windows are then handled singularly using the previously defined approach.

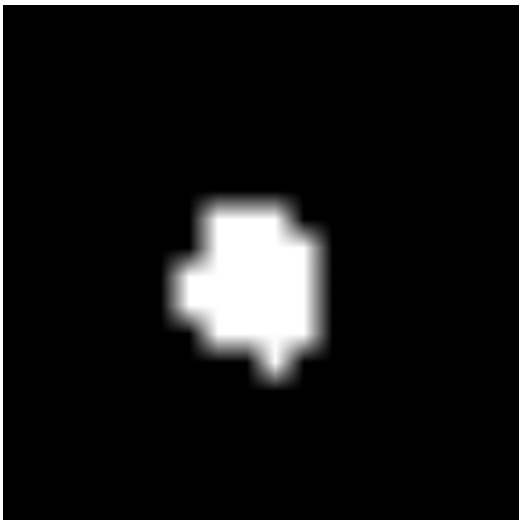


Figure 21: Correctly segmented white part of a solid ball

# 6 Balls Tracking

To perform the balls tracking along the video we decided not to run the ball detection algorithm on each frame since it would have been inefficient and could, even if rarely, generate false positives and negatives. Instead, we preferred using the *Tracker* classes provided by *OpenCV* that are supposed to provide greater stability and performance with respect to manual detection. Initially, we adopted the *MultiTracker* class but it didn't provide the level of control we needed. In fact, using such class it was not possibile to distinguish between tracked balls leading to problems such as not being able to know which ball is lost when the tracker fails. Therefore, we decided to use a vector of single trackers, each one following a specific ball along the video.

Since the are multiple types of trackers provided by *OpenCV*, a testing phase was needed to understand which one better suited our problem. Firstly, all trackers relying on Deep Learning were removed from the candidate list since their use is prohibited by the project guidelines. Some tests were run with the remaing ones, and the best tracker, albeit the one with the longest running time, proved to be *TrackerCSRT*.

Having chose the tracker type, the *TrackBalls* class was created. This class has as attributes a vector of *TrackerCSRT* objects, a vector of *Ball* structs and an integer constant which use will be later explained. In the following paragraphs the methods of the class and therefore the actual use of the tracker will be presented.

The class constructor takes as parameters a vector of *Ball* structs, which, to recall, are composed by a ball type and a bounding box (Rect object), and the initial frame. These objects are used to initialize the class attributes. From practical experience we have seen that using tight bounding boxes can sometimes lead to a tracker failing to recognize the tracked ball between frames. Therefore, before the attributes initialization, the boxes width of the given *Ball* structs are expanded by the constant previously mentioned.

To update the trackers and obtain the new state (bounding box position) of the balls, the *update* method was implemented. The latter receives as parameters the next video frame and a vector of integers passed by reference which is shared with the renderer. For each tracked ball, the relative tracker is updated using the given frame. If the tracker peforms such operation succesfully, then the bounding box of the ball is updated only if the intersection over union (IoU) between old and new box is under the threshold of 0.8 and the relative box index is added to a specific vector. This condition was set to avoid updates of balls which are still, resulting into a better final video rendering in which only moving balls actually move and the drawn trajectories are more straight. Instead, if the tracker fails, the ball box isn't updated and the relative index is added into a different vector.

If at the end of the update phase this last vector has positive size, ball recovery operations are carried and if unsuccessful, the relative ball removal from the attributes must be carried. Now these steps will be explained. In this case, the ball localization algorithm is run on the new video frame. If the number of detected balls equals the one of the tracked balls then the recovery of the lost balls can be done. Firstly, for each ball which tracking update was succesfully performed, the closest detected bounding box is assigned to it and removed from the list of the found ones. Then, the same procedure is carried on the lost balls with the remaining boxes. For each pair of lost ball and box, the relative tracker and *Ball* struct is re-initialized accordingly.

Instead, in the case in which the localization returns a different number of boxes, the tracking update is declared as failed and the lost balls are deleted. The *removeBalls* method was implemented to perform such operation. Using the given vector of indexes representing the list of balls to delete, the respective tracker and *Ball* struct are erased from the vector attributes.

Additionaly, a *getRealBalls* method was implemented to obtained the de-scaled (true size) bounding box versions of the balls using the same constant adopted in the constructor. This was useful to obtain the real-size boxes at the end of the tracking, in the last video frame, and therefore to be able to compute correctly the performances also in such case.

# 7 Map Rendering

The renderer for the minimap was developed in order for it to use the videoPlayer and tracker directly. In order to represent the balls correctly the perspective transformation matrix, that maps the balls to a predermined rectangle with top left side $(0, 0)$. We decided the dimension of such rectangle to be fixed to $500 \times 1000$ pixels, as the usual ratio for a billiard table is $1 : 2$. For the dimension of the balls we used $\frac{1}{70}$ the minimap columns. This was decided empirically on the videos provided, however such radius does not work with the same precision on all videos, as there are multiple standards for the billiard table dimensions.

For the rendering one buffer is kept for the trajectories, at each frame the line between the previous and current position is drawn as a black line. This buffer is then copied and the balls are added. To this the `table.png` image representing the borders is superimposed (the resizing is hard-coded as the dimensions are always the same). This needs the executable to be always in the same relative path as the image is contained in the data folder.

Another role of the render is to determine whether a ball went into a hole or not. We decided to give this role to this class (even though architecturally it wouldn't be correct) as it is the only one that has both knowledge of the trackers and of the actual positions of the balls on the table. The method used is to determine the closeness of the center of the ball bounding box to one of the centers. This verification is done only to balls that have moved in the current frame. The threshold used is 20 pixels, determined empirically. This method is however not perfect, as the tracker may behave unexpectedly when a ball enters a hole, as the object it is tracking is disappearing. This also is problematic when the hole is occluded by the table border itself, as the tracking may fail to be precise. Another problem is when a ball is particularly close to a hole but not inside, as it might be falsely put into a hole. If the renderer detects a ball entering a hole it communicates this to the tracker which drops the ball. We also decided to generate a struct for the `table.png` file, in order for the executable to not be dependent on its position in the filesystem.

# 8 System Performance Evaluation

The project specifications require two distinct performance measures to judge the quality of the segmentation produced by the program:

1. Mean intersection over union (mIoU) for the class segmentation;
2. Mean average precision (mAP) for ball localization.

## 8.1 Intersection over Union (IoU)

This is the more straightforward of the two measures. Given any kind of prediction-truth pairs that represent areas (be it a binary mask or a 2d rectangle), the IoU is simply the ratio between the area of the intersection of the two surfaces and their union.

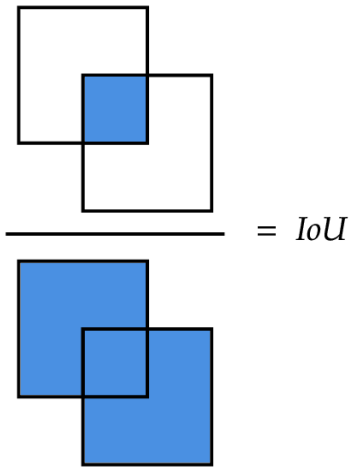



Figure 22: Graphical representation of IoU

Intuitively, this can be used to represent how well a singular prediction in an object detection task fits on the corresponding ground truth. OpenCV provides easy ways to intersect both binary masks and `cv::Rect` objects, the first by pixel-wise logical AND and the second by a dedicated function. Union area is instead directly calculated by adding the areas of the two surfaces and subtracting the area of their intersection.

## 8.2 Average precision

For each ball class for which we make predictions, we are going to calculate a metric known as average precision (AP), which is the area under the precision-recall curve of the class, per-image, estimated using the technique described in the linked article on the project specifications. For each predicted bounding box, we calculate its maximum IoU with the set of ground truths, then we consider it a true positive if the value is more than a certain threshold (0.5 in this case). We keep a tally of true vs false positives and use it to calculate the partial precision and recall values, which we plot in a sparse graph (here represented as a `std::map` object).

The area under this graph gives us information on how well the localization is performing (1 denotes a perfect classifier, 0 a completely wrong one). Since we've build a sparse graph, the approximation method PASCAL VOC is used to calculate the underlying area: 11 equally spaced points are taken in the recall interval (from 0 to 1). For each of those, the corresponding precision value is defined as the highest one among all the points to the right of it.

The estimation is the average of the 11 values found this way. The "mean" in Mean Average Precision is simply the average of all the AP scores calculated for each ball class.

In must be noted that in some clips we might have empty classes (for example, an endgame position where all the striped balls have been scored). The AP for such cases is defined as 1.

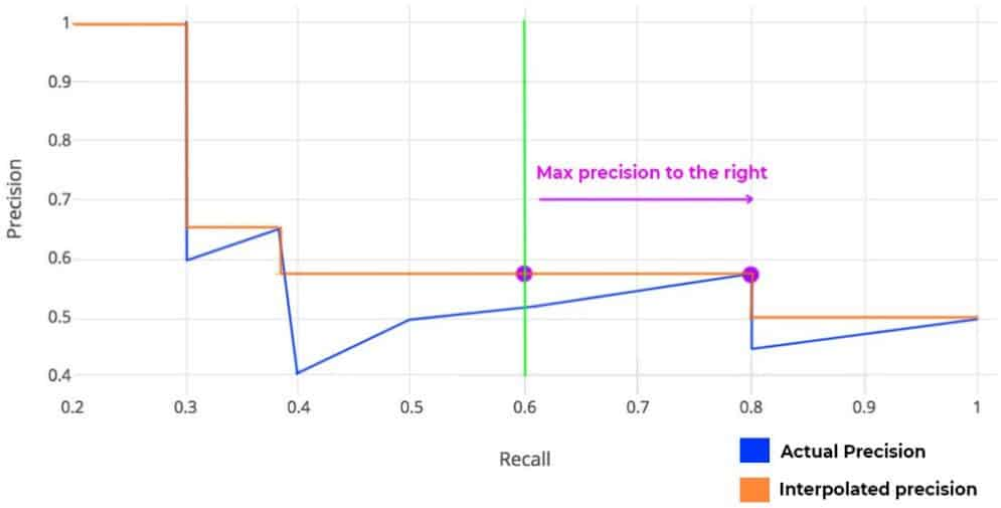
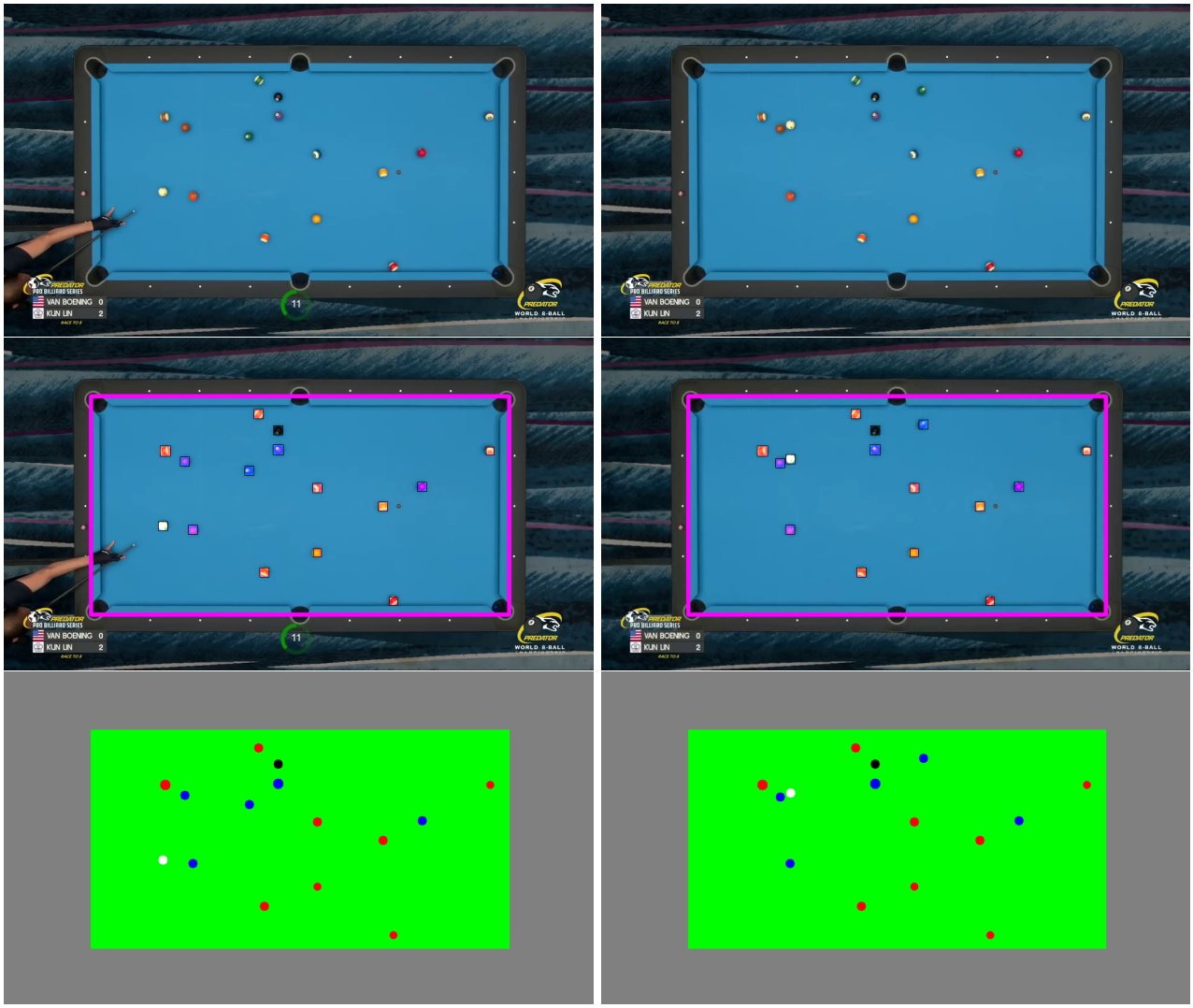
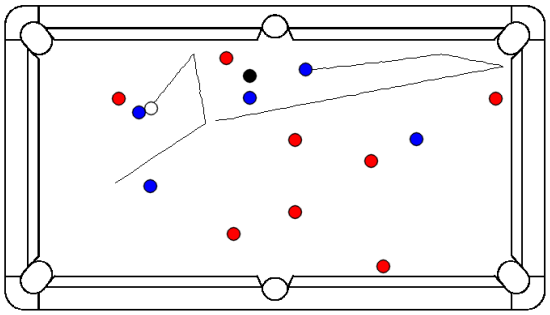



Figure 23: Example of underlying area estimation of a sparse graph

# 9 Results

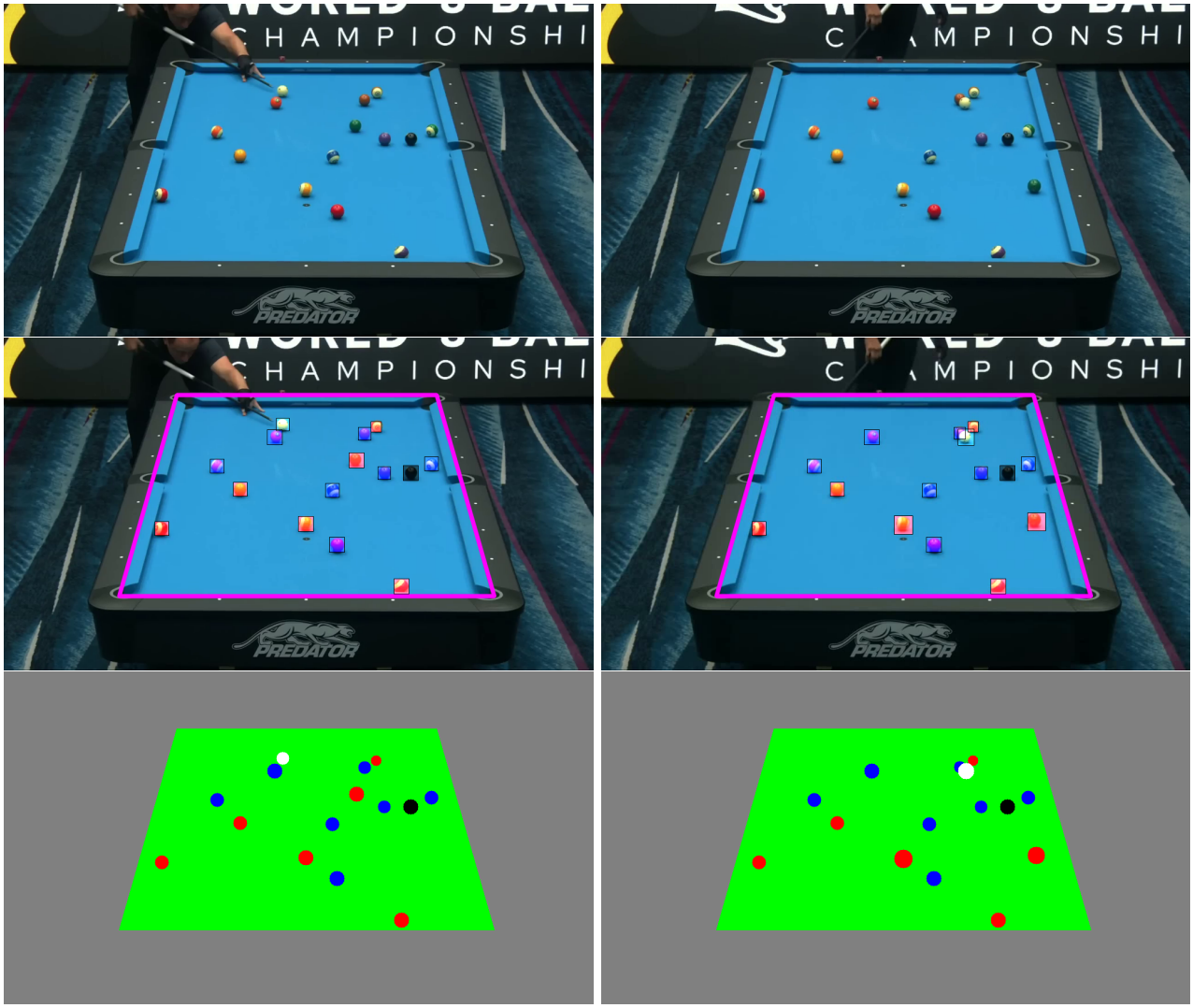
In this section the full output of the developed system is presented with the exception of the rendered videos. For each game event first and last video frames, the generated table and balls segmentation, along with their classification expressed as a color choice, are presented in form of contours and bounding boxes in the first case, and masks in the second one. In addition, the generated map in its state at the last video frame is reported. Along with these images, the requested performance metrics were computed and reported in specific tables. Additionally, in order to highlight the influence of each sub-task solution performance on the final outcome, also the intermediate results of the metrics computation are reported.

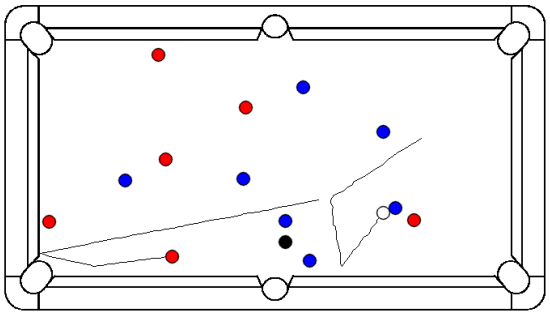




game1_clip1

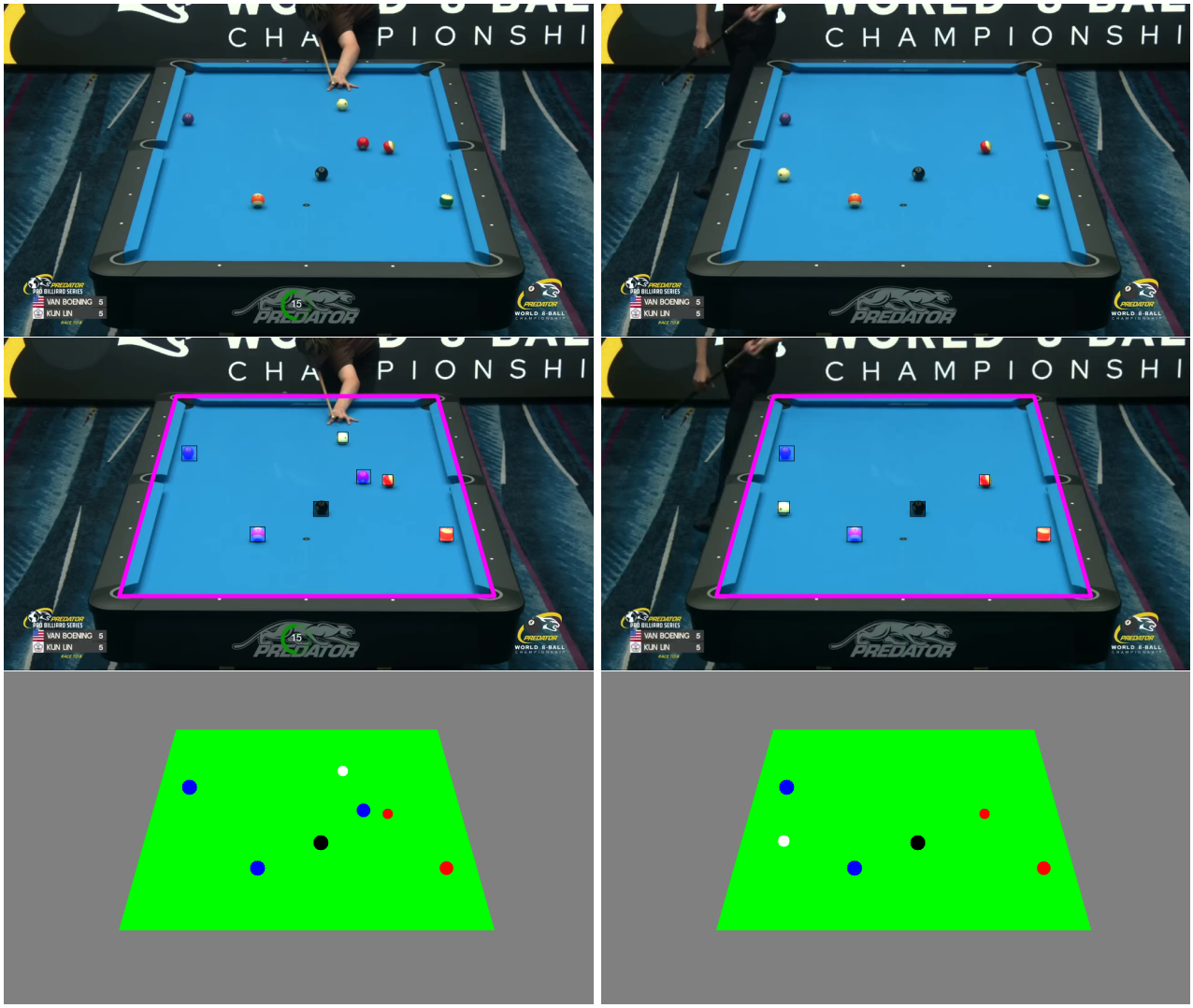
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.731613 | 0.761944 |
| IoU table | 0.959636 | 0.969971 |
| IoU background | 0.969238 | 0.977193 |
| IoU segmentation cue | 0.70082 | 0.827586 |
| IoU segmentation eight | 0.858491 | 0.863208 |
| IoU segmentation solid | 0.653061 | 0.644719 |
| IoU segmentation striped | 0.638473 | 0.693391 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.818182 | 0.818182 |
| AP striped | 1 | 1 |
| mIOU | 0.79662 | 0.829345 |
| mAP | 0.954545 | 0.954545 |

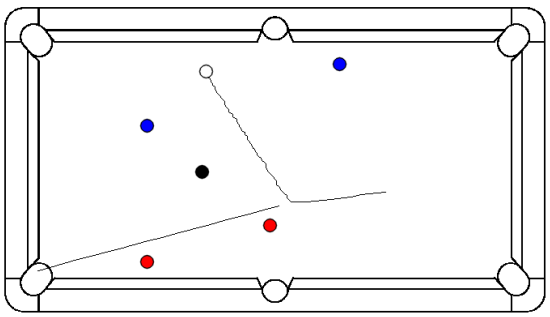




game1_clip2

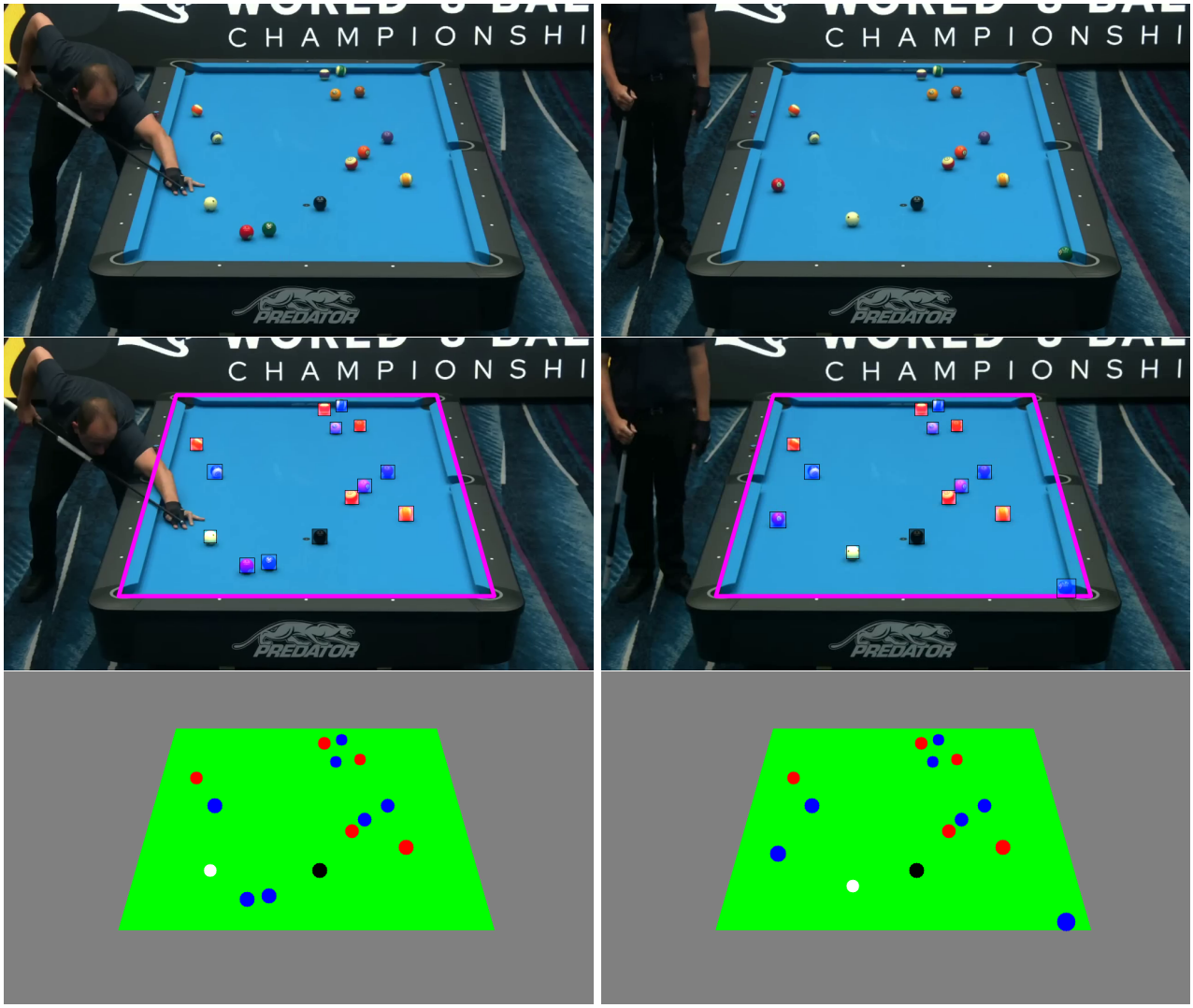
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.813507 | 0.779797 |
| IoU table | 0.945216 | 0.95625 |
| IoU background | 0.978064 | 0.983876 |
| IoU segmentation cue | 0.824607 | 0.625 |
| IoU segmentation eight | 0.765595 | 0.765595 |
| IoU segmentation solid | 0.379028 | 0.337329 |
| IoU segmentation striped | 0.385635 | 0.373494 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.597403 | 0.597403 |
| AP striped | 0.454545 | 0.454545 |
| mIOU | 0.713024 | 0.673591 |
| mAP | 0.762987 | 0.762987 |

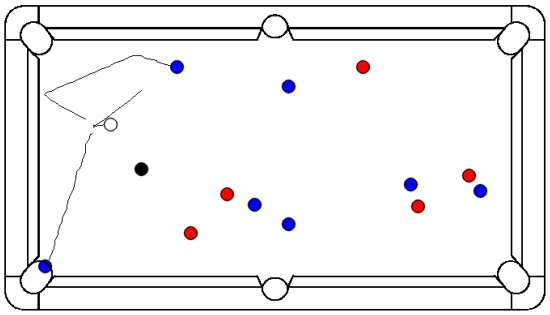




game1_clip3

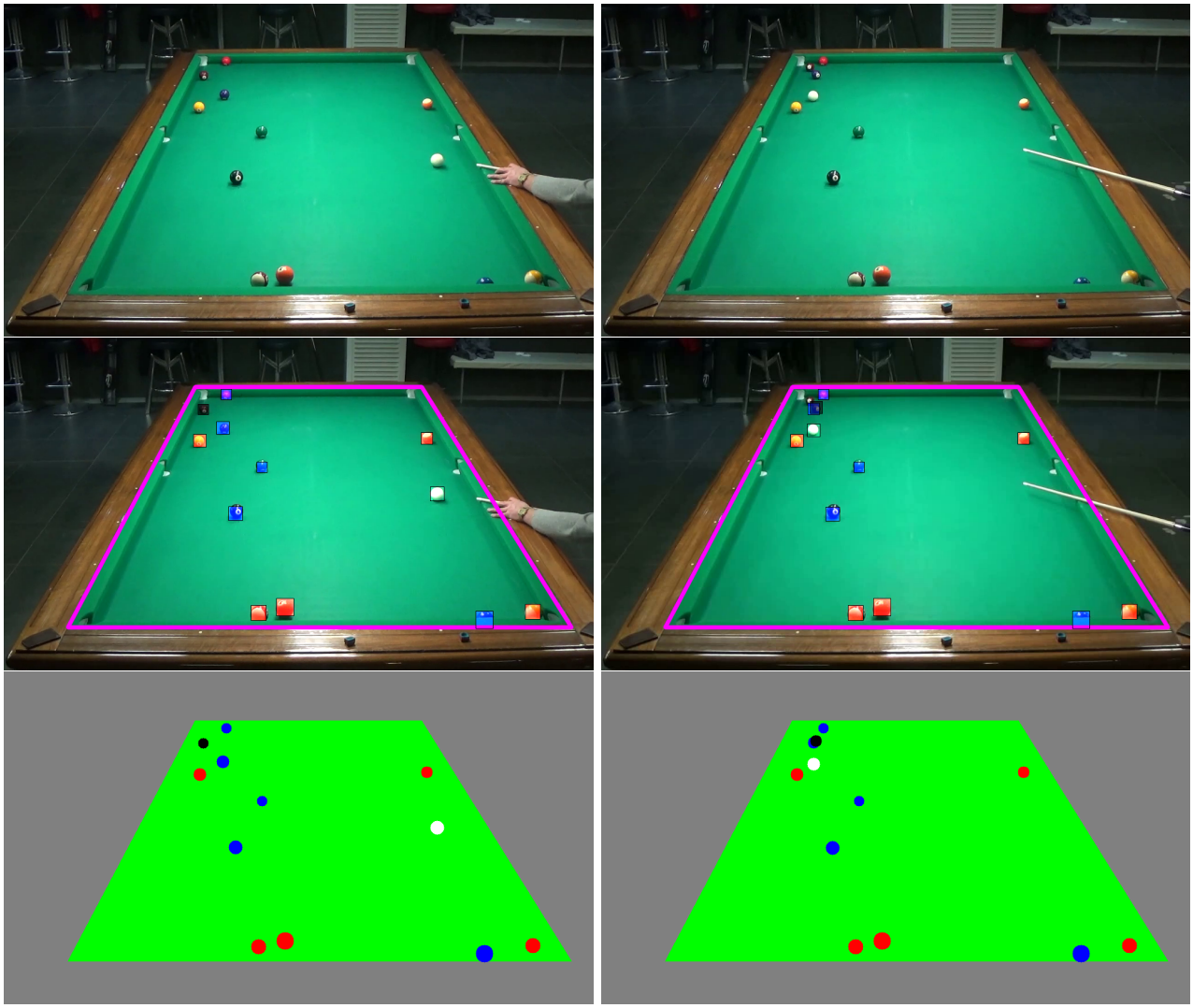
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.760456 | 0.750082 |
| IoU table | 0.954954 | 0.959823 |
| IoU background | 0.980998 | 0.983055 |
| IoU segmentation cue | 0.688202 | 0.706935 |
| IoU segmentation eight | 0.737213 | 0.738053 |
| IoU segmentation solid | 0.48728 | 0.35206 |
| IoU segmentation striped | 0.469625 | 0.465529 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.666667 | 0.5 |
| AP striped | 0.636364 | 0.636364 |
| mIOU | 0.719712 | 0.700909 |
| mAP | 0.825758 | 0.784091 |

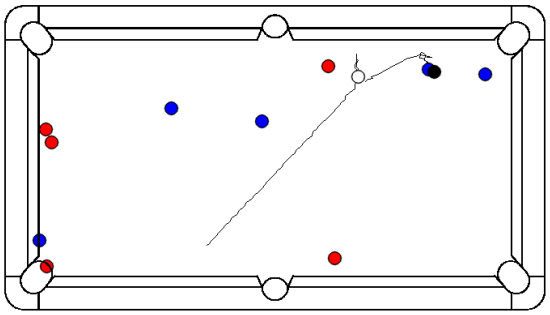




game1_clip4

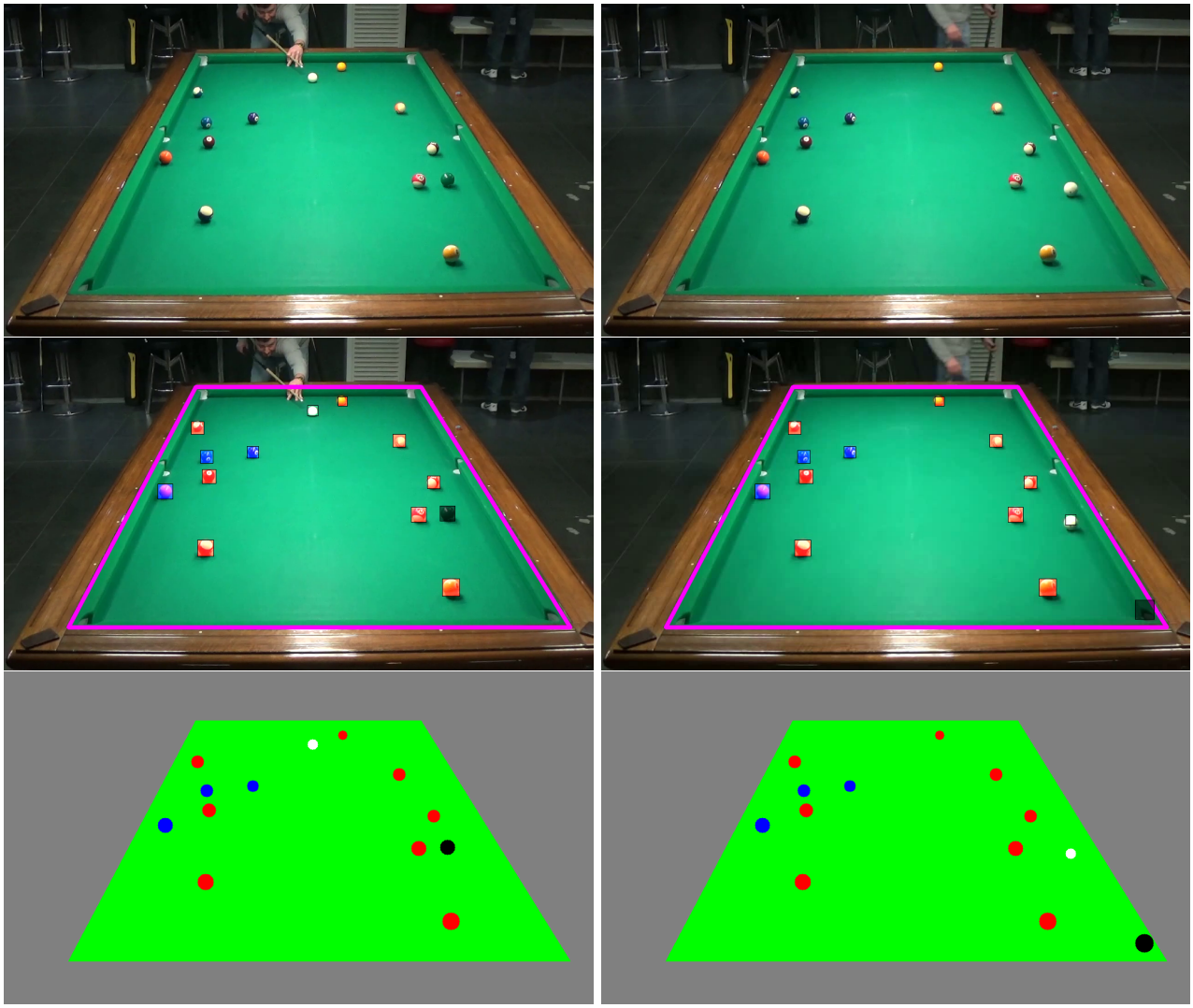
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.816838 | 0.791127 |
| IoU table | 0.921635 | 0.946427 |
| IoU background | 0.967132 | 0.977882 |
| IoU segmentation cue | 0.712121 | 0.711321 |
| IoU segmentation eight | 0.860759 | 0.877514 |
| IoU segmentation solid | 0.570841 | 0.530105 |
| IoU segmentation striped | 0.515582 | 0.531087 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.772727 | 0.772727 |
| AP striped | 0.636364 | 0.636364 |
| mIOU | 0.758012 | 0.762389 |
| mAP | 0.852273 | 0.852273 |

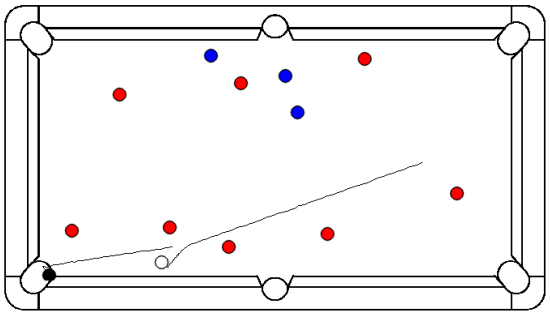




game2_clip1

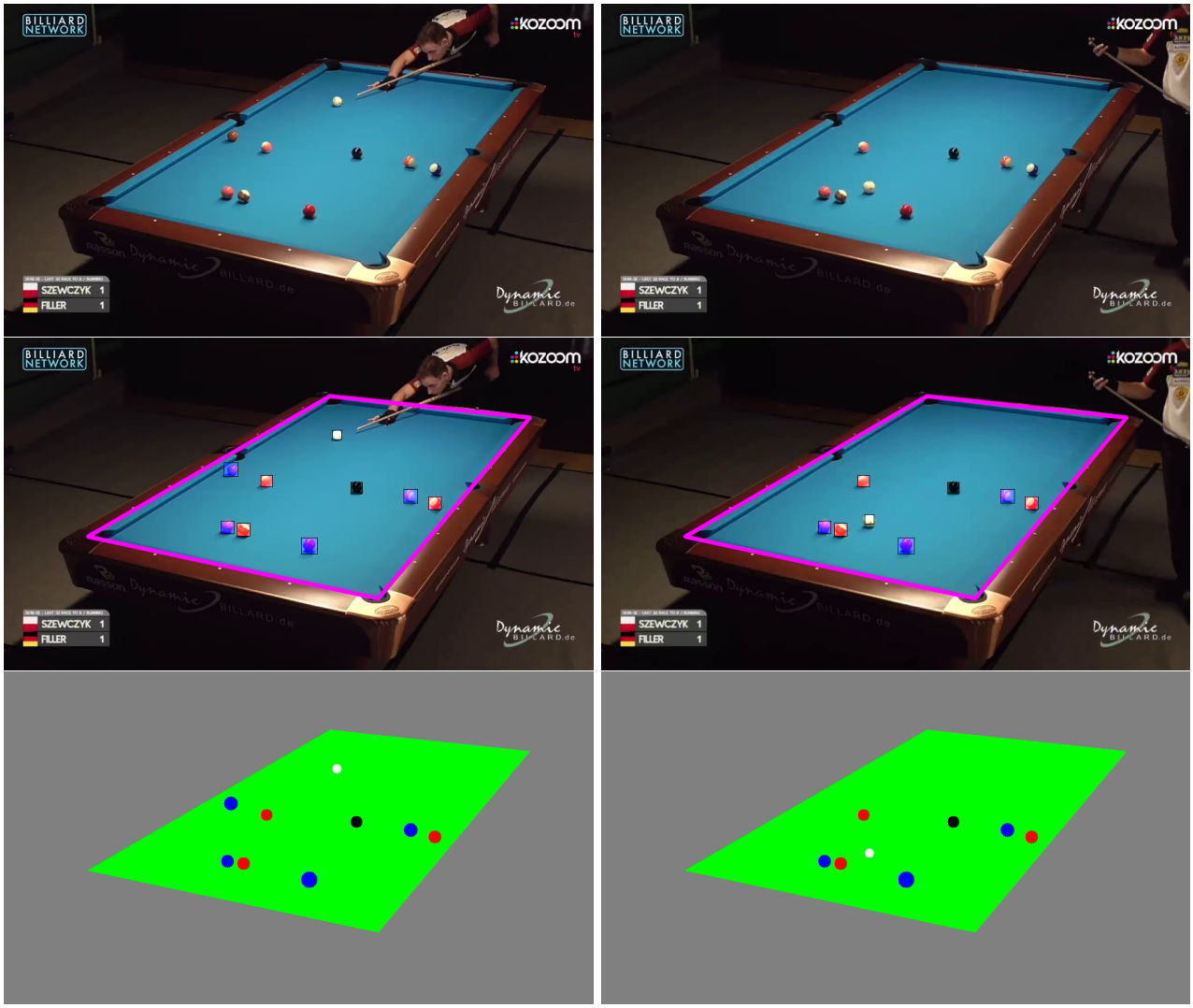
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 0.833333 | 0.833333 |
| IOU localization (alone) | 0.689992 | 0.651972 |
| IoU table | 0.974774 | 0.968989 |
| IoU background | 0.985378 | 0.981357 |
| IoU segmentation cue | 0.95207 | 0.777188 |
| IoU segmentation eight | 0 | 0 |
| IoU segmentation solid | 0.25741 | 0.251919 |
| IoU segmentation striped | 0.455304 | 0.467236 |
| AP cue | 1 | 1 |
| AP eight | 0 | 0 |
| AP solid | 0.272727 | 0.272727 |
| AP striped | 0.6 | 0.6 |
| mIOU | 0.604156 | 0.574448 |
| mAP | 0.468182 | 0.468182 |

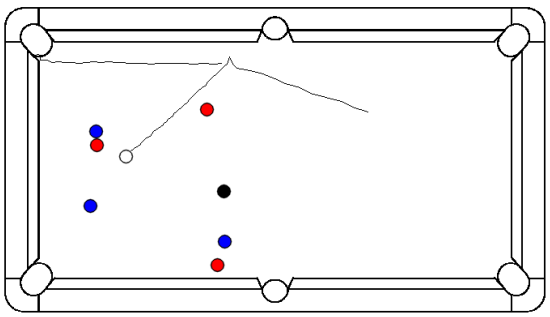




game2_clip2

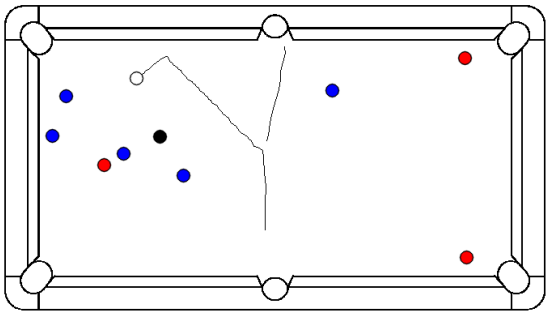
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 0.909091 |
| IOU localization (alone) | 0.843287 | 0.706597 |
| IoU table | 0.974328 | 0.974187 |
| IoU background | 0.982659 | 0.984523 |
| IoU segmentation cue | 0.795848 | 0.443015 |
| IoU segmentation eight | 0 | 0 |
| IoU segmentation solid | 0.566568 | 0.43809 |
| IoU segmentation striped | 0.699113 | 0.69289 |
| AP cue | 1 | 0 |
| AP eight | 0 | 0 |
| AP solid | 0.636364 | 0.545455 |
| AP striped | 0.924242 | 0.924242 |
| mIOU | 0.669753 | 0.588784 |
| mAP | 0.640152 | 0.367424 |



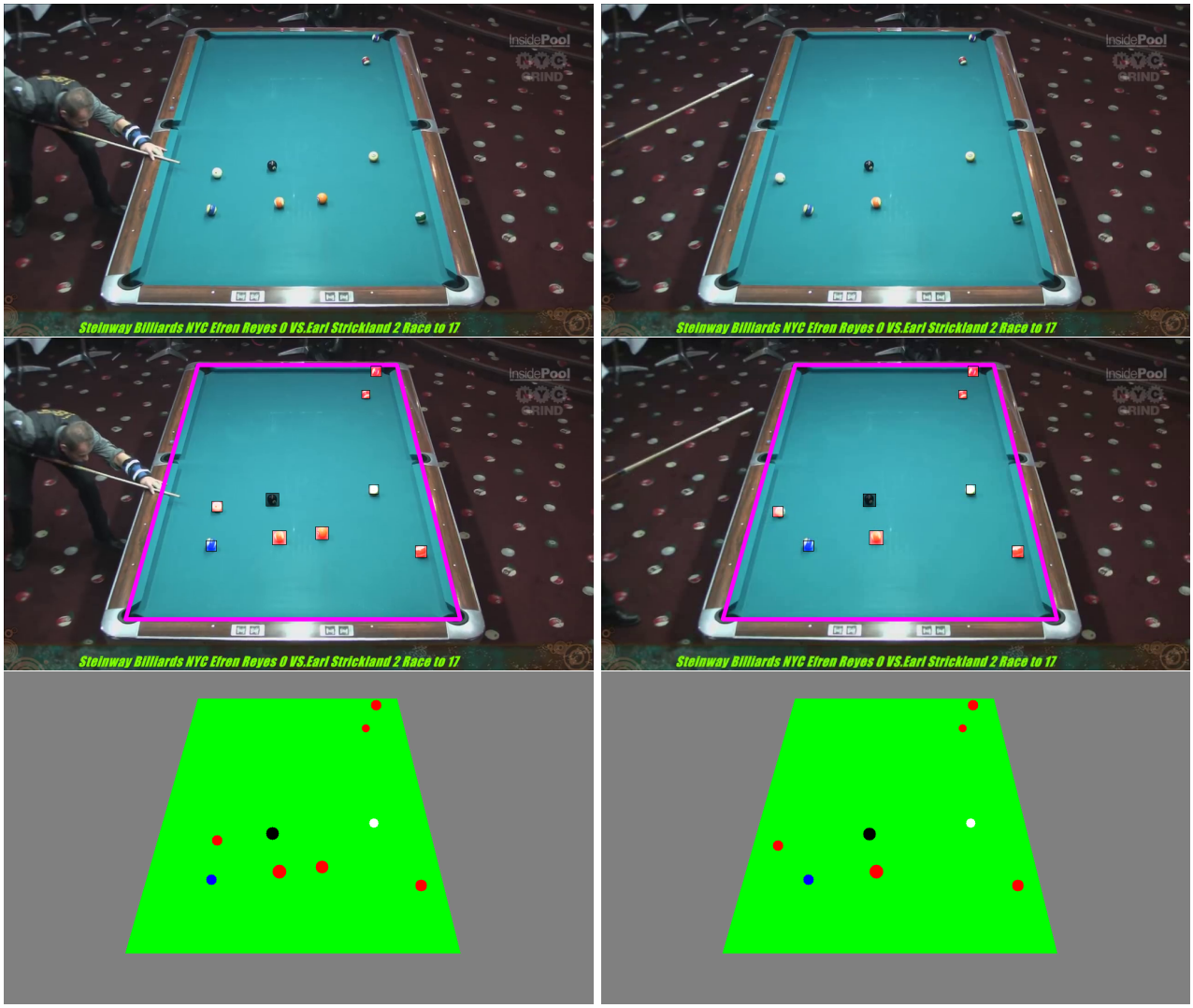
game3_clip1

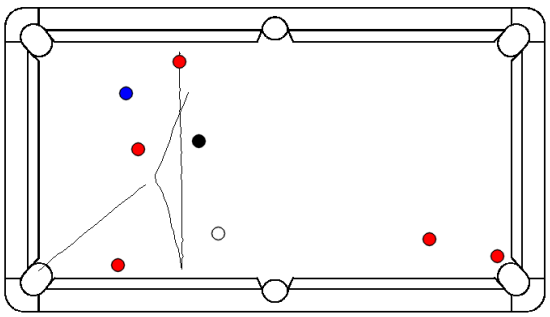
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 0.806818 |
| IOU localization (alone) | 0.784004 | 0.771163 |
| IoU table | 0.956168 | 0.969899 |
| IoU background | 0.988177 | 0.992291 |
| IoU segmentation cue | 0.736842 | 0.464623 |
| IoU segmentation eight | 0.750663 | 0.754011 |
| IoU segmentation solid | 0.594297 | 0.539229 |
| IoU segmentation striped | 0.623494 | 0.633212 |
| AP cue | 1 | 0 |
| AP eight | 1 | 1 |
| AP solid | 1 | 1 |
| AP striped | 0.727273 | 0.727273 |
| mIOU | 0.77494 | 0.725544 |
| mAP | 0.931818 | 0.681818 |

game3_clip2

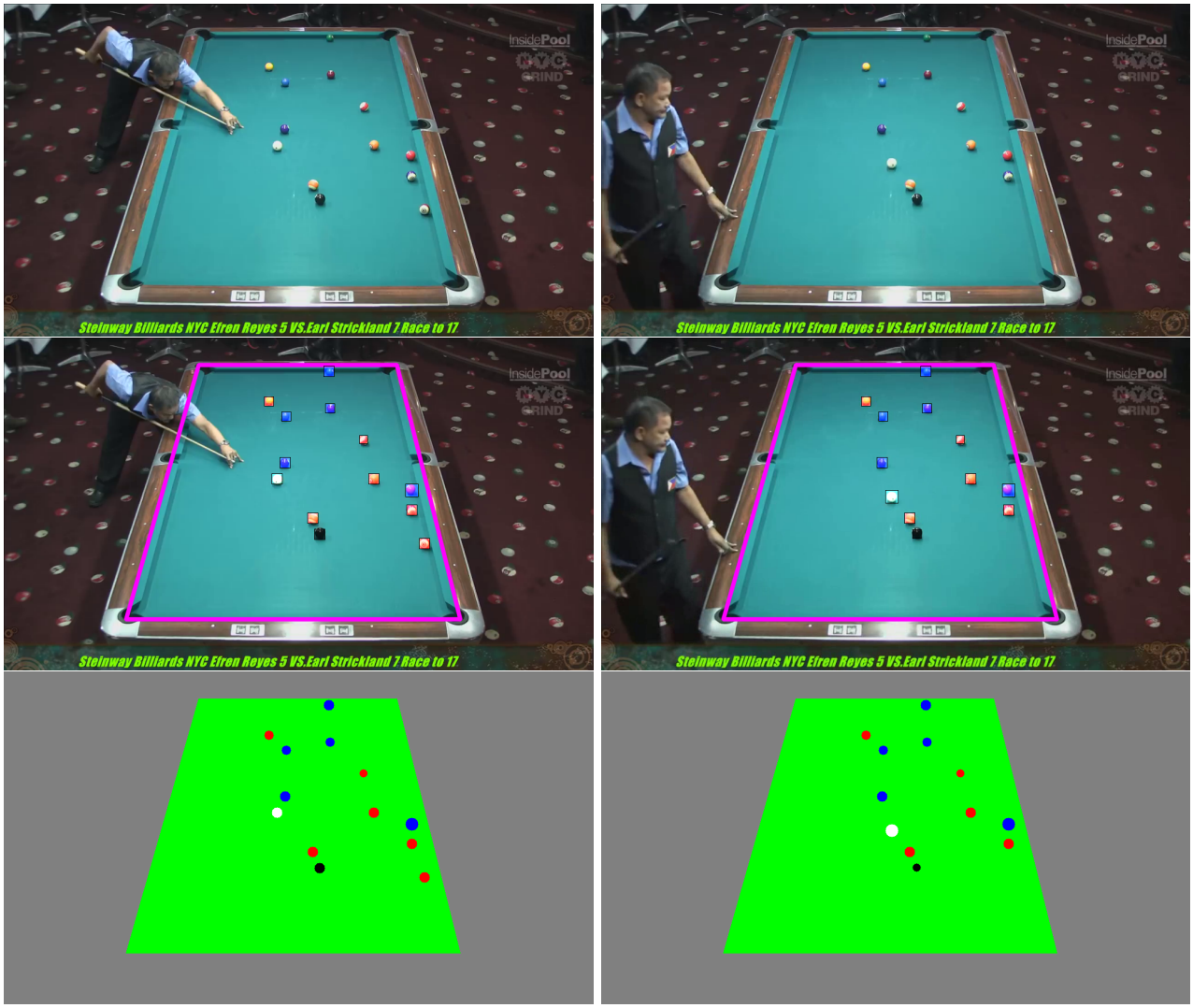
| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.771727 | 0.792737 |
| IoU table | 0.946953 | 0.949756 |
| IoU background | 0.985853 | 0.986171 |
| IoU segmentation cue | 0.792627 | 0.855362 |
| IoU segmentation eight | 0.861413 | 0.859079 |
| IoU segmentation solid | 0.22206 | 0.151909 |
| IoU segmentation striped | 0.162825 | 0.162678 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.290909 | 0.272727 |
| AP striped | 0.242424 | 0.242424 |
| mIOU | 0.661955 | 0.660826 |
| mAP | 0.633333 | 0.628788 |

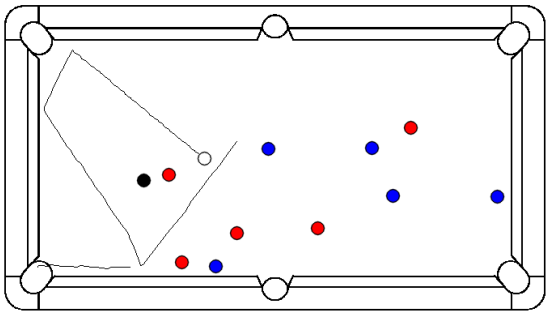


game4_clip1

| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.748102 | 0.698225 |
| IoU table | 0.96674 | 0.975676 |
| IoU background | 0.984729 | 0.989402 |
| IoU segmentation cue | 0 | 0 |
| IoU segmentation eight | 0.748011 | 0.722955 |
| IoU segmentation solid | 0 | 0 |
| IoU segmentation striped | 0.380842 | 0.439496 |
| AP cue | 0 | 0 |
| AP eight | 1 | 1 |
| AP solid | 0 | 0 |
| AP striped | 0.484848 | 0.545455 |
| mIOU | 0.513387 | 0.521255 |
| mAP | 0.371212 | 0.386364 |



game4_clip2

| Metric | First | Last |
|---|---|---|
| AP localization (alone) | 1 | 1 |
| IOU localization (alone) | 0.747309 | 0.711855 |
| IoU table | 0.954033 | 0.977441 |
| IoU background | 0.978839 | 0.990904 |
| IoU segmentation cue | 0.771044 | 0.734748 |
| IoU segmentation eight | 0.71988 | 0.468553 |
| IoU segmentation solid | 0.519915 | 0.526399 |
| IoU segmentation striped | 0.53949 | 0.46773 |
| AP cue | 1 | 1 |
| AP eight | 1 | 1 |
| AP solid | 0.727273 | 0.727273 |
| AP striped | 0.727273 | 0.642424 |
| mIOU | 0.7472 | 0.694296 |
| mAP | 0.863636 | 0.842424 |

# 10 Project Structure and Execution Instructions

The project has been developed using the CMake build system and it is organized with the following structure:

- `include` directory, contains the header source code files;
- `src` directory, contains the implementation source code files;
- `data` directory, contains the background image used to generate the 2D map;
- `samples.zip` file, contains the system input, outputs, predictions and ground truths;
- `output_videos.zip` file, contains the produced video renderings;
- `CMakeLists.txt`, contains CMake instructions to organize the build of the project.

The `sample` directory is structured into `samplei` subfolders, where `i` refers to the sample index (starting from zero). As mentioned before, each subfolder contains inputs, outputs, prediction and ground truths relative to a specific sample.
Furthermore, folders `include` and `src` are further divided into:

- `segment` directory, contains the files for table and balls segmentatation;
- `recognition` directory, contains the files for balls identification and side recognition;
- `tracking` directory, contains the files for balls tracking;
- `rendering` directory, contains the files for the map rendering;
- `utils` directory, contains additional files that implement various utilities.

Additionaly, `src` folder contains:

- `output_main.cpp` file, implements the full video rendering and masks generation;
- `performance_main.cpp`, implements the computation of all performance metrics across all videos.

After successfully build the project using CMake, the two *mains* can be executed under the following conditions. `output_main.cpp` requires the following three parameters as argument:

1. path to the `.mp4` file corresponding to the video;
2. path to the `.png` file corresponding to the first video frame;
3. path to the folder in which the program output shall be saved.

`performance_main.cpp` instead, requires the following parameters:

1. path to the `samples` folder;
2. total number of samples in the folder.

`sample` is provided in `.zip` format for convenience. Therefore, in order to run the respective executable successfully, the `.zip` file must be extracted.

# 11 Experimental Setup

The system has been tested under the following environment:

- C++ 11.4.0;
- CMake 3.22.1;
- OpenCV 4.5.4;
- Linux 6.9.3;
- Intel Core i7-10510U, 16GB RAM.

# 12 Workload Split

In this Section, how the workload was split between the authors is shown along with the amount of working hours spent by each one.

The following table associates each identified task with an author. Each author was responsibile for the creation and development of the owned tasks solution. In some cases, minor contributions in terms of ideas were given by authors different than the task owner.

| **Task** | **Owner** | **Minor contributors** |
|---|---|---|
| Table Segmentation | Giovanni Artico | |
| Table Sides Recognition | Giovanni Artico | |
| Balls Localization and Segmentation | Mattia Toffolon | Giovanni Artico |
| Balls Identification | Marco Giacomin | |
| Balls Tracking | Mattia Toffolon | Giovanni Artico |
| Map Rendering | Giovanni Artico | Mattia Toffolon |
| System Performance Evaluation | Marco Giacomin | Giovanni Artico |

Table 1: Table representing the workload split among the authors

For what concerns the production of the project report, each author was responsibile for the writing of the sections relative to the owned tasks.

The writing of the remaing sections was split in the following manner. Sections: Introduction, Project structure, Experimental setup, Workload split, Conclusions and Future work, were written by Mattia Toffolon. Section: Results, was written by Giovanni Artico.

The amount of working hours spent on the project (code and report) by each author is represented in the following table.

| **Author** | **Working Hours** |
|---|---|
| Giovanni Artico | 110 |
| Marco Giacomin | 28 |
| Mattia Toffolon | 120 |

Table 2: Table representing the amount of working hours spent by each author

# 13 Conclusions and Future Work

The developed system has achieved satisfactory performance in segmentatation of table, balls, background and balls localization, especially considering that it is exploiting only traditional computer vision techniques without the use of Machine Learning or Deep Learning. The ball identification step instead, showed major flaws in multiple cases, bringing down the complexive system performance. Despite this, the generated 2D maps proved to represent fairly well the game events on each frame, leading to the creation of a modified version of the given videos in which useful informations are shown.

To improve the system, every sub-task solution could be refined. In particular, one way to better the performance could be the development of a method to generate accurate bounding boxes of the tracked objects at any time. In fact, our system always returns boxes in their starting shape which can change during the video due to ball movement combined with perspective effects, leading to possibile false positives (IoU between true and predicted below 50% while the system still counts it as a ball).

Other than that, another possibile way in which the project solution could be further improved is the adaptation to live-streaming billiard games. In this case the program should be modified in order to process the video frames in real-time. Additionaly, in order not to lose any frame, strict time constraints on the images processing shall be added.