8BallPool video analysis

Michele Sprocatti[1], Alberto Pasqualetto[2], Michela Schibuola[3]

July 21, 2024

Department of Information Engineering, University of Padua
Via Gradenigo 6, 35131 Padova, Italy
{michele.sprocatti[1], alberto.pasqualetto.2[2], michela.schibuola[3]}@studenti.unipd.it

Introduction

This is the report of the final project for the Computer Vision course, which goal is to develop a computer vision system for analyzing video footage of various "Eight Ball" billiard game events.

How the work has been split

The three group members collaborated by dividing the work. Each member was responsible for producing specific files, all of which were annotated with their respective author.

Split of the work

The source code reflects a division of labor across three key areas:

1. Detection and segmentation of balls and the playing table: this area, along with the main program functionality, was handled by Michele.
2. Metrics calculation and tracking: responsibility for this area fell to Alberto.
3. Transformation code and mini-map management: this area was overseen by Michela.

Working hour per member

The approximate number of working hours for each member of the group are:

Michele: 50 hours

Michela: 50 hours

Alberto: 50 hours

Elements of our project

Executables

The program contains 4 different executables:

- `8BallPool`: the main executable that, given a video file, it processes it and creates the output video with the superimposed minimap.
- `TestAllClip`: it is the executable used to test the detection and segmentation in the first and last frame of all videos through AP and IoU by comparing them with the ground truth.
- `ShowSegmentationColored`: is an helper which has been used to show the ground truth of the segmentation of a particular frame using human-readable colors and it was also used as a test for the code that computes the metrics because it computes the performance of the ground truth on itself.
- `ComputePerformance`: is used to compute the performance across the dataset so the mAP and the mIoU.

Table detection

A mask-based approach was implemented for table detection, exploiting the fact that in billiard games' footage, the table should be always located in the middle of the frame; this assumption is confirmed by all the videos in the dataset. The mask was generated by identifying the most common color in the image's central columns; the color is represented by a range in the Hue component (with respect to HSV color space). Building upon this initial step, Michele exploits the Canny edge detector and OpenCV's HoughLinesP function. Then the function analyzes the found intersections and merges nearby points, ensuring the consistent identification of 4 corner points corresponding to the table's corners in the processed dataset.

Table segmentation

In order to isolate the table with high precision, Michele employed a combination on two different binary masks through a voting system. The involved masks are:

Color-based mask Created by identifying pixels corresponding to the table's color range. This mask is very robust on shadows and reflections since it defined through the Hue component of the HSV color space, which is invariant to brightness.

k-means clustering mask Generated by applying the k-means algorithm on the image to separate the table from the background. Useful when the table color range is too broad to be captured by the color-based mask.

These two masks re combined through a voting system, which classifies as table pixels classified as such by at least one of the two masks gaining the best of both approaches. Finally the isolated area is limited to the pixels inside the polygon defined by the previously detected corners.

Balls detection

To detect balls, Michele proposed a multi-step preprocessing approach. Initially, the table region is isolated using an approach similar to the segmentation described before. Then the corners area is removed to prevent Hough Circle transform to find them as false positives. Subsequently k-means clustering was applied to the image with k=5 (the number of balls type plus the playing field). The resulting clusterized `Mat` is converted to gray-scale to be used as `HoughCircle` input. The gray-scale output colors were selected to be as different from each other once the color space is changed.

Circle parameters, such as radius and center color, were analyzed to identify potential ball regions. By calculating the mean radius of in-table circles with center not selected by the color mask, a radius range was established. Circles within this radius range were considered for further analysis.

Ball classification involved creating a circular mask, computing the gray-scale histogram, and excluding background pixels from the values of the histogram. Peak values in the histogram were used to differentiate between striped and solid balls, while HSV color space analysis is used to distinguish white and black balls.

After finding the balls, the team identified an optimization opportunity. Since there's only one white ball and one black ball, Michele implemented non-maxima suppression for white and black balls independently, in order to improve performance.

The result of the detection process is then used to segment the balls.

Attempt to find the ball radius relative to the distance and perspective of the camera with respect to the table

To try to increase the performance of the ball detection, Michela had the idea of to computing an interval of values for the ball radius relative to the pixels of the image and the position of the camera with respect to the table; this would have been used in the `HoughCircles`. For that purpose, she wrote the method `radiusInterval`.

This method starts by computing the mean radius value by using a proportion between the diagonal of the table in pixels and the dimensions of the diagonal of the table and the balls in centimeters (such dimensions in centimeters are not standard, so an average value has been selected).

$$mean\_radius = \frac{ball\_radius\_cm}{table\_diagonal\_cm} \times longest\_diagonal\_px \tag{1}$$

Then, a percentage of the slope between the camera direction and the table has been computed, by using one of the angles ($< 90°$) that the table detection provides; this angle is compared with

the $\frac{\pi}{2}$ angle, and a value between 0 and 1 is computed:

$$percentage\_slope = 1 - \frac{minimum\_angle}{\pi} \tag{2}$$

- If the value is 1, then the camera is parallel to the table;
- If the value is 0, then the camera is perpendicular to the table;
- Otherwise it is a value between 0 and 1, which indicates the percentage of slope between the camera and the table; for example, if the value is 0.5, then the camera is about 45° from the table.

To compute the final interval, the minimum and maximum values are computed by subtracting and incrementing a value, which increases with the percentage of slope (more the slope, more the variance) by multiplying the percentage of slope with the mean radius previously computed, and a precision value is added due to some other variables in the images.

$$min\_radius = mean\_radius - mean\_radius \times percentage\_slope - precision \tag{3}$$

$$max\_radius = mean\_radius + mean\_radius \times percentage\_slope + precision \tag{4}$$

Tracking

The tracking has been performed exploiting the `TrackerCSRT` class of the OpenCV library.

In our application a new class `BilliardTracker` has been implemented, which is responsible for the tracking of all the balls on the billiard table while adding the lower level implementation of OpenCV `TrackerCSRT`. The class works by creating a new `TrackerCSRT` object for each ball to be tracked.

During the tracking process, at every frame, the tracker updates the global `Ball` vector with the new position of the ball through a pointer which will be used to draw the ball and its trace on the minimap. The position of the ball is only updated if the updated bounding box is moved by at least a certain threshold, which is set to the 70% of the IoU between the previous and the new bounding box. This is done to avoid some false positives that may occur during the tracking which lead to small wiggles in the ball position, even if they are steady. The IoU threshold value is a trade off between the wiggle reduction and the time frequency sampling of the ball position, used to draw the trace of the ball on the minimap.

If the ball is no more visible since it has been scored, then the relative tracking is stopped.

Based on our experiments the tracking is performed on a 10 pixels padded version of the ball's bounding box since the tracker gains much more performances in its ability to track the ball, even without occlusions.

This tracking implementation reveals to be very robust and accurate on all the frames of all videos of the dataset, but it is also very slow; this is the bottleneck of the application, since the tracking is performed on every frame of the video, and the tracking of each ball is performed independently from the others.

Minimap creation

To create the minimap are needed:

- An image that contains an empty billiard table and some information about it;
- The position of the balls in the current and previous frames;
- A transformation matrix that computes the position of the balls in the minimap.

Empty minimap image

As a first step, an image of an empty billiard table has been selected, and its corner positions and dimensions have been stored in constant variables by testing different values. In particular Alberto had the idea of converting the image into a byte array and inserting it in a header file through ImageMagick (https://imagemagick.org/). This step has been performed with the aim of creating a self-contained executable without the need of the png image dependency. The byte array is then used to create a `Mat` object through the `imdecode` function.

Computation of the transformation matrix

The `computeTransformation` method has been written to compute the transformation matrix, which allows for the computation of the positions of the balls in the 2d table represented in the minimap. To do that, a relationship between the corners of the table in the frame and the corners of the table in the minimap has been found by the OpenCV `getPerspectiveTransform()` method, which "calculates a perspective transform from 4 pairs of the corresponding points" and returns a transformation matrix. At first, it is supposed that the corners are given in clockwise order and that the first corner is followed by a long table edge. To check this information, `checkHorizontalTable` has been written.

Check if the corners are in the required order

The `checkHorizontalTable` method checks, using the image in input and the corners of the table in that image, if the corners are oriented such that the first corner is followed by a long table edge. To check this information, the "percentage of table" with respect to the pocket in a rectangle placed in the center of the edge (with dimensions proportional to the real table and pocket dimensions) has been computed for all the edges. This computation has been done in the table image previously transformed and cropped to the table dimensions; in this way, the center between two corners corresponds to the real one (otherwise, if the table has some perspective effect, the center between the two corners may not correspond to the real one). Then, the edges have been ordered by using this percentile. To understand how the corners were oriented, three cases have been considered:

- If the edges with "more pocket" are opposite edges, then they are the longest edges; This happens, for example, in Figure 2.

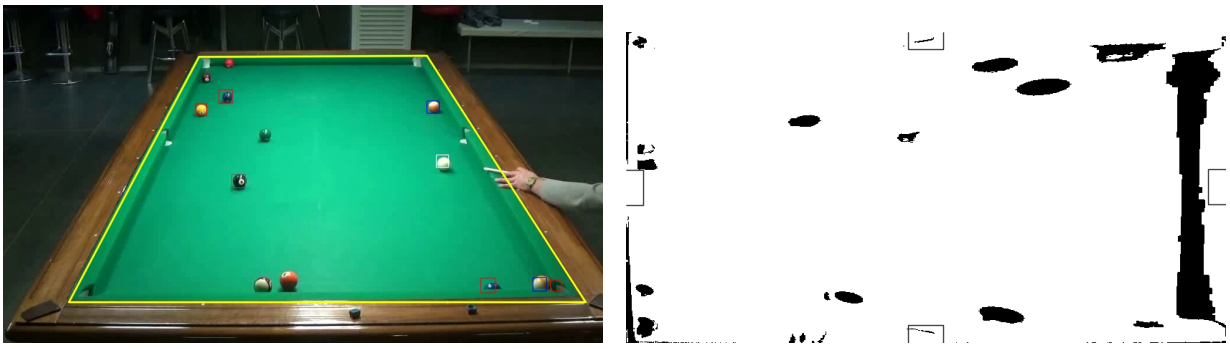
- If the edge with ”more pocket” is opposite to the one with ”less pocket”, then they are not the longest edges; This happen, for example, in Figure 3 and Figure 4, when there is an occlusion or much noise in the center of the edge with ”more pocket”.
- Otherwise, there is uncertainty, and then, probably, the one with ”more pocket” is the longest edge.

If the table is not horizontal as expected (for example in Figure 1), then all the edges are rotated and the transformation matrix is re-computed.

(a) Detection of the table and the balls with the colors representing the classes.

(b) Transformation of the table to the minimap table size

Figure 1: game1_clip1 first frame. The table is transformed in a wrong way, because the pockets are located in the shortest edges rather than the longest ones.

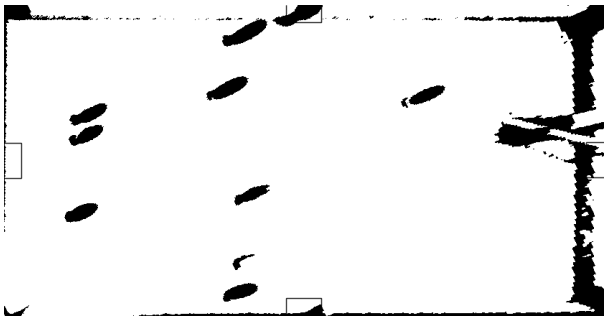


(a) Detection of the table and the balls with the colors representing the classes.

(b) Transformation of the table to the minimap table size.

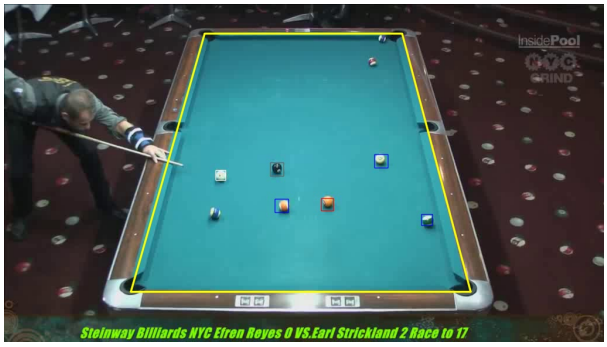
Figure 2: game2_clip1 first frame. The table is correctly transformed. In this case the pockets are lightly visible, but they allow to detect the correct orientation.

(a) Detection of the table and the balls with the colors representing the classes.

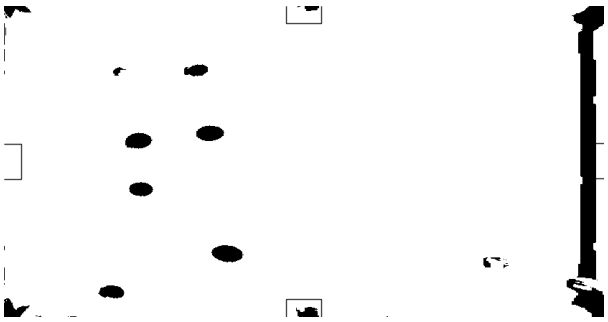


(b) Transformation of the table to the minimap table size.

Figure 3: game3_clip1 first frame. The table is correctly transformed. In this case, the center of one of the shortest edges has some noise due to the person playing the game; the result is correct, because in the opposite edge there is no noise.



(a) Detection of the table and the balls with the colors representing the classes.

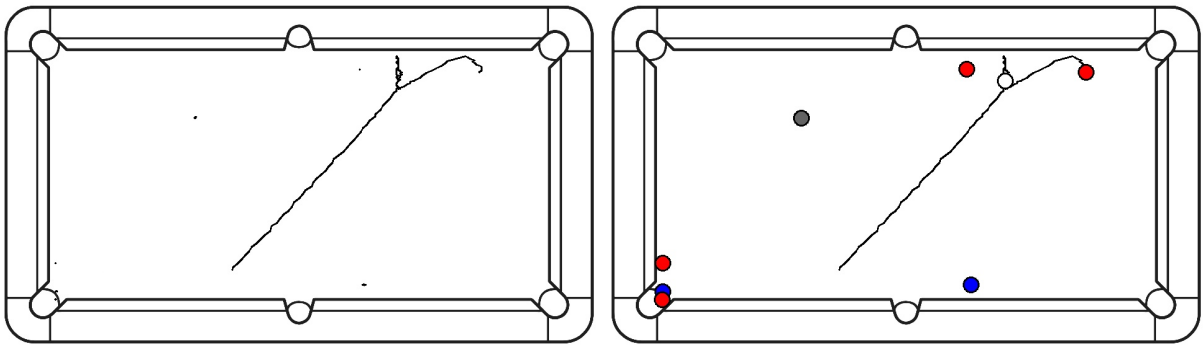


(b) Transformation of the table to the minimap table size.

Figure 4: game4_clip1 first frame. The table is correctly transformed. In this case, the center of one of the shortest edges has some noise due to the light of the table; the result is correct, because in the opposite edge there is no noise.

Draw the minimap with tracking lines and balls

Given the transformation matrix and the ball positions in the frame, it is possible to compute the positions of the balls in the minimap. This computation has been done in the `drawMinimap` method. Every time this method is called, the ball positions and the positions of the balls in the previous frame (if they have been computed by the tracker) are computed by using the `perspectiveTransform` method. For each ball in the frame, a line between the previous position and the current position is drawn on the minimap image, passed as a parameter by reference such that all the tracking lines are kept in a single image (Figure 5a). Then this image is cloned into a copy, and the current balls are drawn on it. This image is then returned (Figure 5b). This implementation idea comes from Alberto.



(a) Minimap with tracking lines (b) Minimap with tracking lines and balls

Figure 5: game2_clip1. Minimap of the last frame.

The ideas of using and the implementation of `getPerspectiveTransform` and `perspectiveTransf` and how to check the orientation of the table were from Michela.

Video creation

To create the output video Michele used the `VideoWriter` class of OpenCV. For each frame of the input video tracking is performed and the minimap is created, then the minimap is superimposed on the frame and the frame is saved in the output video.

To avoid creating the output video file and then run in some exception resulting in a non-readable file, Alberto thought about using a temporary file and then renaming it to the output file only at the end if no exception has been thrown and the program is sure that the file is complete and readable.

Metrics

The `computePerformance` executable handles both mAP and mIoU calculations.
mAP (mean Average Precision):

1. Predictions are performed for all first and last frames in a video.
2. Since we lack alternative *confidence scores*, we opted to calculate mAP using IoU as the measure of confidence, because it is a good indicator of how good the detection is.
3. For each object class (e.g., ball type), the Average Precision (AP) is calculated.
4. Then the final mAP is obtained by averaging the AP values across all classes.

mIoU (mean Intersection over Union):

1. IoU is calculated for the first and the last frame for each video.

2. The average IoU is then computed for each object class across all 20 images (10 videos each one with 2 frame) in the dataset.

3. Finally, the mIoU is obtained by averaging the IoU values obtained in the last step.

The 8BallPool executable displays the performance metrics (AP and IoU) achieved by the method for the specific input video.

Results

Table detection exhibits very high accuracy across the dataset, in particular for each initial frame of each video 4 corner points are always identified, the assumption that we made is that the camera does not move during a single clip so, once the table is detected in the first frame, we can use that information for all the other frames in the same video.

In contrast, ball detection is influenced by k-means clustering. To achieve consistent and satisfactory results, a fixed random seed is incorporated into the code.

This method results in an average mAP of 0.51 for the dataset.

Quantitative results

The *computePerformance* executable calculates the detection and segmentation performance across the entire dataset.

Class	**AP**
white	0.90
black	0.44
solid	0.31
striped	0.38

Table 1: AP of the detection across the dataset

Class	**IoU**
background	0.97
white	0.67
black	0.75
solid	0.36
striped	0.39
playing field	0.95

Table 2: IoU of the detection across the dataset

mAP	mIoU
0.51	0.69

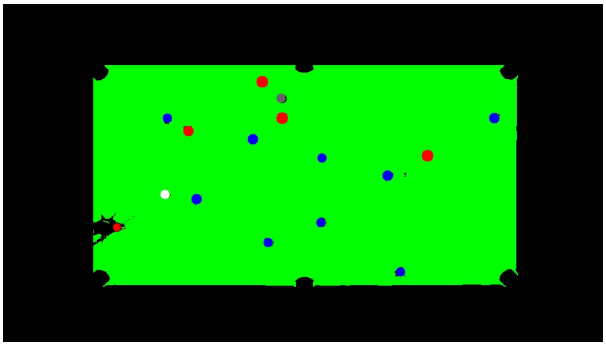
Table 3: Performance of the detection and segmentation across the dataset

While the algorithm successfully detects tables, backgrounds, and both white and black balls with high Intersection over Union (IoU), it struggles with solid and striped balls due to inaccurate distinction. This leads to a lower overall mean Average Precision (mAP) that is focused only on balls, but a good mean Intersection over Union (mIoU) because the background and playing field are well segmented.

Qualitative results

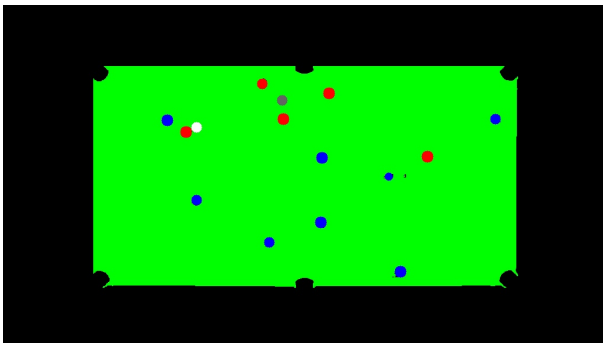
ome qualitative results are presented below.

(a) Detection *game1_clip1* first frame

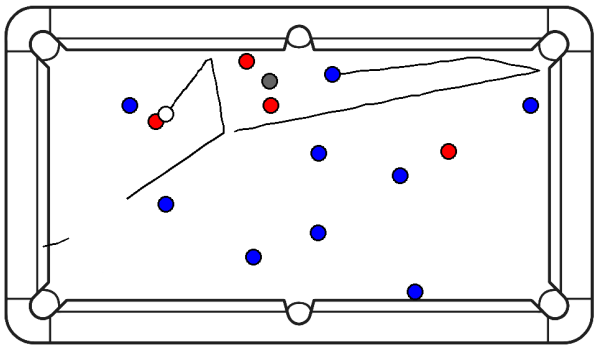


(b) Segmentation *game1_clip1* first frame

(c) Detection *game1_clip1* last frame



(d) Segmentation *game1_clip1* last frame

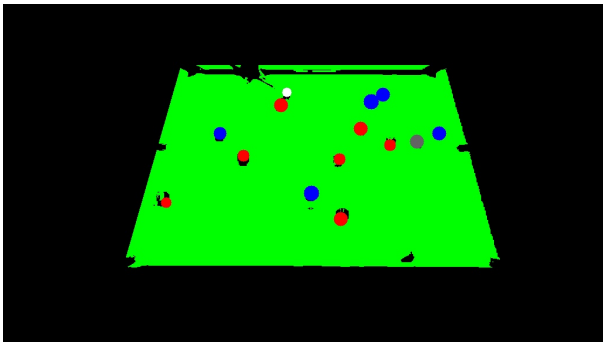


(e) Minimap *game1_clip1*

(f) Video *game1_clip1*

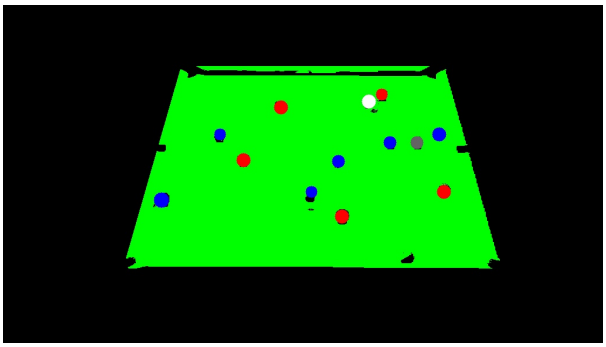
Figure 6: *game1_clip1*

(a) Detection *game1_clip2* first frame

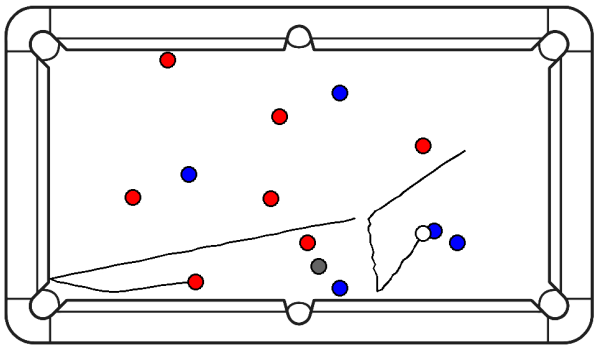


(b) Segmentation *game1_clip2* first frame

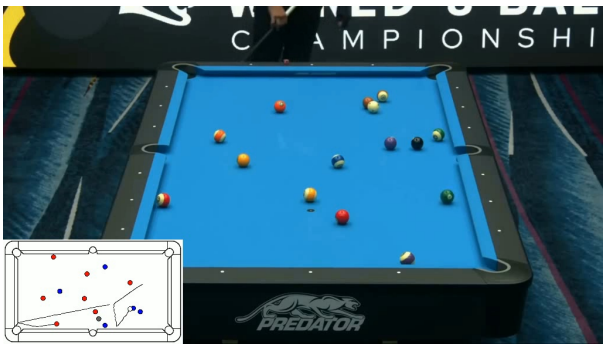
(c) Detection *game1_clip2* last frame



(d) Segmentation *game1_clip2* last frame



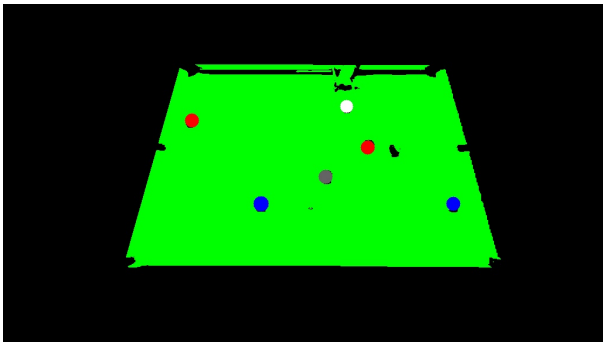
(e) Minimap *game1_clip2*



(f) Video *game1_clip2*

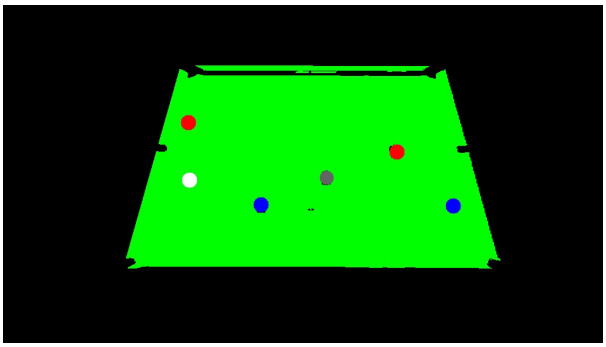
Figure 7: *game1_clip2*

(a) Detection *game1_clip3* first frame

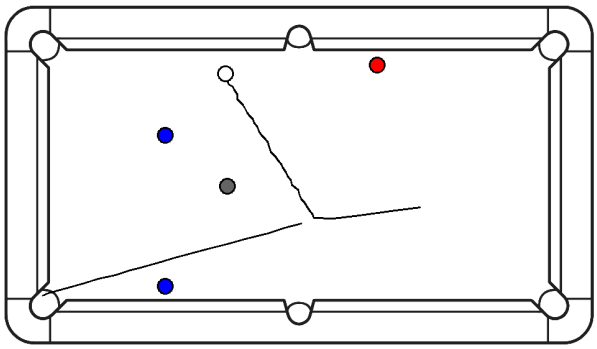


(b) Segmentation *game1_clip3* first frame

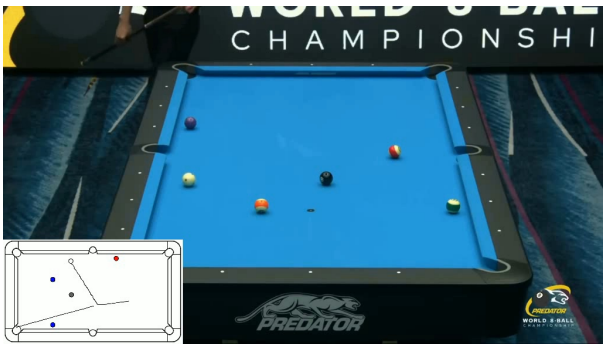
(c) Detection *game1_clip3* last frame



(d) Segmentation *game1_clip3* last frame



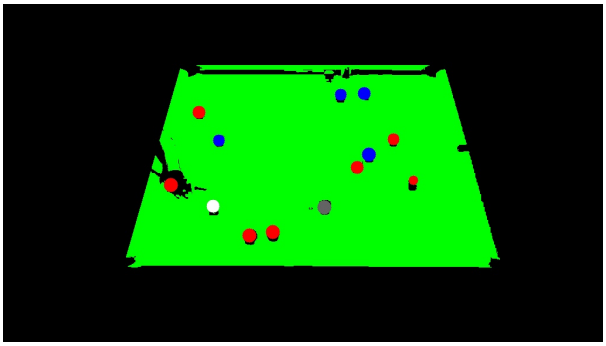
(e) Minimap *game1_clip3*



(f) Video *game1_clip3*

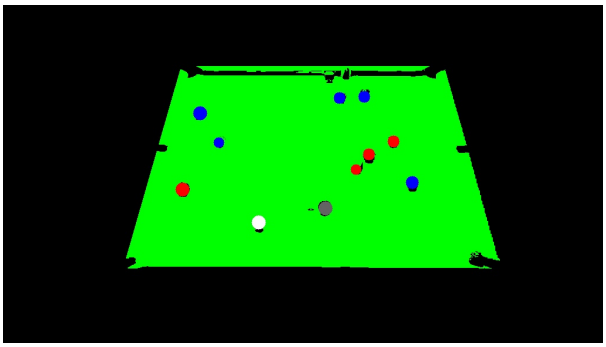
Figure 8: *game1_clip3*

(a) Detection *game1_clip4* first frame

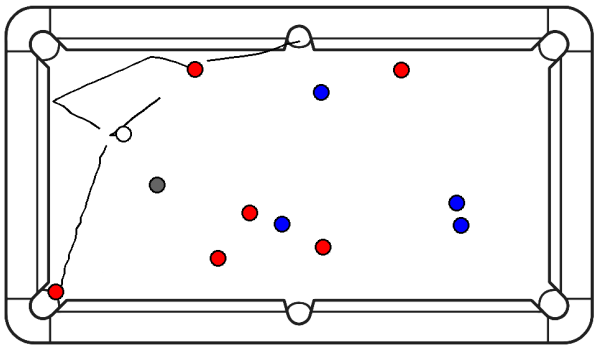


(b) Segmentation *game1_clip4* first frame

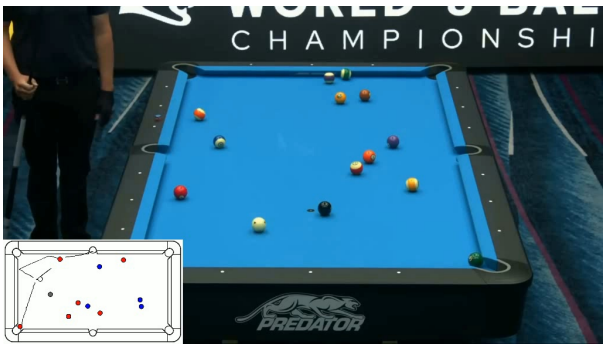
(c) Detection *game1_clip4* last frame



(d) Segmentation *game1_clip4* last frame



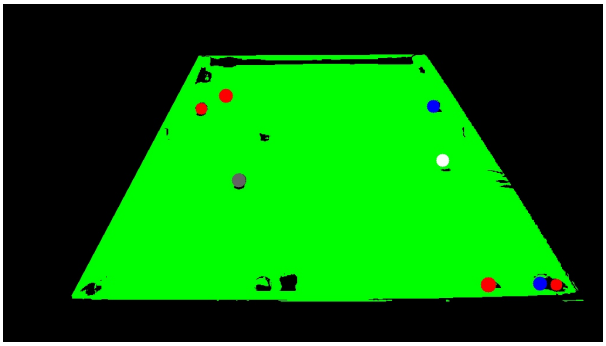
(e) Minimap *game1_clip4*



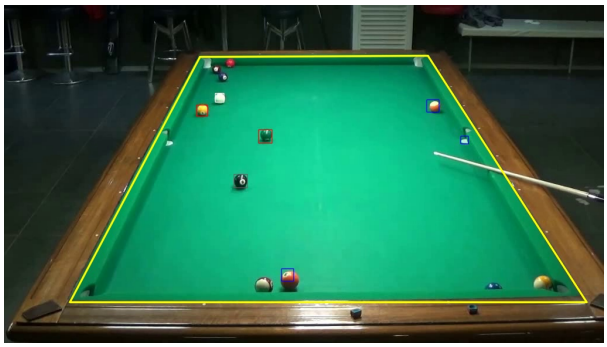
(f) Video *game1_clip4*

Figure 9: *game1_clip4*

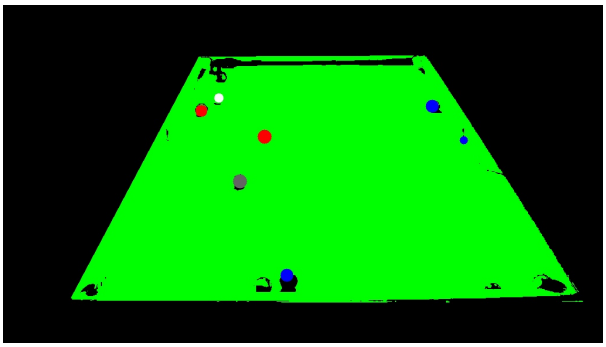
(a) Detection *game2_clip1* first frame



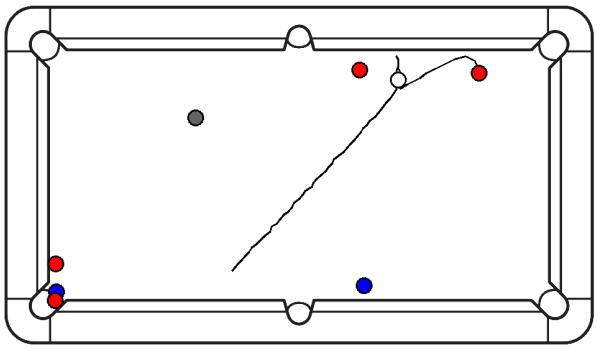
(b) Segmentation *game2_clip1* first frame



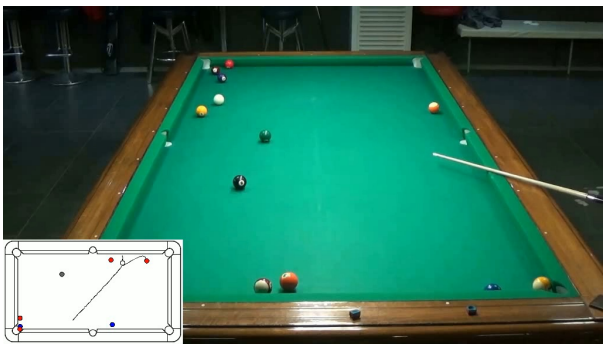
(c) Detection *game2_clip1* last frame



(d) Segmentation *game2_clip1* last frame



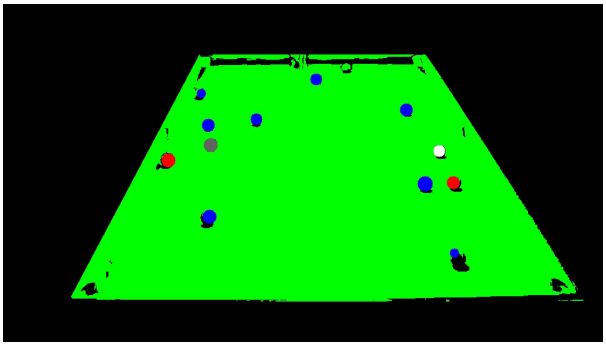
(e) Minimap *game2_clip1*



(f) Video *game2_clip1*

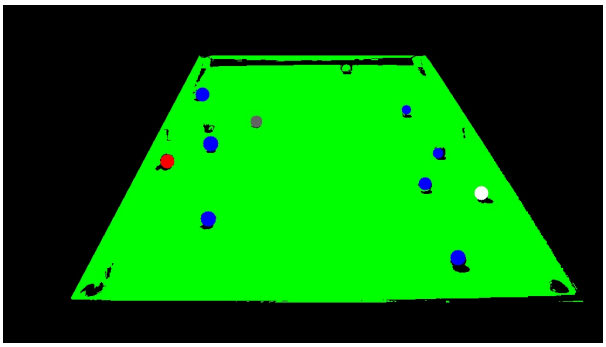
Figure 10: *game2_clip1*

(a) Detection *game2_clip2* first frame

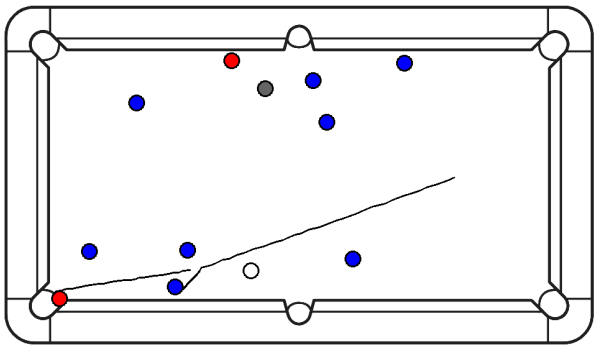


(b) Segmentation *game2_clip2* first frame

(c) Detection *game2_clip2* last frame



(d) Segmentation *game2_clip2* last frame

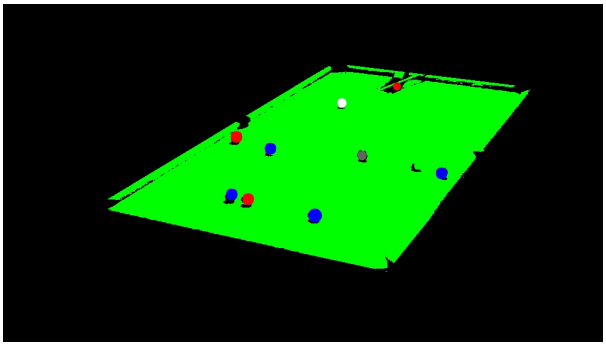


(e) Minimap *game2_clip2*

(f) Video *game2_clip2*

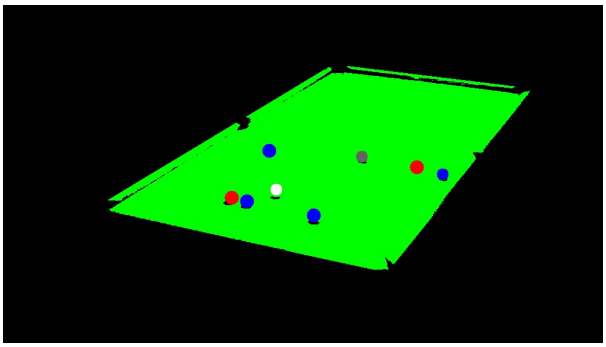
Figure 11: *game2_clip2*

(a) Detection *game3_clip1* first frame

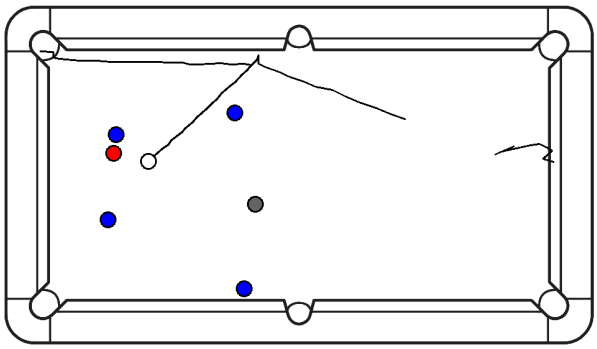


(b) Segmentation *game3_clip1* first frame

(c) Detection *game3_clip1* last frame



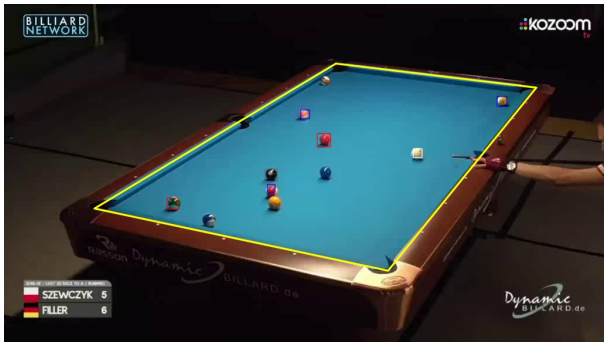
(d) Segmentation *game3_clip1* last frame



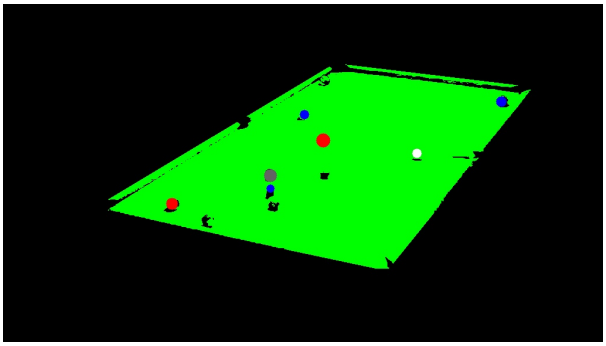
(e) Minimap *game3_clip1*

(f) Video *game3_clip1*

Figure 12: *game3_clip1*

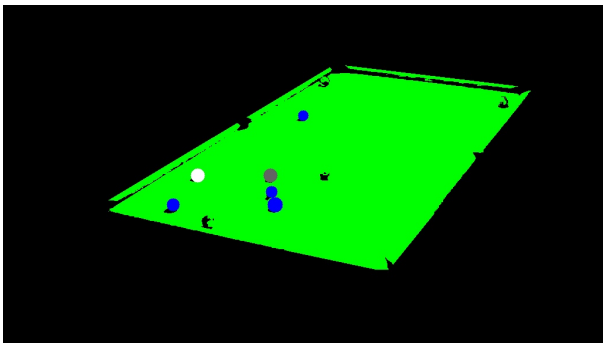


(a) Detection *game3_clip2* first frame

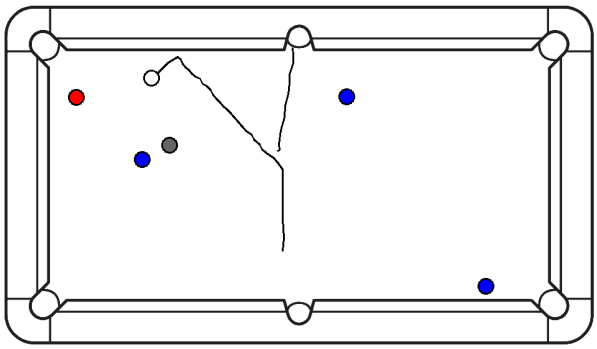


(b) Segmentation *game3_clip2* first frame

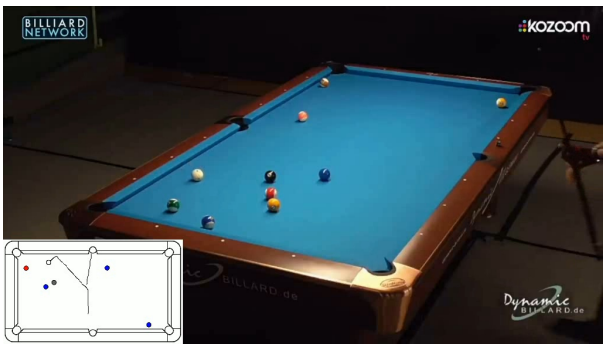
(c) Detection *game3_clip2* last frame



(d) Segmentation *game3_clip2* last frame

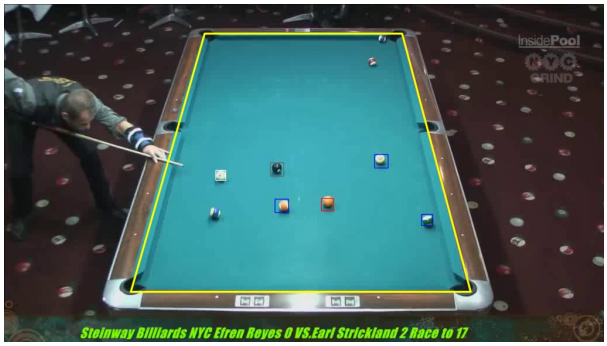


(e) Minimap *game3_clip2*

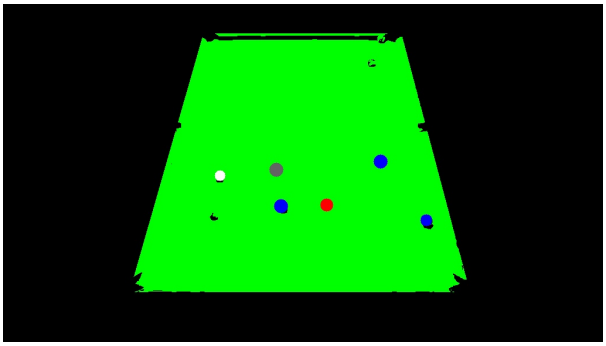


(f) Video *game3_clip2*

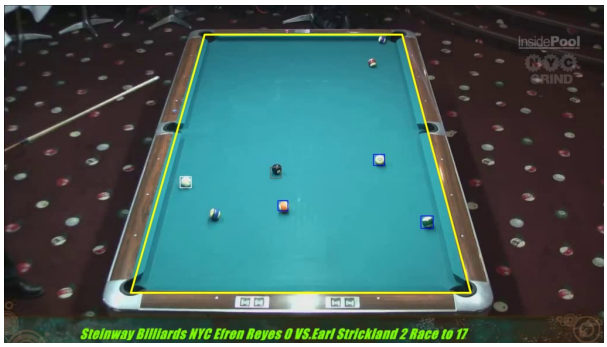
Figure 13: *game3_clip2*



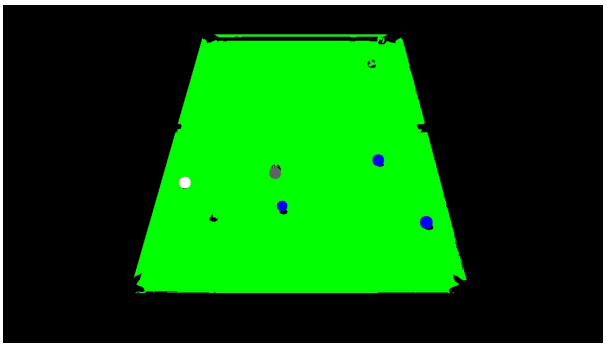
(a) Detection *game4_clip1* first frame



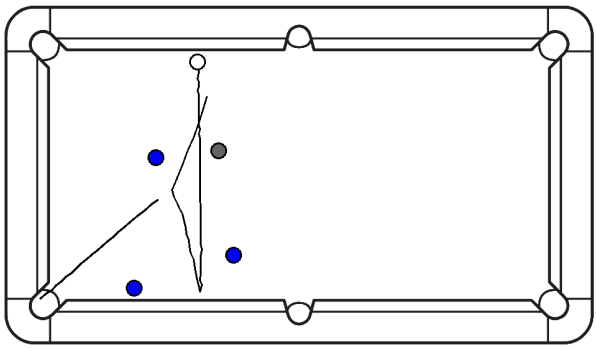
(b) Segmentation *game4_clip1* first frame



(c) Detection *game4_clip1* last frame



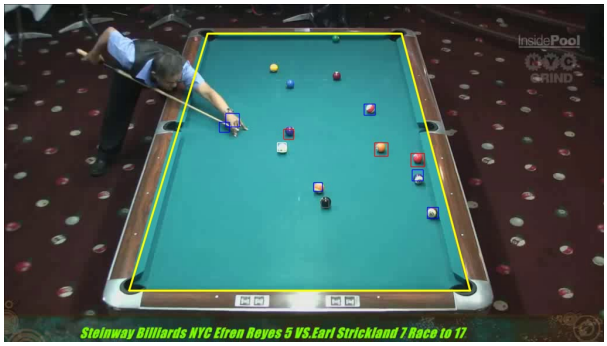
(d) Segmentation *game4_clip1* last frame



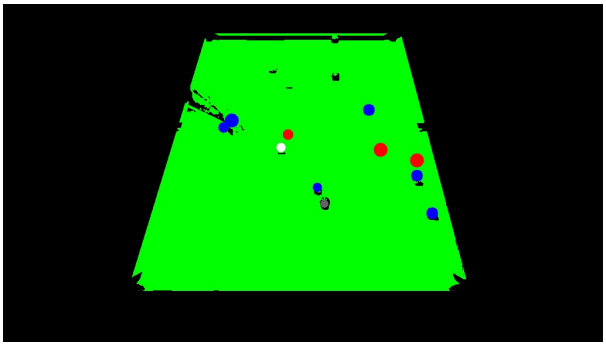
(e) Minimap *game4_clip1*

(f) Video *game4_clip1*

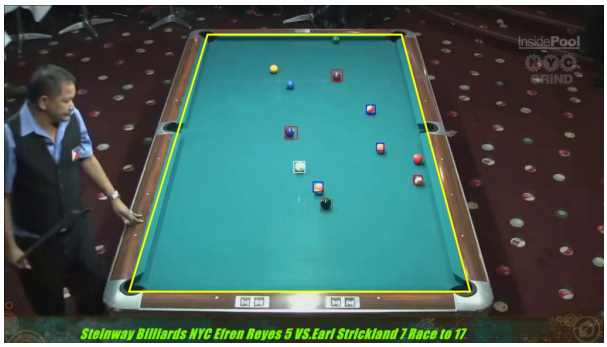
Figure 14: *game4_clip1*



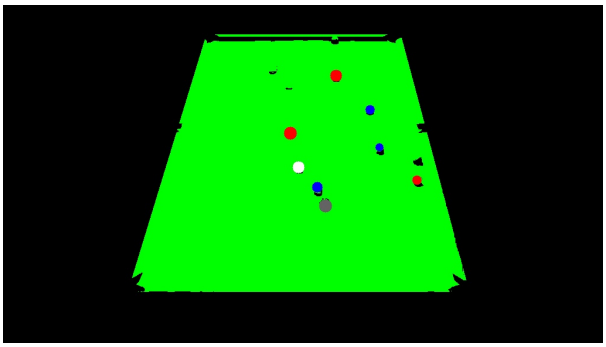
(a) Detection *game4_clip2* first frame



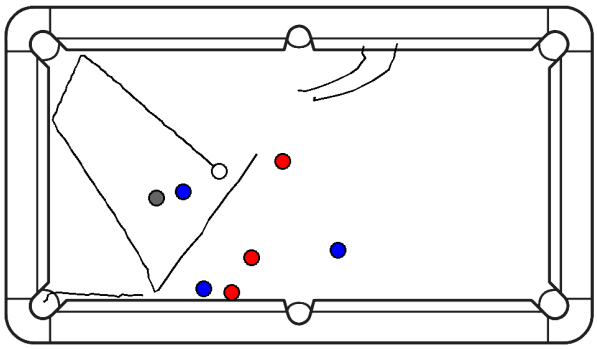
(b) Segmentation *game4_clip2* first frame



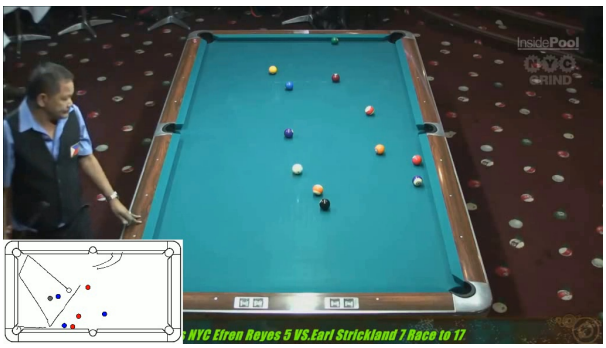
(c) Detection *game4_clip2* last frame



(d) Segmentation *game4_clip2* last frame



(e) Minimap *game4_clip2*



(f) Video *game4_clip2*

Figure 15: *game4_clip2*

Conclusions

Our approach demonstrates consistent performance across the dataset. Notably, table detection achieves high accuracy. However, ball classification presents some challenges due to their varying sizes and colors that can sometimes are similar to the one of the playing field, also solid and striped balls are difficult to distinguish because