8BallPool video analysis

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

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 these ones:

1. Michele: 40 hours.
2. Michela: 40 hours.
3. Alberto: 40 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.
- `TestAllClip`: the executable that was used to test the detection and segmentation in all the first and last clip of all videos.
- `ShowSegmentationColored`: the executable that was used to show the ground truth of the segmentation of a particular frame and it was also used as test for the code that computes the metrics because it computes the performance of the ground truth it self.
- `ComputePerfomance`: This executable was used to compute the performance across the dataset so the mAP and the mIoU.
- `InteractiveTracking`:

Table detection

A mask-based approach was implemented for table detection, exploiting the consistent central positioning of the table within the dataset. The mask was generated by identifying the most common color in the image's central columns. Building upon this initial step, Michele exploits the Canny edge detector and OpenCV's HuoghLinesP function. Then the function analyzes intersections and merges nearby points, ensuring the consistent identification of four 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 two-mask combined with k-means algorithm. The masks involved are:

1. Color-based masking: A mask was created that identified pixels corresponding to the table's color.
2. Corner detection masking: A separate mask was generated to capture the table's geometric features starting from the corners already detected.

By combining these two masks with the output of a k-means clustering algorithm with two clusters, the table was effectively isolated from the background.

Balls detection

To detect balls, Michele proposed a multi-step preprocessing approach. Initially, the table region was isolated by constructing a polygon using its corners and a color-based mask is generated. Subsequently, pixels outside the table were nullified, and k-means clustering was applied to the image. The resulting clusters were converted to gray-scale for Hough

Circle Transform application. 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 of 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, it has been attempted to compute 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, the `radiusInterval()` method has been written. This method starts by computing the mean radius value by using a proportion between the diagonal of the table in pixels and the approximate dimensions of the diagonal of the table and the balls in centimeters.

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

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 detected table creates; this angle is compared with the 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 it;
- If, for example, 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.

$$half\_interval = mean\_radius * percentage\_slope + prec \tag{3}$$

$$min\_radius = mean\_radius - half\_interval \tag{4}$$

$$max\_radius = mean\_radius + half\_interval \tag{5}$$

The idea of trying this method was from Michela.

Tracking

Mini-map creation

To create the mini-map 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 mini-map.

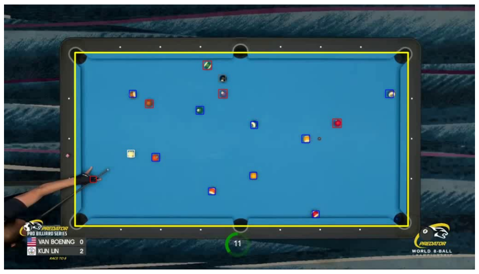
Empty mini-map image As a first step, an image of an empty billiard table has been selected, and its corner positions and dimensions have been stored in constant variables by testing different values. In particular we decided to use an header version of the image to have a self-contained executable without the need of the image. This version has been created with an online tool (ImageMagick - https://imagemagick.org/).

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 table of the mini-map. To do that, a relationship between the corners of the table in the frame and the corners of the table in the mini-map has been made. This relationship has been made by the OpenCV `getPerspectiveTransform()` method, which "calculates a perspective transform from four 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 order needed 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 pool in a rectangle placed in the center of the edge (with dimensions proportional to the real table and pool 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 pool" are opposite edges, then they are the longest edges; This happens, for example, in Figure 2.
- If the edge with "more pool" is opposite to the one with "less pool", then they are not the longest edges; This happen, for example, in Figure 3 and Figure 4.
- Otherwise, there is uncertainty, and then, probably, the one with "more pool" is the longest edge.

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


(a) Detection of the table and the balls.e

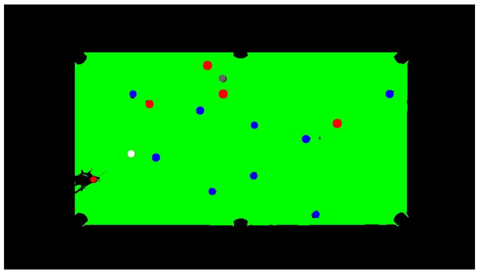
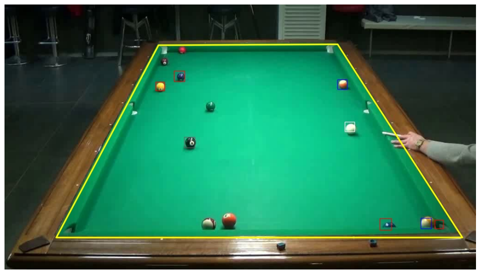

(b) Transformation of the table to the minimap table size.e

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


(a) Detection of the table and the balls.

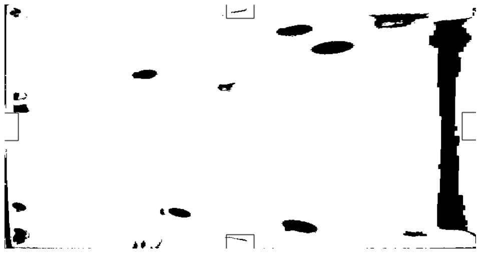
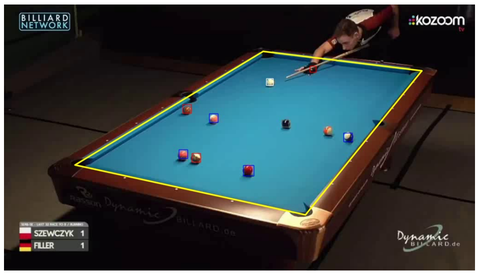

(b) Transformation of the table to the minimap table size.

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


(a) Detection of the table and the balls.

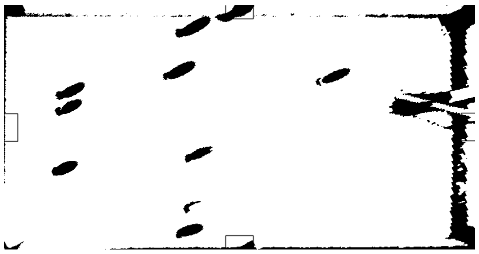
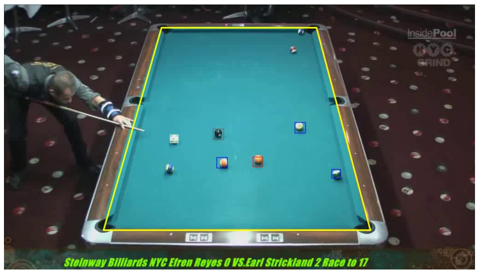

(b) Transformation of the table to the minimap table size.

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.

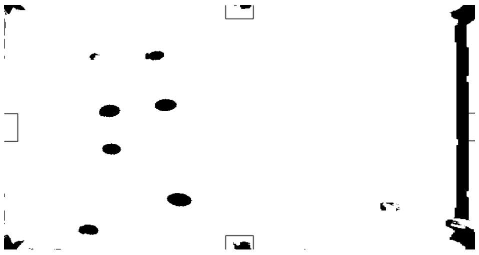
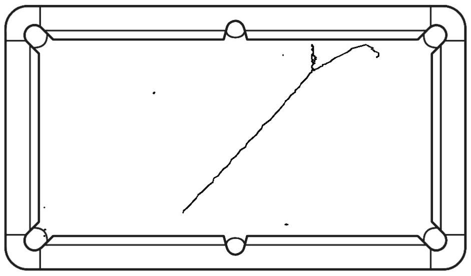

(b) Transformation of the table to the minimap table size.

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 mini-map 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 mini-map. 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 mini-map 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).


(a) Mini-map with tracking lines

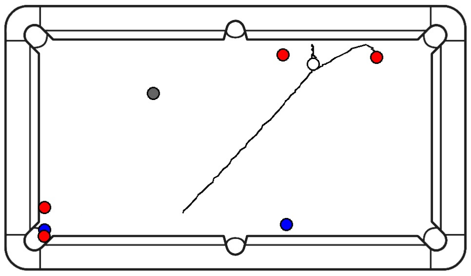

(b) Mini-map with tracking lines and balls

Figure 5: game2_clip1. Mini-map of the last frame.

The ideas of using `getPerspectiveTransform()` and `perspectiveTransform()`, and how to check the orientation of the table were from Michela; the ideas of using an header version of the mini-map image and of drawing the balls on a copy of the mini-map image, rather than the one that contains the tracking, were from Alberto.

Video creation

To create the output video we take the minimap and the current frame and Michele decide to do as follows: At the beginning the function compute two values: the scaling factor for the minimap and the offset (the first row where the minimap is placed). For the scaling factor Michele thought that the minimap should have 0.3 of the total columns of the frame, so the scaling factor is computed as follows:

$$scaling\_factor = \frac{(0.3 \times frame.col)}{minimap\_with\_balls.cols} \tag{6}$$

Instead for the offset, Michele thought that the minimap should be placed at the bottom right of the frame and it must be attached to the bottom border so first a factor is computed as follows:

$$value = scaling\_factor \times minimap\_with\_balls.rows \tag{7}$$

$$percentage = \frac{frame.rows - value}{frame.rows} \tag{8}$$

Then the offset is computed as follows:

$$offset = percentage \times frame.rows \tag{9}$$

After that the minimap is resized and placed in bottom right corner of the output frame that is saved in the output video. To avoid creating the output video file and then run in some exception resulting in a not readable file, Alberto thought about using a temporary file and then rename it to the output file only at the end where it is sure that the file is readable.

Metrics

The *computePerformance* executable handles both mAP and mIoU calculations.

- mAP (mean Average Precision):
  - Predictions are made for all first and last frames in a video.
  - 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.
  - For each object class (e.g., ball type), the Average Precision (AP) is calculated.
  - Then the final mAP is obtained by averaging the AP values across all classes.
- mIoU (mean Intersection over Union):
  - IoU is calculated for the first and the last frame for each video.
  - The average IoU is then computed for each object class across all 20 images (10 videos each one with 2 frame).
  - Finally, the mIoU is obtained by averaging the IoU values obtained in the last step.

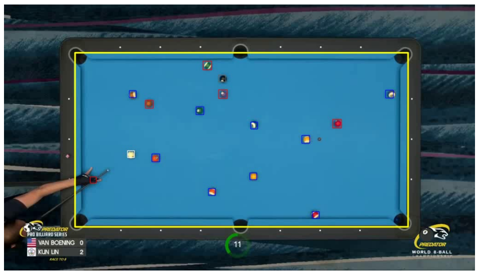
The 8BallPool executable displays the performance metrics (AP and IoU) achieved by the method for the specific input video.

Results

Table detection exhibits very high accuracy, in particular for each initial frame of each video four corner points are consistently identified across the dataset, 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 frames of 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.

Qualitative results

Below some qualitative results are presented.


(a) Detection game1 clip1 first frame

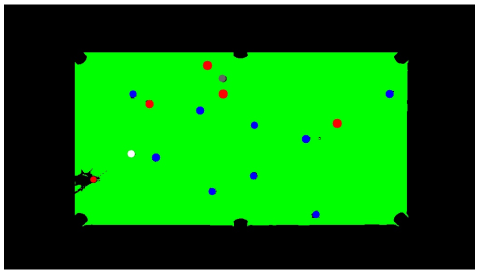
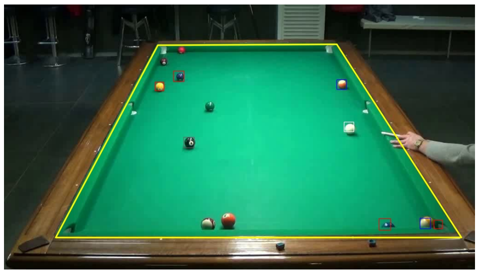

(b) Segmentation game1 clip1 first frame

Figure 6: game1 clip1 first frame


(a) Detection game2 clip1 first frame

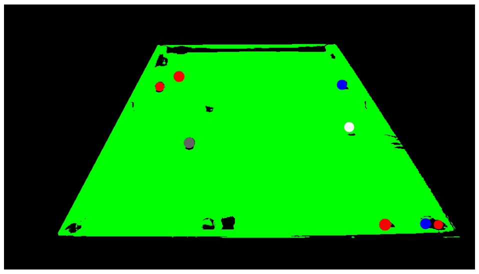
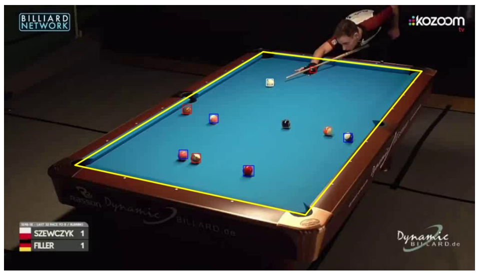

(b) Segmentation game2 clip1 first frame

Figure 7: game2 clip1 first frame

mAP	mIoU
0.51	0.69

Table 1: Performance of the detection and segmentation


(a) Detection game3 clip1 first frame

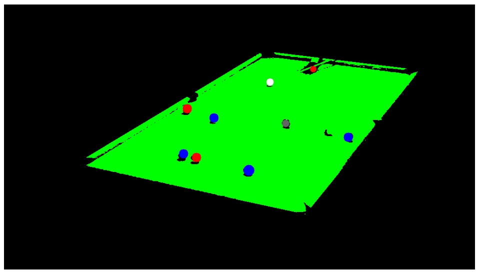
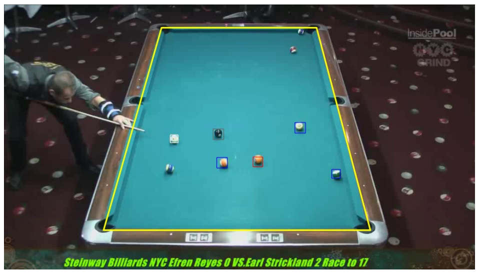

(b) Segmentation game3 clip1 first frame

Figure 8: game3 clip1 first frame


(a) Detection game4 clip1 first frame

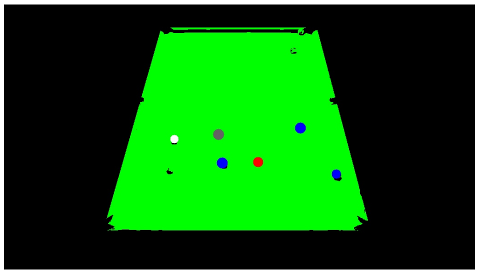

(b) Segmentation game4 clip1 first frame

Figure 9: game4 clip1 first frame

Quantitative results

The *computePerformance* executable calculates the detection and segmentation performance across the dataset. While the method 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 classification. 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 identified well.

Conclusions

Our program 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 table.