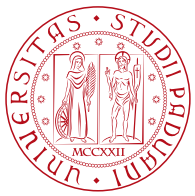


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Pool Game Tracking System

Project report

Authors: Federico Adami, Alessandro Bozzon, Simone Peraro

21/07/2024

1 Introduction *by Simone*

The project consists of creating a system to track the billiard balls of a pool game. Ideally, the system should receive as input a video directly from a camera pointed at a billiard table, and should be able to detect the position and the class of the balls. Said system will be tested, rather than with a real live game, with a set of small video clips extracted from real life games. Moreover, this set of test videos provides a variety of different camera angles, light sources, tables and balls which push to the limit the task of detecting and classifying the balls in the scene correctly. Thus is not enough to fine tune the system according to a single ideal situations, but rather the system should be imagined as a complex set functionalities which aim at making the whole project robust, but also efficient.

Moreover, the system will be evaluated against a set of provided ground truth files, in a offline computation of metrics which are the intersection over union (over the masks of the segmented scene) and the mean average precision (over the bounding boxes of the detected balls). Finally the system is required to draw a schematic representation of the game status, tracking the balls' trajectories and showing them from a top view representation.

In the following sections, we will discuss how the project has been developed, which difficulties we faced and how we tried to solve them with different techniques.

2 Structure of the project *by Simone*

As discussed in the introduction, the project cannot be reduced to a single set of consecutive functions to reach a perfect result: firstly because such perfect system is probably really difficult to obtain with the variety of situations provided (and even if perfection is nearly reached, the system will probably fail with a new unseen game situation), and secondly because our system should also be imagined running in an *offline-mode* to evaluate performance against a provided ground truth, thus the system should be able to run and produce some outputs when required, in order to perform the evaluation.

I spent a few hours thinking how to effectively split the project code, while *Alessandro* and *Federico* started experimenting with the segmentation task.

Probably, the most natural way to divide the project is to identify the main components of the system and imagine how they should run and interface with each other. Specifically, considering also the project requirements to be delivered, I have identified the following main components/modules:

- table segmentation module
- ball detection module
- ball classification module
- ball tracking module
- game status drawing module
- performance evaluation module
- live game executable
- offline performance evaluation executable

My initial idea was to run the first five modules in an independent way frame by frame, but as my colleagues pointed out, to keep the system efficient and assuming a static camera angle, the first three modules can be runned (ideally) just one time. The main advantage of having a system divided this way, besides from obvious better code maintainability with respect to a single library/set of functions, is that we can test and tune each module (almost) independently, providing a good way to improve the system without necessity of rebuilding the entire project. Ideally, each module will collect and provide information to the next module, without making redundant computations which may be slow down the system a lot. However, this efficiency comes at cost of an almost-independence between modules: each one depends on quality of data provided by the previous module.

All those modules will be instantiated and executed in the two executables files, one for the *live game* mode, and the other for an *offline* performance evaluation mode, but the execution sequence of the modules will be the same. In Figure 1, Figure 2 we can see some initial basic implementation idea, both for live and offline running modes.

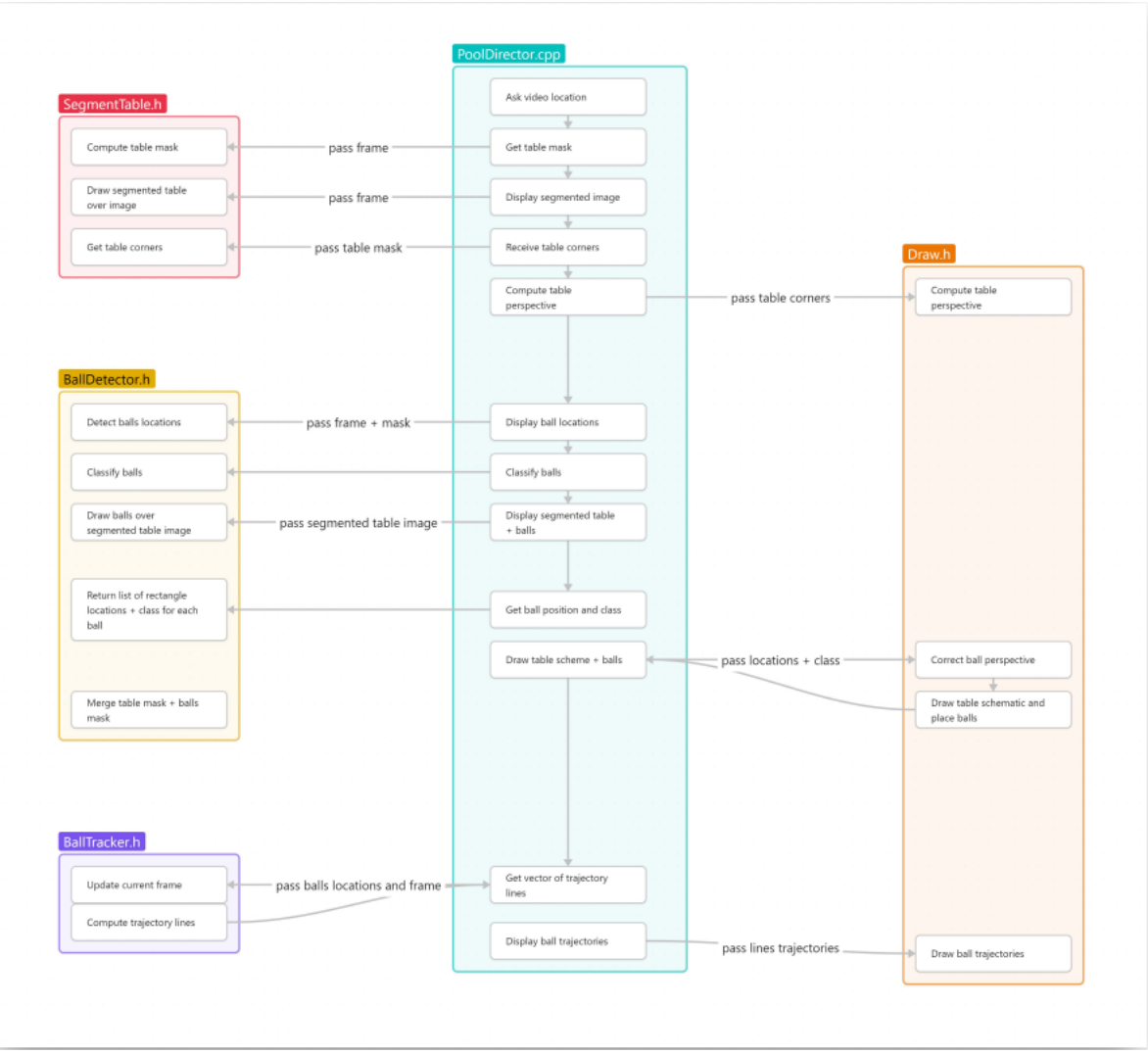


FIGURE 1: First implementation idea of the system running in live mode. In the center, the sequence of operations performed by the executable, while on the sides the module's functionalities are called.

In the end, the system is composed by the following main classes and files:

- TableSegmenter class, developed by *Alessandro*,
- BallDetector class, developed by *Alessandro*,
- BallClassifier class, developed by *Federico*,
- BallTracker class, developed by *Federico*,
- Draw class, developed by *Simone*,
- EvaluationMetrics class, developed by *Simone*,
- run-live executable, developed by *Simone*,
- run-evaluation executable, developed by *Simone*.

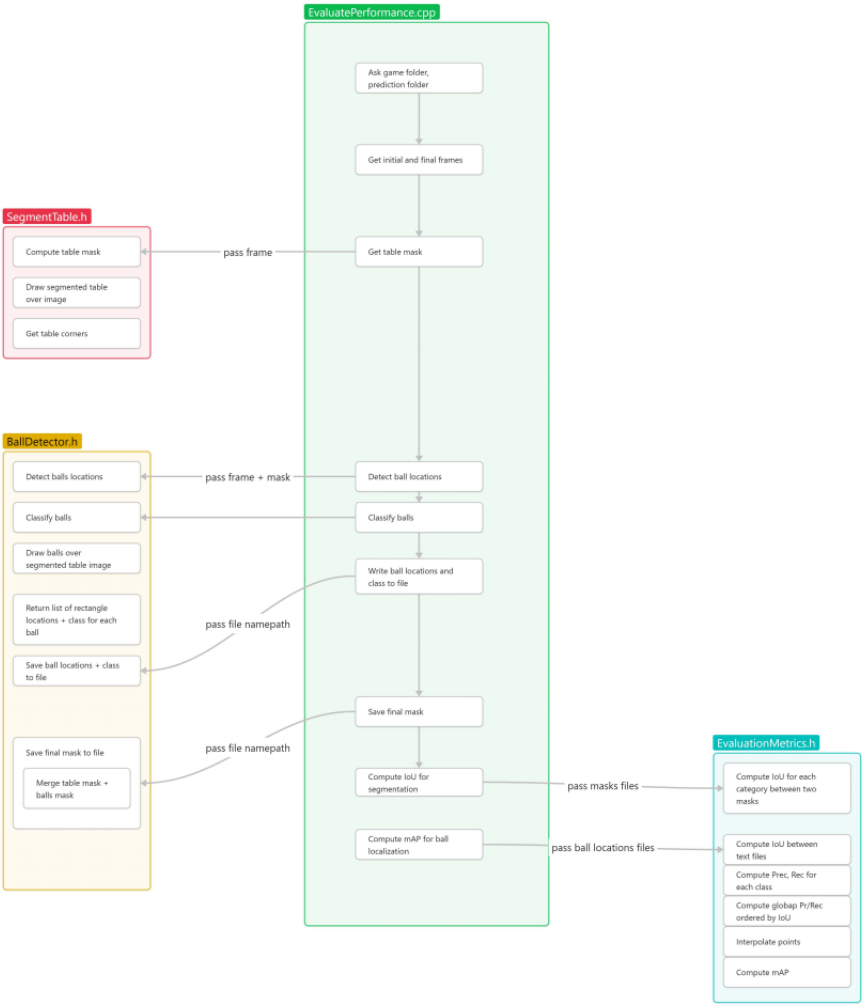


FIGURE 2: First implementation idea of the system running in offline mode. In the center, the sequence of operations performed by the executable, while on the sides the module's functionalities are called.

The purpose of each class is almost self explanatory and follows a natural definition of the sequence of operations that the system should do: we want to segment the table, find the balls inside the table, classify them into different classes, track them along video frames and finally draw the schematic view of the game status. The interface for each class was suggested using the functionalities represented in the two figures Figure 1, Figure 2, but each member was free to better define the interface accordingly, naturally with some limitations to prevent making the two main executables a complex sequence of too specific functions' calls. In the following subsections, the purpose and functionality of each class is discussed by each team member.

2.1 Table segmentation and masking *by Alessandro*

The mask of the field is computed starting from the frame that is obtained from the video that we are evaluating. First of all the frame is transformed into the HSV color space for an easier manipulation of the colors. Then the mean color in a window of size 11x11 centered on the center of the image is computed. Since the pool table is the main subject, by looking at the center pixel of the frame it is almost certain to look at a portion of the table itself. The only weakness of this method is that if the center of the image correspond exactly with the center of a ball, the mean color obtain as result is not the one of the table cloth. Statistically, given the size of the table and the size of the balls it has a high probability of obtaining the correct mean color of the table cloth and in fact, on all of the 10 video clips, it was always able to compute the mask of the table starting from the mean color. After mean color computation, a rough mask of the field is computed by assigning a value of 255 in a gray scale image to all the pixels that in the original frame have a color that is similar to the mean color computed up to a given threshold whose values were defined by hand by looking at the average result on all the 10 video clips first frames. After the rough mask computation, the mask contours is computed and filled to obtain a mask of the playing field, as in figure 3, without any hole caused by the balls.
This mask is then used to isolate the playing field from the rest of the stuff present on the image, labeled as background as can be seen in figure 4.
The final mask that contains background, playing field and balls is then computed only after the balls are identified and

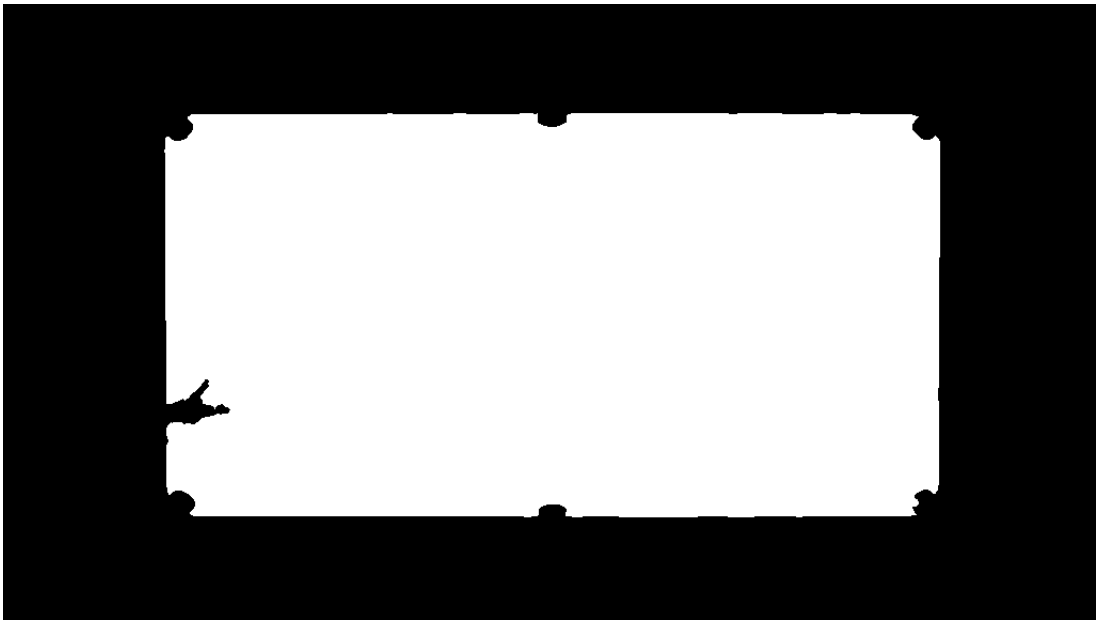


FIGURE 3: example of segmented table without the balls on top of it

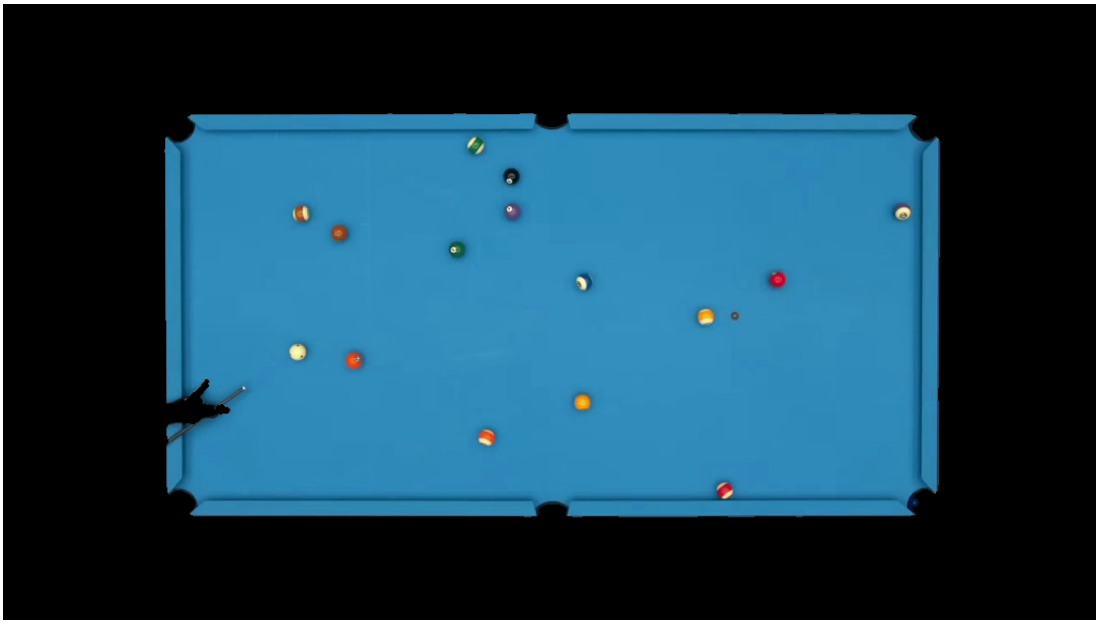


FIGURE 4: example of isolated field starting from field mask in figure 3

classified using the functions of the class that is responsible of identifying the balls on the playing field.

The final mask has the following colors:

1. gray scale level 0: background
2. gray scale level 1: white ball
3. gray scale level 2: black ball
4. gray scale level 3: balls with solid colors
5. gray scale level 4: balls with stripes
6. gray scale level 5: playing field

Although this color choice is not the greatest for human eye quick distinction of the classes, it was used because it followed the same color mapping used in the ground truth masks inside the dataset used.
From the table mask without the balls on top of it, playing field corners are computed in order to let us correlate the playing field on the video with a sketched playing field where the trajectories are then plotted alongside the balls to simulate the match being played.

The segmentation of the table is performed using the class TableSegmenter which has three public functions:

1. getTableMask to compute the mask of the field without the balls on top of it, it basically distinguish the table from the background
2. getFieldCorners to obtain the four field corners that are fundamental to link the frame to the sketched playing field
3. getMaskedImage to obtain an image from the frame where only the table is present and the rest of the image is set to black color which identifies it as the background that does not need to be considered

2.2 Ball localization *by Alessandro*

First of all a class Ball was created because it felt necessary to have an object for every single detected ball that in the end contains all the informations related to the ball, that are:

1. the position of the center of the circle that identifies the ball
2. the radius of the circle that identifies the ball
3. the values of the bounding box that encloses the ball
4. the class of the ball that is defined via an enum class for better reading of the code
5. the white ratio of the ball that contains the percentage of white-ish pixels of the ball as suggested and requested by Federico in order to simplify the classification of the ball

In the early stages of the development, center point of the ball and the radius were contained inside a single cv::Vec3f variable of the class Ball because of the output format of the circles found by the HoughCircles function. In later stages, as suggested by Federico, the single variable was split into two separate ones to make them easier to manage and the code easier to read.

The first major operation was the detection of the balls starting from the frame extracted from the video. In order to do so, the frame was processed to obtain an image as in figure 4 that removes all the background and makes it easier to focus on the field only removing the noise that background objects can cause further down the balls detection pipeline.
From an input as figure 4, the 3 HSV channels were split and only the S channel was used for the detection of the balls on the playing field because from various test it happened to be the channel that gave the most informations to detect the highest number of balls. My colleague Federico also tried to have a look at how to improve HoughCircles by using also channel H alongside S but it resulted in less precise circle placement for some balls and a lot of wrongly detected circles that most of the time were removed, but sometime they ended up in spots of the field that the outliers removal was not able to detect, for example circles given by the shadow of a ball. Once the image was split into the 3 separate channels, the opencv HoughCircles function was called in order to detect circles on the image that can correspond to balls and wrongly detected circles. Once again the parameters used in the HoughCircles function were defined after several tries on the first frame of each of the 10 video clips of the dataset used. From tests, it came out that the balls had a radius that ranged from 6 to 13 pixels. Once all the circles were computed by the HoughCircles function, then the process to erase all the wrongly detected circles starts.
The first thing that the algorithm for the removal of cirlces that do not correspond to balls does is compute the mean color of the table cloth using a kernel window of size 11x11 centered on the center of the image. From that value three individual thresholds values are set, one for each of the HSV channels. The values were hard-coded after some evaluation tests on all the first frame of the 10 video clips. It turned out that the parameter that did not matter at all for the removal of the incorrect detected circles was the value of the S channel, meanwhile the H channel played a really important role because it is the channel that defines the color of the pixel. After the mean color was computed and the thresholds set, the process of removal of the wrong cirlces start by looping through all the detected cirlces stored inside a vector. For each one of the circles, the first thing the we checked was the fact that the center of the circle lied inside the mask that defines the field. Then the color of the central pixel of the circle was extracted and if it was similar to the mean color previously computed up to the defined threshold then the circle was removed because it was a circle that was detected on the table cloth and not on a ball. After those two checks another two are performed, one check if the center of the circle is too close to the bounding polygon that encloses the field, which means that a circle is detected on top of the railing of the playing field, the second one checks if the circle center is extremely close to one of the four corners of the field which means that the algorithm most probably detected the hole of the playing field as a circle. This last check can present some minor problems if the perspective of the field compresses the distances of far away balls from the camera, so in order to limit this problem, the threshold value on the distance from the four corners was kept low, but still it was useful to erase some cirlces that did not correspond to balls. After all this processing, the final step was the creation of a vector that contains all the balls that are initialized starting from the remaining circles after the process of removal of the wrongly detected circles.
Once the vector of balls was initialized, the process of defining the bounding box for each ball starts. First of all, all the balls are plotted over the mask that correspond to the field, as can be seen on figure 5, where the balls have already the color based on the class, as defined in the section regarding the table segmentation, meanwhile the field is kept with a gray scale level of 255 to enhance the contrast between the balls and the field that is later used to compute the bounding box for each single ball.

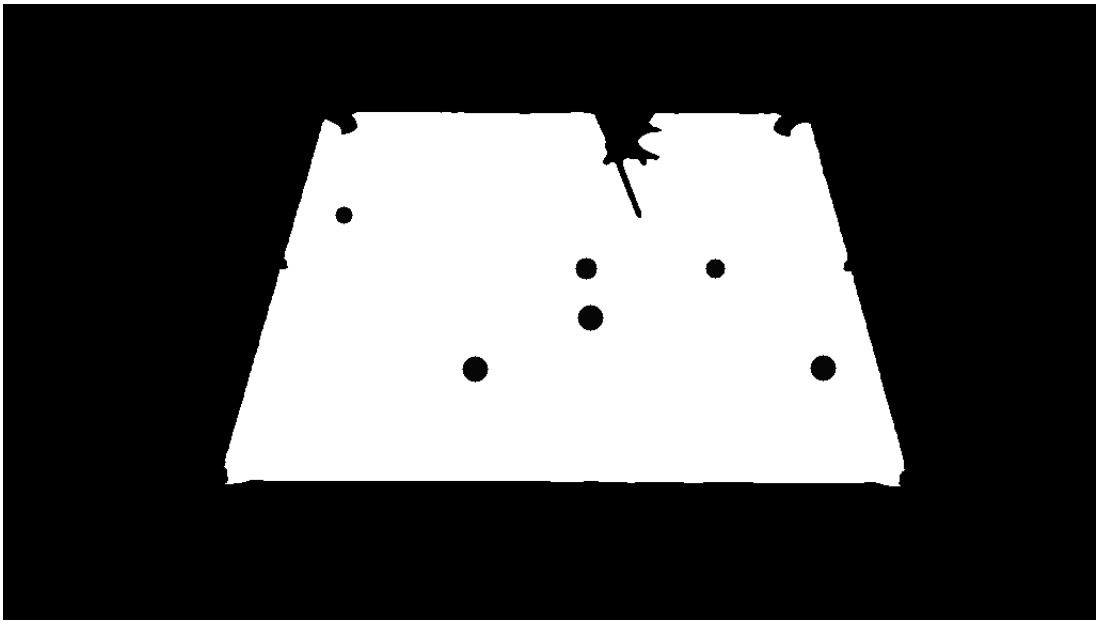


FIGURE 5: example of field mask with balls from first frame of video clip 3 of game 1

Starting from a mask like the one of figure 5, Canny edge detection is performed and from the computed edges the contours of the objects on the field are found using the opencv function findContours that stores all the contours that has been found inside a vector of vectors of points, where each inner vector corresponds to a contour. It then proceeds to compute each individual bounding box for each one of the contours, removing all the bounding box that are found multiple times for the same contour. It then removes all the bounding boxes that have the color of the central pixel of said bounding box different from the colors that correspond to the classes of the balls that are plotted on the mask. The last check that is performed is the removal of all the bounding boxes that have an area that is larger or smaller than a given threshold, which means that those bounding boxes do not frame any ball.
Once all the bounding boxes are found, they are associated with the corresponding ball. This process is done by matching ball and bounding box that have a center that has at most an offset of 1 pixel alongside the x axis and the y axis.
The last step in the ball localization pipeline consist in removing all the balls that have a bounding box that still has the default constructor value, meaning that no bounding box has been associated to the ball.
Once that is done, the vector containing all the detected balls is returned.
The class for the detection of the balls is also responsible for saving on the disk at a given location passed by parameter the final mask of the field and the balls with the correct colors, as defined in the section about segmentation. The last function present in the ball detection class is the one used for saving of a file the bounding box and class of each ball. These two function are essential in order to be able to compute the evaluation metrics as explained in the later section of this report.

2.3 Ball classification *by Federico*

This whole task was carried out by Federico Adami. In order to have consistent testing and easily compare different classification algorithms, the first thing necessary to do was to create a dataset of all the balls in the game. In order to obtain single cutout images for every ball, the localization algorithm was employed. By having both information about the bounding box relative to a ball and the circle outlining it, it was possible to extract from a video frame small images each containing one ball. By exploiting the circle outlining the ball, all the pixels of the small image, outside the circle were set to black, isolating the ball from the background. This process was performed for all the first frames of each game clip. The result was a folder for each game clip containing the cutout images of all the localized balls. For each game clip the set of images was then subdivided into two subfolders *full* and *half*, respectively containing only balls with full color and only balls half colored and half white. The black ball was considered as full color and the white one as half color, an additional classification would be then performed for these two balls. The main idea was to try to classify a ball based on its occurrence of white color. This comes from the observation that half balls should have a higher occurrence of white pixels with respect to full balls. In general the algorithms explored had the following structure:

1. Image preprocessing
2. Binary thresholding
3. Computation white pixels ratio
4. Classification by ratio thresholding

A first attempt was made just by trying binary thresholding on each BGR channel and by computing the white pixels ratio on the resulting image, tha aim was to find a good combination of threshold values for each channel that could best separate the different classes. The issue with this approach was its sensibility to lighting conditions and so threshold parameters that would work in one clip couldn't work as well in another with different view angle and lighting. So the next step was to try to make the process independent from lighting and a preprocessing step was added. This converted the images into grayscale and then both image equalization and normalization were tried to achieve independence from lighting. This system both with equalization and normalization didn't improve much with respect to the previous approach. Because these approaches didn't provide satisfying results another way was explored. Firstly by converting the images to HSV color space and performing binary thresholding only on the S channel of the converted image, this approach seemed to be more robust to the variance of lighting and was performing a little bit better than the previous ones. Lastly discussing with *Alessandro* it was decided to try classification by converting the image into HLS color space and analyzing the L luminance channel which we believed could give more information about how "white" the color of a pixel was. By testing around different combinations of thresholds on HLS channels the best performing method was obtained by applying the opencv function `inRange` and the ranges specified for the function where: (0, 150, 30) for the lower bound HLS values, while the upper bound was (180, 255, 255). Then from the binary image output of `inRange` the white pixels ratio was computed. This method was the most robust with respect to illumination changes and it was possible to set the threshold to classify half and full images at 9.9% of white pixels ratio. Then from this it was necessary to classify the black and white balls. To do so the previous algorithm was exploited, during the various tests it was noticed that for almost every game the ball with the highest white ratio was the white ball, while the ball with the lowest white ratio usually was the black one. So to classify these two balls in every game, the ball with the highest white ratio is classified as white, while the one with the lowest white ratio as black.

The major flaws of this approach are when a billiard ball has a lot of reflections from the light above the pool table and so the ball might be labelled as white even if of full color and lastly in some conditions solid balls of blue or green can be classified as black.

FIGURE 6: example of black ball with strong reflection

This is phenomenon is possible to observe in figure 6 where the black ball was not labelled as such firstly beacuse of the heavy reflections from the lighting and secondly because the white circle containing the number "8" was very visible. To tackle the issue of the white circles containing the ball numbers, Hough transforms were initially used in an attempt to remove these circles. However, this method proved ineffective because the circles were often partially occluded. As a result, this approach was not adopted.

2.4 Tracking the balls *by Federico*

This whole task was carried out by Federico Adami. For the development of the tracker initially the tutorial provided by opencv was followed. Then the tutorial was modified to handle more than one tracker by creating a vector of trackers. This way it was possible to test on the different clips the various tracking algorithms provided by opencv. The best performing two were KCF and CSRT trackers. On one side KCF was really fast and computationally light even with a high number of trackers but it was also really sensible to occlusions and couldn't recover tracking after such events. On the other hand CSRT was computationally more expensive, slowing down the real time tracking of the balls but despite this, was more robust to occlusions than KCF and many times could recover after losing tracking. Because of this it was decided to use CSRT as the tracker for the BallTracker class. The problem to solve was to speed up the process and optimize the tracking of all the balls to achieve real time performance. One important aspect, noticed empirically, was that the computing time of tracking program was directly proportional to the number of tracked entities in the game. So it was important to reduce the number of tracked balls and select just the ones which were moving while the ones not moving didn't need to be tracked, their position was known from the localization algorithm. In this view the BallTracker class has two vectors of balls: still balls and moving balls. Each moving ball has a tracker assigned to update the positions over the frames while the still balls don't. Given a video to start the tracking it is necessary to identify the moving balls. From the rules of the game it was known that the first ball to move would have been the white one. So at the beginning of a video the first frame is analyzed by localizing and classifying the balls in the image, then a vector of balls is created and its elements are ordered by white pixels ratio, the three with the highest ratio are selected and placed in the moving balls vector; then the trackers are initialized with these balls. During the game some balls may hit each other, increasing the number of moving balls to track. To account for this scenario a collision distance was determined, so if a moving ball is at a distance lower than the collision distance from a still ball, the static one will start to move so we can start tracking it, remove it from the still balls

vector and insert it in moving balls. The collision distance was determined by doing different trials and was set to 25 pixels. The last problem we needed to face was when a player hit the white ball so hard that tracking was lost. This is due to a big and sudden change in position of a ball from one frame to the next, causing motion blur due to the high speed of the ball that deforms it as can be seen in Figure 7 and Figure 8 which represent two consecutive frames in which the white ball tracking was lost.

FIGURE 7: White ball before beign hit

FIGURE 8: White ball after hit, with motion blur

Discussing with *Simone* we came up with the idea to check when the tracking of a ball has failed. So in this case by repeating ball classification and localization on the critical frame, it was possible in some cases to recover the position the of the lost ball. Then by assigning to the trackers the new position it was possible to recover the tracking. This solution may increase the computing times, especially with consecutive tracking losses but on the other hand it increases the robustness of the tracker CSRT. Sometimes the tracking is lost but our system doesn't detect it because the CSRT algorithm starts tracking the tip of the billiard cue instead of the white ball, in this case it might be a problem with the CSRT algorithm itself.

2.5 Drawing the game status *by Simone*

The last operation required by the system running in live mode is the task of drawing the game status from a top point of view, which is exactly the purpose of the Draw class. Moreover, the Draw class is one of the two classes that must be run at each frame, along with the ball tracker class, in order to keep the game drawing consistent with each frame of the video, thus we want to keep the drawing class simple and efficient.

The idea to generate a drawing is very simple: we want to place balls over a fixed background (which represents our billiard table) according to their position on the real table, also showing the class of each detected balls. In order to do this, the first problem we encounter is the perspective effect: how can we represent the game status from a top view when the camera is placed on the side of the table? This is where OpenCV library can help us: we can map the corners of the table to our game schematic drawing using the function *cv::getPerspectiveTransform()* which will compute the transformation matrix to map each point of the video frame to a point in the top view drawing. Of course we need information about the position of some points in the two images: for the drawing, we can easily retrive the information of the corners of the drawn table, while for the corners of the real table, we rely on those conveniently computed by *Alessandro's* class TableSegmenter: the corners are not only detected and passed, but also they are sorted in order starting from top-left one and proceeding in clockwise order. In this way, we can unequivocally associate each corner of the real table to the corresponding corner in the drawing image. At this point we have all the elements to effectively compute the perspective-correction matrix, which we can store in class instance, since (assuming the camera angle will not change) the matrix will be fixed and associated with

a specific game clip. This operation can be done just as an initialization step at the beginning of the system, preventing to slow down the computation at each frame to compute the matrix.

Moving to the core functionality of the class, that is to draw the balls over the playing field, we need to receive information about the position and the class of each ball. To this purpose, the Ball class is useful to store those informations: at each frame, the ball tracker class will update a vector of balls with their new position, and the drawing class will receive this information. Each ball element will also have a corresponding class associated, thus we can use this information to show the different class of each ball, coloring them accordingly. Using the already computed perspective-correction matrix, we compute each ball position in the drawing, and we put a colored circle in such position. To draw the trajectories, we can simply put a little dot over the position of each ball, and generate two images: one with trajectories dots and balls, returned at each frame, and another one with just trajectory dots which will be stored in class instance. At each frame we take the previous dotted image, we put the new trajectory dots, we update the instance image and we return this image with also the drawn balls. The last thing we want this class to do is to put the drawing over the video frame. To do this, we take our drawn image, we scale it to a certain ration (default 0.5) and we place it in bottom left corner of the frame. Draw class can be easily configured: a set of constants is used to define the background table drawn (which can be changed providing a path to image both vertical and horizontal), the padding used to align table corners, the drawing size, the scaling ratio, and the ratio to distinguish vertically or horizontally aligned tables. In Figure 9, Figure 10 we can see the outputs of the Draw class from game1_clip3 clip.

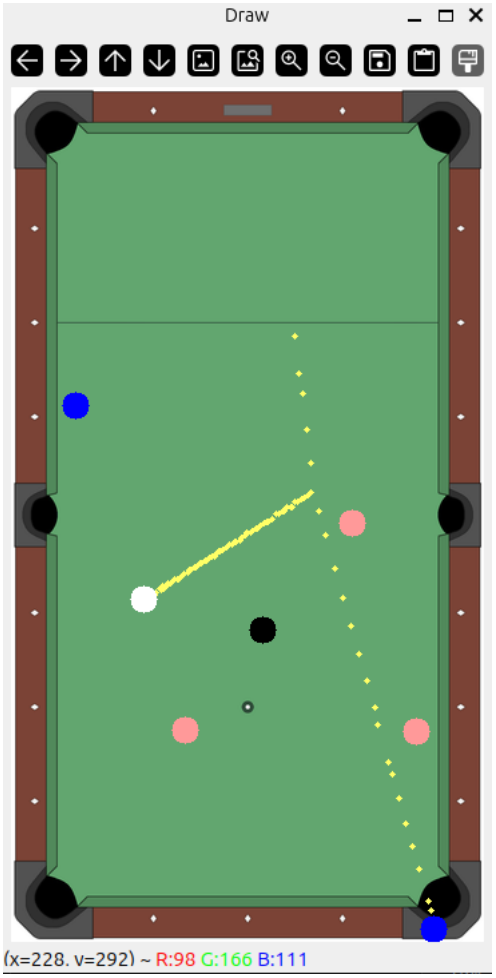


FIGURE 9: Output of the Draw class showing the status of the game. Blue balls should be the full balls while red balls should be the half balls. Dotted spots are the trajectories of the moving balls (from game1_clip3).

2.6 Running in live mode *by Simone*

All the classes described above have a public interface which is used while running the system in live mode. The main executable file will use those provided public functions to perform a simple sequence of operations, that is:

1. Open the video received as command line argument
2. Read the first frame to segment the table and detect corners
3. Detect all the balls in the first frame and classify them
4. Compute the perspective-correction matrix to be used in the drawing

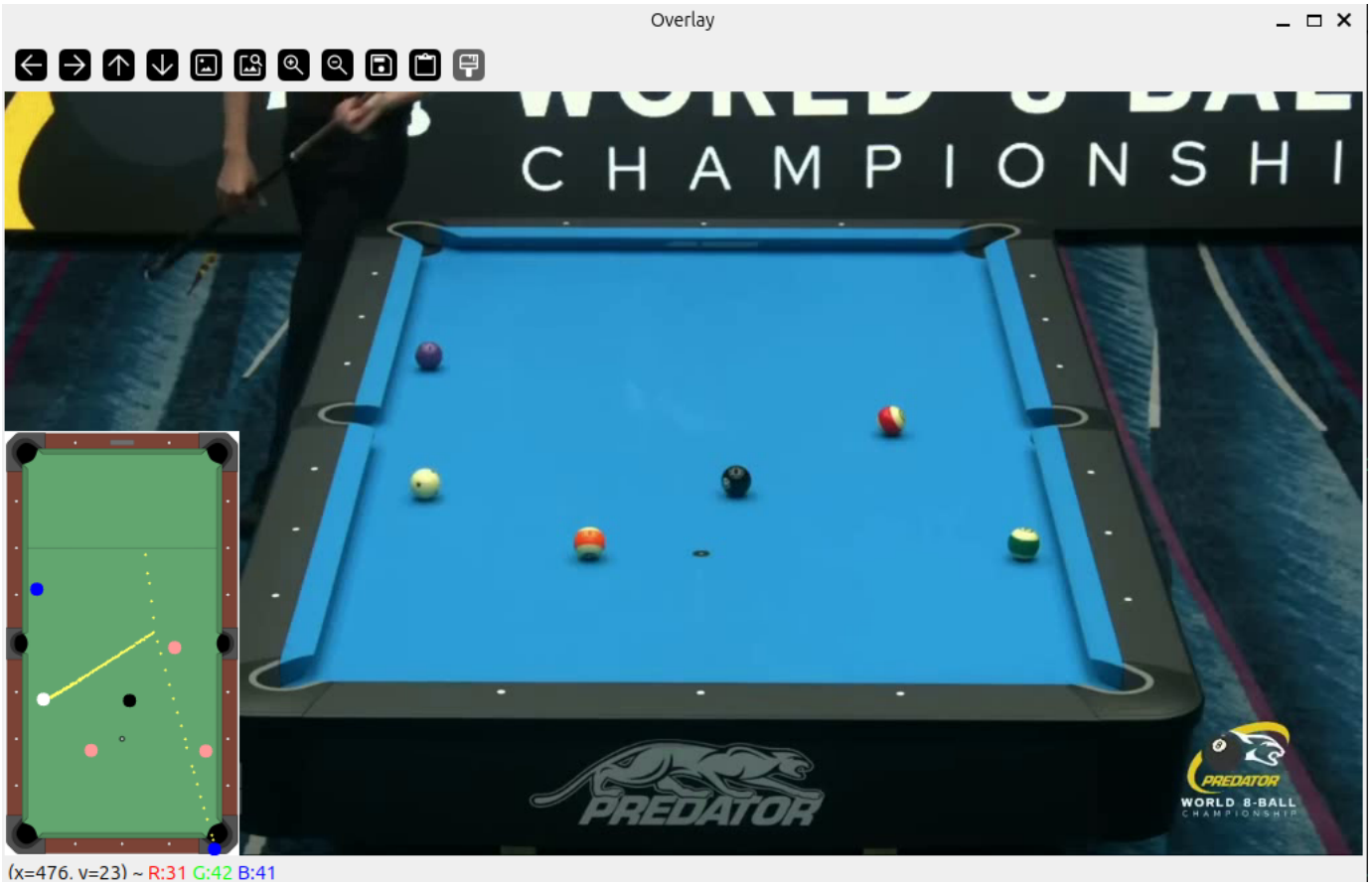


FIGURE 10: Output of the draw class showing the drawing overlapping the video frame (from game1_clip3).

5. Setup a tracker object to follow each ball in the frame
6. Loop over every frame to update the tracking of the ball and show the updated drawing

As opposed to my initial idea of having the detection and classification running at each frame of the video, my colleagues *Federico* and *Alessandro* pointed out that since the camera is in a fixed position, we can compute those only during an initial setup process, letting the system to focus on tracking the balls (which is, as discussed by *Federico*, very expensive in terms of computational power.). However, during our tests, we find a little compromise to make the system more robust: we want to perform a new detection and classification of the balls when one of the tracked balls is lost, initializing a new tracker objects over the new set of detected balls. In this way, even if the system is not perfect at the beginning of the video, it can recover a little bit during the game when the situation changes.

As already said, we want our system to be running in real time, and just to monitor the speed performance I added a few lines of code to track time spent by the system at each frame. Considering that each video has a frame rate of ≈ 30 fps, this leaves us with about $\approx 33ms$ of time to perform tracking of the balls of the system at each frame. During my test, I was able to reach mean frame elaboration time of $\approx$ 60ms on my system (running Ubuntu 24-04LTS with a 12 threads i7-10th gen. cpu), while on virtual laboratories this time rise to $\approx$ 120ms. Although the frame time is still above the target time, the result is still pretty usable.

Finally, the live mode executable will store the video of the clip overlapped by the game drawing in a path provided as a second command line argument.

3 Evaluating performance *by Simone*

As required by the project, the system should be able to generate an evaluation of itself matching its outputs against a provided set of ground truth elements. To this purpose an EvaluationMetrics class has been developed by me, as long with an offline executable file to run the system over the set of provided frames (first and last) and to produce the required output files (that are the bounding boxes and the masks). Over those generated files, the EvaluationMetrics class will perform the computation of the IoU and mAP metrics. The implementation of such class is described in the next section.

3.1 Evaluation class *by Simone*

The EvaluationMetrics class is not just a simple class to store mathematical functions, but rather a little more complex class that will provide full path names of required output files of the system. The class is adapted to work with the folder

structure provided in the dataset: a folder gameX_clipY with the subfolders masks, frames and bounding boxes. This folder structure is actively checked to be consistent when an instance of EvaliationMetrics class is created, using the path provided to its constructor. The same folder structure is also created (if not existent) for the output of our system, in a specified path provided as a second parameter of the constructor. After checking the consistency of the two folders, the class can be used (but also configured) to retrive the name of the files which are needed to perform the evaluation of the metrics, using a set of getter functions which will produce strings including directories and name of required files. We will pass those file names to the previously described classes to save the required output files.

The metrics required to be computed by our system are two: the IoU of the masks images and the mAP of the file containing the bounding boxes of the detected and classified balls.

Starting with the mask files, we need to compute the intersection area and union area of each class in the predicted and true images. Counting all the pixels with a certain value by hand seems a really inefficient way of doing this task, thus we can exploit OpenCV library which provides us with a *cv::countNonZero()* method to find the number of non zero pixels. This method for counting the pixels leaves us with two problems: first of all, the background must be considered as a class, and we need to compute IoU even if all pixels of this class have a value of zero in the provided mask. Secondly, we need to count only non zero pixels of a specific class at each i-th class iteration, without considering other classes. Solving the first problem is easy, because OpenCV library provides us a simple *cv::LUT()* function to remap values of an image using a Look-Up-Table vector, while the second problem seems to be resolved by taking the *bitwise_and* operations between the i-th class value and the value of the pixels in the image. However the real problem arise when we notice that in binary representation the value 5 shares some positive bits with value 1 for instance, thus making all pixels belonging to class 5 to belong to class 1. The best way that I have found to solve both those problems, is to use the *cv::LUT()* function to remap pixel values to the value of 2^i, where i is the original value corresponding to the class. In this way we obtain an image where pixel values do not share digits in binary code (i.e. 5 will be encoded as 100000 while 1 will be 10), and we can efficiently extract class values using the *bitwise_and* operation to extract only the pixels belonging to a specified class. Then computing the IoU of the class means counting the non zero pixels of the result of the and operation and dividing them by the number of non zero pixels resulted from an or operation (between the pixel values of the true and predicted masks, considering only the pixels belonging to the class).

Computing mAP between the bounding boxes files is not a so simple task. I have spent quite some time in developing this and I will try break down the process in details here. The mean average precision metric is usually computed using the bounding boxes of the detected objects: we compute the IoU of those, and we check if the IoU is above a certain threshold. If it is, the detected object is correct, and will count as a new True Positive element, while if it is below the threshold, it will be a False Positive. We also need to make sure that the True Positive element is of the correct class as the true object, or the object will be counted as a False Positive. Then we want to compute the precision-recall values for each detected object and save them in a structure that we will use to compute the average precision, obtained interpolating all points using using PASCAL VOC 11 Point Interpolation Method. Finally, we take the average precision of each class and compute the mean value.

So the first thing we need to do in out Metric class is to compute the IoU score of each detected object. Starting with two sets of bounding boxes, we take one box from each set and check if they are overlapping: since the boxes are given with top left corner and sizes, we need to check if one of the two boxes has its top left corner inside the corner+size area of the other box (both for x and y coordinates). Then, if the two are intersecting, we can compute the intersection as the area of the rectagle defined by the corner obtained combining the two bigger corner coordinates and as size the combination of the earliest ending side of each corner. The union area will be the sum of the two areas minus the intersected area. In order to associate an IoU score to each object, we need to match its bounding box with ne of the true bounding box, for example we might want to take be true box that gives the best IoU score to de predicted object box. In this way, each object in each class will have associated a IoU score and we can also decide to store if the predicted class is the same of the true class. I decided to store those values inside a tuple structure, so in this way we can order the tuples according to a certain score and count the number of True/False positive elements. As regard the order of detected object, since our system doesn't really provide a confidence score for a detected object, I decided to use the IoU score as a confidence score: the object with higher IoU scores will be evaluated earlier. After computing the total ground truth elements in the true bounding box file for each class, we can compute precision and recall values and store them in a map: in this way, we can find a precision value associated to a recall value, used as key of the map. In case of multiple precision values for a single recall value, we keep the best one. Now we can proceed at computing the average precision using the PASCAL VOC 11 Point Interpolation Method: starting from a recall value of 1.0 we scan the map starting from the end, and we take note of the best precision value so far. We store in a vector the best precision value at each of the 11 points, and we continue until we passed all the recall values of the map. At this point we have a vector of 11 elements, we take the mean of those elements and we compute the mean of them, which will be the average arecision of the class. Finally, repeating this procedure for all classes, we can compute the mean average precision.

The EvaluationMetrics class is now complete.

3.2 Evaluating the system in offline mode *by Simone*

Running the system to evaluate performace is really similar to running the system in live mode, with a few diffecrences. First of all, we dont need to perform tracking at each frame, since we only have to evaluate against a set of frames. Moreover we are not required to provide a drawing of the game, so we can skip to compute the perspective-correction matrix. We

need however to store the segmentation mask and the detected bounding boxes in a set of files, which we will use to evaluate the metrics of the system. As discussed in the previous section, the name of those files and the metrics are all managed by the EvaluationMetrics class, so we just need to run the system through each frame and evaluate the metrics, which will be conveniently saved in a text file. The runner requires as command line arguments the path to the gameX_clipY ground truth folder and the path to the folder where all the outputs will be stored. A copy of all the obtained outputs will be provided along with the source code.

3.3 Evaluation outputs *by Simone*

In this section are reported all the required outputs for each game clip. The mask values have been remapped with higher values in order to show them in this report (for this purpose, a simple executable *open-mask* is left in the project to conveniently show two mask files), while the masks obtained from running the system in evaluation mode are consistent with the mask provided in the dataset (but are difficult to see by human eye). For each game clip, the required metrics IoU and mAP are reported divided by each frame in Table 1. All the metrics are also reported in generated metrics.txt file after running the system in evaluation mode, in the specified folder along with masks anf bounding boxes files. The video of the elaboration and the final drawing are instead generated when running the system in live mode.

Game/Clip/Frame	mIoU	mAP
1/1/first	0.564453	0.617424
1/1/last	0.609188	0.65
1/2/first	0.484217	0.292045
1/2/last	0.579699	0.514773
1/3/first	0.808025	1
1/3/last	0.77418	1
1/4/first	0.694136	0.727273
1/4/last	0.785668	0.87987
2/1/first	0.515271	0.505682
2/1/last	0.552538	0.509091
2/2/first	0.412162	0.09375
2/2/last	0.370761	0.0551948
3/1/first	0.622827	0.563636
3/1/last	0.551338	0.480303
3/2/first	0.500408	0.420455
3/2/last	0.601429	0.623106
4/1/first	0.624688	0.645455
4/1/last	0.389094	0.113636
4/2/first	0.469273	0.323864
4/2/last	0.643219	0.727273

Table 1: Computed values for mIoU and mAP for each frame in each game clip.

As we can see, the metrics are not always excellent in each frame, however as discussed, the main objective was to provide a system which can work in very different kind of environments: missing a black or white ball in a certain situation due to illumination has a drastic effect on the metrics. However the system seems to be enough stable in detecting the balls in the playing field, even if sometimes, the tracker provided seems to stick multiple balls together or losing some fast moving balls.

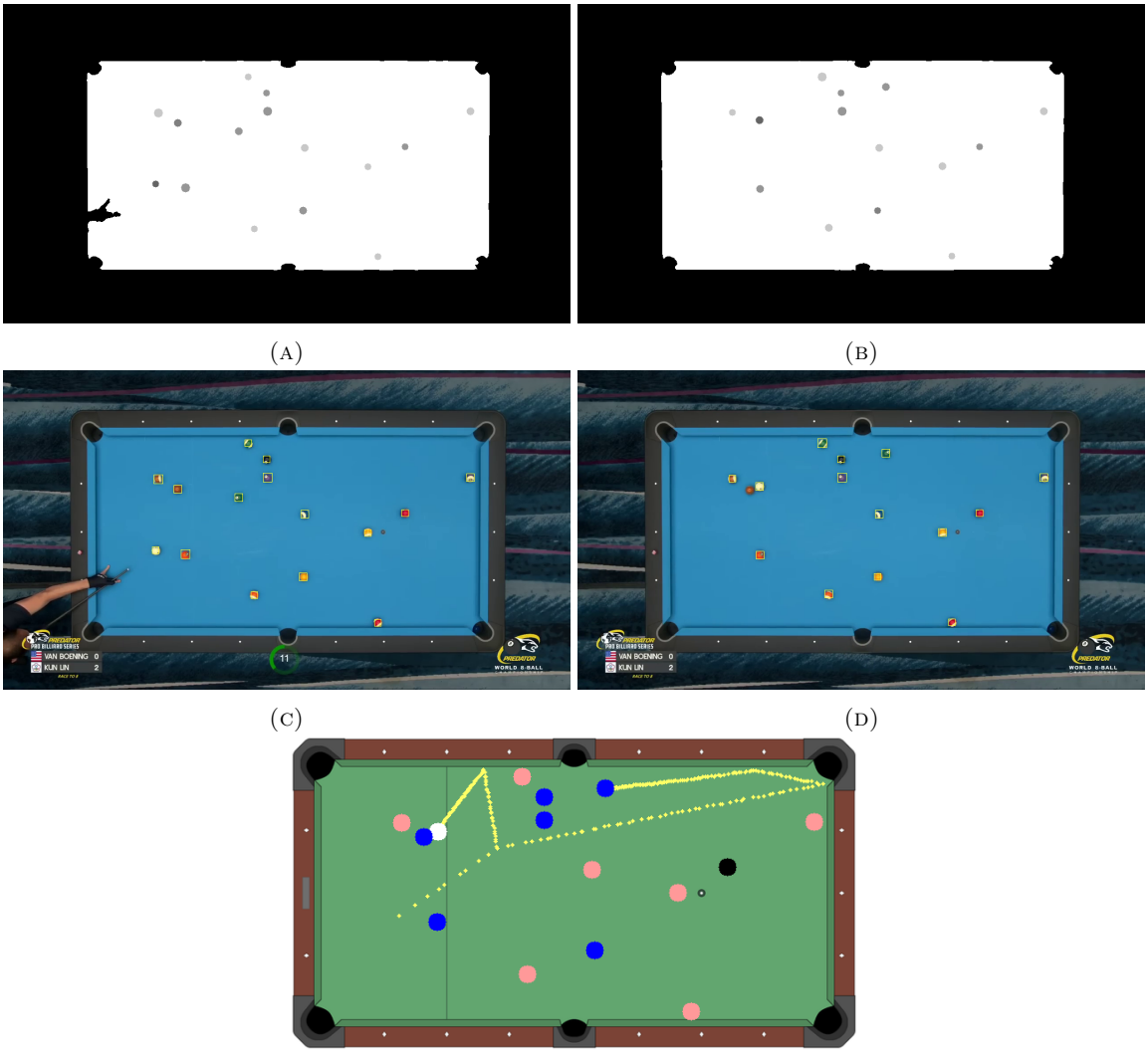
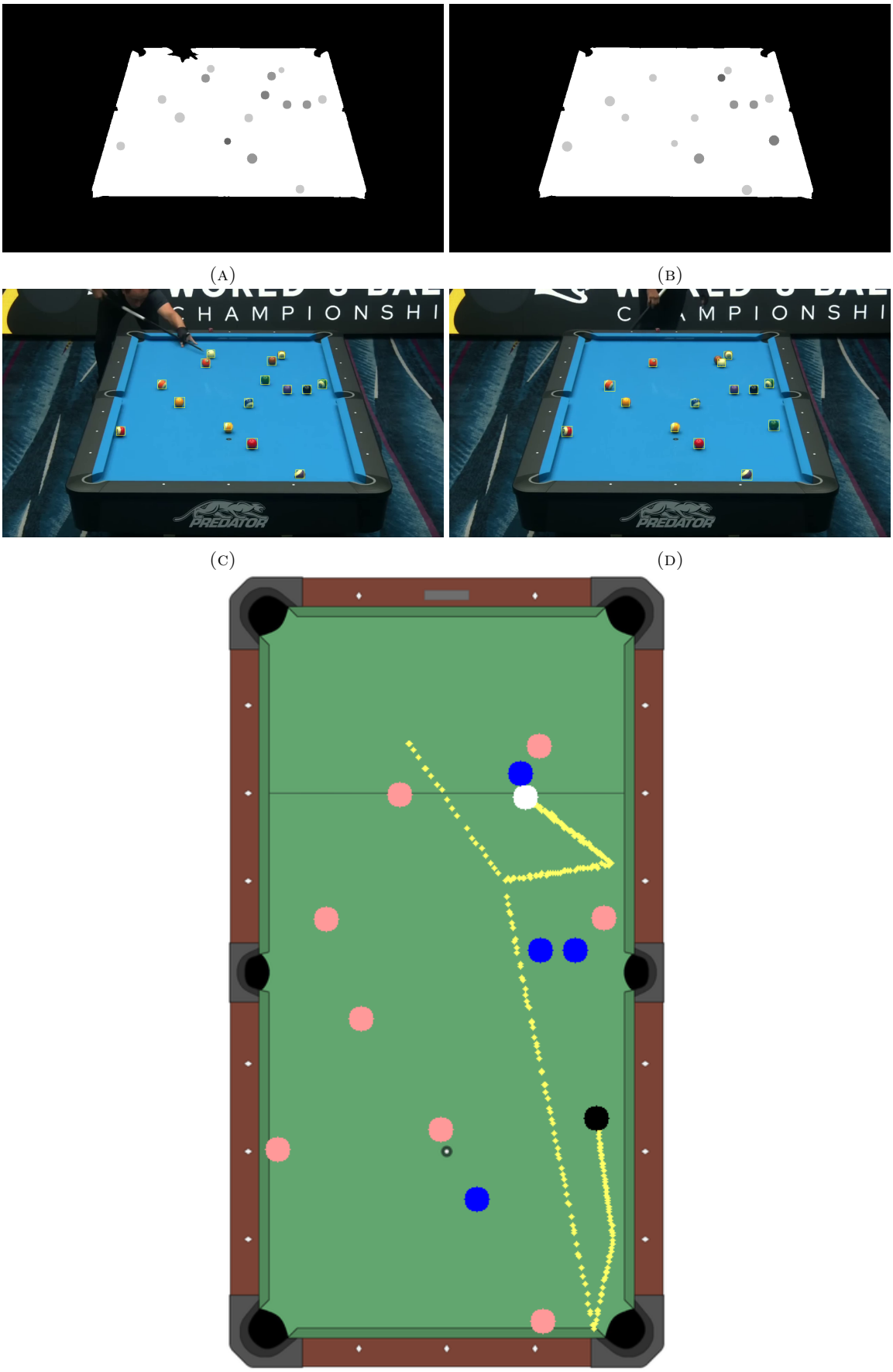


FIGURE 11: Output for game1_clip1: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.



(A) (B)

(C) (D)

(E)

FIGURE 12: Output for game1_clip2: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

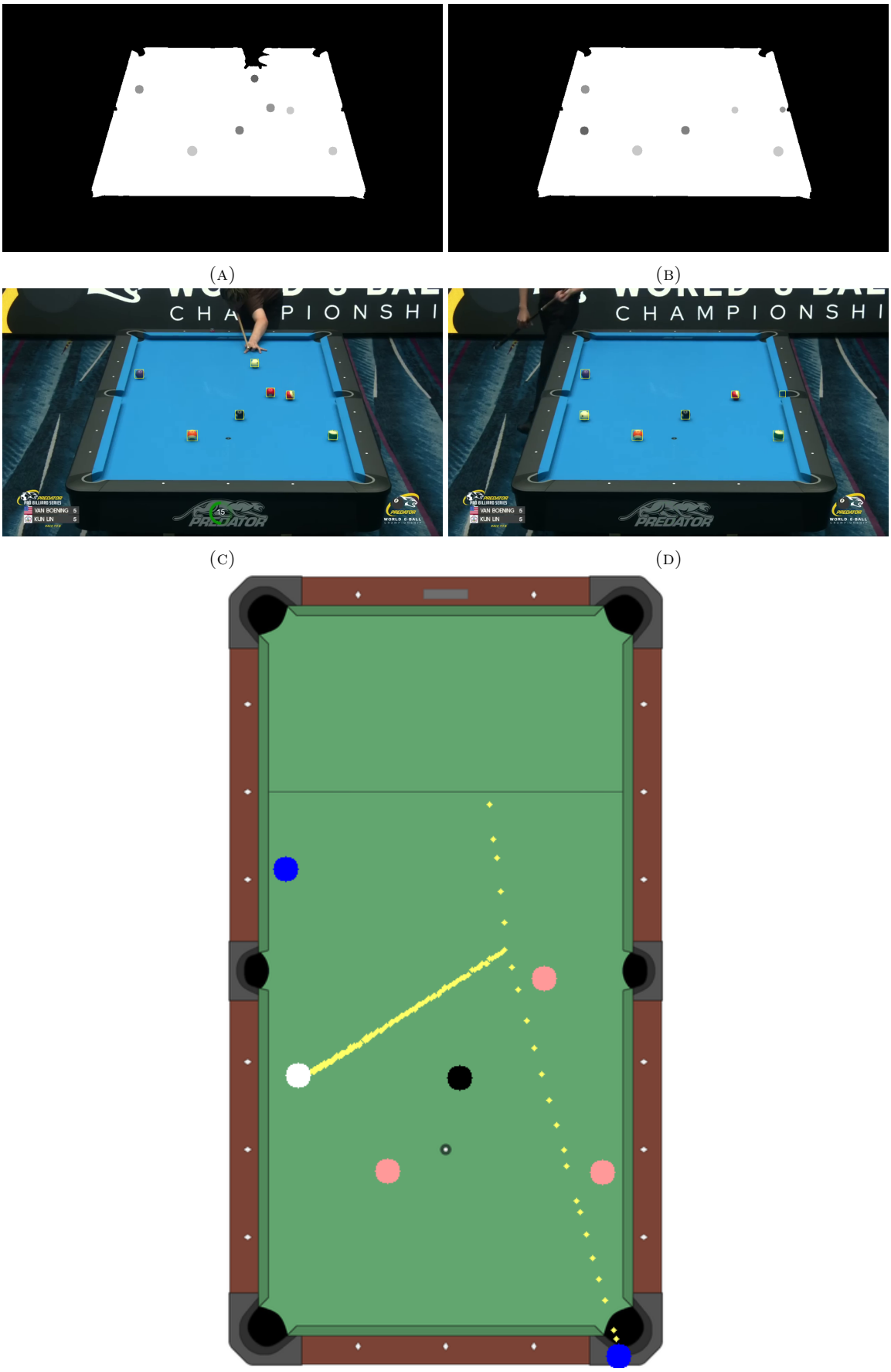
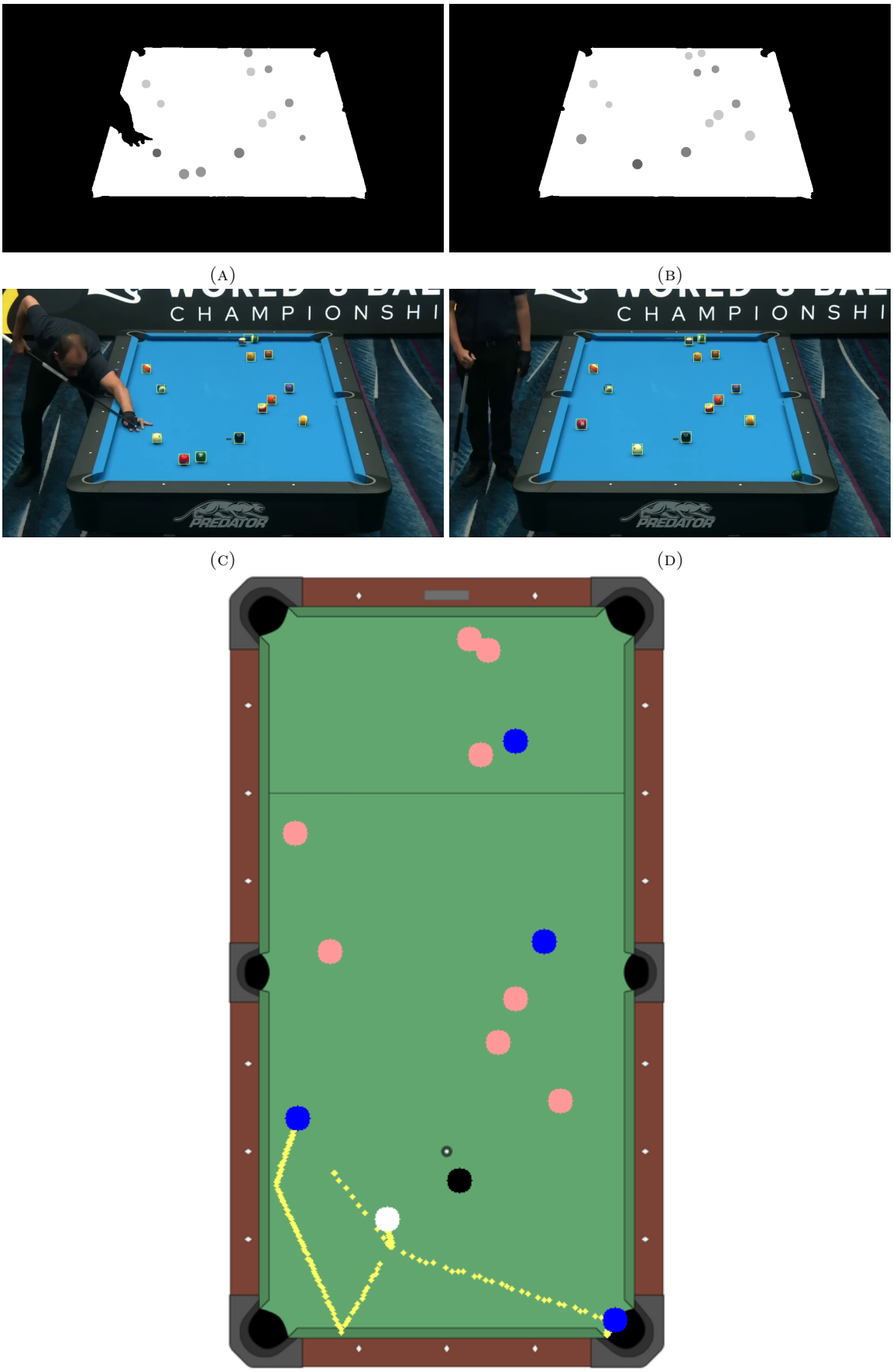
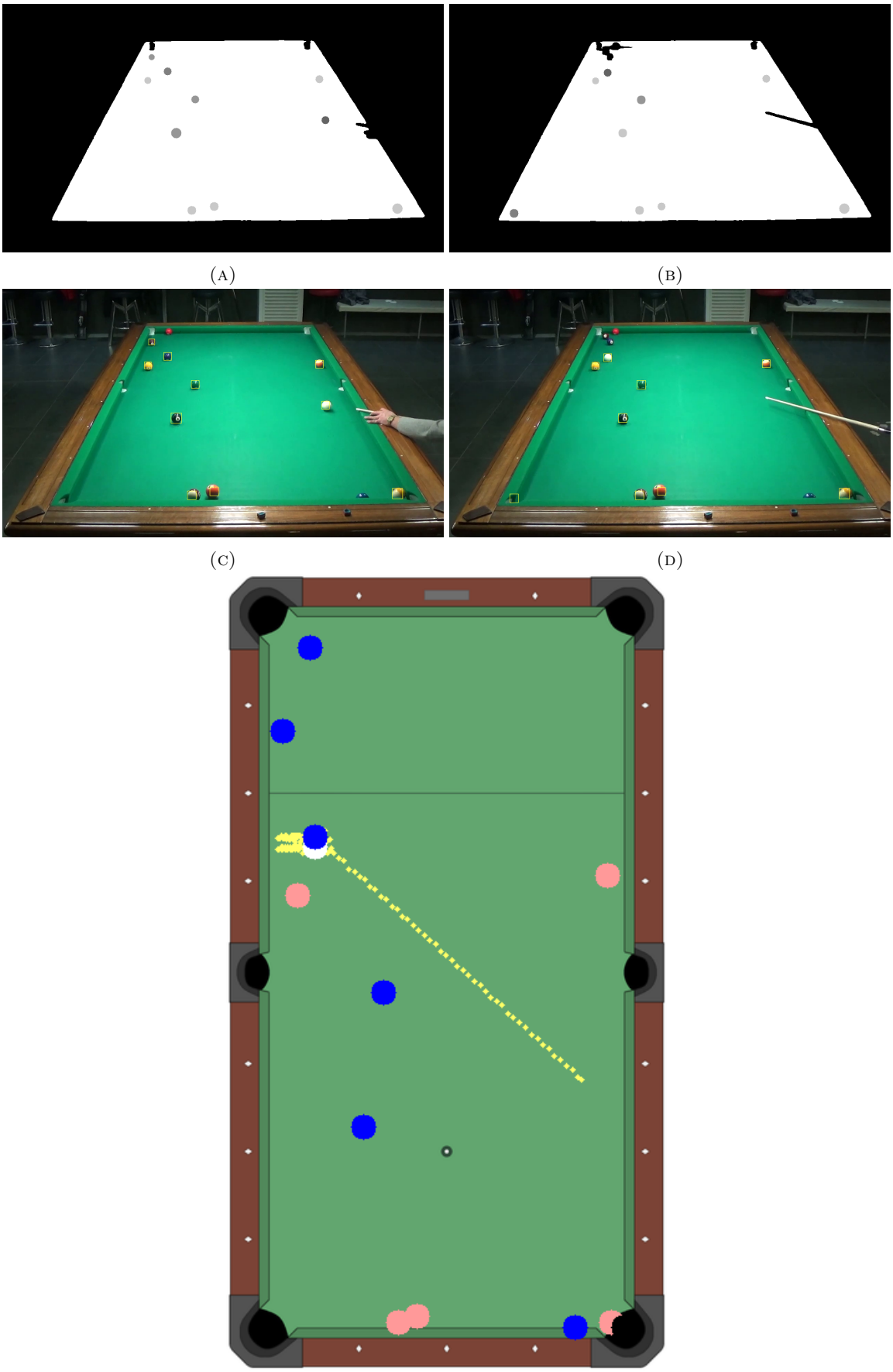


FIGURE 13: Output for game1_clip3: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.



(A) (B) (C) (D) (E)

FIGURE 14: Output for game1_clip4: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

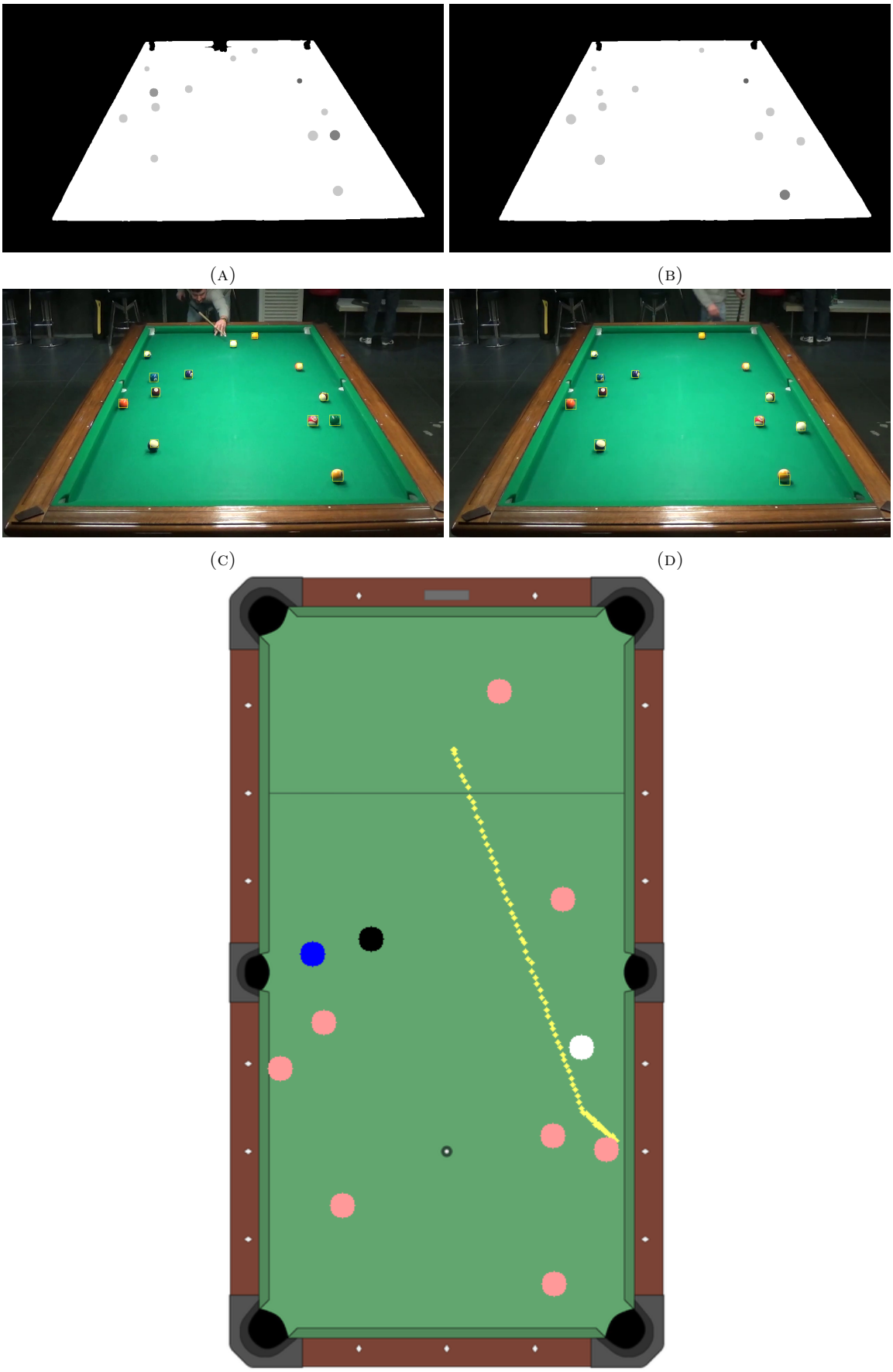


(A) (B)

(C) (D)

(E)

FIGURE 15: Output for game2_clip1: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

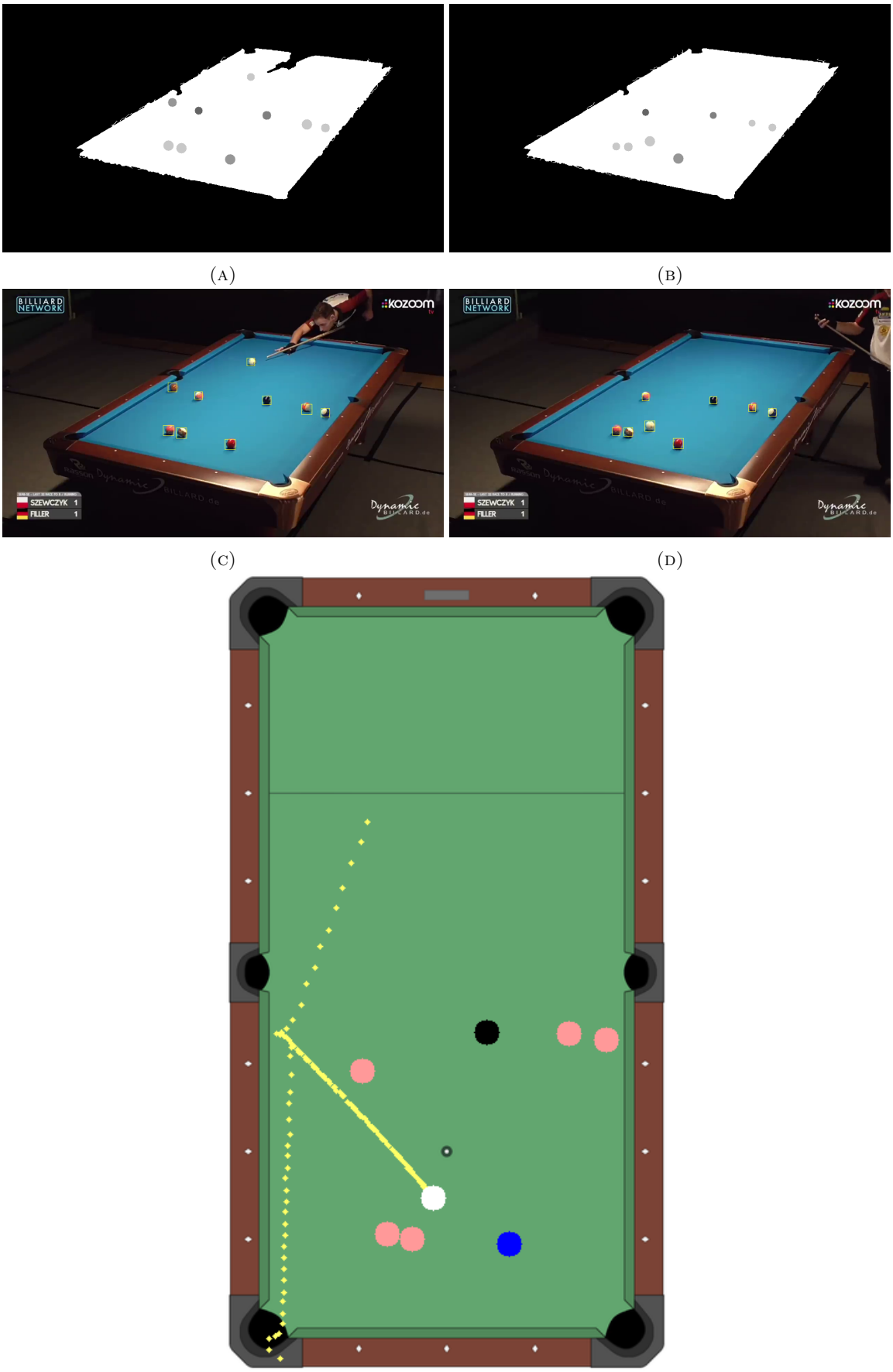


(A) (B)

(C) (D)

(E)

FIGURE 16: Output for game2_clip2: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.



(A) (B)

(C) (D)

(E)

FIGURE 17: Output for game3_clip1: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

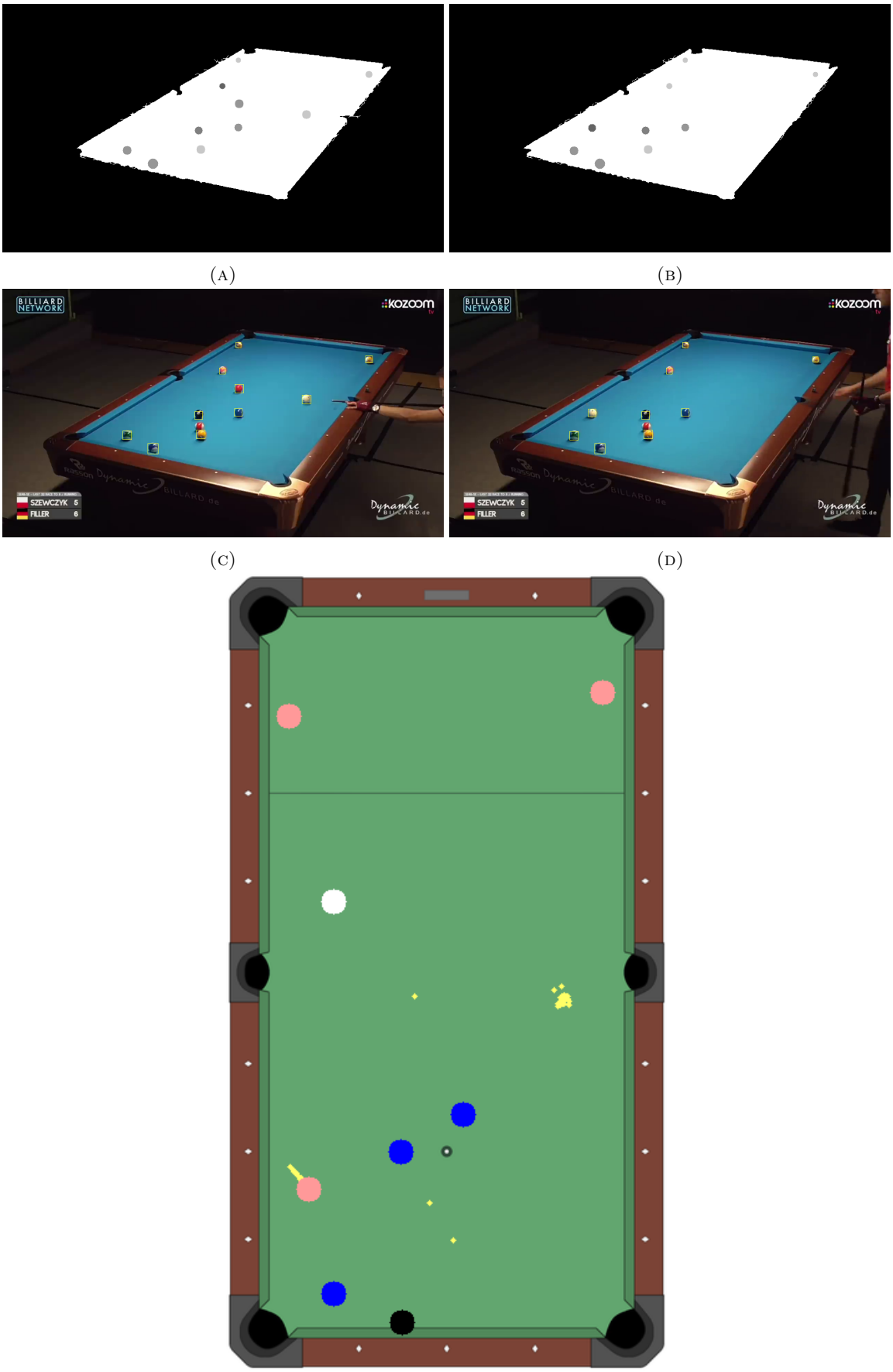


FIGURE 18: Output for game3_clip2: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

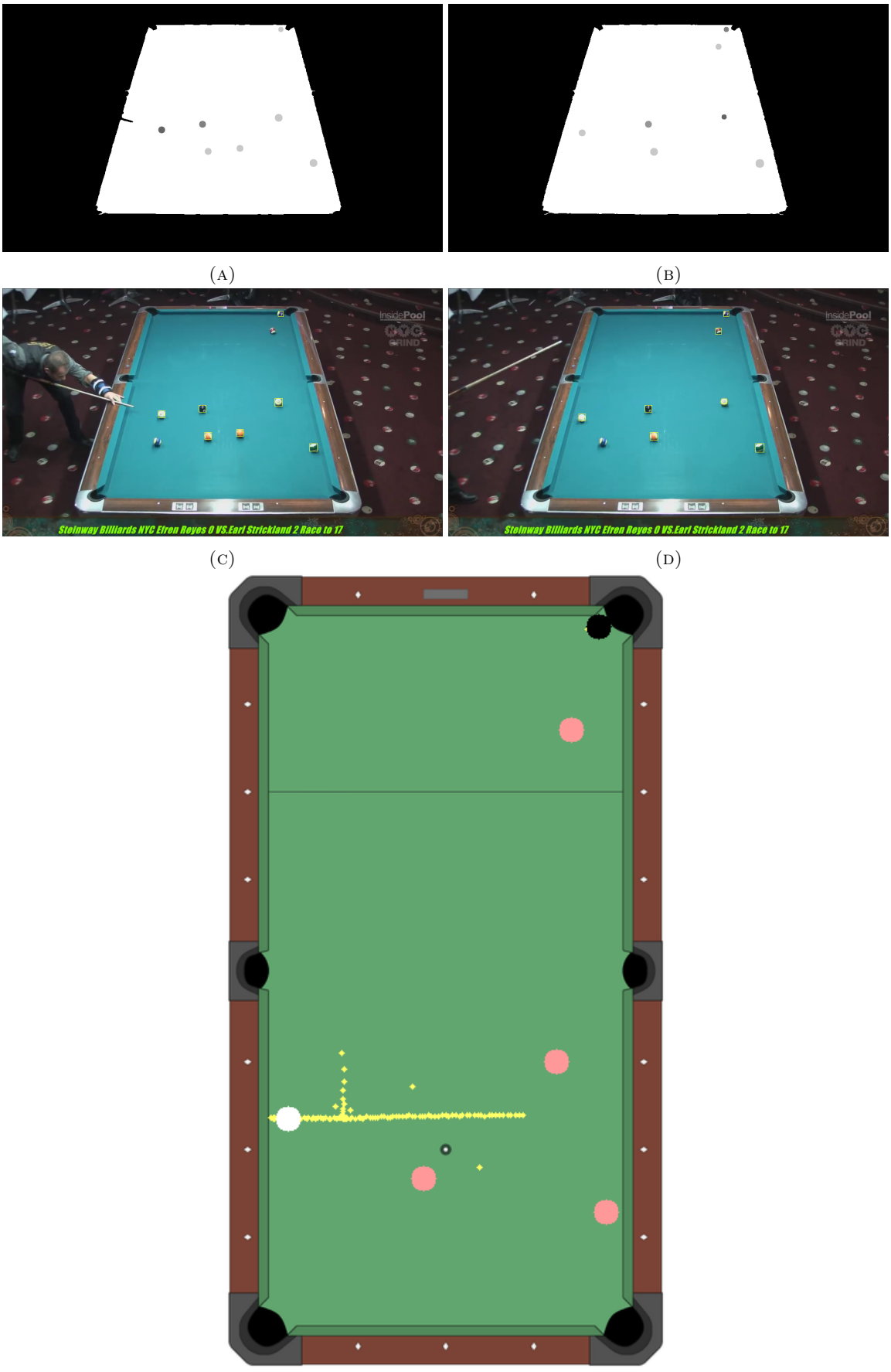


FIGURE 19: Output for game4_clip1: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

FIGURE 20: Output for game4_clip2: a) Mask of first frame, b) Mask of last frame, c) Detected boxes of first frame, d) Detected boxes of last frame, e) Drawing result with trajectories.

4 Final considerations *by Simone*

Having reviewed the output of our system, we can say that even if the system is not perfect, we are satisfied with the result achieved. The system works and can provide useful results, and probably can be adapted to each situation to improve the obtained results. We did never expect to have a perfect system in each one of the clips provided, but in some clips it seems to be almost perfect. Obviously some improvements are required, especially in the classification and in the tracking of the balls, maybe defining a new kind of tracker extending the one provided by OpenCV. Another thing to consider is that nowadays a lot of Deep Learning architectures have been developed that could easily perform this kind of task with a great accuracy, however running those networks is not always suitable and requires a lot of training data. The system we developed proves that the same task can be implemented without using any deep network at all, but just exploiting the conventional functions provided by one of the biggest library for Computer Vision tasks.

If you read so far, thank you for your attention.

Working hours

Total number of working hours is roughly reported in Table 2, but each member of the group spent some time thinking abut approaches, problems and solutions even when not actively working over the project.

Member	#hours
Federico	108
Alessandro	106
Simone	110
TOTAL	324

Table 2: Number of hours spent on the project by each member of the group