Billiard ball CV report

github repository to see all the history:
https://github.com/ma-jafari/CV-2024-final-project

Authorship of various cpp files (and associated hpp):

Matteo De Gobbi: field_detection.cpp, ball_classification.cpp, ball_tracking.cpp (main.cpp the command line argument parser and table segmentation)
Alessandro Di Frenna: ball_detection.cpp, minimap.cpp
Mohammadali Jafari: measurements.cpp, most of the main.cpp, (Initial parts of ball classification)

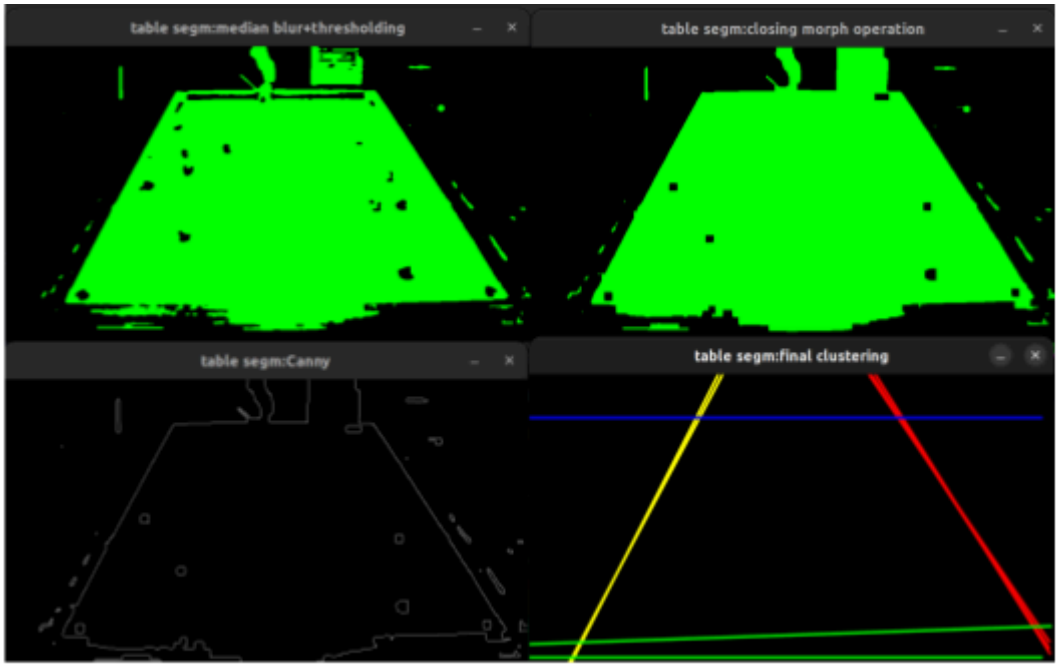
Usage and command line arguments (Matteo De Gobbi):

The program takes one positional argument and 4 optional command line arguments:

- The optional command line arg -h (or equivalently --help, --usage, ?) display the following help screen:

```
Billiard Analisys
Usage: BilliardAnalysis [params] path

        -?, -h, --help, --usage (value:true)
                print the help message
        -i, --intermidiate
                show intermediate steps of the algorithm
        -s, --save
                save the output video on a file
        --savepath
                path of the directory where the output video wil be saved if -s or --save are used

        path (value:../data/game2_clip2)
                path to the directory of the clip
```

- The positional mandatory argument path should contain the path to the directory that contains the clip and the other data structured in the same way as the files provided as samples
- The optional argument -i (or --intermediate) that will show to the user the outputs of some intermediate steps of the algorithm, this is useful for debugging purposes or to understand how the different steps in the pipeline affect the output



Example of some of the outputs when the program is ran with --intermediate

- The argument -s (or --save) is used if we want the program to save the processed video to a file
- The argument --savepath is used in case also -s is provided and it stores the path of the directory where the processed video will be saved

Also to use the program the table.png must be in the directory above where the executable is, for example if the executable has the path project/build/BilliardAnalysis.exe the table.png should be stored inside the project directory.

To run the code just unzip the zip file then
> mkdir build && cd build
> cmake ..
> make
>./BilliardAnalysis
(after ./BilliardAnalysis you should add the command line args explained previously)

Table segmentation procedure(Matteo De Gobbi):

We decided to exploit the fact that the viewpoint doesn't change during the clips, this allows us to use a slower but more precise pipeline for the table segmentation since it will only be executed once per video. For this reason we always segment the table using the first frame of the video.

To segment the table we defined a function called detect_field which takes as input an image, (that will be the first frame of the video) and returns a Vec4Points which is just a typedef of cv::Vec<cv::Point2i, 4> that will contain the four vertices of the table.
This diagram outlines the steps for segmenting the table, now we will explain the steps one by one and the reasons behind them.

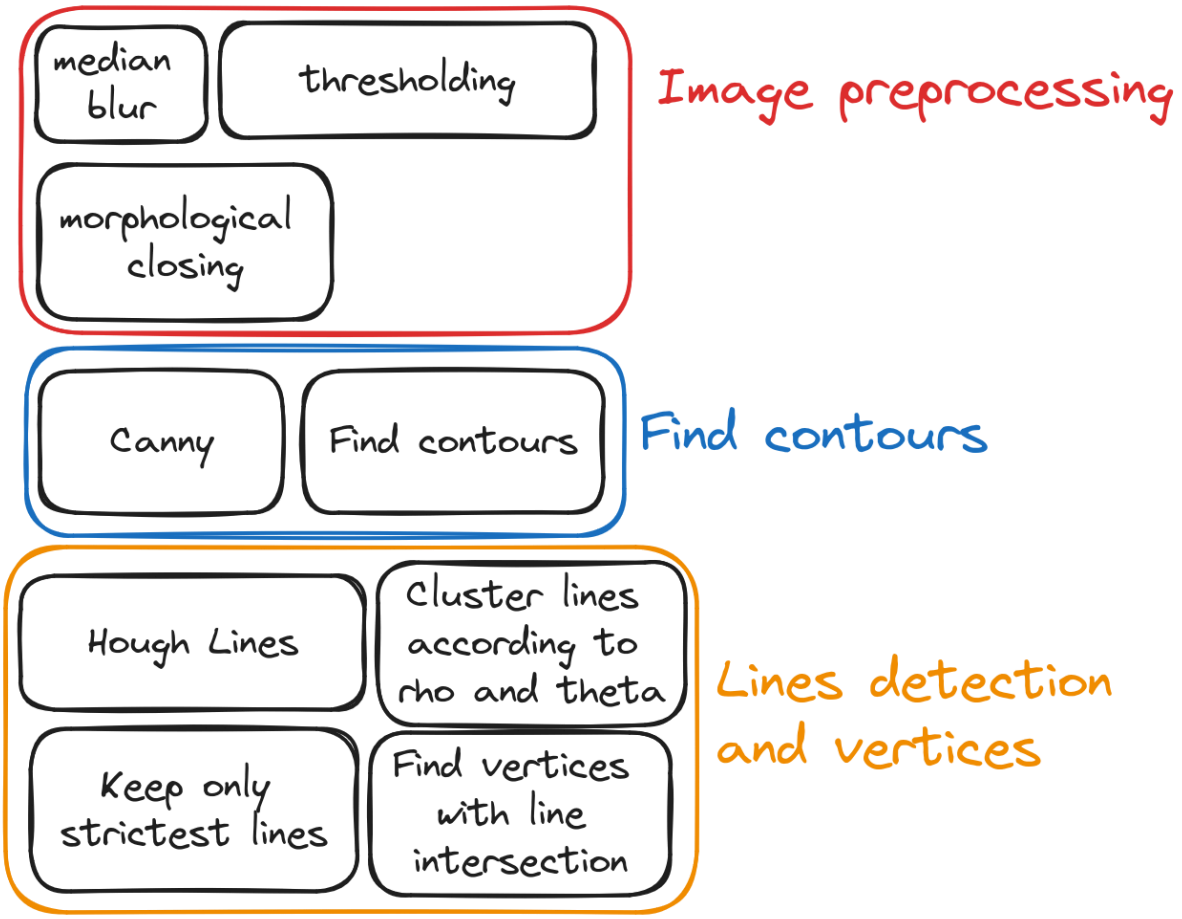


Image preprocessing

Before proceeding with the rest of the algorithm we apply some preprocessing to remove noise and unwanted features.
The first step is applying a median blur on the original image to remove noise, we chose median blur instead of gaussian blur because we saw empirically that it works better. Normally using median blur could be much slower than gaussian blur but since the segmentation improves a lot and we only blur the image once per video we decided to go with median blur.

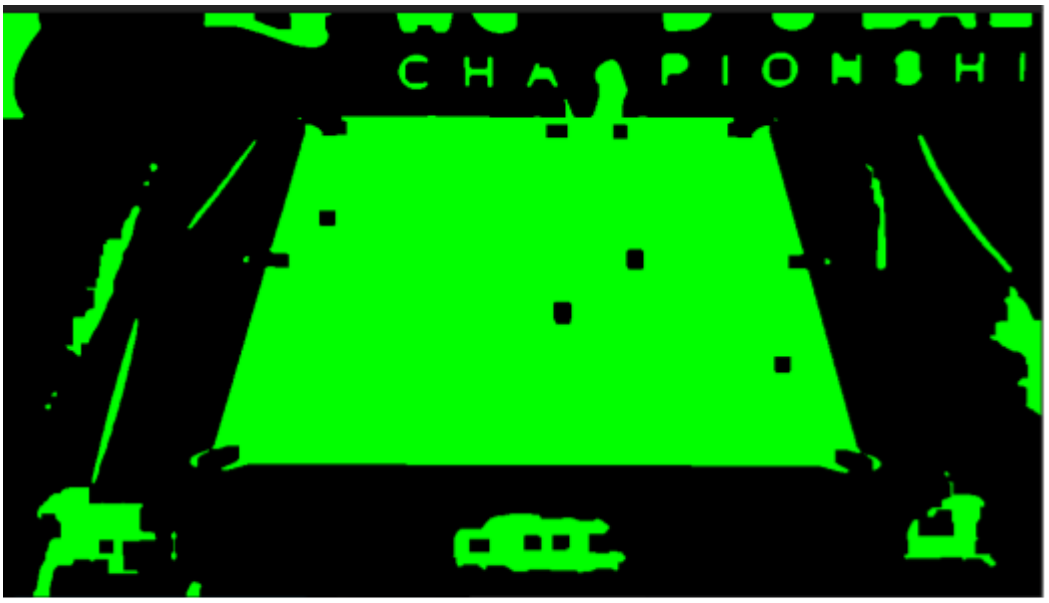
Blurred image after applying median blur

After the blurring we apply a threshold of 100 on the green channel, this is not a high threshold but it easily removes a lot of the background and other objects that are not needed for the table segmentation.

Example after thresholding: a large part of the background gets removed

Since we will use the hough transform in the line detection part of the pipeline it helps to have smooth straight lines and to remove small imperfections due to shadows, lightning, arm

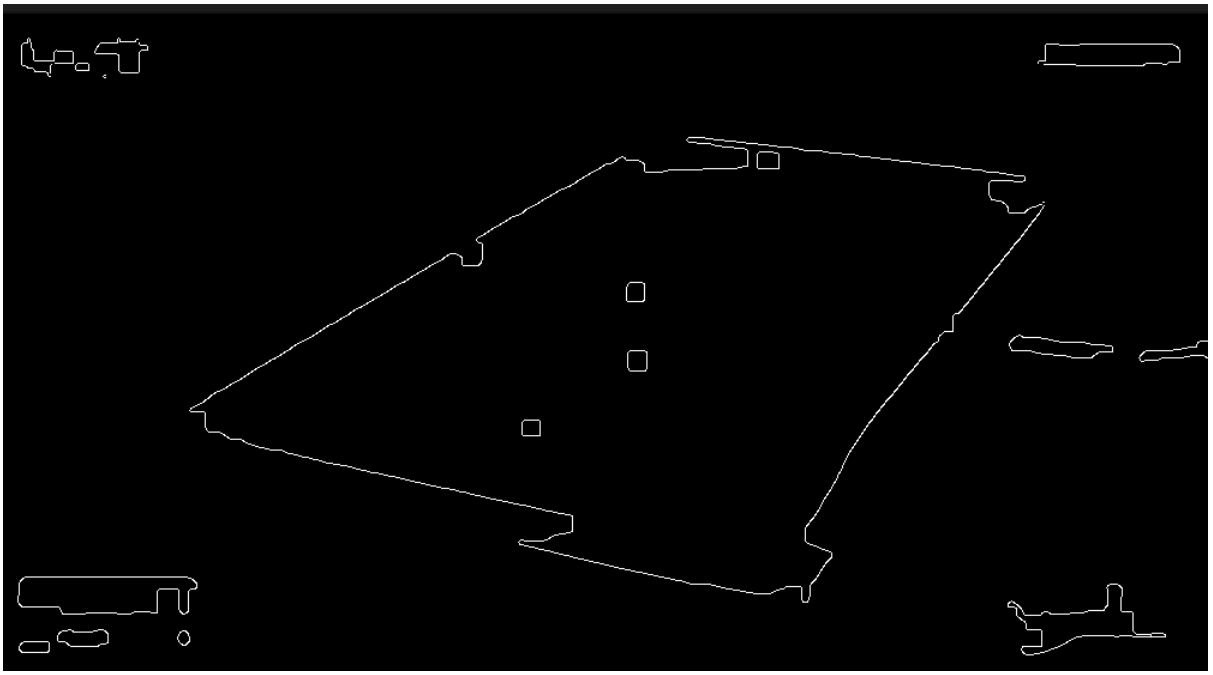
of the player and balls. To achieve this we can use the closing morphological operation (dilation followed by erosion) that allows us to close holes without changing the size of the objects. We use a 13x13 MORPH_RECT structuring element that allows us to close big holes.



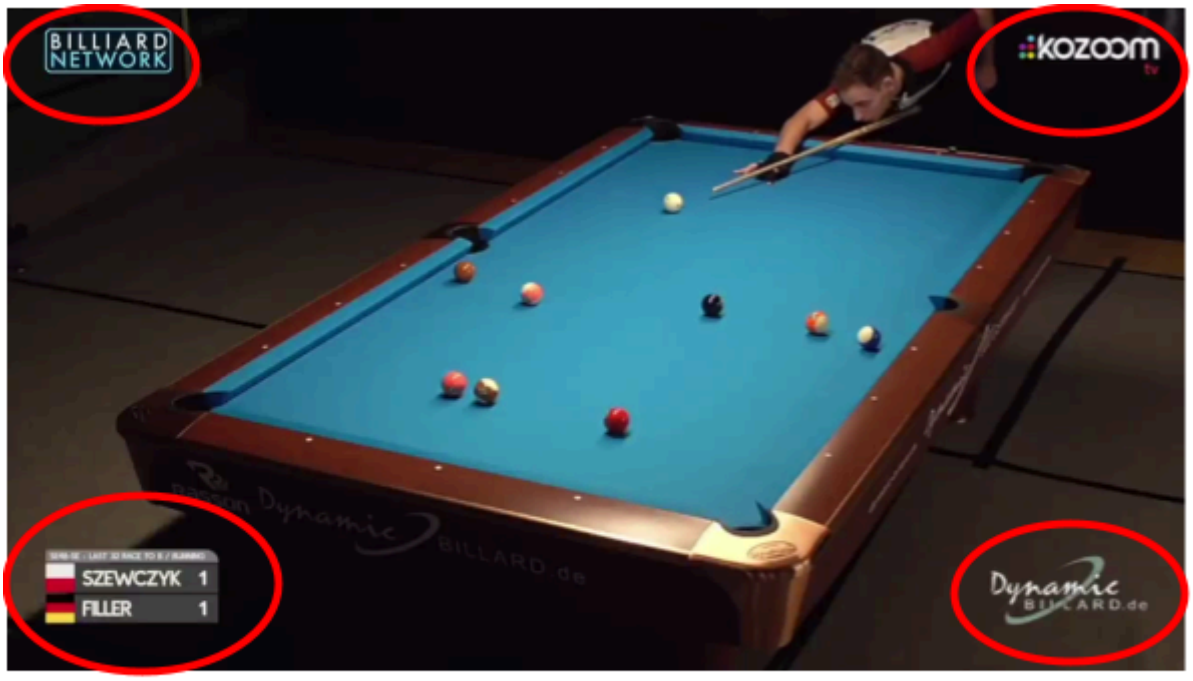
After closing morphological operation, the breaks due to the shadows/illumination get fused together

Detect edges of table

This step is carried out after the preprocessing and consists in detecting the edges in the images, this is a necessary step in order to be able to apply HoughLines in order to detect lines in the images using the Hough Transform.
To detect the lines we first apply the Canny edge detector:



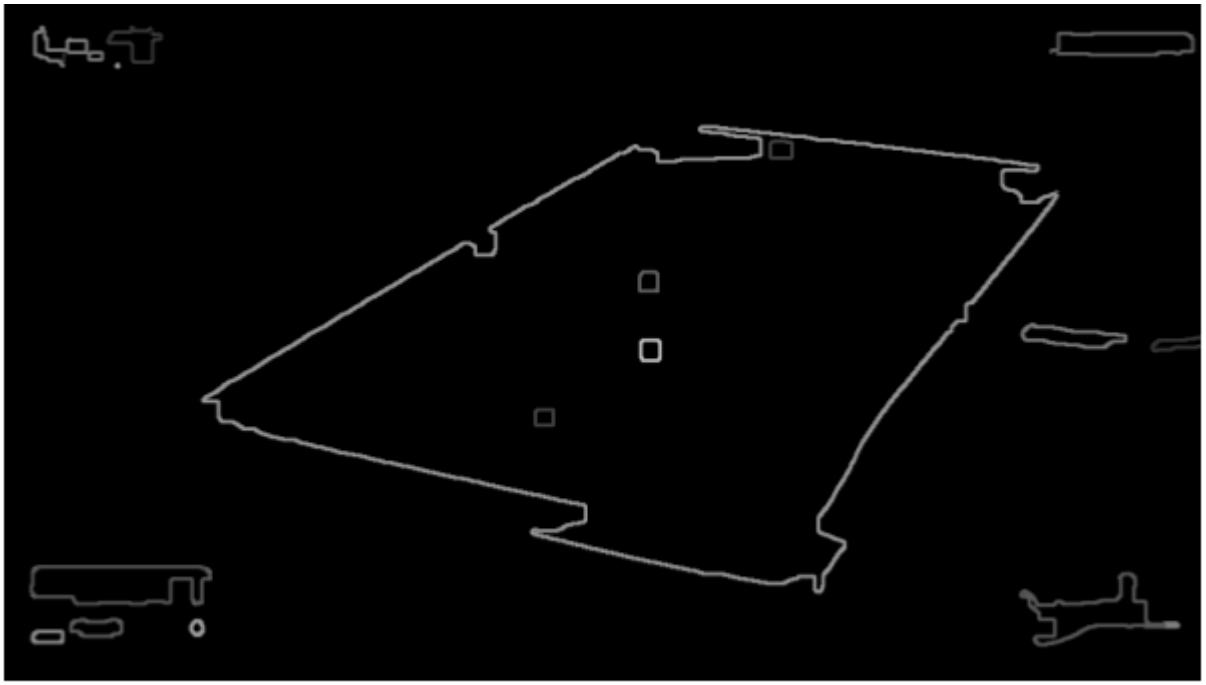
After the Canny edge detection usually in the image we have left the edges of the table, some background edges due to carpets, chairs and other objects and most of the time some edges due to the overlays of the game footage:



Some of the edges in the previous image are due to these highlighted game overlays

Canny makes very thin edges thanks to its non maxima suppression step but since we want to apply HoughLines to detect the lines we want to have thicker edges.

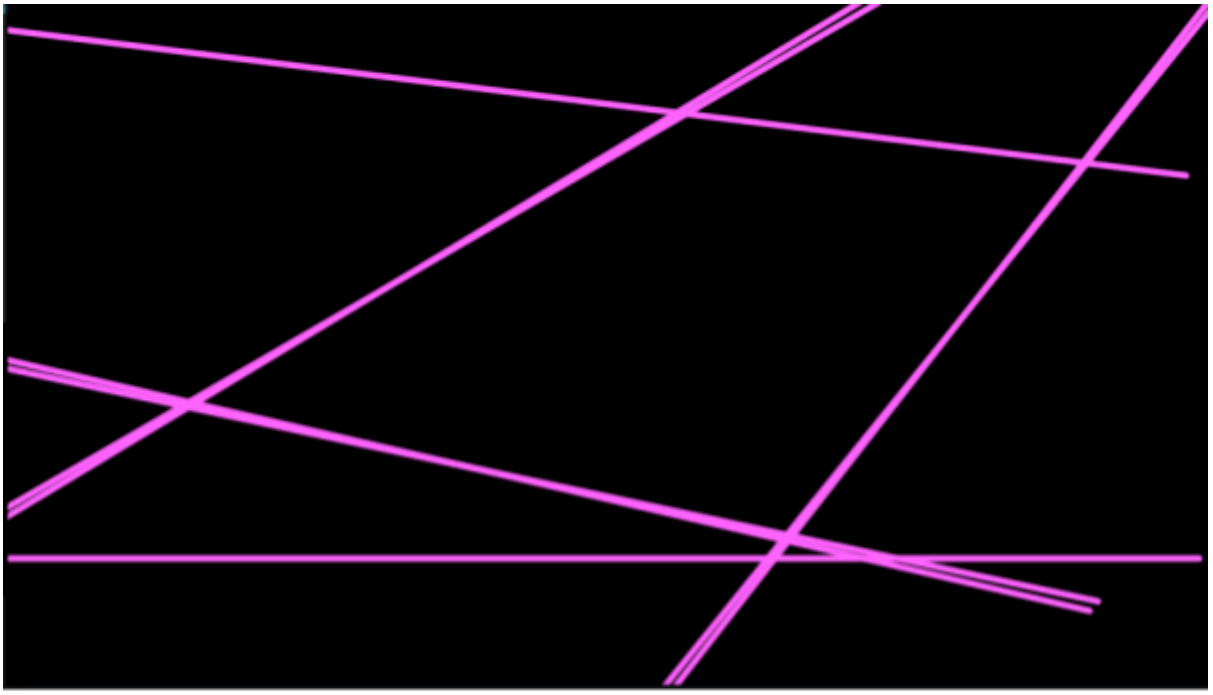
To obtain this we could apply a dilation step but it has the problem of fusing together corners where lines meet (the same problem applies to the closing morphological operation), instead we use find_contours and draw_countours to draw the edges again but thicker without applying morphological operations.



Result after applying draw_contours

Detect lines and vertices

We want to detect the lines in the image to be able to segment the table. We use the HoughLines function by opencv using a quantization step for the thetas of 1 radian and for the rho of 1.3 pixels, using 1.3 instead of 1 as the resolution of rho allows the HoughLines function to detect the lines even if the points are not perfectly aligned, granting some additional robustness to the detection. We also have to be careful of not choosing a quantization step that is too big otherwise we will detect spurious lines.

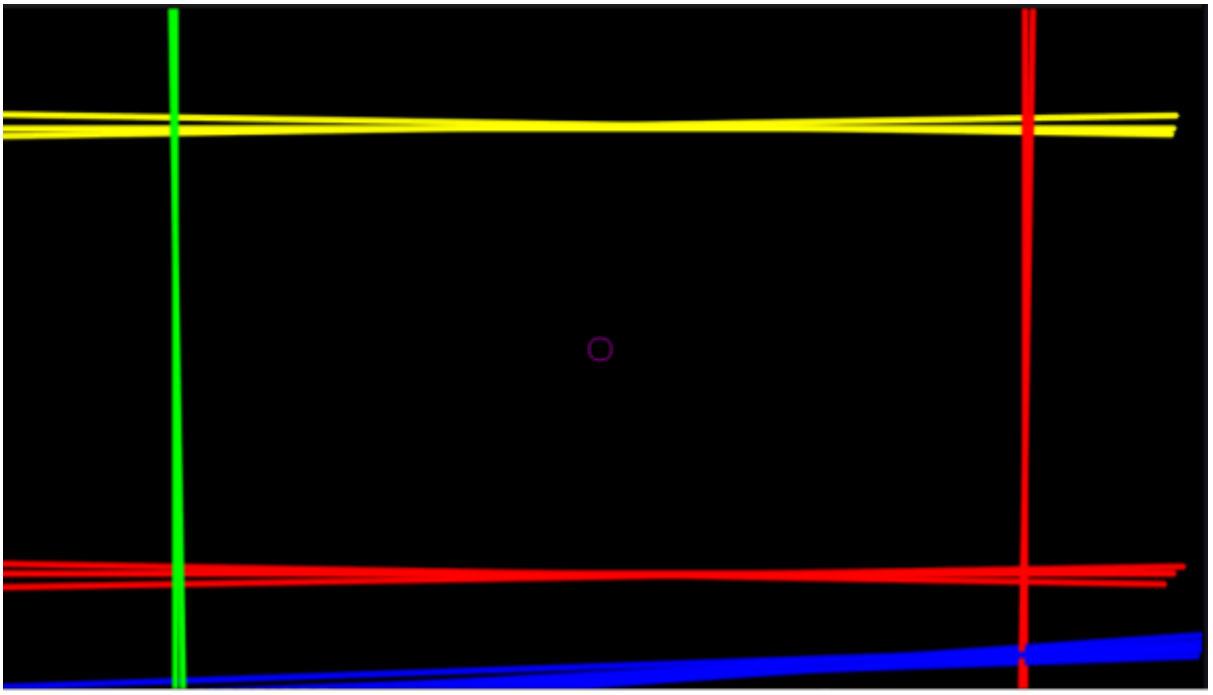


Lines detected by using the Hough transform

We can see from the previous image that we detect multiple lines for each edge of the table, sometimes we also detect some other lines due to other features of the images (like the carpet , the player or the background).
Our objective is to get the vertices of the table, we can get the vertices by intersecting the lines.
To intersect the lines we would need to only have one line per edge so we need a way to cull redundant lines (and also spurious lines that are not table edges).
A possible way to get one line per table edge is to select one line on the left, one on the right, one on the top and one on the bottom.
This approach has some notable problems, namely:

- We need to define what being the “leftmost line” means
- For some configurations of the table (diagonal perspective) the leftmost line could be also the top one (or bottom and right etc)
- We are more susceptible to spurious lines that are usually found outside the table due to background objects.

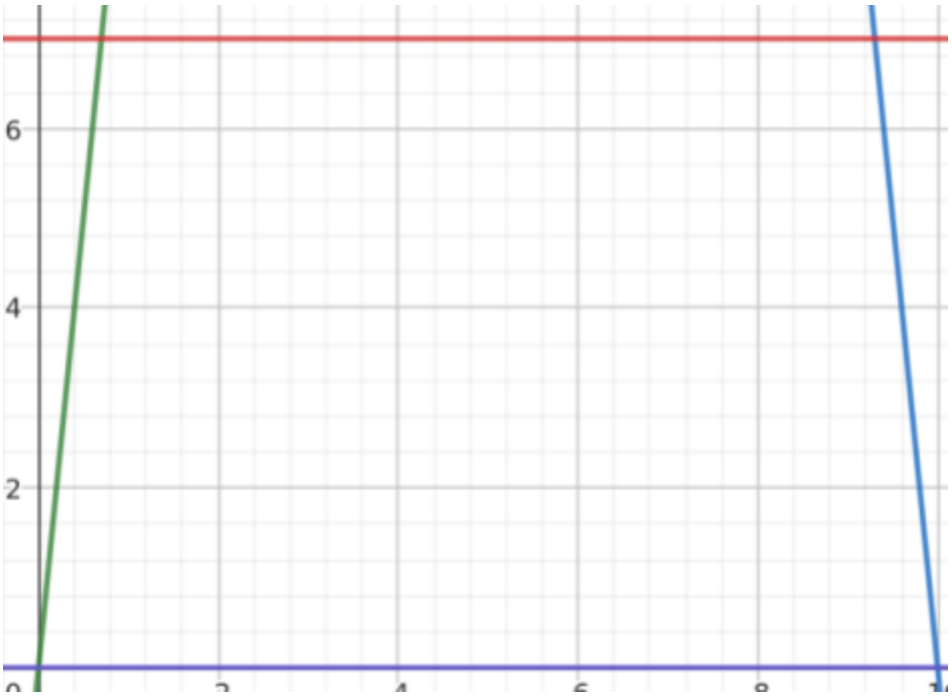
To avoid these problems we developed another approach, we can use the Hough space representation for all the lines that we have already computed and cluster the lines in the Hough parameter space.
For this clustering we can use the KMeans clustering function provided by opencv but we have to be careful: at the end we want 4 clusters, one for each side of the table; but we cannot immediately cluster the lines in 4 different clusters otherwise it’s likely that one of the spurious lines will be in a cluster by themself. This happens because spurious lines usually have a different orientation from the table lines, we can see this problem in the next image.



Having K=4 leads to wrong clustering due to the spurious lines

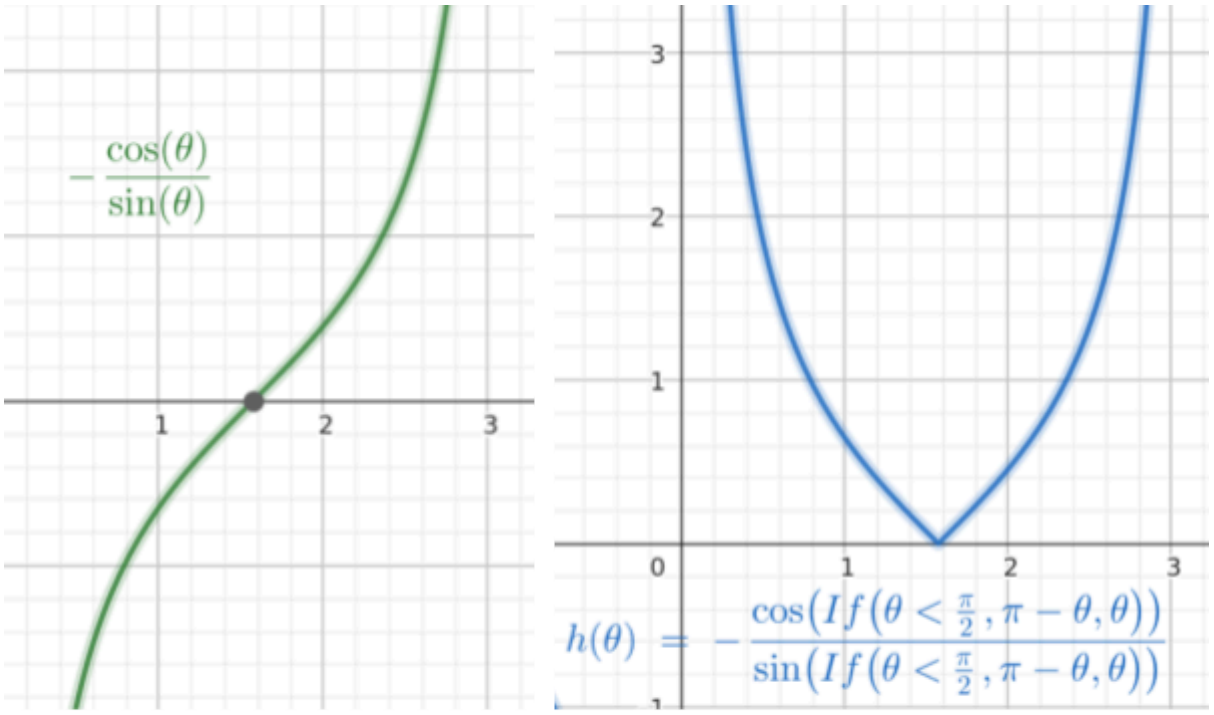
To solve this problem we can do a first clustering with K=2 using the theta parameter of the lines; this will divide the lines in two clusters based only on their orientation. Then we do a subsequent clustering splitting the two previous clusters using the rho parameter in order to separate parallel edges of the table according to their position in the image.

Theta will be approximately the same for parallel lines, this is because the angular coefficient is computed as $m = -\frac{\cos(\theta)}{\sin(\theta)}$ but this is not enough because parallel lines in real life get projected by the camera and are not parallel in the image.
This makes some lines that should have the same angular coefficient have opposite angular coefficients e.g. in the next plot the green line has m=10 and the blue line has m=-10, if we want these two lines to cluster together we can cluster using the absolute value of their angular coefficient.

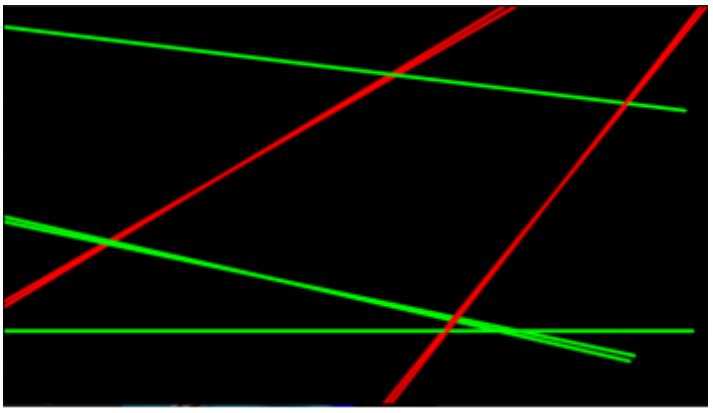


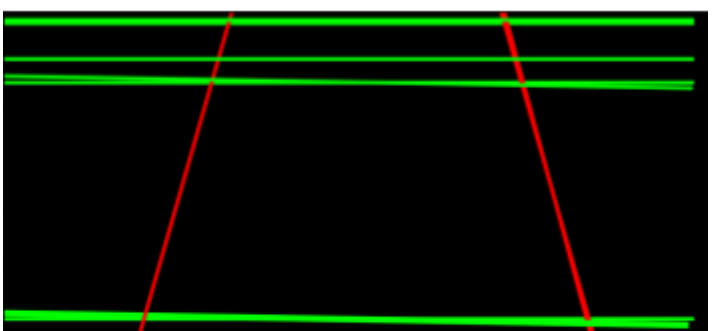
In the actual code we are not using angular coefficients because they approach infinity for vertical lines, instead we cluster using a modified version of the thetas computed by the Hough Transform.

We update $\theta = \pi - \theta$ if $\theta < \frac{\pi}{2}$, this is equivalent to applying the absolute value to the angular coefficient as we can see in the following plot:



Left: plot of angular coefficient as a function of theta
Right: plot of angular coefficient as a function of the updated value of theta

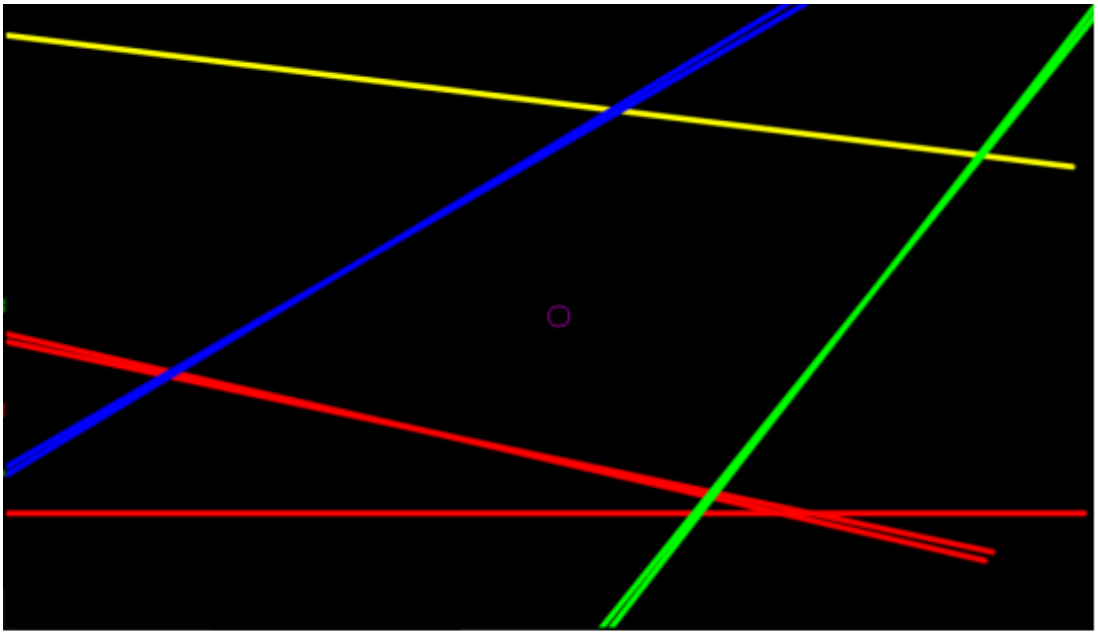




From these examples we see how from this first clustering we have divided the detected edges in two sets based on their approximate direction.

For the next step we need to split in half these two sets based on the position of the lines. To do this we can proceed with a further clustering again with K=2. This time though we don't use the theta for clustering but we use the value of rho. The reason behind using only rho is because we

have already grouped the lines with the same direction together and we just want to consider their distance.



Final result of the clustering, each cluster corresponds to an edge of the table

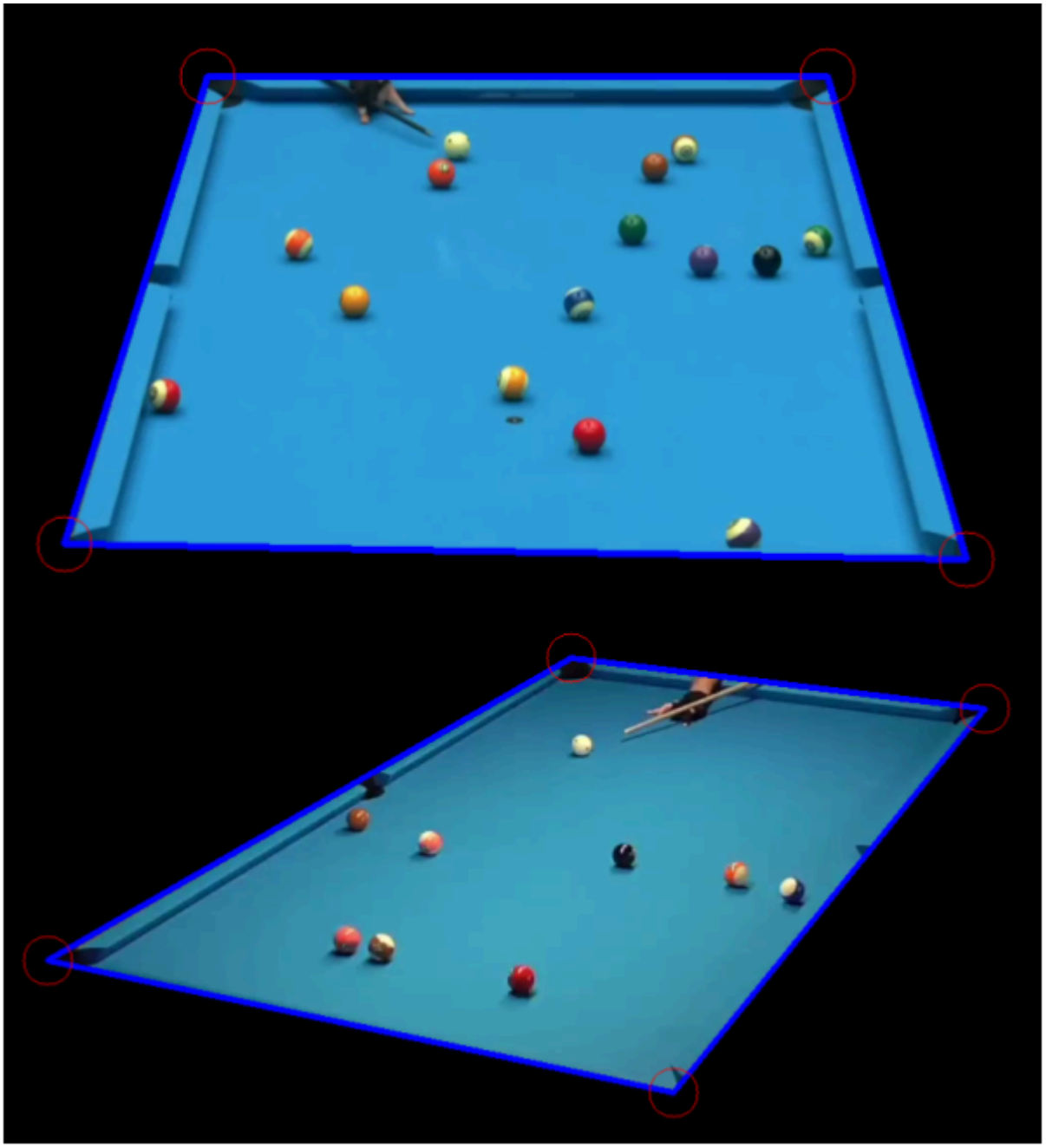
Now that we have one cluster per edge of the table we can select one line per cluster and get the selected edges of the table we will use for the segmentation.
We select the innermost edge for each cluster, this will exclude spurious lines outside the table and select a conservative estimate of the playing field.

To find the four vertices of the table we compute the intersection of the blue line with red and yellow ones and the green line with the red and yellow ones.
Again to avoid problems with infinite angular coefficients for vertical lines we directly compute the intersections in the Hough parameter space with the following function:

```
Vec2i intersect_hough_lines(Vec2f line1, Vec2f line2) {

  cv::Mat A = (Mat_<double>(2, 2) << cos(line1[1]), sin(line1[1]),
               cos(line2[1]), sin(line2[1]));
  cv::Mat b = (Mat_<double>(2, 1) << line1[0], line2[0]);

  cv::Mat x;
  cv::solve(A, b, x);

  int x0 = static_cast<int>(std::round(x.at<double>(0, 0)));
  int y0 = static_cast<int>(std::round(x.at<double>(1, 0)));

  return Vec2i(x0, y0);
}
```

The variables line1 and line2 are two Vec2f parameters containing rho and theta for the line. The solve function inverts matrix A to solve the system of equations.

After having computed the vertices we can mask the original image to get only the table, this is useful so the ball detection process can focus only on the area of the image where balls could be.
The vertices are also used in the later steps to do the projection of the table to draw the top view.

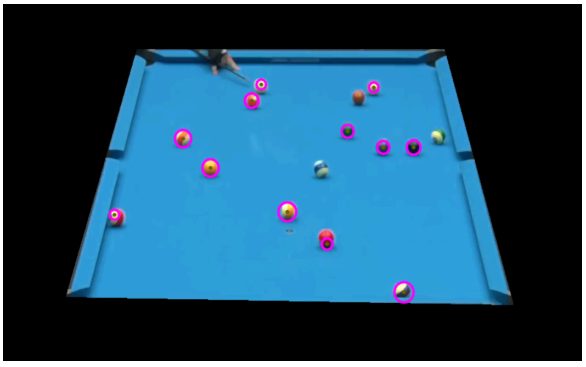


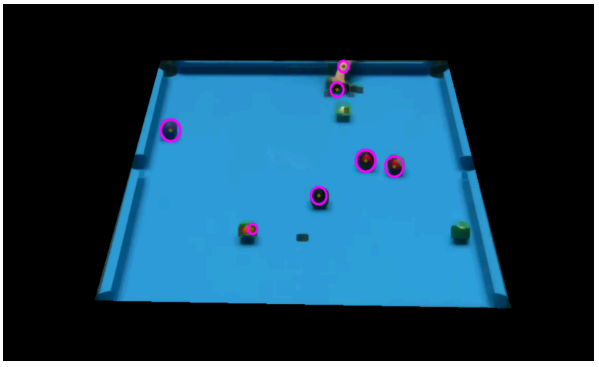
Two examples of tables after the masking with the detected vertices

Ball detection (Alessandro Di Frenna):

We decided to use the Hough Circle algorithm to detect the balls. This is an appropriate choice due to the projection of the balls on the camera. Namely, from whatever perspective we capture the balls, the projection of a sphere is always a perfect circle, and the Hough Circle algorithm is designed to accomplish this task. Additionally, the output of the Hough Circle algorithm provides us with the center of the detected circle, which is essential for calculating the bounding box of the detected object and subsequently calculating the mAP (mean Average Precision) and accuracy.Even though the projections of the spheres are circles, the algorithm suffers from several errors due to the presence of noise such as shadows, changes in lighting, and details on each ball such as the number on the surface or the drastic change in color of the striped balls (from white to another color). To overcome this problem, we decided to do some preprocessing on the given frames. The goal is to remove all the noise around each ball without ruining its shape, allowing the algorithm to capture it with high accuracy.Practically, we apply erosion and dilation separately from the original image with kernel sizes of 1 and 3, respectively. The dilation highlights the lighter balls, namely the striped balls and the cue ball. On the other hand, the erosion highlights the darker ones, namely the eight ball and the solid balls. For each processed image, we then apply a Gaussian blur with a kernel size of 9x9 and a standard deviation of 2 for both axes. Finally, we apply the Hough Circle algorithm.So far, we have been able to detect the darker and brighter balls very well for each set of images. In practice, we have two `vector<cv::Vec3f> circles`, one for each set.

Below, the results of the Hough Circles algorithm after applying preprocessing for each set.





Now, mathematically speaking, we have to simply do the union and then remove the intersection of these 2 sets. In this manner we'll obtain the final set which is the detection of all the balls(both darker and brighter ones).

- For the Union: we append one of the 2 vectors to the other one.
- For the Intersection: we have to select the balls detected from both Hough circles algo, and then remove them from the vector obtained from the previous step.

We use "select_Circles" algorithm:

void select_Circles(vector<cv::Vec3f>& circles, float TH_Circ_A, float TH_Circ_a, float TH_Circ_B, float TH_Ratio_B, float TH_Circ_C, float TH_Ratio_C)

Code Explanation:

1. **Initialization:**

- `n_balls` is the number of circles in the `circles` array.
- `new_size` is initialized with the same size as `circles`, but it will be used to track the new size of the selected circles array.
- `is_selected` is a boolean array, initialized to `true`, of the same size as `n_balls`, which indicates whether each circle is selected or not.

2. First part

```
for (int i = 0; i < n_balls; i++) {
    for (int j = i + 1; j < n_balls; j++) {
        float x_first = circles[i][0], x_second = circles[j][0];
        float y_first = circles[i][1], y_second = circles[j][1];
        float r_first = circles[i][2], r_second = circles[j][2];

                            float min = r_first < r_second ? r_first : r_second;
                            float max = r_first >= r_second ? r_first : r_second;
                            float ratio = min / max;

        float distance_Centers = sqrt( pow((x_second - x_first),2) +  pow((y_second - y_first), 2)  );
        float distance_Circ = distance_Centers - (r_first + r_second);
```

Double `for` loop:

- The first `for` loop with index `i` iterates through each circle in the array.
- The second `for` loop with index `j` compares circle `i` with each subsequent circle `j`.

Extraction of coordinates and radii:

- `x_first`, `y_first`, and `r_first` are the coordinates of the center and the radius of the first circle `i`.
- `x_second`, `y_second`, and `r_second` are the coordinates of the center and the radius of the second circle `j`.

Calculation of `min`, `max`, and `ratio`:

- `min` is the radius of the smaller circle between `i` and `j`.
- `max` is the radius of the larger circle between `i` and `j`.
- `ratio` is the ratio of the smaller radius to the larger radius (i.e., `min / max`).

Calculation of distances:

- `distance_Centers` is the Euclidean distance between the centers of circles `i` and `j`. It is calculated using the Euclidean distance formula.
- `distance_Circ` is the distance between the surfaces of the two circles. It is obtained by subtracting the sum of the radii of the circles from `distance_Centers`. If this value is negative, the circles overlap.

3. Selection

This section of the code within the select_Circles function deals with deciding whether a circle should be selected or not based on various distance and radius ratio criteria. Use thresholds (TH_Circ_a, TH_Circ_A, TH_Circ_B, TH_Circ_C, TH_Ratio_B, and TH_Ratio_C) to filter circles.

```
if (r_first < 10 && r_second < 10) {

        if (distance_Circ < TH_Circ_a) {
            is_selected[r_first < r_second ? i : j] = false;
            new_size--;
        }
        else {
            if (distance_Circ < 2 && ratio < 0.91) {
                is_selected[r_first < r_second ? i : j] = false;
                new_size--;
            }
            else if (distance_Circ < 6 && ratio < 0.75){
                is_selected[r_first < r_second ? i : j] = false;
                new_size--;
            }
        }
}
else {
        if (distance_Circ < TH_Circ_A) {
            is_selected[r_first < r_second ? i : j] = false;
            new_size--;
        }
        else {
                if (distance_Circ < TH_Circ_B && ratio < TH_Ratio_B){
                    is_selected[r_first < r_second ? i : j] = false;
                    new_size--;
                }
                else if (distance_Circ < TH_Circ_C && ratio < TH_Ratio_C){
                    is_selected[r_first < r_second ? i : j] = false;
                    new_size--;
                }
        }
}
```

Handling Small Circles (`r_first < 10` and `r_second < 10`):

This condition checks if both circles being compared have a radius smaller than 10. If true, stricter thresholds are applied for small circles. Otherwise, different values for thresholds are applied for larger circles.

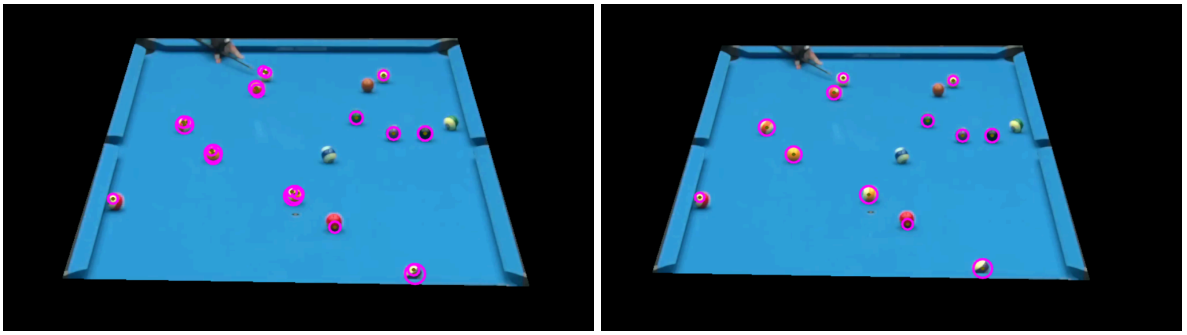
```
if (distance_Circ < TH_Circ_a) {
    is_selected[r_first < r_second ? i : j] = false;
    new_size--;
}
else {
    if (distance_Circ < 2 && ratio < 0.91) {
        is_selected[r_first < r_second ? i : j] = false;
        new_size--;
    }
    else if (distance_Circ < 6 && ratio < 0.75){
        is_selected[r_first < r_second ? i : j] = false;
        new_size--;
    }
}
```

1. **Threshold Based on `distance_Circ`:**

If the distance between the surfaces of the circles (`distance_Circ`) is less than `TH_Circ_a`, it indicates that the circles are overlapping too much, and one of them is discarded (`is_selected` set to `false`). The circle with the smaller radius is removed.

2. **Additional Threshold Conditions:**

When two circles have a very different radius and they are very near each other then there is an inconsistency with projection' rules. If two balls are very near each other, they should have the radius with a very similar magnitude, from a perspective point of view, since all balls are physically the same radius. Therefore, except for some particular cases with a very specific angulation, we set, in an empirical way, the parameters that satisfy this concept quite well. The parameters have to be different if we consider different distances from the 2 balls.So, if we consider 2 spheres not so near each other the "else if" is applied and we apply the same concept but with different values. If the 2 balls are very distant any assumptions shouldn't be strong enough to hold due to the instability of the radius' s output of the algorithm. Below, before and after applying the "select_Circles" algorithm.

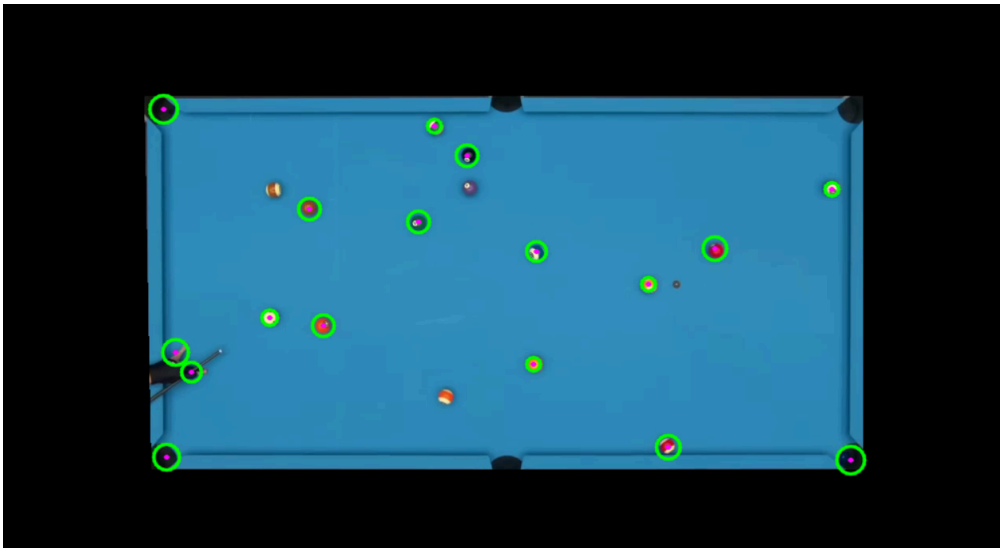


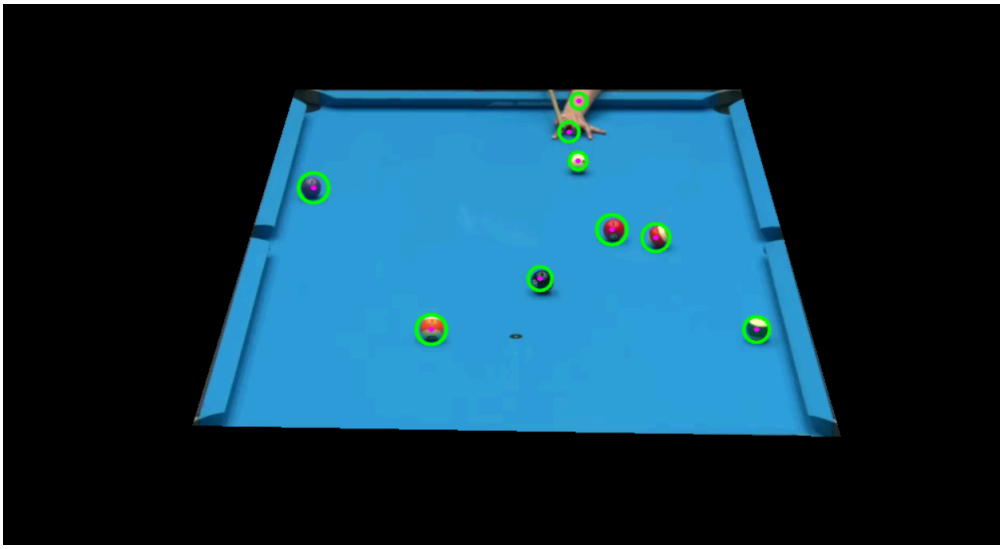
3. **Final part**

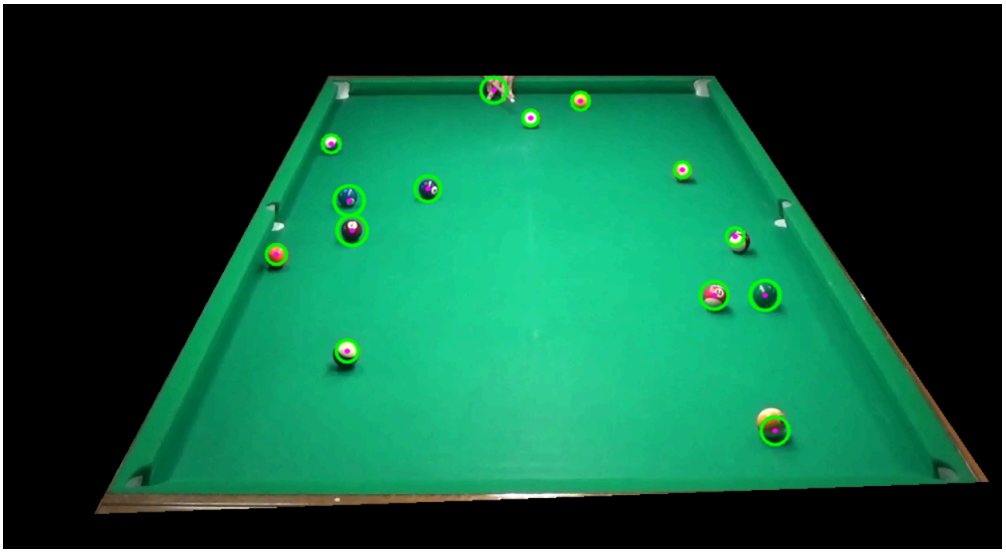
```
vector<cv::Vec3f> selected_circles;
for (int i = 0; i < n_balls; ++i) {
    if (is_selected[i]) {
        selected_circles.push_back(circles[i]);
    }
}
```

At the end of this loop, `selected_circles` contains all the circles that were marked as selected by the criteria applied earlier in the function. This filtered list can now be used for further processing or analysis.

Examples of final results:

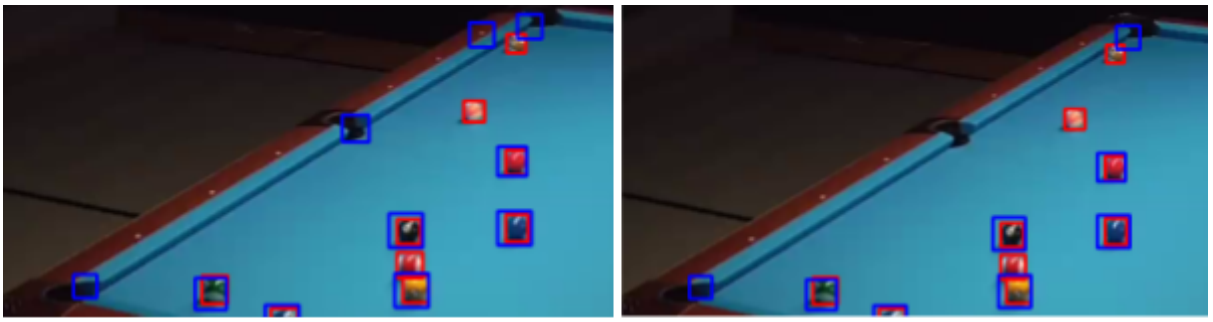






Ball classification (Matteo De Gobbi):

Before proceeding with the actual classification we added a step that discards detected balls whose radius overlaps with the table edge, this is needed because often the ball detection step detects a lot of false positives due to shadows on the edge of the table.

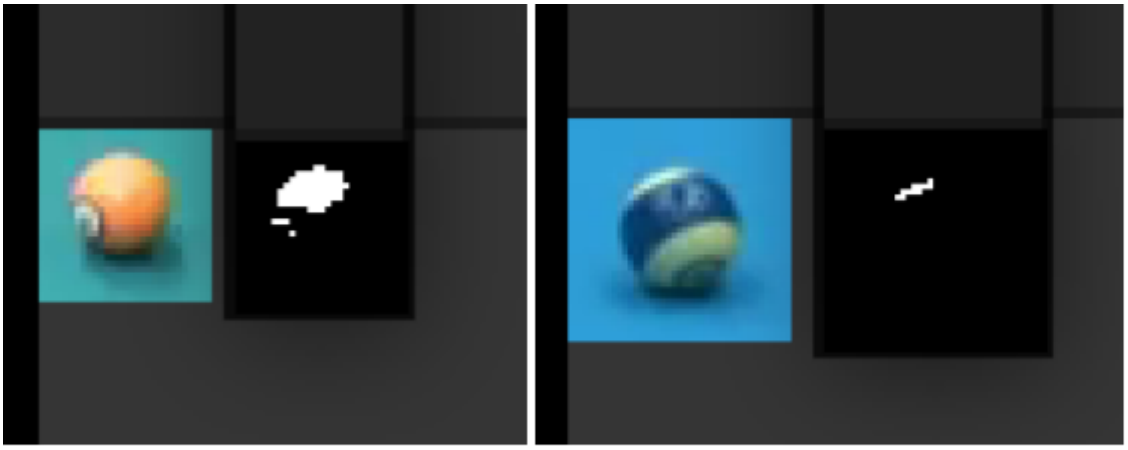


Two of the false positives on the edge of the table get suppressed by this simple technique

For the ball classification we decided to go with a simple approach and we decided to avoid using machine learning techniques because they require a big labeled dataset that is different from the test set provided.
Our approach consists in using thresholds to keep only relevant pixels to the class of the ball we are trying to detect and then counting the percentage of pixels in the bounding box that are left after the thresholding.

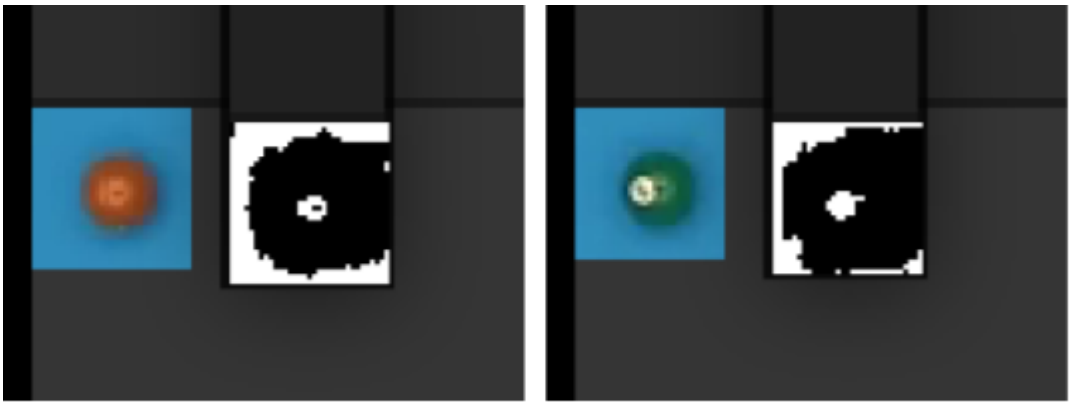
The first approach we tried was using the threshold function in opencv to keep only bright pixels; we compute the percentage of total pixels left after the thresholding.
If the percentage is more than the threshold Tcue we classify the ball as the white cue ball, while if the percentage is less than Tcue but more than another threshold Tstriped we classify the ball as striped, the same idea is used but with the percentage of dark pixels for black and solid balls.

This first approach was simple but depends a lot on the color of the ball, for example yellow balls have a lighter color because both the red and green channels tend to have high values on the opposite side brown balls have low values on all channels.



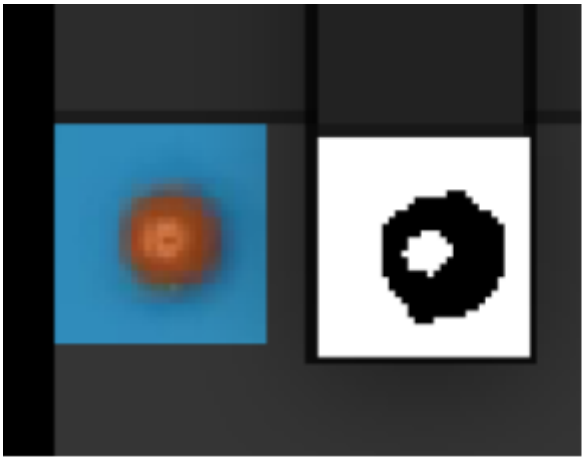
Problems with the first approach: on the left the yellow ball has a lot of pixels detected as stripes while on the right the striped ball has very few pixels detected as stripes

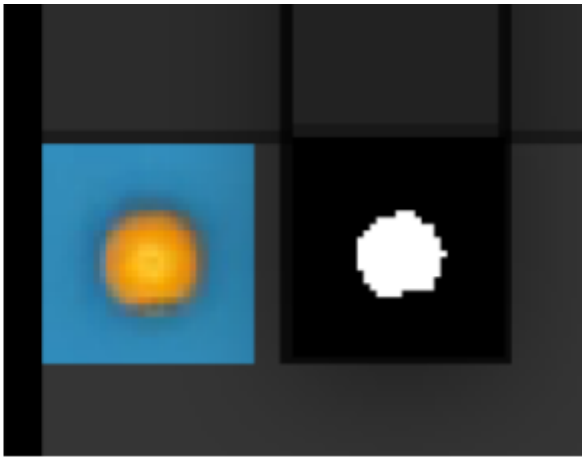
We tried a few ways to improve this method that ended up not working:
1. Equalizing the histogram of the area inside the bounding box, this approach doesn't work because most of the time the background intensity level profuces a spike in the histogram that leads to some background pixels to be set above the threshold.



Problems with histogram equalization: some background pixels get mapped above threshold, some below threshold

2. Using Otsu's threshold or adaptive threshold, these other two approaches fail in the same way as histogram equalization, since Otsu tries to maximize inter class variance we get that often white parts of the ball get assigned the same class as the lighter parts of the backgrounds.



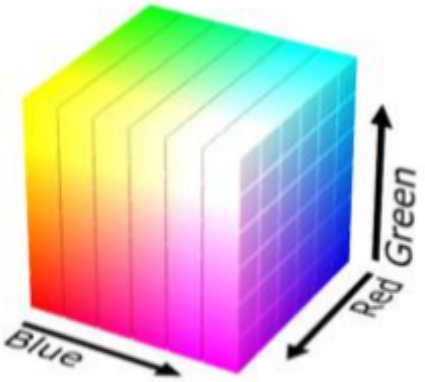


Examples of Otsu failing

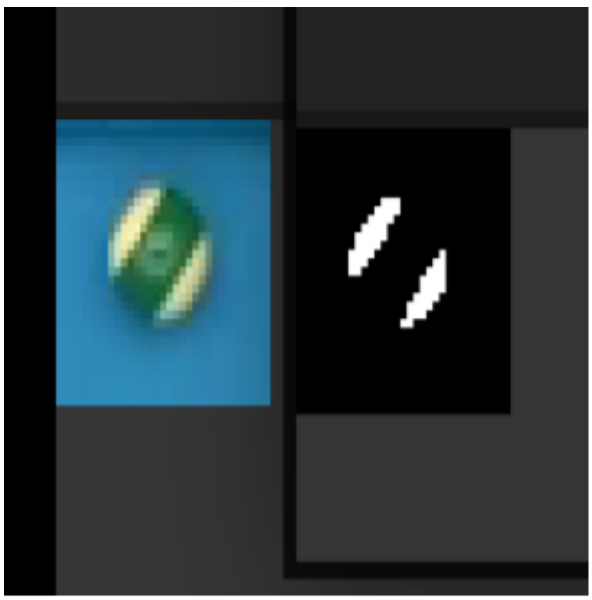
After trying these approaches we realized that by using thresholding we are not exploiting the color information since we only work on the grayscale image.

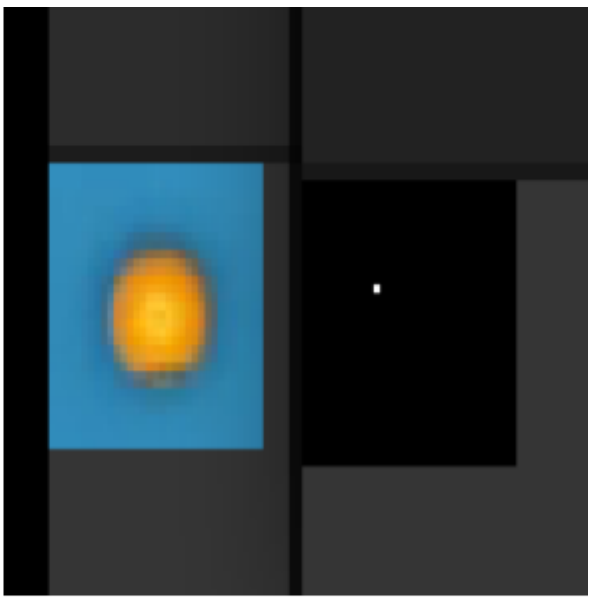
The important point is that we don't want to divide bright pixels from light pixels but we want to get white stripes which are characterized by all three channels having about the same value and all three channels being above a certain brightness.

For applying this idea we used the inRange function provided by opencv which allows us to only select pixels that have a color inside a certain slice of the BGR color space.



RGB color space cube By SharkD [GFDL or CC BY-SA 4.0], from Wikimedia Commons





Examples when using the inRange function

After having applied inRange we can count the percentage of white pixels in the result image and choose the class based on a threshold on the percentage. We use the same approach to also detect black 8-balls but using inRange to select pixels where all channels have low values instead of high values.

Minimap (Alessandro Di Frenna)

The function `MINIMAP` is designed to create a minimap representation of pool balls based on their positions in a given space. It takes as inputs:

- A vector of rectangles (`cv::Rect`), each representing a bounding box around an object.
- A set of four points (`Vec4Points`) representing the corners of the quadrilateral (the projection of the table on the camera).
- A vector of labels (`ball_class`) corresponding to the types of objects: `CUE`, `EIGHT_BALL`, `STRIPED`, `SOLID`.

The `vertices` are not ordered, and we need the corners to follow the convention adopted by OpenCV for the functions to work correctly. Specifically, the first corner should be the top-left one, followed by the others in clockwise order.

To achieve this, we run a function called `reorderVerticesClockwise`. This function finds the first and third corners by comparing the magnitudes of their coordinates. The top-left corner is the one with both coordinates being the smallest, while the third one has the largest coordinates. The second corner shares the same x-coordinate as the first one, and the last corner shares the same y-coordinate as the third one.

After reordering the vertices, we need to determine if the camera is recording along a short side or a long side. To do this, we calculate the average lengths of two parallel sides and then compare them. We assume that the first side is a long side, following the convention of placing the minimap horizontally. If the average of the shorter sides is greater than that of the longer sides, it means the camera is recording along a short side. In this case, we rotate the order of the corners counterclockwise using the `rotateCounterclockwise` function.

The function `cv::getPerspectiveTransform` takes two sets of four points each: `points` (which are the vertices of the table in the camera view) and `dst_points` (which are the corners of the rectangle where we want to map those vertices). It calculates a "homography matrix", which is a transformation matrix that allows you to map or transform the perspective from the points seen by the camera to the new points in the output image.

`points_to_map` is a vector of points representing the centers of the rectangles (bounding boxes around the pool balls) in the original camera view.

The function `cv::perspectiveTransform` takes the `points_to_map` (original points), the `homography_matrix` (transformation matrix), and outputs the `mapped_points` (new points in the transformed image).It applies the homography matrix to each point in `points_to_map`, warping each point from the original perspective to the new perspective defined by `dst_points`.

Finally the loop goes through each pool ball, for each ball, it gets its type (e.g., eight ball, cue ball) and it draws a circle on the image at the corresponding position, with the color and size based on the ball type.

```
void MINIMAP(const vector<cv::Rect> rectangles, Vec4Points vertices,
const vector <ball_class> balls) {
int width = 700;
int height = 350;
std::string imagePath = "C:\\Users\\Alessandro Di
Frenna\\OneDrive\\Desktop\\minimap.jpg";

    reorderVerticesClockwise(vertices);
    vector<cv::Point2f> points(vertices.val, vertices.val + 4);

                float sideAC = calculateDistance(vertices[0],
vertices[1]);
                float sideBC = calculateDistance(vertices[1],
vertices[2]);
                float sideBD = calculateDistance(vertices[1],
vertices[2]);
                float sideAD = calculateDistance(vertices[3],
vertices[0]);

                std::vector<float> sides =
{sideAC,sideBC,sideBD,sideAD};

                float average_shortest = (sides[1] + sides[3]) / 2;
                float average_longest= (sides[0]+ sides[2]) / 2;

                if (average_shortest > average_longest)
                    rotateCounterclockwise(points);

    vector<cv::Point2f> dst_points(4);
    dst_points[0] = cv::Point2f(0, 0);
    dst_points[1] = cv::Point2f(width, 0);
```

```
    dst_points[2] = cv::Point2f(width, height);
    dst_points[3] = cv::Point2f(0, height);

    cv::Mat homography_matrix = cv::getPerspectiveTransform(points,
dst_points);

    vector<cv::Point2f> points_to_map(rectangles.size());
    for (int i = 0; i < rectangles.size(); i++) {
    int min_x = rectangles[i].x;
    int min_y = rectangles[i].y;
    int width = rectangles[i].width;
    int height = rectangles[i].height;

        points_to_map[i] = cv::Point2f(min_x + width/2 ,min_y +
height/2);
    }

    vector<cv::Point2f> mapped_points;
    cv::perspectiveTransform(points_to_map, mapped_points,
homography_matrix);

    cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR);
    cv::Size size(width, height);
    cv::resize(image, image, size);
    for (int i = 0; i < rectangles.size(); i++) {

    ball_class label  = balls[i];

        switch (label) {
        case ball_class::EIGHT_BALL:
            cv::circle(image, mapped_points[i], 10, cv::Scalar(0, 0,
0), -1);
            break;
        case ball_class::CUE:
            cv::circle(image, mapped_points[i], 10, cv::Scalar(255,
255, 255), -1);
            break;
        case ball_class::STRIPED:
            cv::circle(image, mapped_points[i], 10, cv::Scalar(0, 0,
255), -1);
            break;
```

```
        case ball_class::SOLID:
            cv::circle(image, mapped_points[i], 10, cv::Scalar(255, 0,
0), -1);
            break;
        }
    }
    for (int i = 0; i < rectangles.size(); i++) {
        cv::circle(image, mapped_points[i], 1, cv::Scalar(255, 0, 255),
-1);
    }
    cv::imshow("Image", image);
    cv::waitKey(0);

}
```

Ball tracking (Matteo De Gobbi):

For the tracking we used the algorithms provided by opencv, all the tracker classes inherit from a base abstract class called Tracker, so to try the different algorithms we just need to change the instantiation method.
Opencv used to provide a MultiTracker class to track multiple objects at once but it was deprecated in opencv 4.5.1 so we decided to avoid using legacy code.
To track multiple objects at once we just use a vector of trackers, one for each object, this is basically the same as what the old MultiTracker implementation did but with the difference that in our case if a tracker loses an object we don't have to destroy and create all the trackers again.
Each tracker will try to detect in the next frame the same object that was assigned to it by looking at a neighborhood of the previous position of the object.

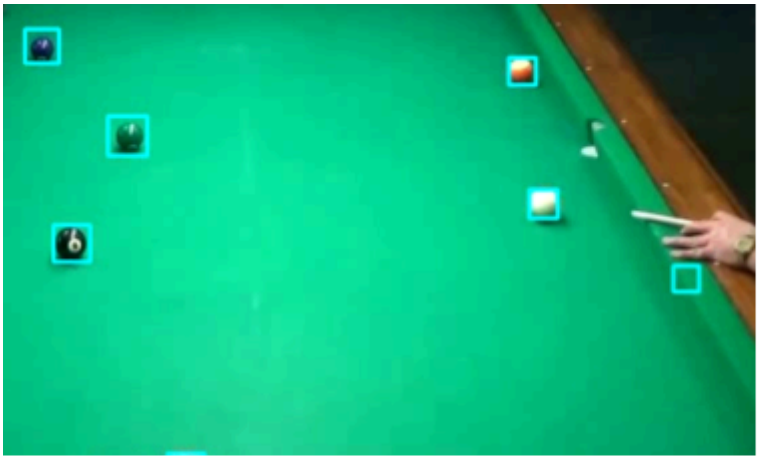
We initialize all the trackers with the first frame bounding boxes we get from the ball detection, then we extract one frame at a time from the video (using a VideoCapture object). For each frame we run the update on all the trackers.
The update function updates the bounding boxes and returns false if it lost track of the object, it returns true instead if the object was found.
Then only if the tracker didn't lose track of the object we draw a rectangle around the object.

```
for (size_t i = 0; i < trackers.size(); ++i) {
  bool isok = trackers[i]->update(frame, bboxes[i]);
  if (isok) {
    rectangle(frame, bboxes[i], Scalar(255, 255, 0), 2, LINE_4);
  }
}
```

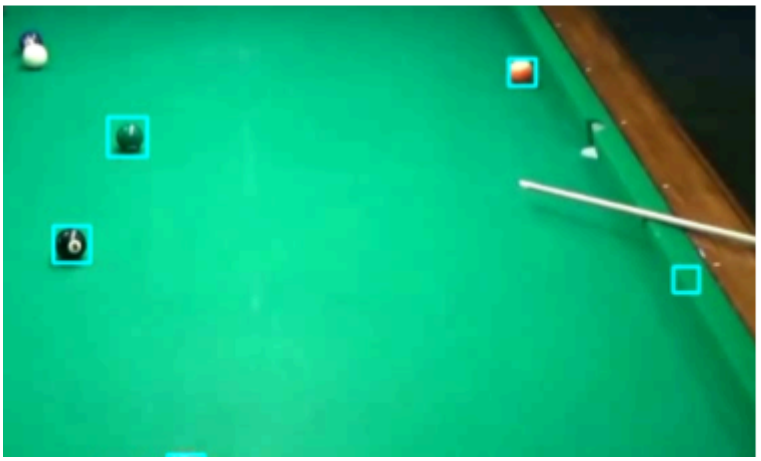
Code for updating the trackers and drawing the rectangle around the new position of the object

Now we will illustrate some of the tracking algorithms we tried, their advantages and their disadvantages:

1. The first tracker we tried was the KCF (Kernelized Correlation Filters) which uses correlations between patches computed in the frequency domain to track the objects. This tracking algorithm runs very fast so it would be good for real time applications. However from our experience KCF doesn't handle objects that move quickly very well and most of the times loses track of the balls when they get hit, for example



Before the ball is hit

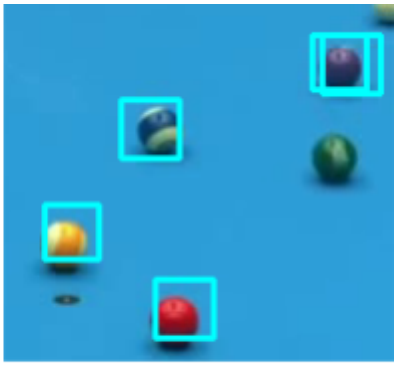


After the white ball is hit KCF loses the object, after the blue ball gets hit by the white one KCF also loses track of the blue one

2. The second tracker we tried is MIL, this type of tracker takes the object in the starting bounding box, extracts some features and considers them as the positive example, then it takes the pixels surrounding the bounding box it extracts features from these patches and considers them as the negative (background) class. This algorithm compared to KCF is able

to handle balls that move at high speeds but it still has a few problems. The algorithm is very slow and it's impossible to use it for real time applications unless we skip some frames, making the accuracy worse.
Also MIL has a problem where the bounding box of balls that are not moving jumps around in a neighborhood of the ball from one frame to another making it unstable. This last problem means that often a bounding box of a ball will start tracking a different ball if it passes near enough, like we can see in the following image:

Before: green and purple ball have two different bounding boxes



After: the tracker of the green ball started tracking the purple ball when they were near

3. The third and best tracker we used is CSRT (Channel and Spatial Reliability Tracker), it is based on the same idea as KCF of using correlation between the features of patches but it adds some other steps to make the algorithm more reliable at the expense of speed of computation.
To increase the accuracy CSRT assigns reliability scores to different patches inside the starting bounding boxes, the idea behind this is that if for example we have a partially occluded ball the region that is occluded will be assigned a lower reliability score this causes the features from that occluded patch to be weighted less when considering the next probable location.
CSRT also assigns a channel reliability score, the idea behind this is for example we are tracking the yellow ball the red and green channel will be more important than the blue channel.
This algorithm also provides some temporal coherence by averaging the filters temporally, this makes it so that if the appearance of the object changes over time CSRT is still able to track it.
In practice CSRT is the best algorithm and it avoids the problem with fast balls of KCF and the problem of unstable bounding boxes of MIL, it still makes some mistakes sometimes, for example in certain cases it continues tracking a ball that went into a hole and leaves a bounding box on the hole:

Before the red ball goes in the hole

After the ball goes in the hole CSRT keeps tracking

Mean of Intersection over Union for segmentation (Mohammadali Jafari):

In this section, we evaluate the performance of the segmentation process using the mean Intersection over Union (mIoU) metric. The mIoU provides a robust evaluation for the segmentation accuracy of different classes, including the playing field and various types of billiard balls (white, black, solid color, and striped).

Dataset and Ground Truth

Ground truth annotations are provided as grayscale masks where each pixel is assigned a category ID:

- **0**: Table Field
- **1**: White/Cue Ball
- **2**: Black/Eight Ball
- **3**: Striped Ball
- **4**: Solid Ball
- **5**: Background

These annotations serve as the benchmark against which the system's output is compared.

Segmentation Process

The segmentation process involves several steps:

1. **Field Detection**: Detect and segment the playing field from the rest of the image.
2. **Ball Detection**: Identify and classify the billiard balls within the detected playing field.
3. **Mask Generation**: Generate binary masks for each class (striped, solid, white, black, and playing field).

4. **Combined Mask Creation**: Combine the individual masks into a single mask where each pixel's value corresponds to its class ID.

Intersection over Union (IoU) Calculation

IoU is calculated for each class by comparing the predicted mask with the ground truth mask. The formula for IoU is: $IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$.

Process for Mean Intersection over Union (MIoU):

We follow these steps:

1. **Ground Truth Comparison**: Each predicted mask is compared against the ground truth mask. The ground truth mask provides the accurate segmentation for each class.
2. **IoU Calculation for Each Class**:
    - For each class C, compute the IoU by identifying the true positives (TP), false positives (FP), and false negatives (FN).
    - **True Positives (TP)**: Pixels correctly identified as belonging to class C.
    - **False Positives (FP)**: Pixels incorrectly identified as belonging to class C.
    - **False Negatives (FN)**: Pixels that belong to class C but were not identified as such.
3. The IoU for class C is then: $IoU_C = \frac{TP_C}{TP_C + FP_C + FN_C}$.
4. **Iterate Over All Classes**: Repeat the above IoU calculation for each class.
5. **Compute mIoU**: Average the IoU values for all classes to obtain the mIoU.

   $mIoU = \frac{1}{C} \sum_{c=1}^{C} IoU_c$.

Results

For each image in the dataset, we compute the IoU for all six classes and then average these values to obtain the mean IoU for that image. The results for each image are summarized as follows:

```
// Compute IoU for each class
vector<double> classIoUs(6, 0.0);
for (int c = 0; c < 6; ++c) {
    classIoUs[c] = ComputeIoUPerClass(combinedMask, gtMasks[i], c);
}

// Compute mean IoU for this image manually
double sumIoU = 0.0;
int numClasses = classIoUs.size();
for (int c = 0; c < numClasses; ++c) {
    sumIoU += classIoUs[c];
}
double meanIoU = (numClasses > 0) ? (sumIoU / numClasses) : 0.0;
```

The mean IoU metric provides insight into the segmentation accuracy of our system. A high mean IoU indicates that the predicted masks closely match the ground truth, signifying accurate segmentation. The results for each class IoU and the overall mean IoU for each image are as follows:

- **Class 0 (Playing Field) IoU**: Varies based on the presence of the playing field pixels in the image.
- **Class 1 (White/Cue Ball) IoU**: Reflects how well the system can segment the white/cue ball.
- **Class 2 (Black/Eight Ball) IoU**: Reflects how well the system can segment the Black/Eight ball.
- **Class 3 (Stripped Ball) IoU**: Reflects the accuracy of the stripped balls segmentation.
- **Class 4 (Solid Ball) IoU**: Reflects the accuracy of the solid balls segmentation.
- **Class 5 (Background) IoU**: Indicates how well the background is segmented.

By averaging the IoUs across all classes for each image, we obtain the mean IoU which provides a single metric representing the overall segmentation performance.

Mean Average Precision(mAP): (Mohammadali Jafari):

The task involves detecting four types of billiard balls: STRIPED, SOLID, CUE, and EIGHT_BALL, in the first and last frames of each video clip. The procedure involves:

1. **Bounding Box Extraction:** Ground truth and predicted bounding boxes are extracted from the dataset and the detection model respectively.
2. **Precision and Recall Calculation:** Precision and recall are computed incrementally as the number of considered bounding boxes increases.
3. **Average Precision Calculation:** Average Precision (AP) is computed using Pascal VOC 11-point interpolation.

4. **Mean Average Precision Calculation:** The mAP is obtained by averaging the AP values for each ball category. The following functions were implemented to compute mAP:

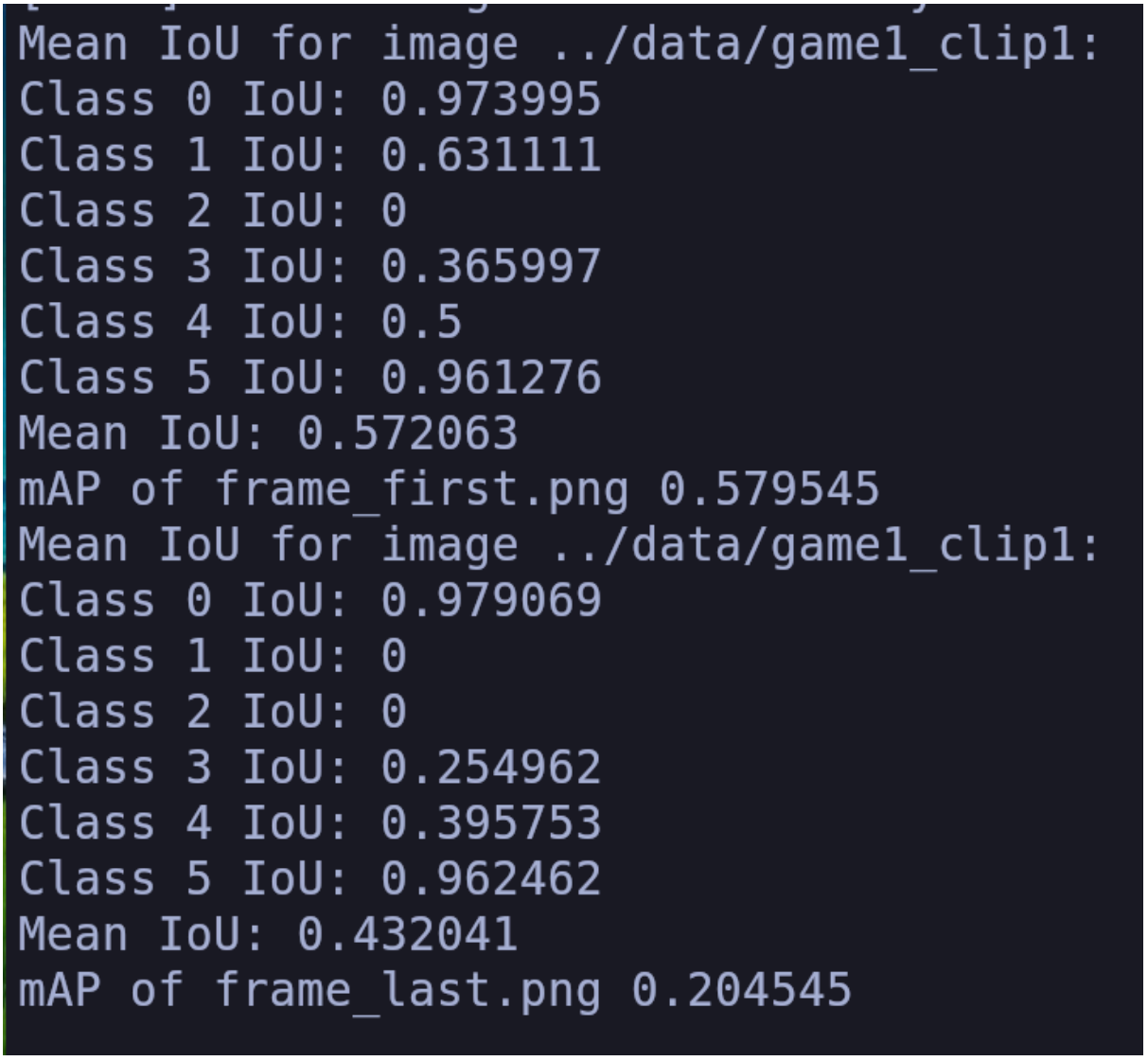
1. **computeAP**: This function calculates the AP for a specific ball category using the Pascal VOC 11-point interpolation method.
    a. Process:
        i. Compute precision and recall values using the `computePrecisionRecall` function.
        ii. Define recall levels at 11 points: {0.0,0.1,0.2,…,1.0}.
        iii. For each recall level, find the maximum precision for recalls greater than or equal to that level.
        iv. Average these maximum precision values to get the AP.
2. **computePrecisionRecall**: This helper function computes precision and recall at different thresholds.
    a. Process:
        i. Match predicted boxes with ground truth boxes based on IoU.
        ii. Compute true positives, false positives, and false negatives.
        iii. Calculate precision and recall at each threshold.
3. **computeMeanAP**: This function calculates the mean AP across all ball categories.
    a. Process:
        i. Compute the AP for each ball class (STRIPED, SOLID, CUE, EIGHT_BALL) using `computeAP`.
        ii. Average the AP values of all classes to get the mAP.

```
double computeMeanAP(const vector<Rect> &gtBoxes, const vector<Rect> &predBoxes,
                     vector<ball_class> &gtClassIDs,
                     vector<ball_class> &predClassIDs) {
  cout << "STRIPED----------------------------------------" << endl;

  double stripedAP = computeAP(gtBoxes, predBoxes, gtClassIDs, predClassIDs,
                               ball_class::STRIPED);

  cout << stripedAP << "----------------------------------------" << endl;

  cout << "SOLID----------------------------------------" << endl;
  double solidAP = computeAP(gtBoxes, predBoxes, gtClassIDs, predClassIDs,
                             ball_class::SOLID);

  cout << solidAP << "----------------------------------------" << endl;
  cout << "CUE----------------------------------------" << endl;
  double cueAP =
      computeAP(gtBoxes, predBoxes, gtClassIDs, predClassIDs, ball_class::CUE);

  cout << cueAP << "----------------------------------------" << endl;
  cout << "8BALL----------------------------------------" << endl;
  double eightballAP = computeAP(gtBoxes, predBoxes, gtClassIDs, predClassIDs,
                                 ball_class::EIGHT_BALL);

  cout << eightballAP << "----------------------------------------" << endl;
  return (solidAP + stripedAP + cueAP + eightballAP) / 4;
}
```

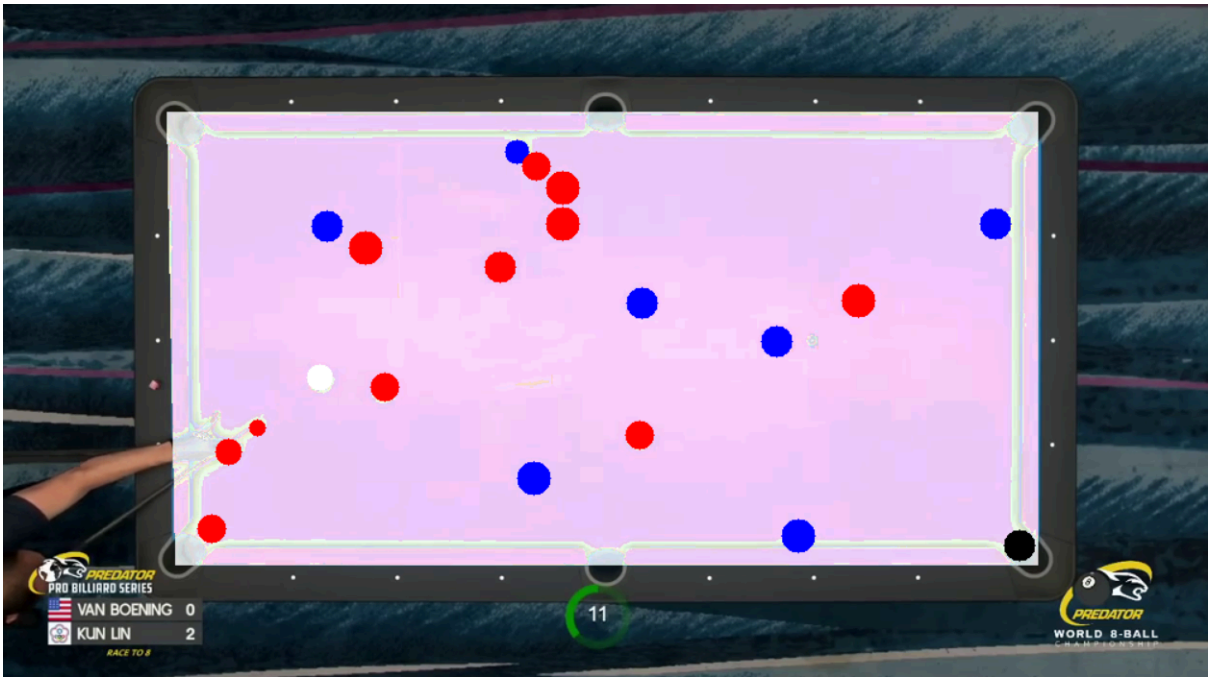
Accuracy of the algorithm for all the provided images: (Matteo De Gobbi)

Here we present the results on the first and last frame of all the images in the test dataset.

game1_clip1

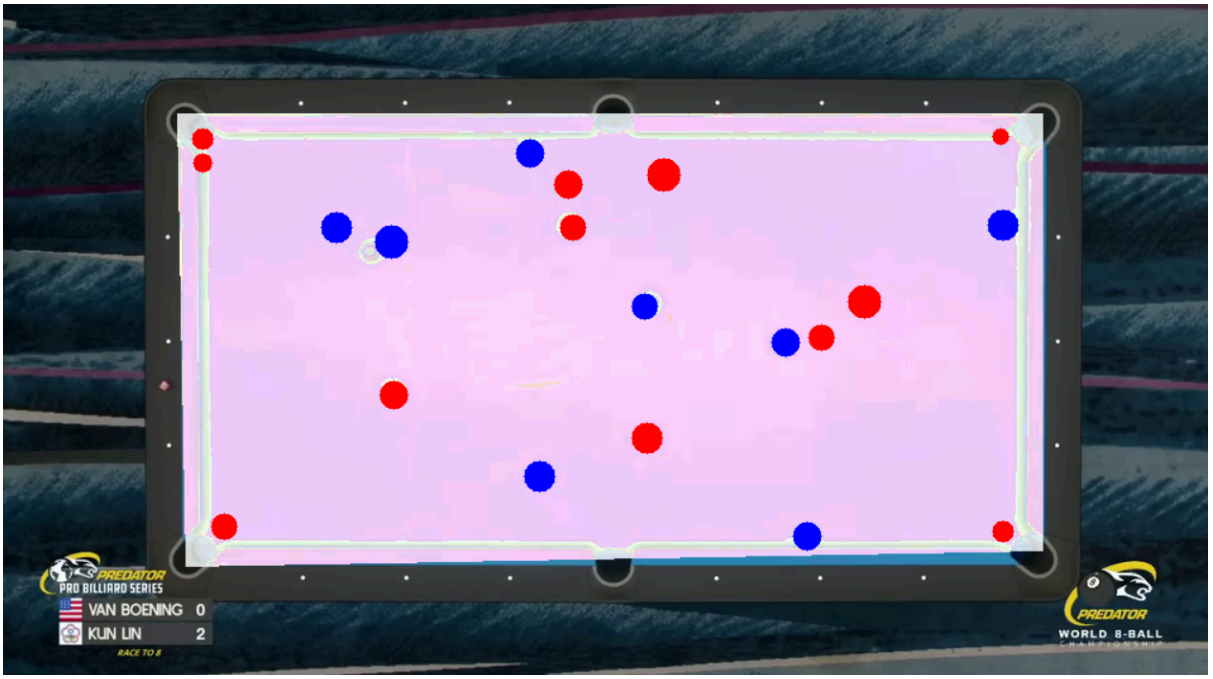


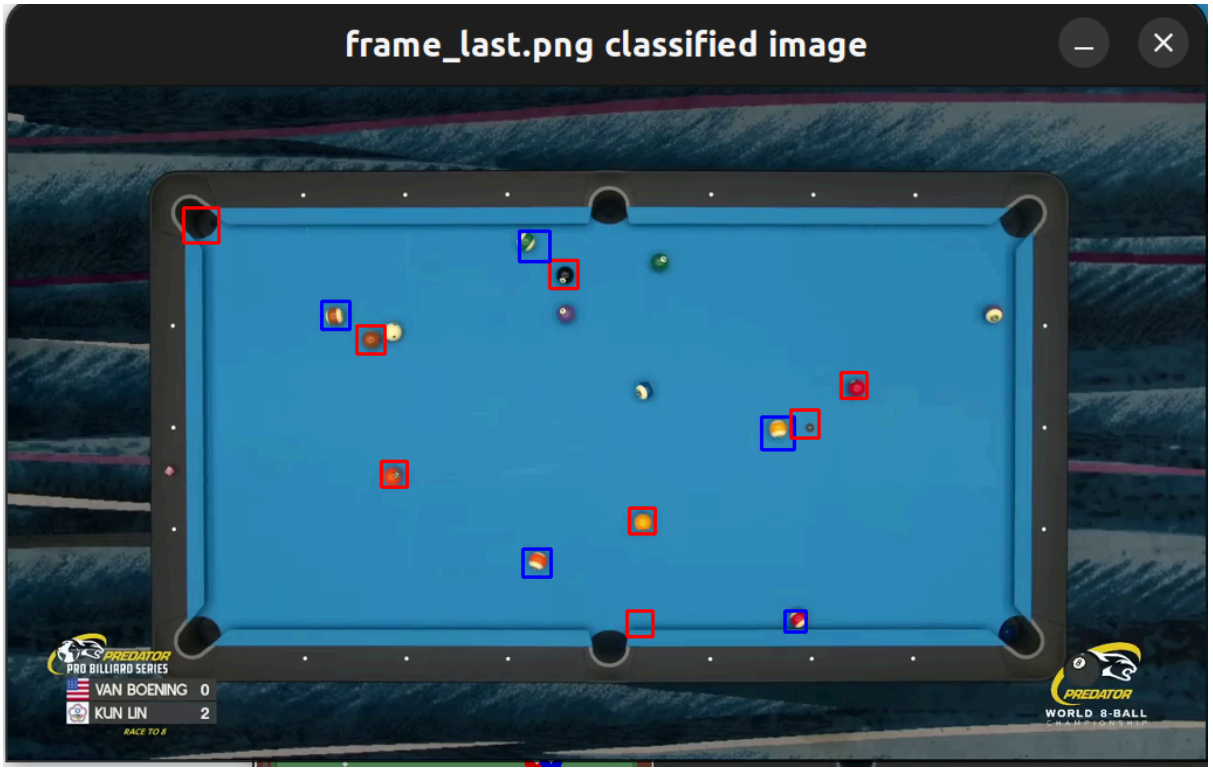
First frame:


PREDATOR
PRO BILLIARD SERIES
VAN BOENING 0
KUN LIN 2
RACE TO 8
11
PREDATOR
WORLD 8-BALL

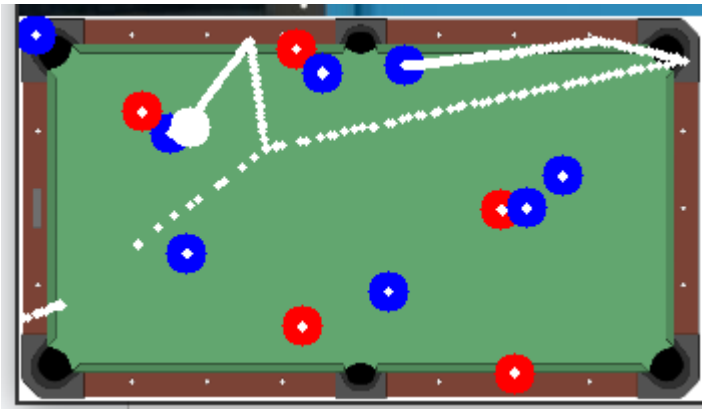

frame_first.png classified image
PREDATOR
PRO BILLIARD SERIES
VAN BOENING 0
KUN LIN 2
RACE TO 8
11
PREDATOR
WORLD 8-BALL

Last frame:


PREDATOR
PRO BILLIARD SERIES
VAN BOENING 0
KUN LIN 2
RACE TO 8
PREDATOR
WORLD 8-BALL



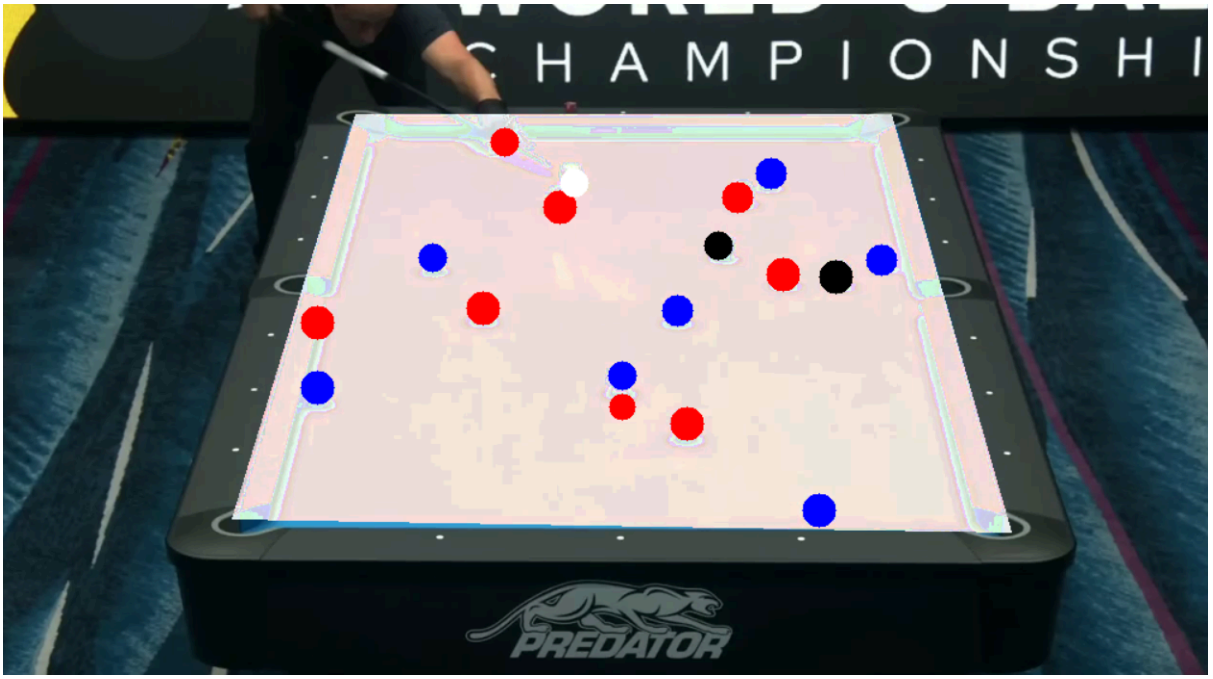
Final top view:

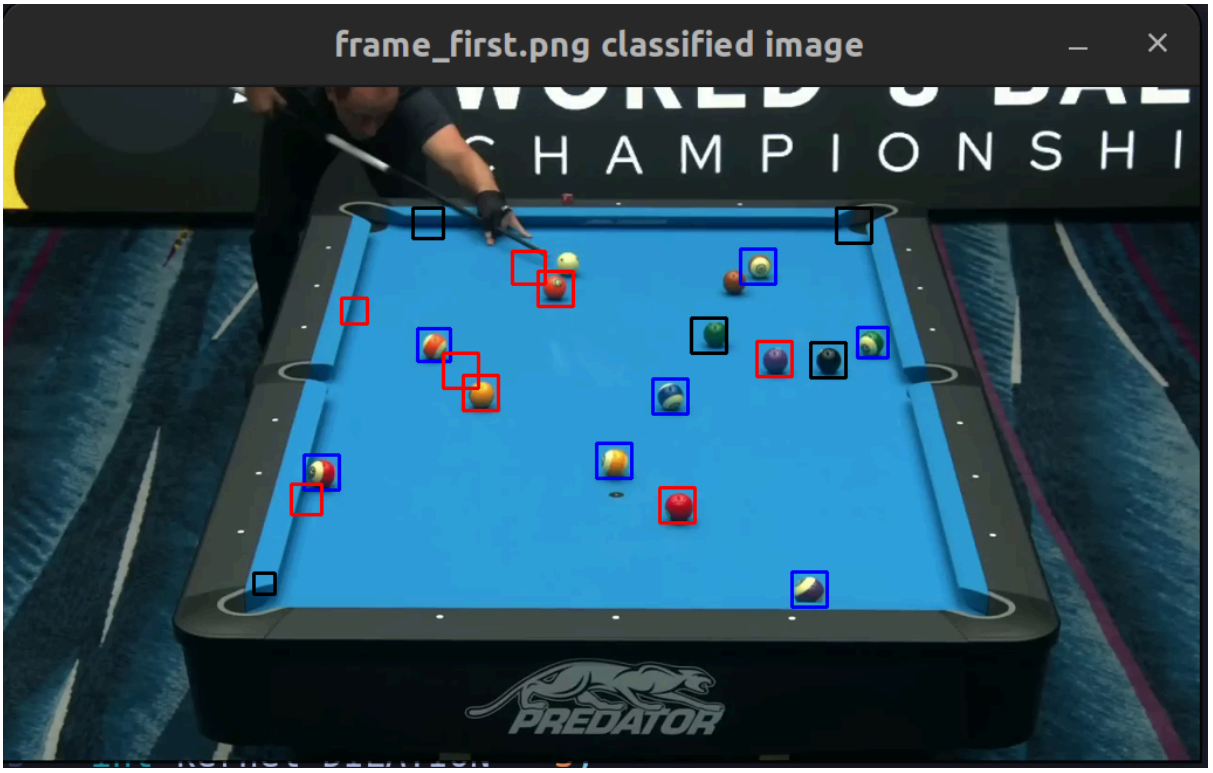


game1_clip2

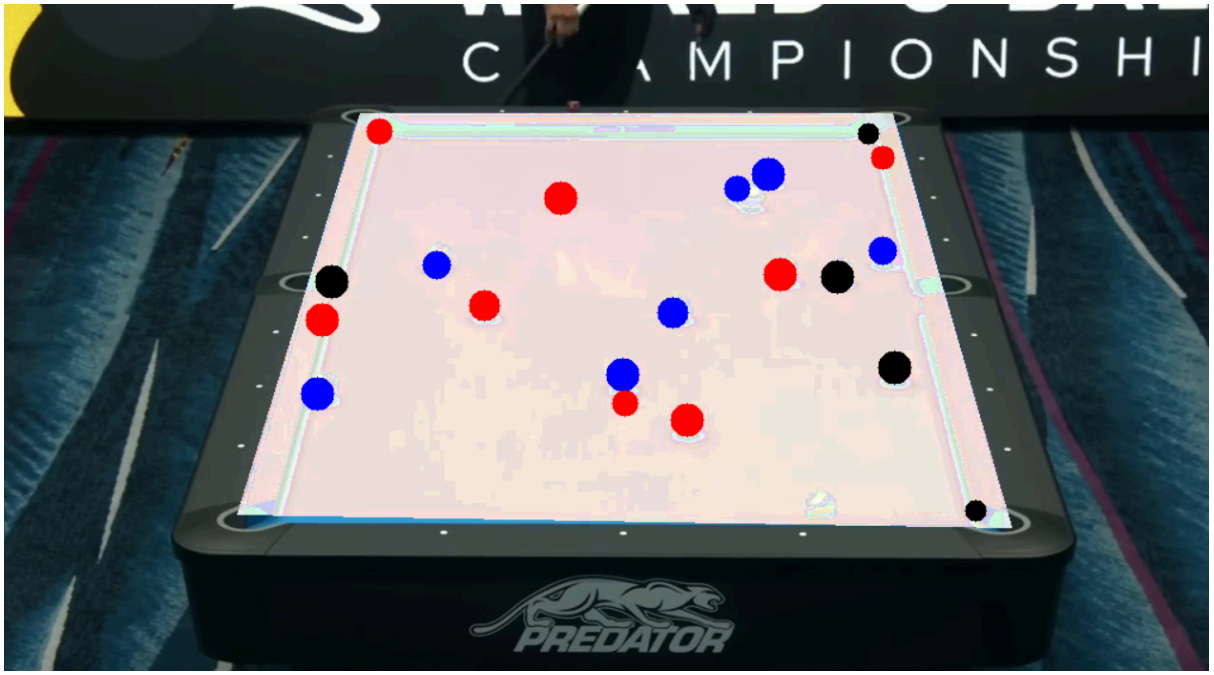
```
Mean IoU for image ../data/game1_clip2:
Class 0 IoU: 0.981793
Class 1 IoU: 0
Class 2 IoU: 0.157895
Class 3 IoU: 0.327911
Class 4 IoU: 0.736842
Class 5 IoU: 0.928371
Mean IoU: 0.522135
mAP of frame_first.png 0.584416
Mean IoU for image ../data/game1_clip2:
Class 0 IoU: 0.988619
Class 1 IoU: 0.49639
Class 2 IoU: 0.273834
Class 3 IoU: 0.437855
Class 4 IoU: 0.695322
Class 5 IoU: 0.958014
Mean IoU: 0.641672
mAP of frame_last.png 0.594156
```

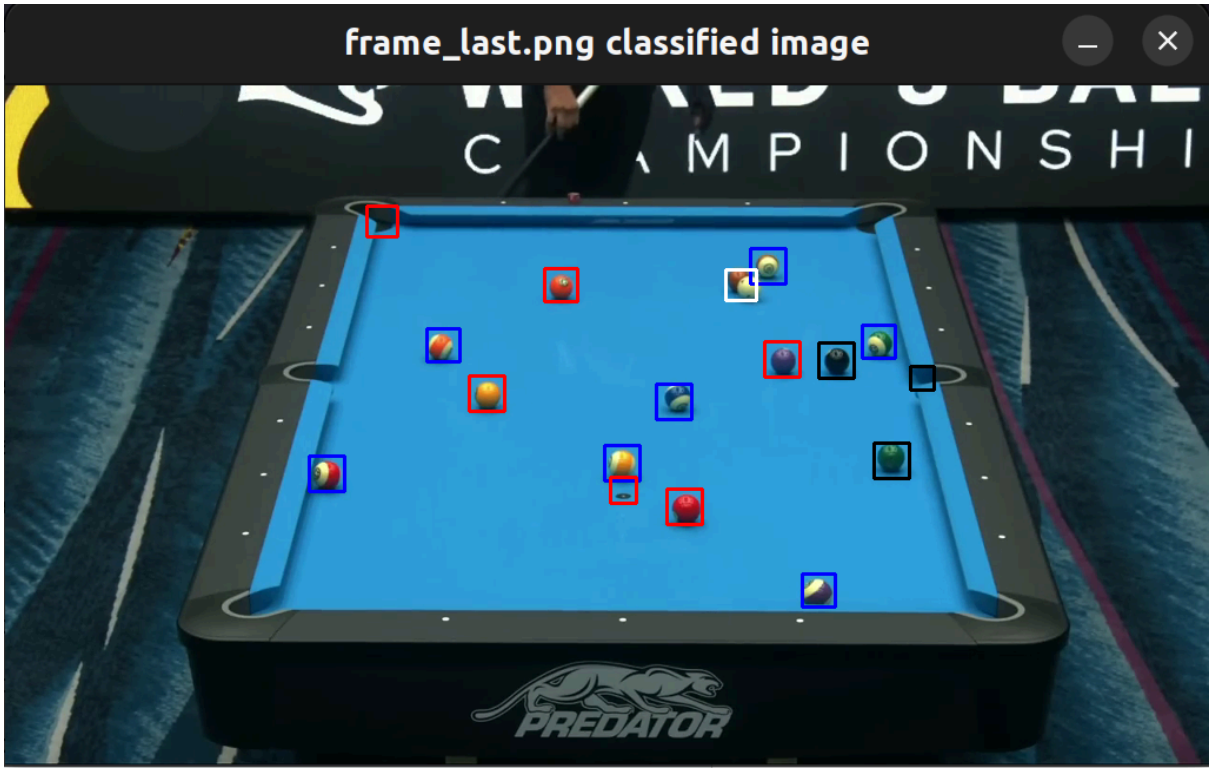
First frame:



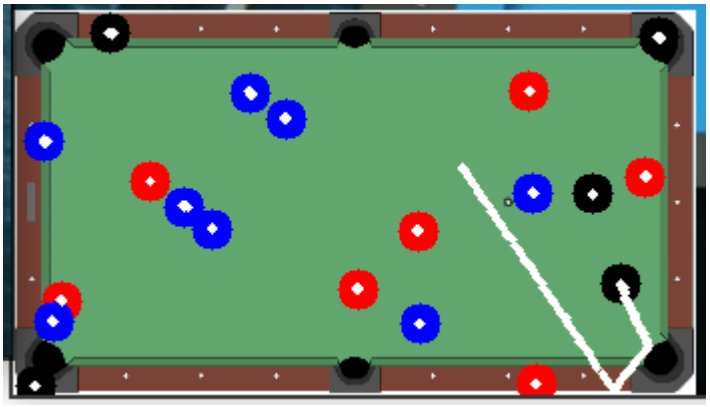


Last frame:





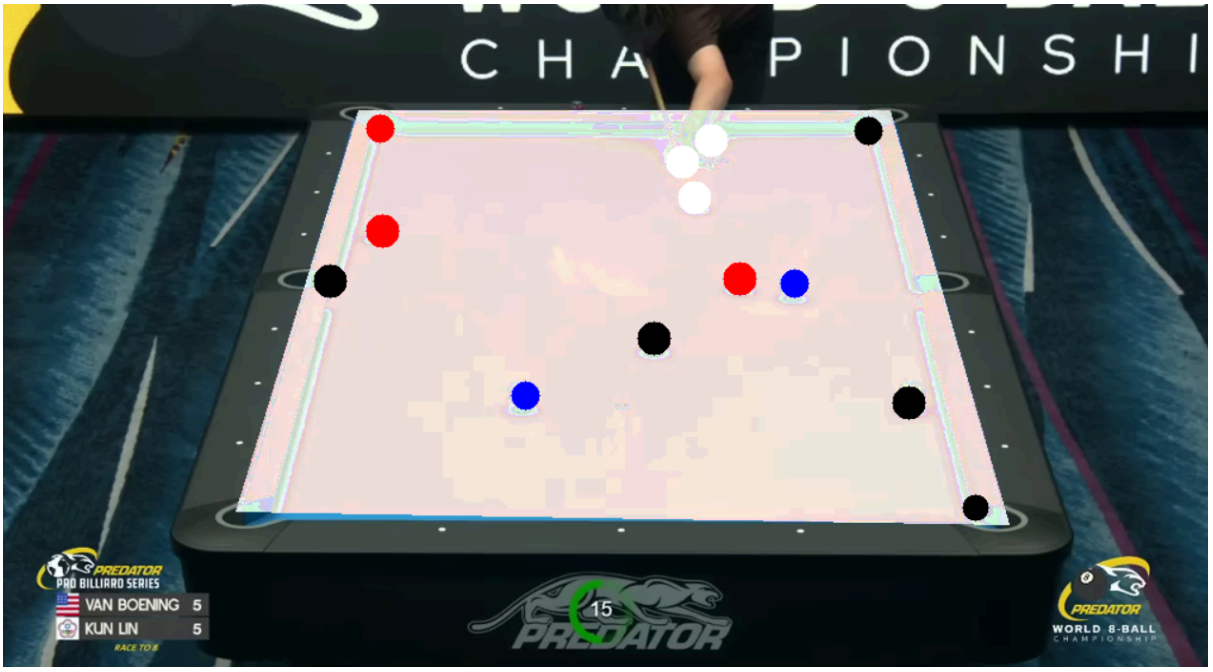
Final top view:

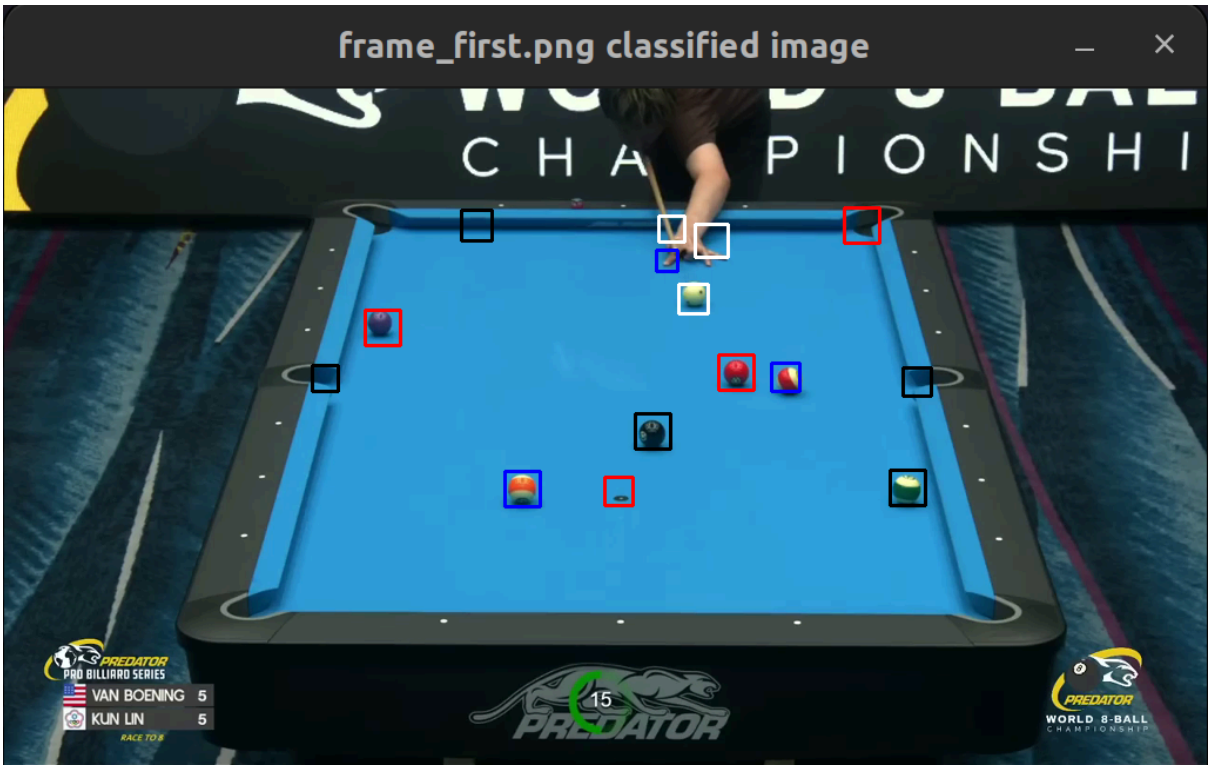


game1_clip3

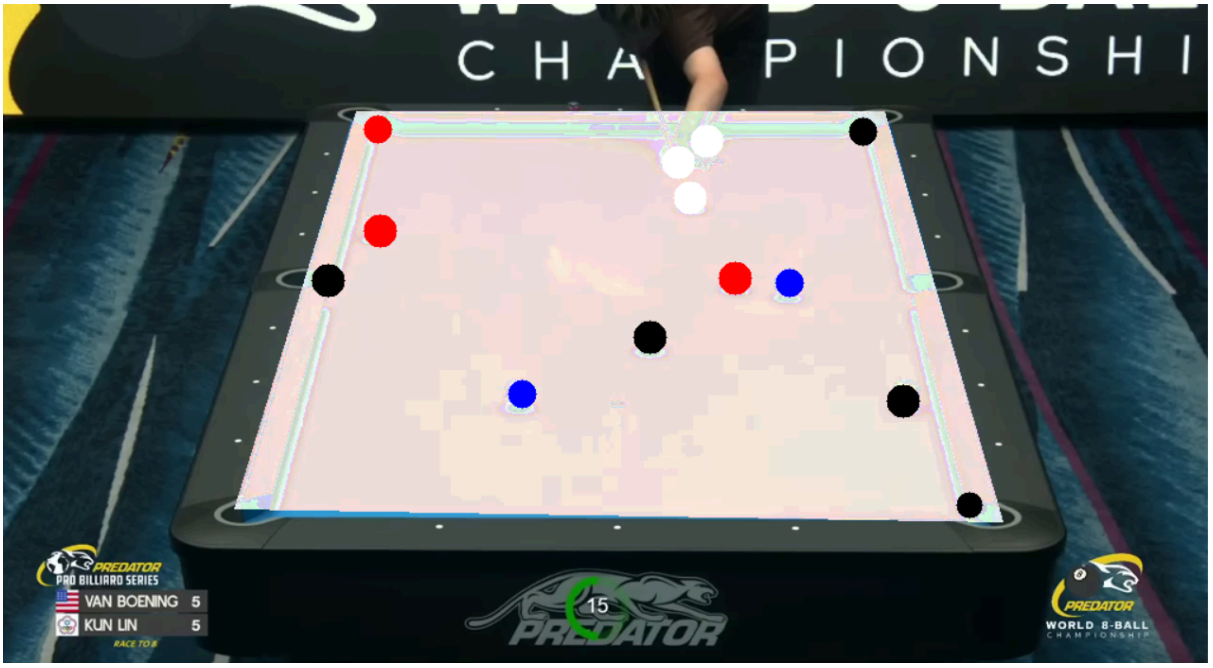
```
Mean IoU for image ../data/game1_clip3:
Class 0 IoU: 0.984243
Class 1 IoU: 0.216798
Class 2 IoU: 0.179395
Class 3 IoU: 0.327281
Class 4 IoU: 0.466741
Class 5 IoU: 0.950041
Mean IoU: 0.52075
mAP of frame_first.png 0.871212
Mean IoU for image ../data/game1_clip3:
Class 0 IoU: 0.986928
Class 1 IoU: 0.441704
Class 2 IoU: 0.363932
Class 3 IoU: 0.253289
Class 4 IoU: 0.545563
Class 5 IoU: 0.964914
Mean IoU: 0.592722
mAP of frame_last.png 0.420455
```

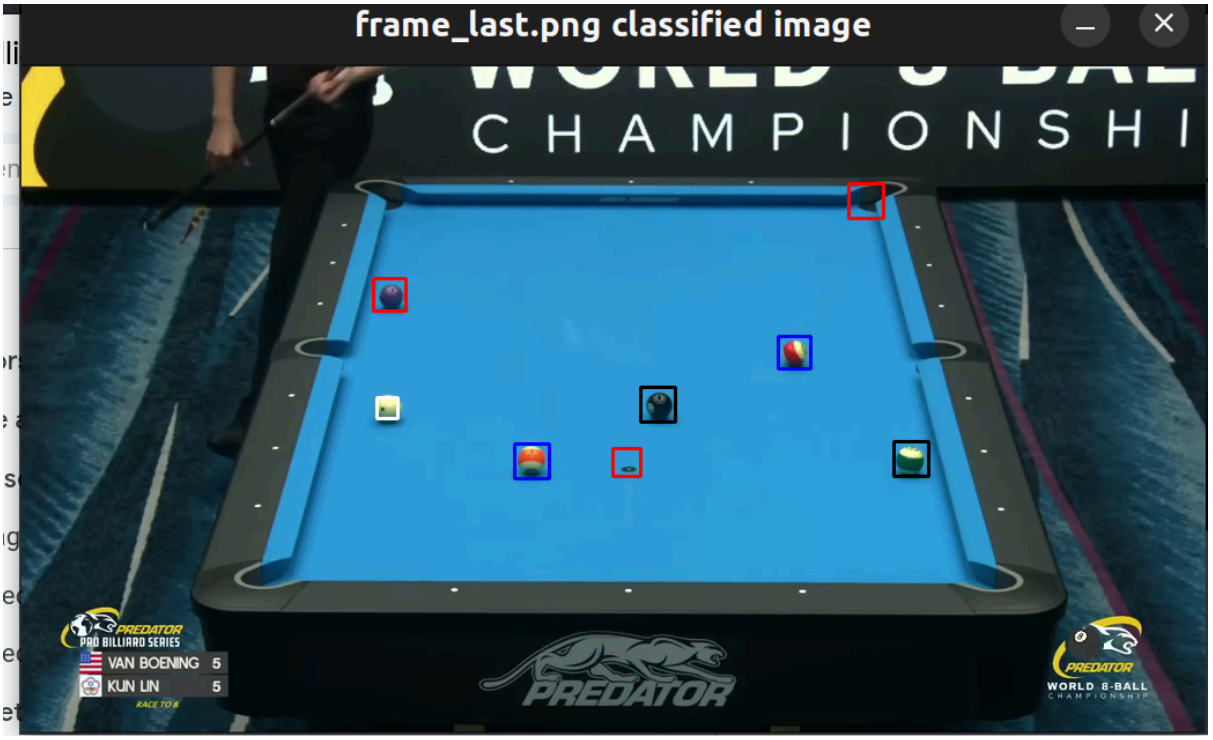
First frame:


PREDATOR
PRO BILLIARD SERIES
VAN BOENING 5
KUN LIN 5
RACE TO 8
15
PREDATOR
PREDATOR
WORLD 8-BALL
CHAMPIONSHIP

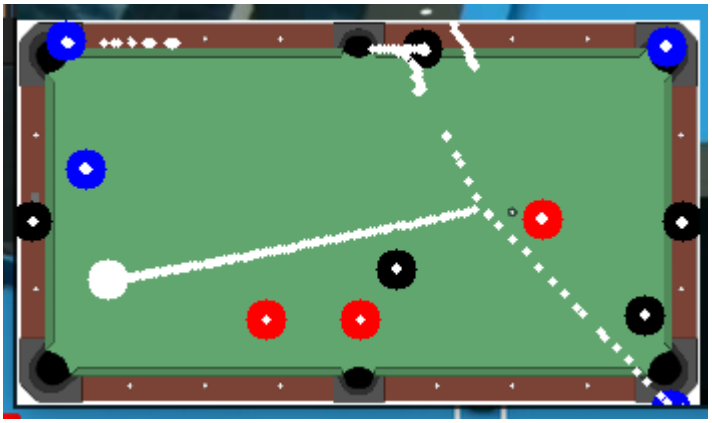

frame_first.png classified image
PREDATOR
PRO BILLIARD SERIES
VAN BOENING 5
KUN LIN 5
RACE TO 8
15
PREDATOR
PREDATOR
WORLD 8-BALL
CHAMPIONSHIP

Last frame:





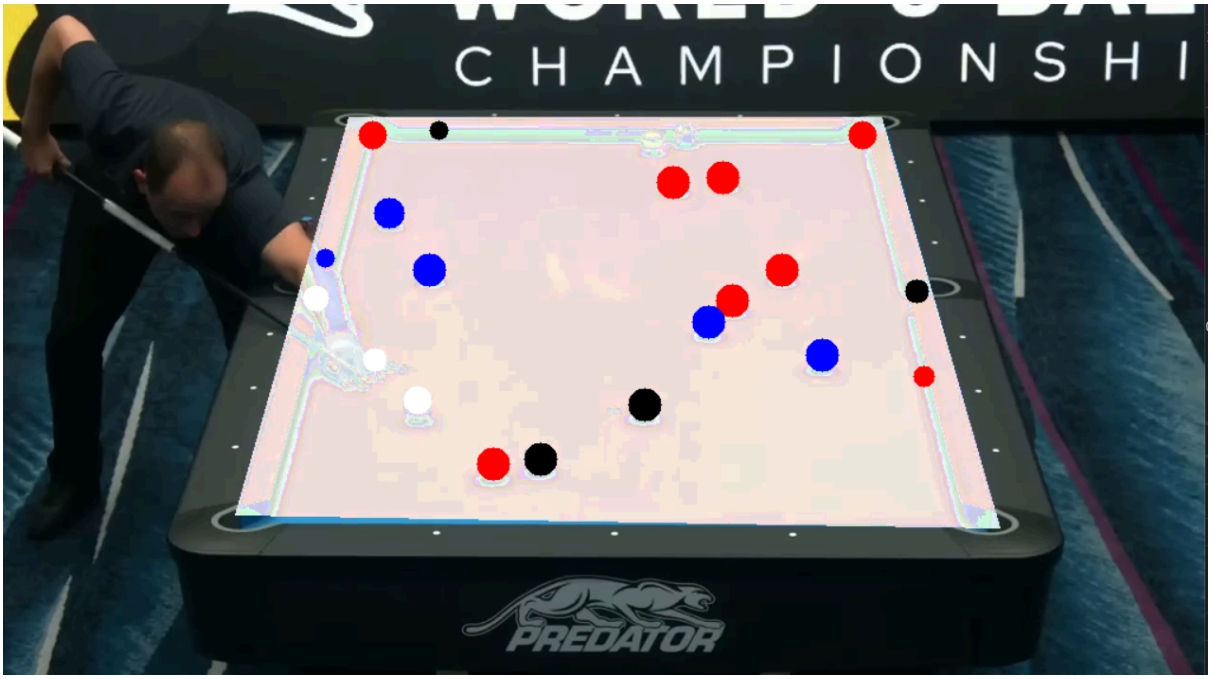
Final top view:

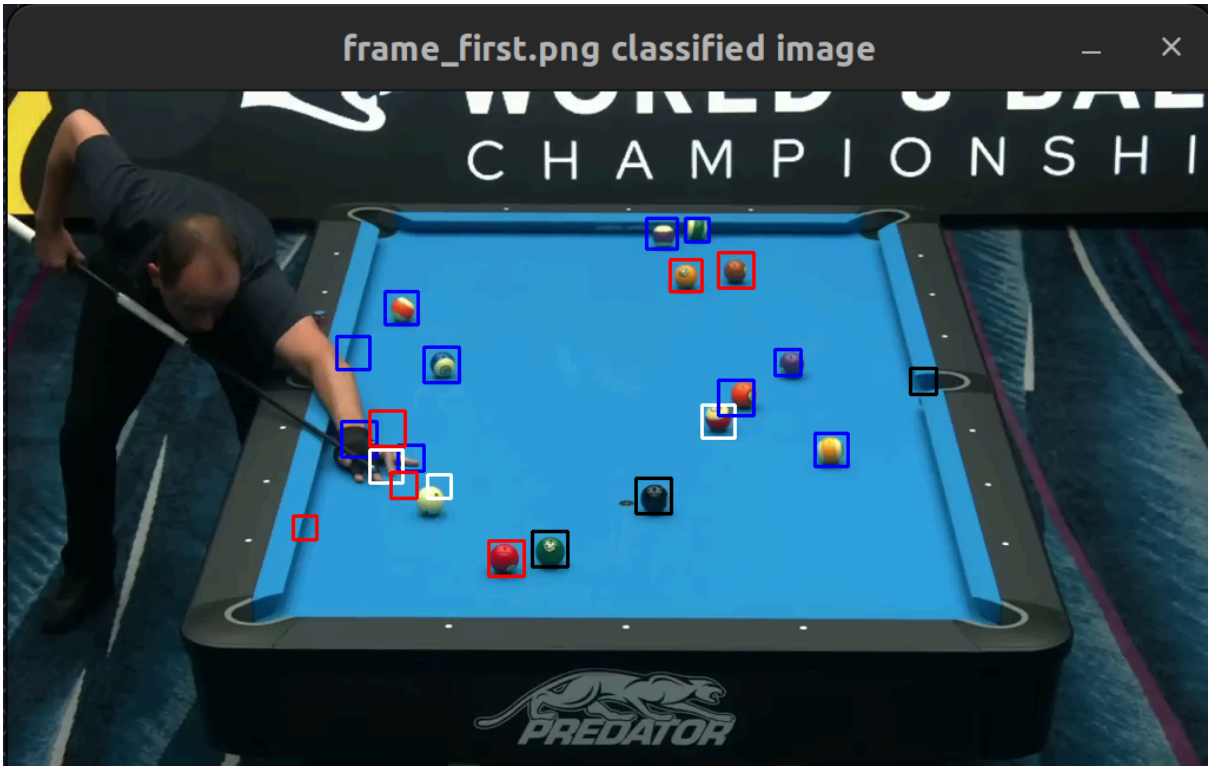


game1_clip4

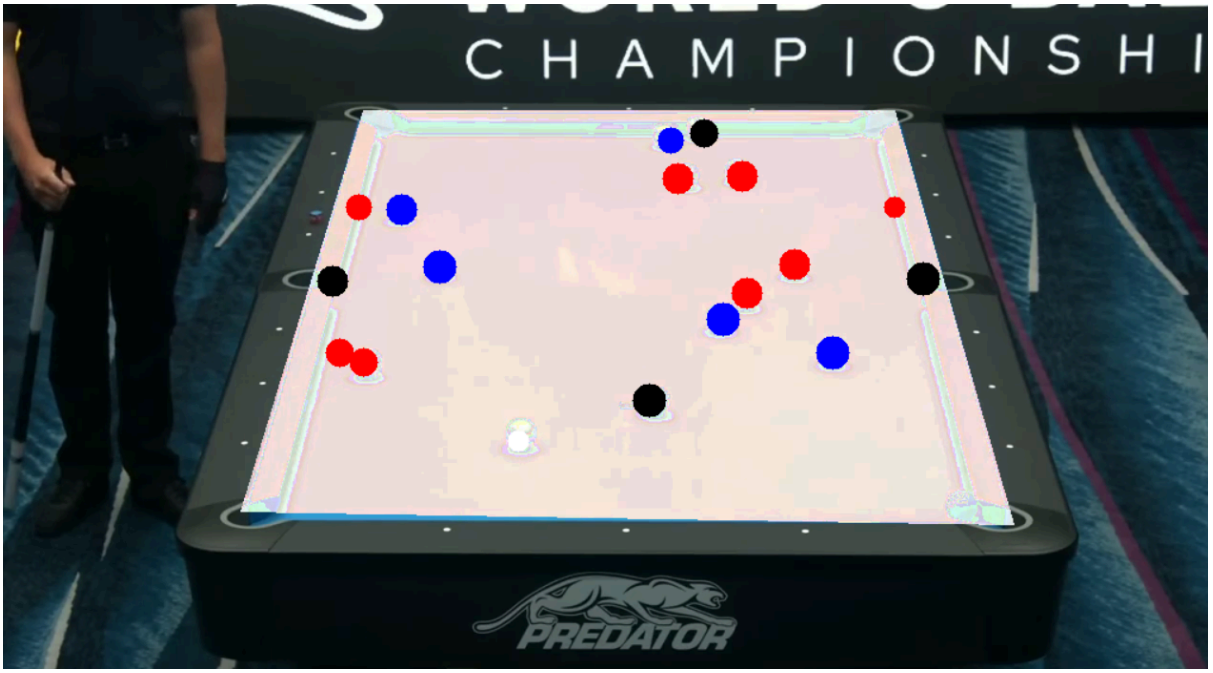
```
Mean IoU for image ../data/game1_clip4:
Class 0 IoU: 0.973661
Class 1 IoU: 0.0603795
Class 2 IoU: 0.290587
Class 3 IoU: 0.272067
Class 4 IoU: 0.329522
Class 5 IoU: 0.922499
Mean IoU: 0.474786
mAP of frame_first.png 0.512987
Mean IoU for image ../data/game1_clip4:
Class 0 IoU: 0.985233
Class 1 IoU: 0
Class 2 IoU: 0.295544
Class 3 IoU: 0.55767
Class 4 IoU: 0.703927
Class 5 IoU: 0.952437
Mean IoU: 0.582468
mAP of frame_last.png 0.5
```

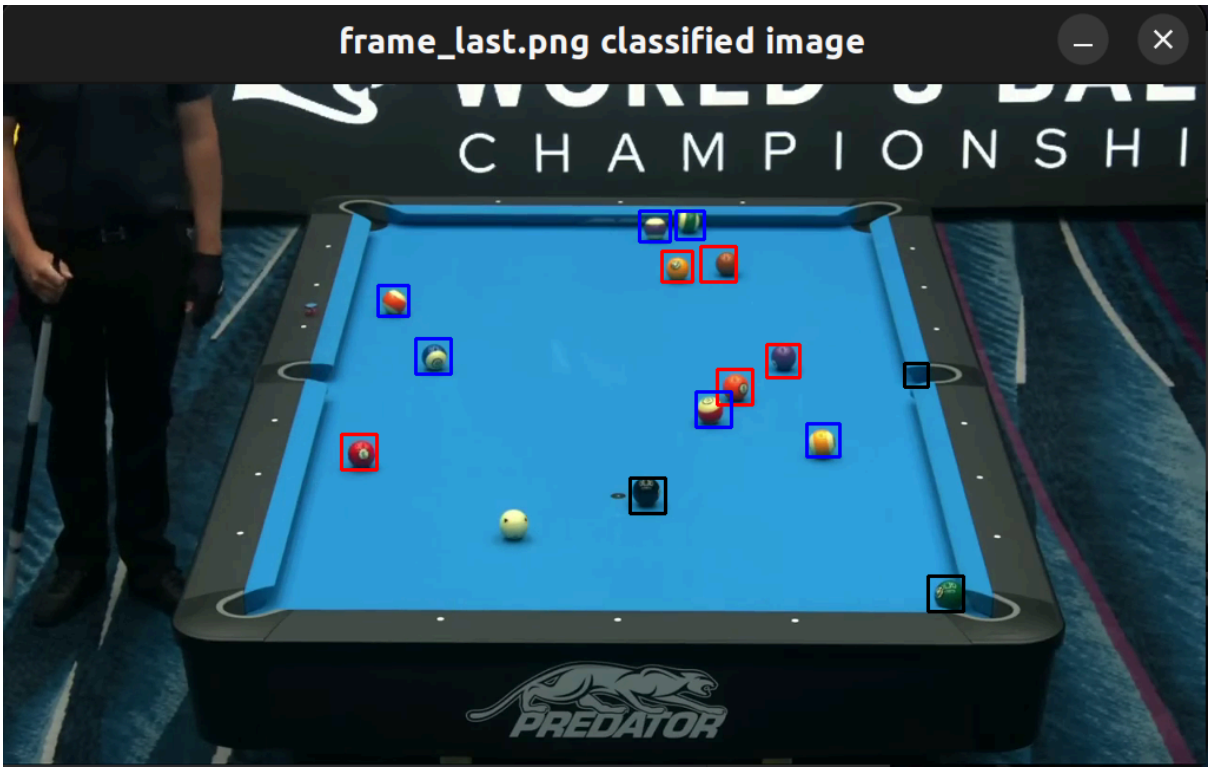
First frame:


CHAMPIONSHI
PREDATOR

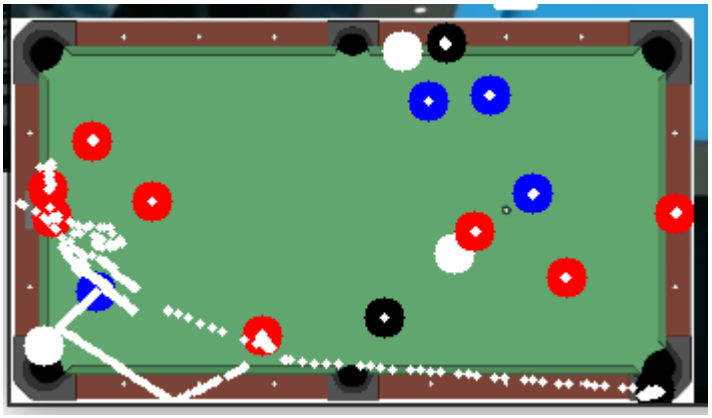

frame_first.png classified image
CHAMPIONSHI
PREDATOR

Last frame:





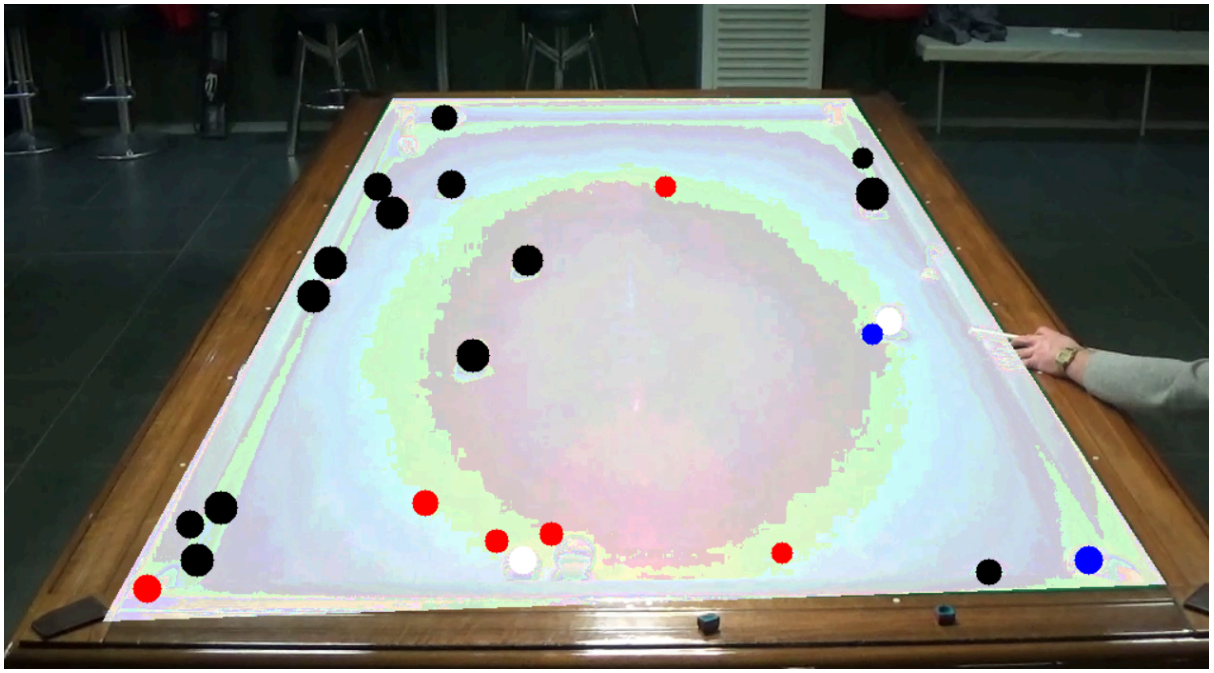
Final top view:

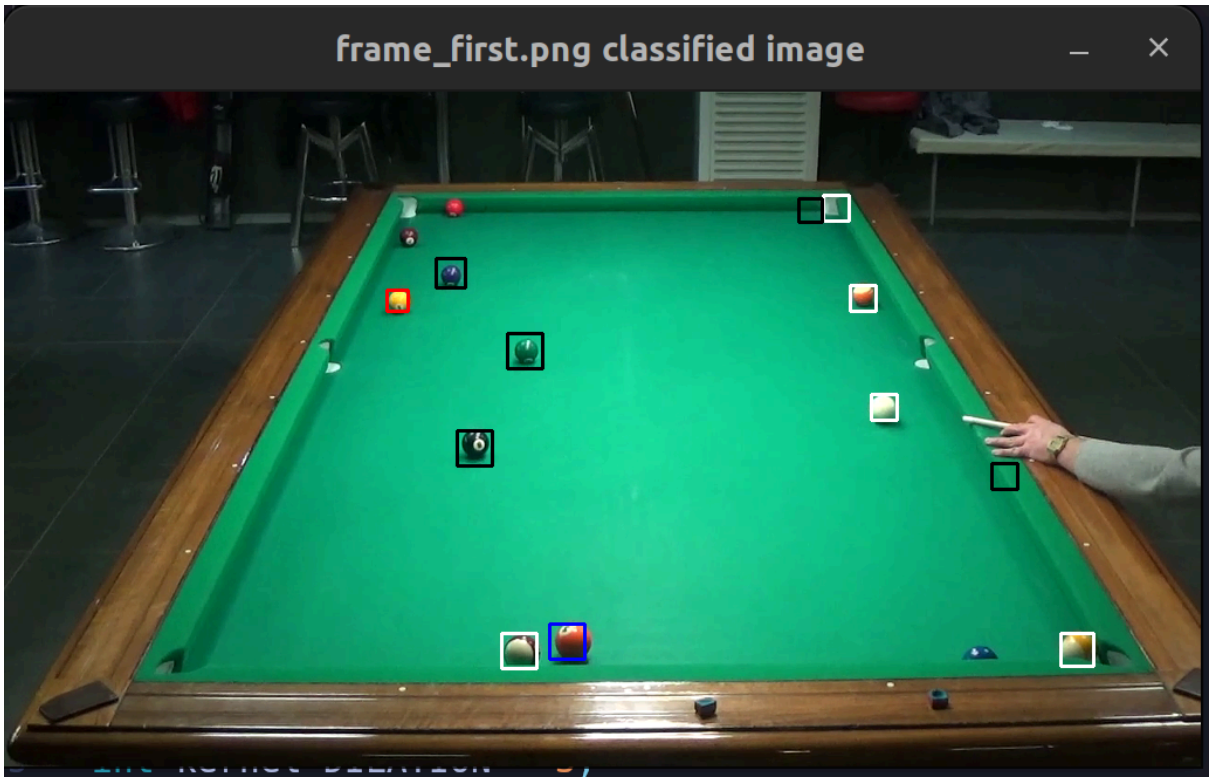


game2_clip1

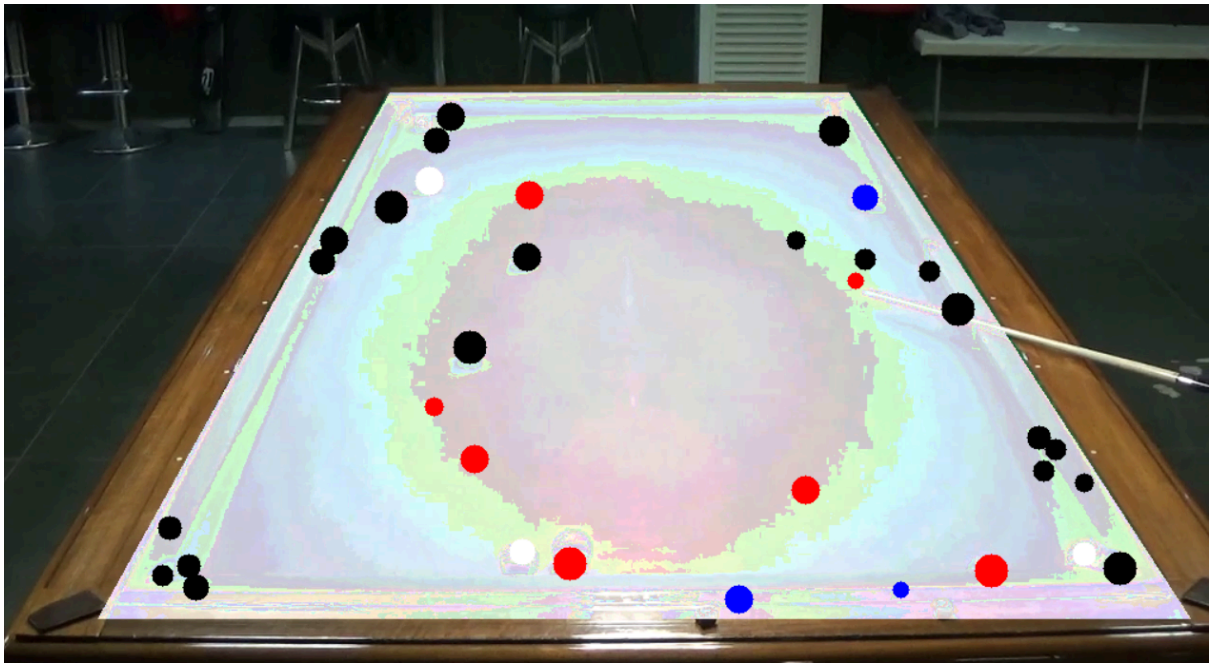
```
Mean IoU for image ../data/game2_clip1:
Class 0 IoU: 0.967545
Class 1 IoU: 0.157484
Class 2 IoU: 0.185231
Class 3 IoU: 0.0736173
Class 4 IoU: 0
Class 5 IoU: 0.946786
Mean IoU: 0.388444
mAP of frame_first.png 0.178788
Mean IoU for image ../data/game2_clip1:
Class 0 IoU: 0.962925
Class 1 IoU: 0.114269
Class 2 IoU: 0.337838
Class 3 IoU: 0.0574402
Class 4 IoU: 0.0912577
Class 5 IoU: 0.944848
Mean IoU: 0.418096
mAP of frame_last.png 0.340909
```

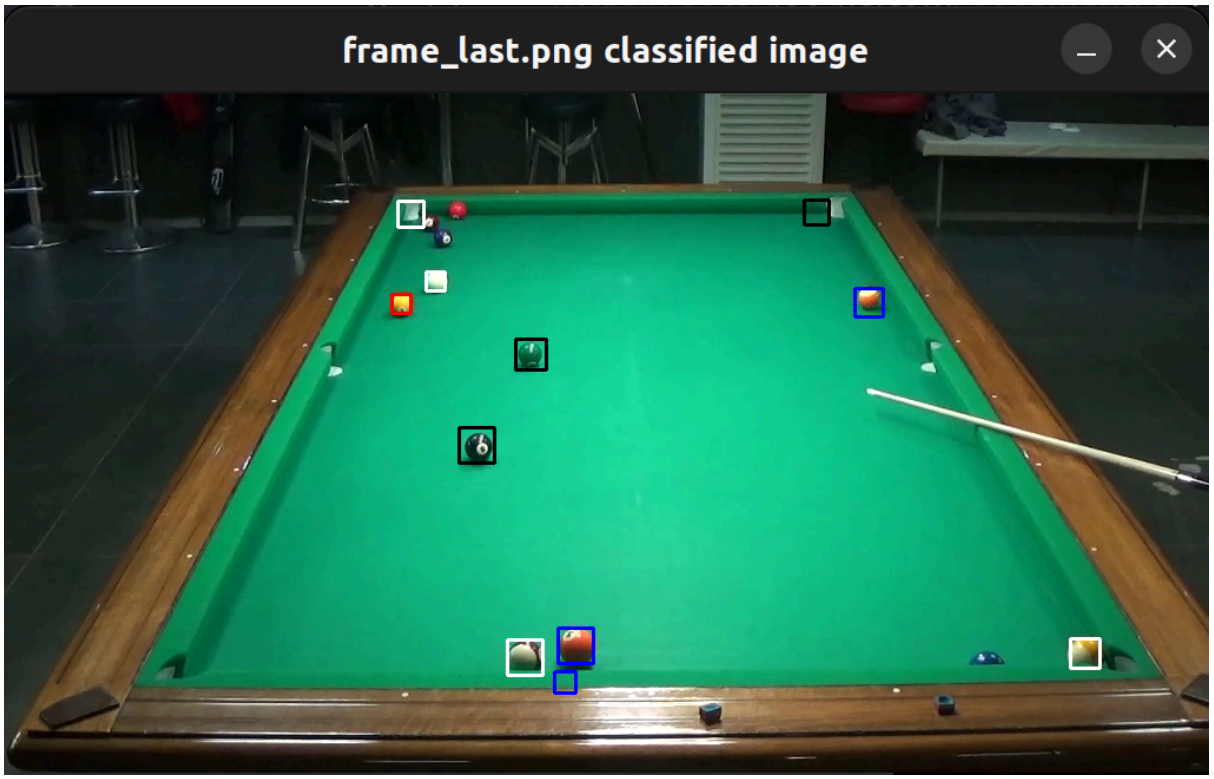
First frame:



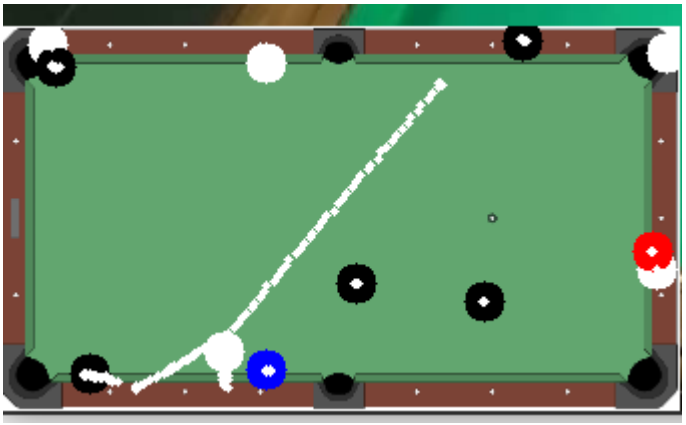


Last frame:





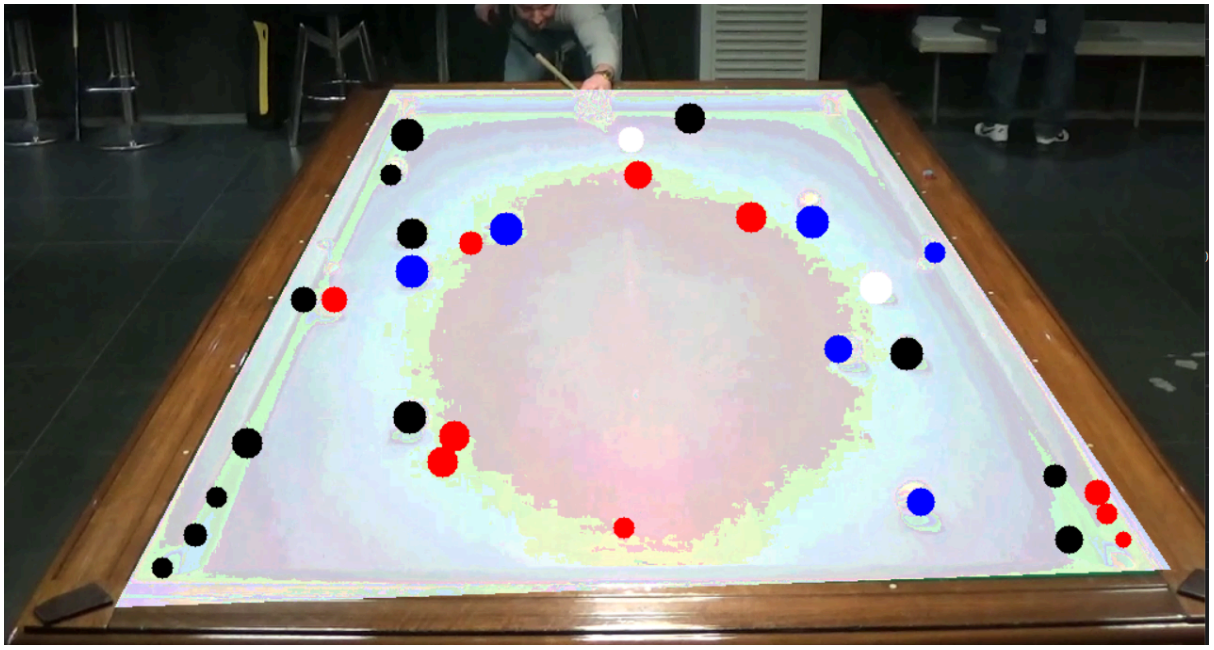
Final top view:

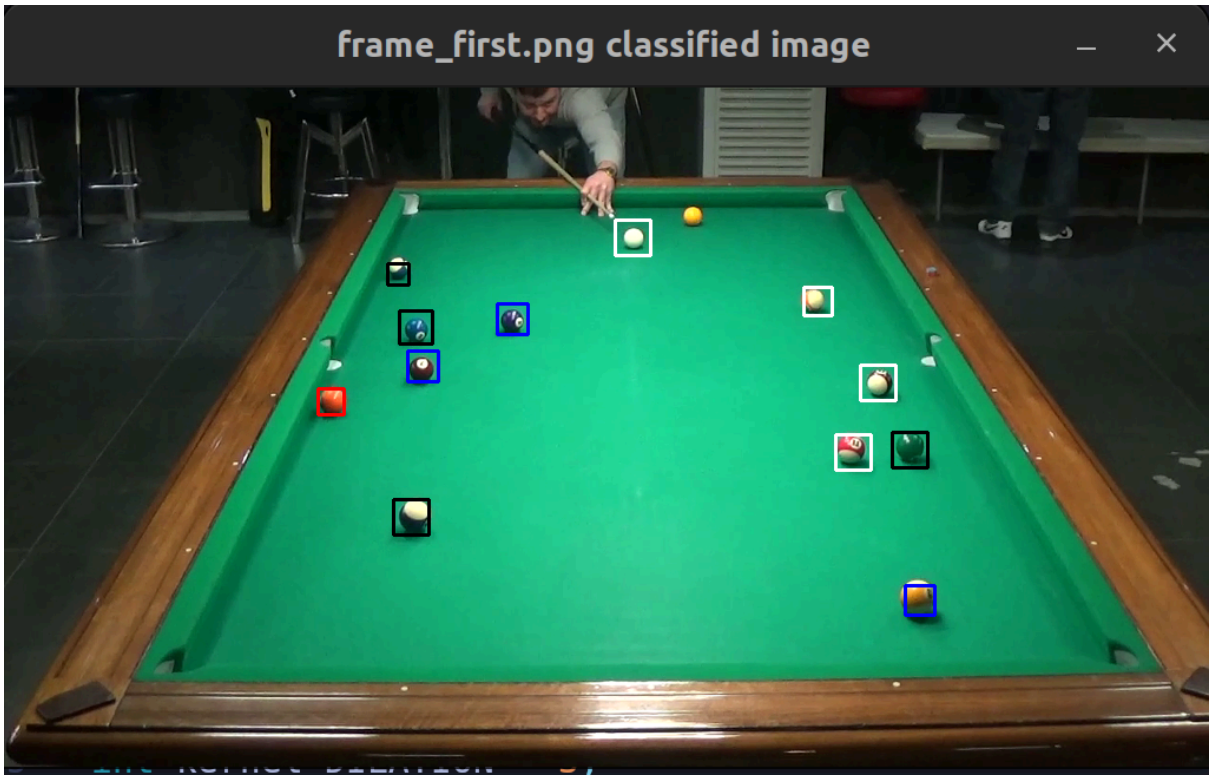


game2_clip2

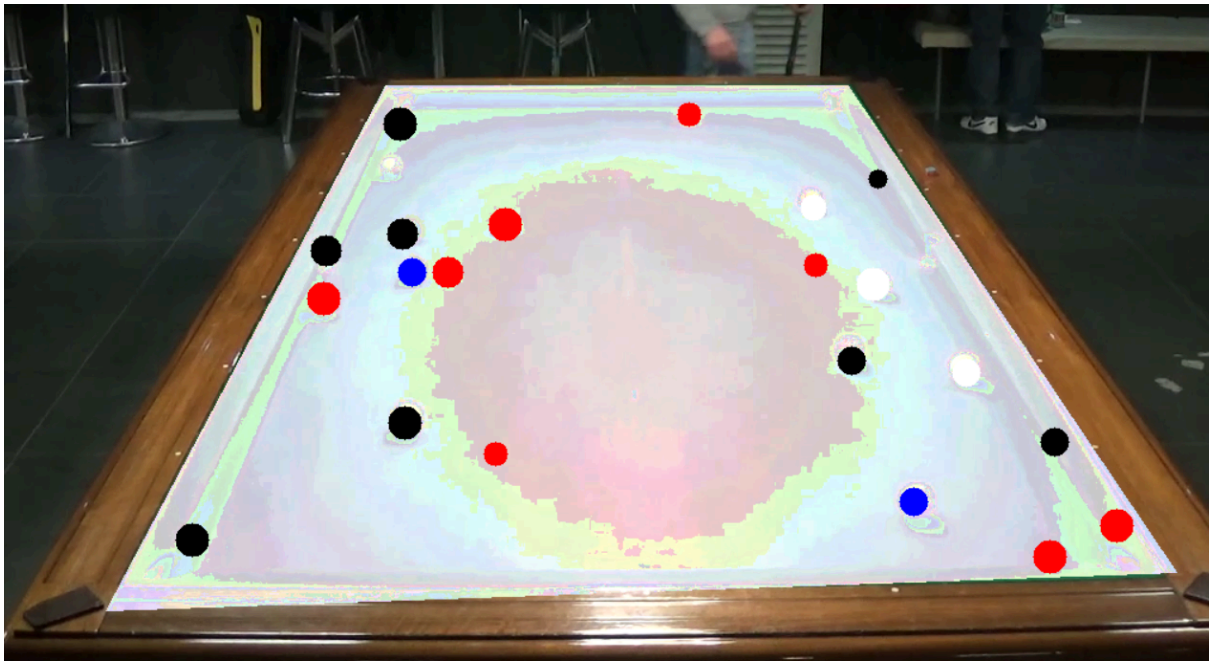
```
Class 0 IoU: 0.963828
Class 1 IoU: 0.116667
Class 2 IoU: 0
Class 3 IoU: 0.162231
Class 4 IoU: 0.111223
Class 5 IoU: 0.945541
Mean IoU: 0.383248
mAP of frame_first.png 0.0833333
Mean IoU for image ../data/game2_clip2:
Class 0 IoU: 0.966151
Class 1 IoU: 0.170856
Class 2 IoU: 0
Class 3 IoU: 0.059167
Class 4 IoU: 0.14773
Class 5 IoU: 0.946272
Mean IoU: 0.381696
mAP of frame_last.png 0.121212
```

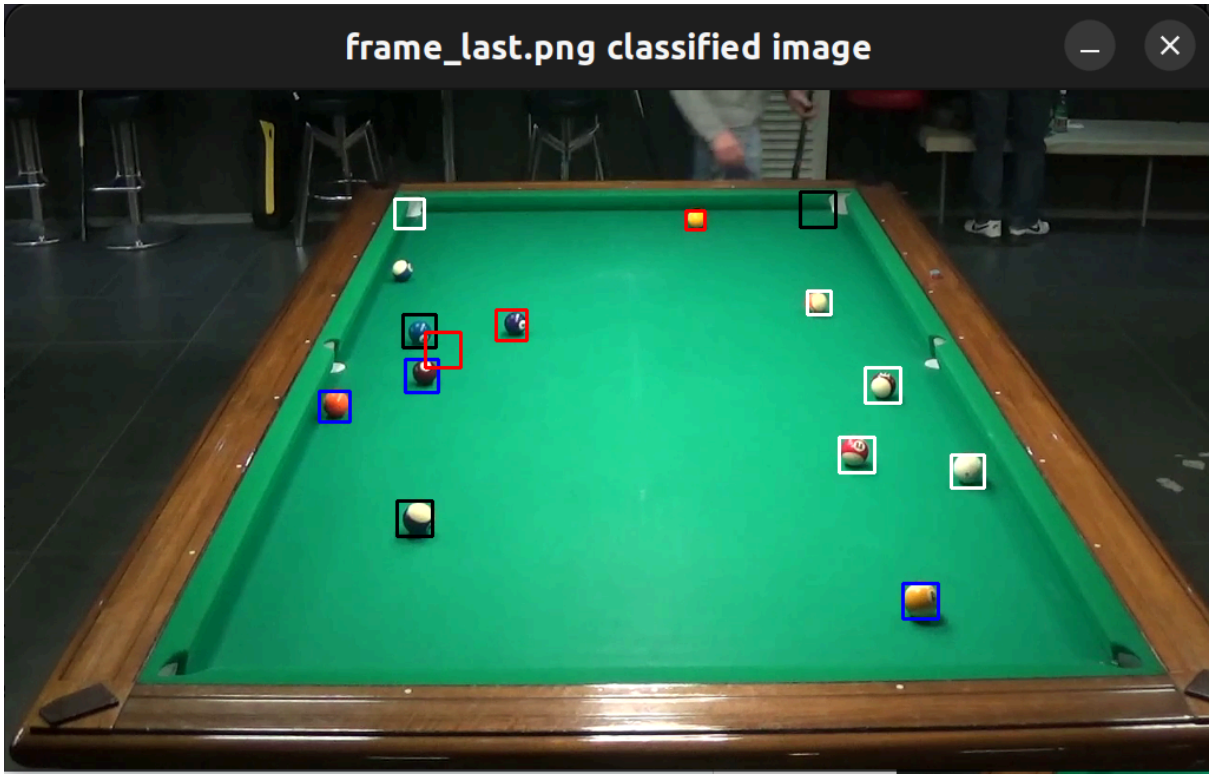
First frame:



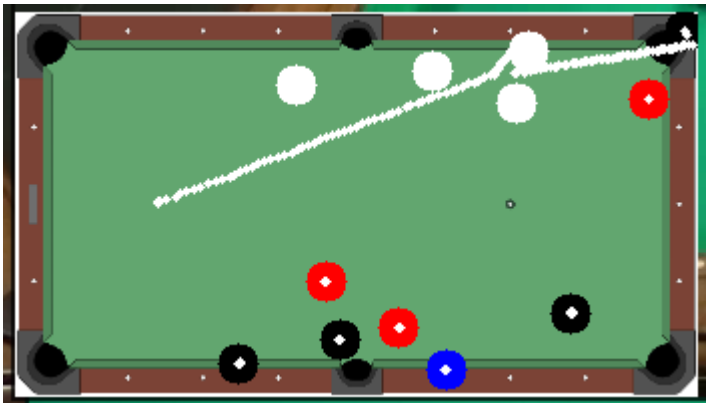


Last frame:

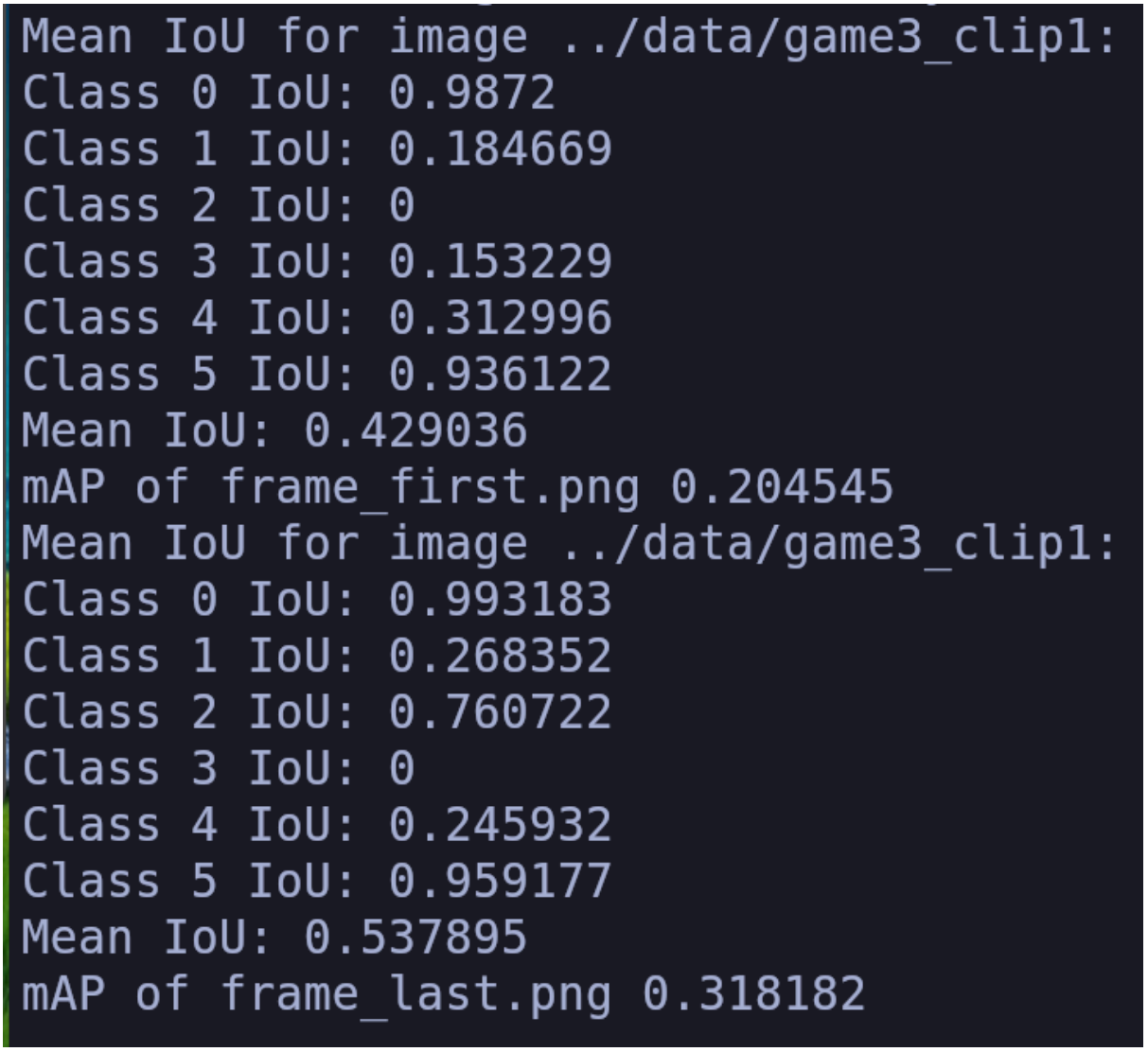




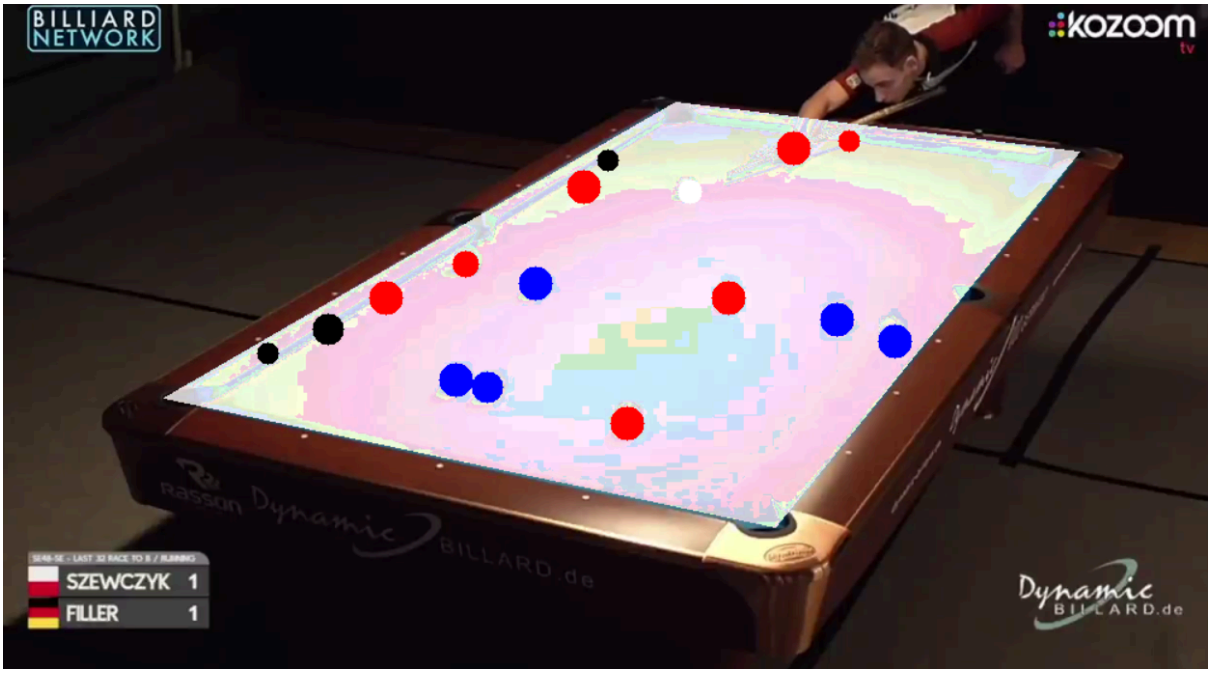
Final top view:

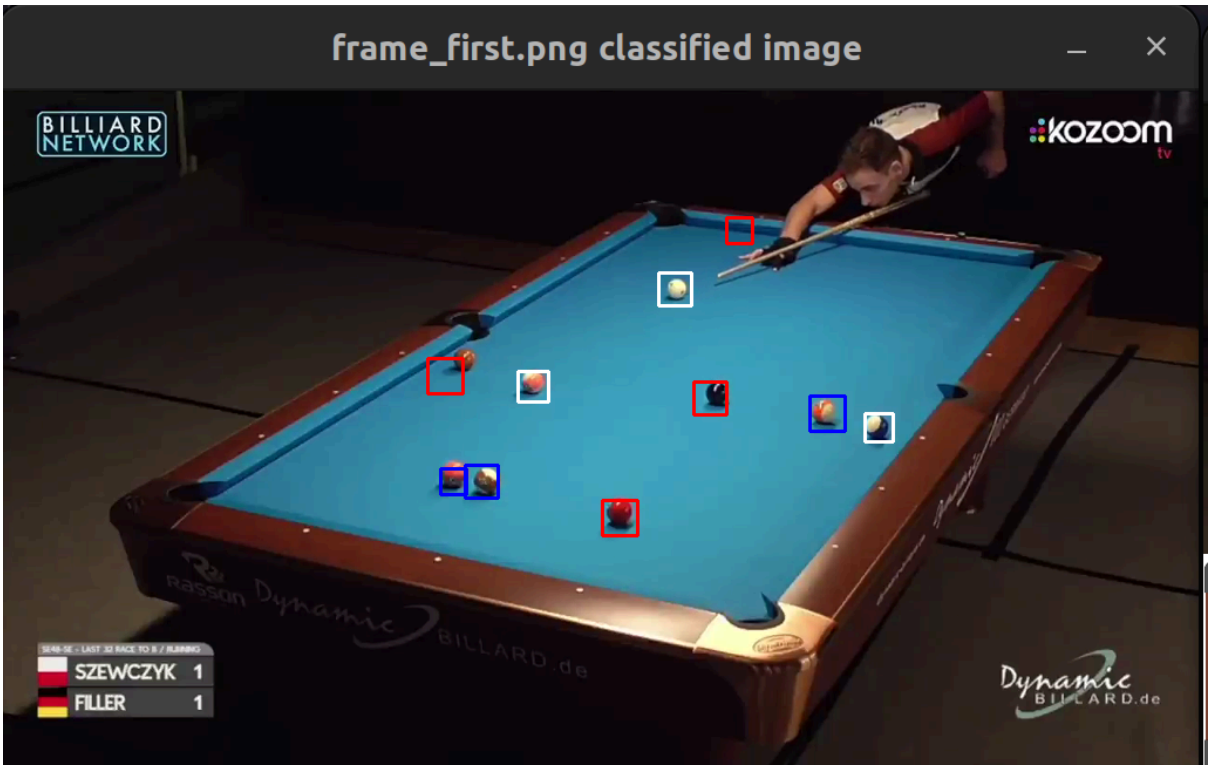


game3_clip1

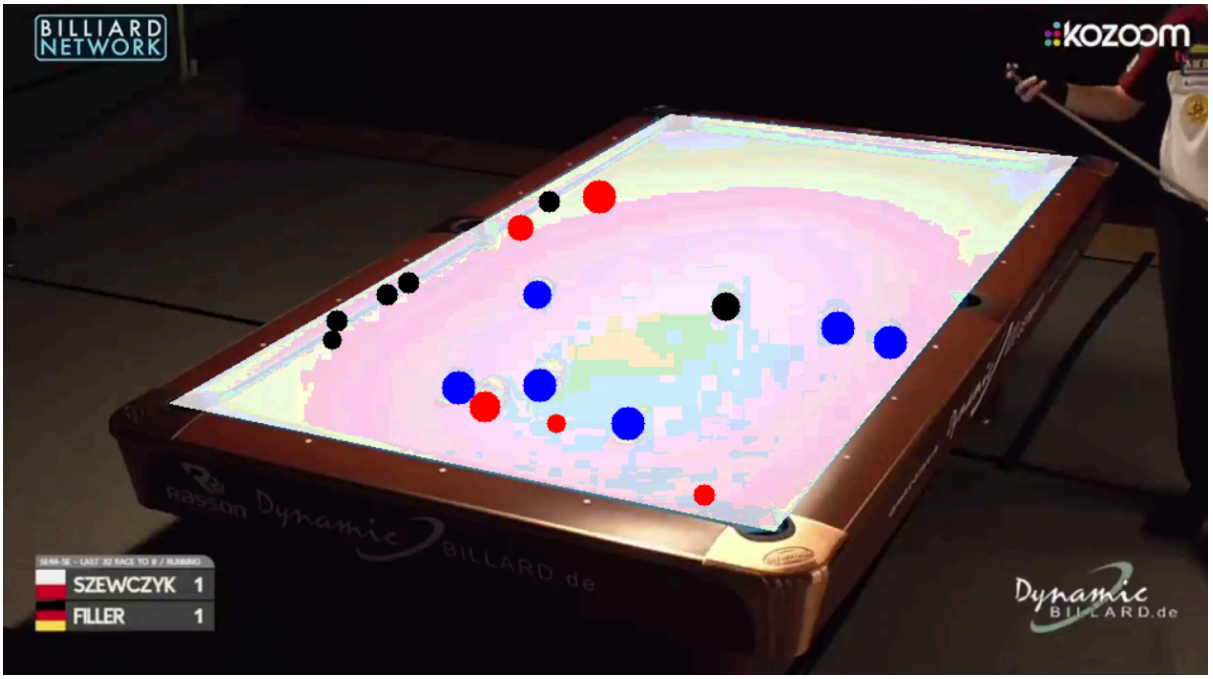


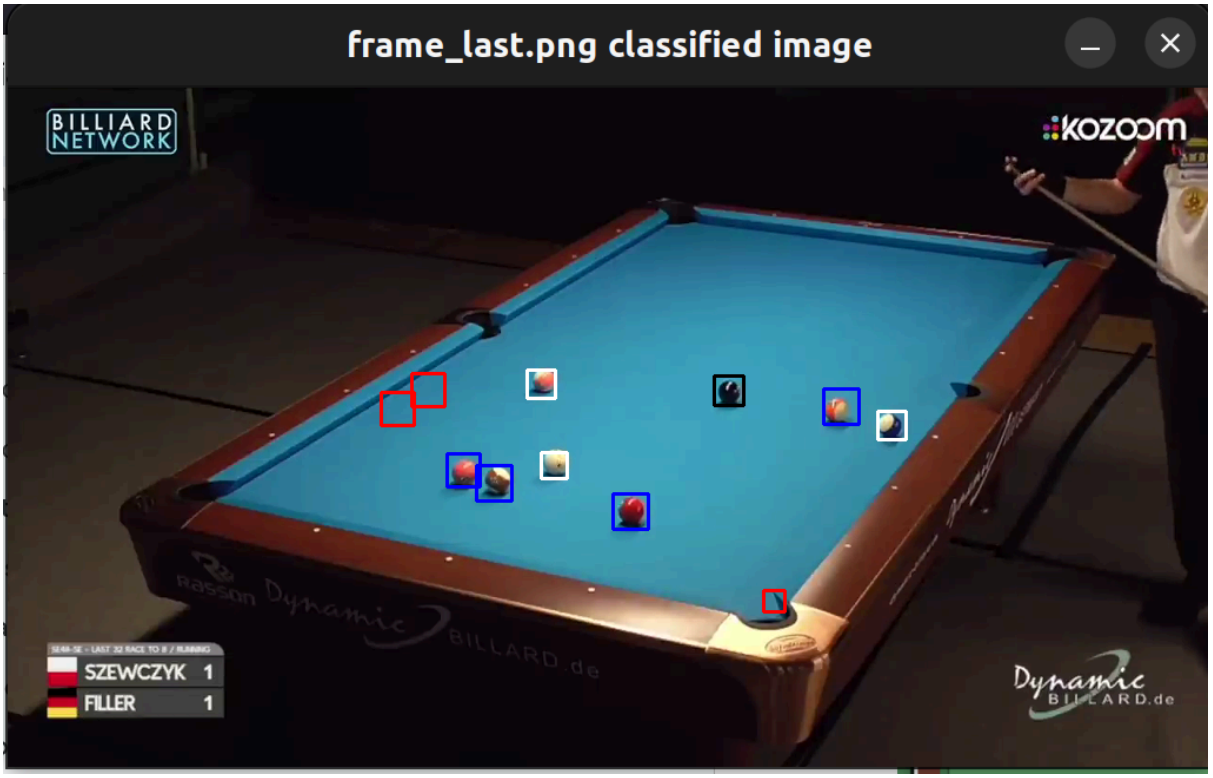
First frame:


BILLIARD NETWORK
kozoom
SZEWCZYK 1
FILLER 1
Dynamic BILLARD.de

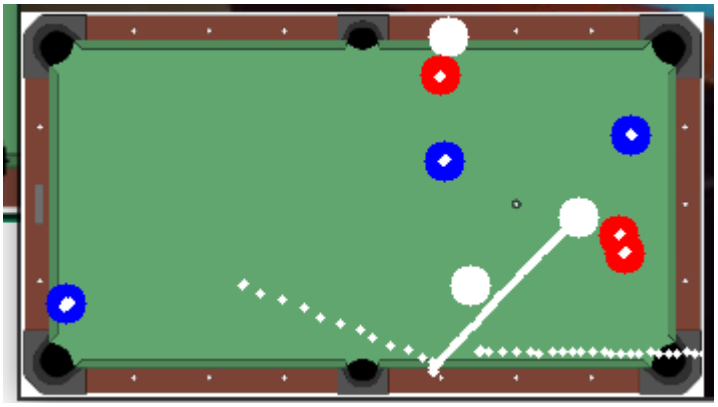

frame_first.png classified image
BILLIARD NETWORK
kozoom
SZEWCZYK 1
FILLER 1
Dynamic BILLARD.de

Last frame:





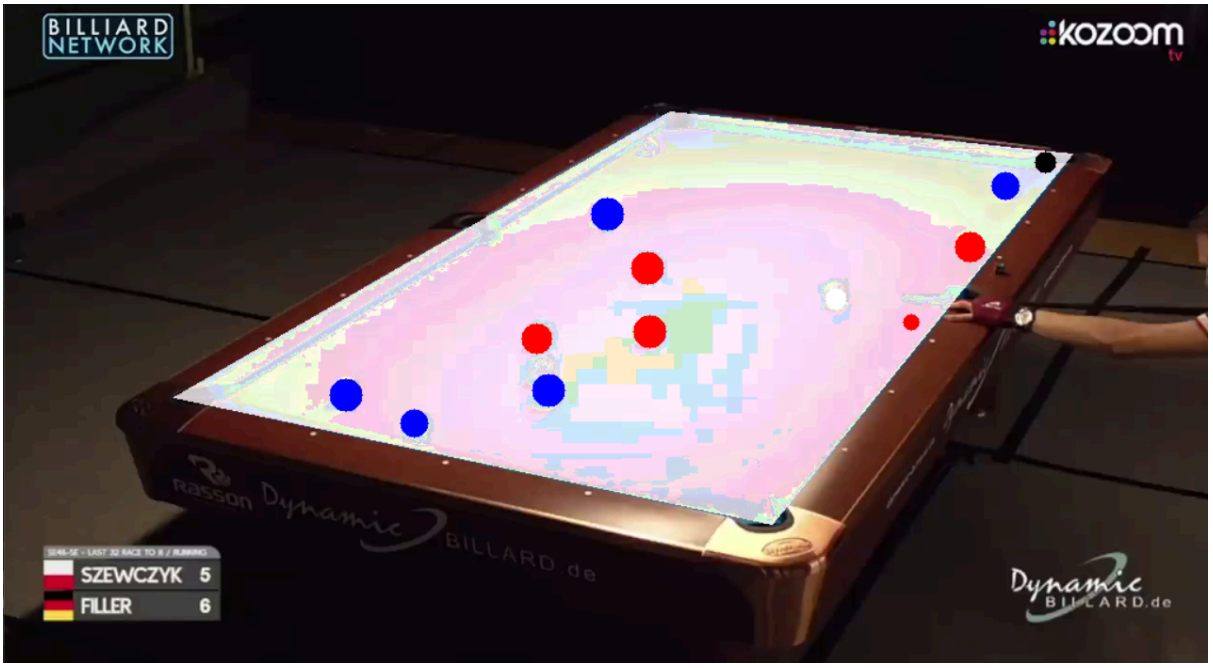
Final top view:

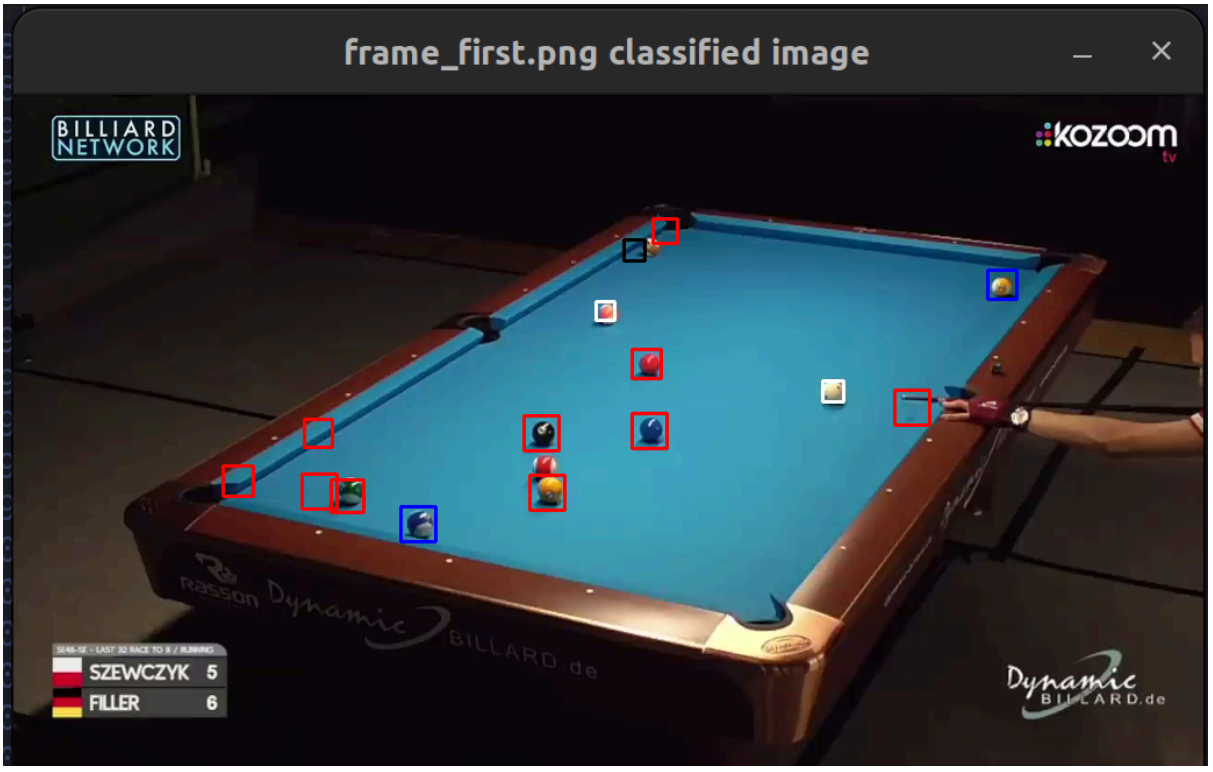


game3_clip2

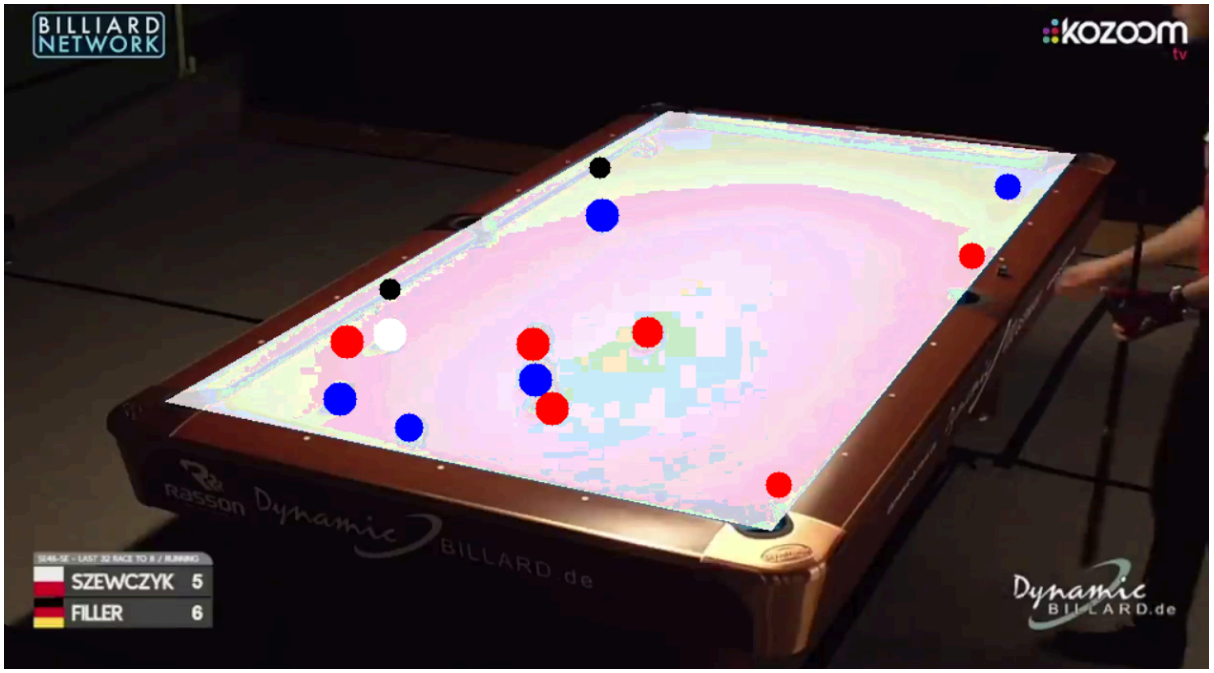
```
Mean IoU for image ../data/game3_clip2:
Class 0 IoU: 0.991707
Class 1 IoU: 0.358182
Class 2 IoU: 0
Class 3 IoU: 0.224498
Class 4 IoU: 0.296132
Class 5 IoU: 0.943405
Mean IoU: 0.468987
mAP of frame_first.png 0.505682
Mean IoU for image ../data/game3_clip2:
Class 0 IoU: 0.992932
Class 1 IoU: 0.340054
Class 2 IoU: 0
Class 3 IoU: 0.231527
Class 4 IoU: 0.292206
Class 5 IoU: 0.950626
Mean IoU: 0.467891
mAP of frame_last.png 0.25
```

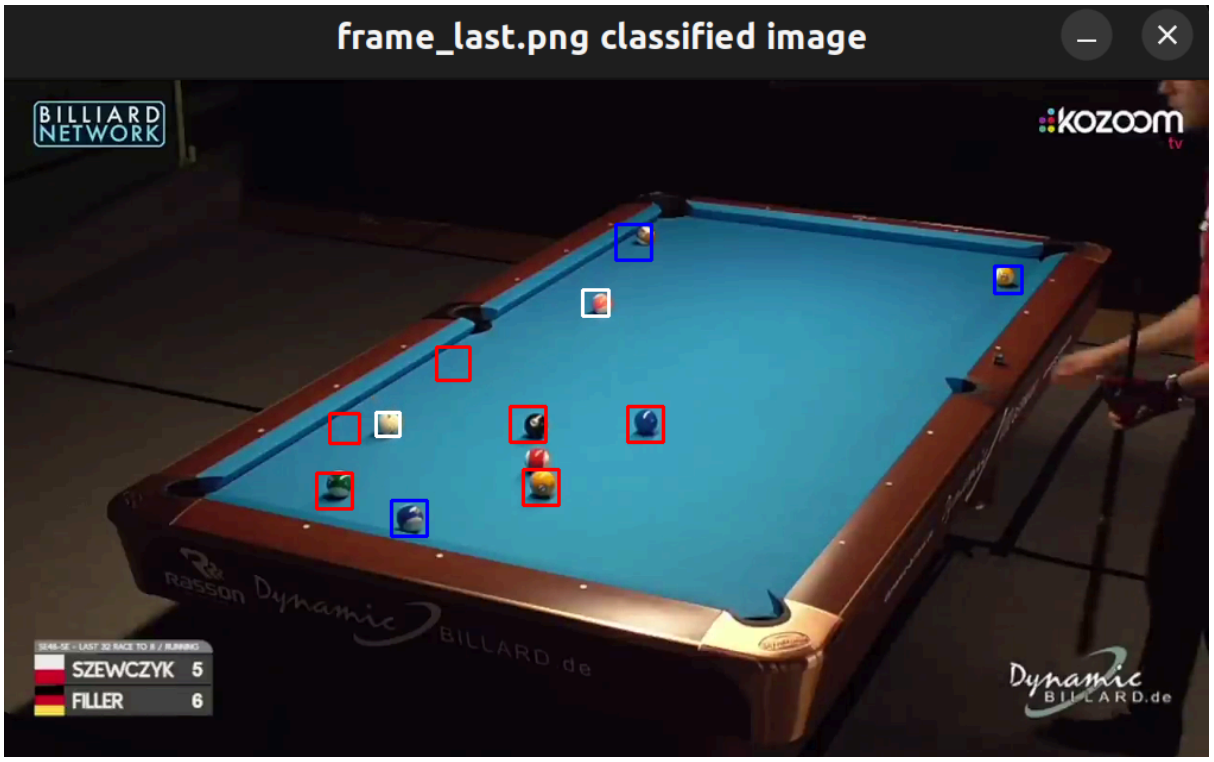
First frame:


BILLIARD NETWORK
kozoom tv
SZEWCZYK 5
FILLER 6
Dynamic BILLARD.de

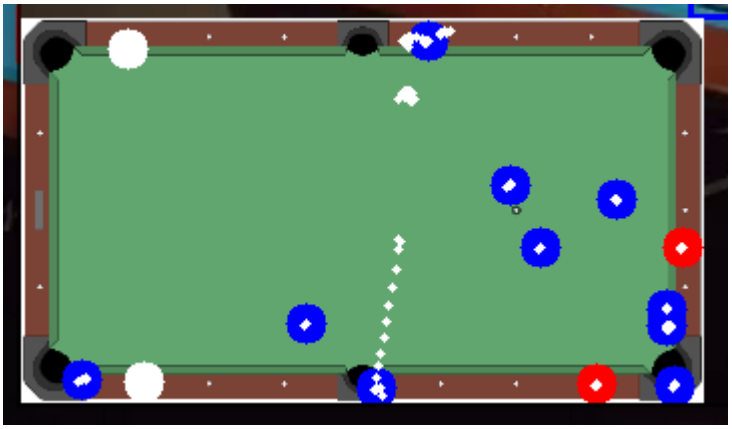

frame_first.png classified image
BILLIARD NETWORK
kozoom tv
SZEWCZYK 5
FILLER 6
Dynamic BILLARD.de

Last frame:





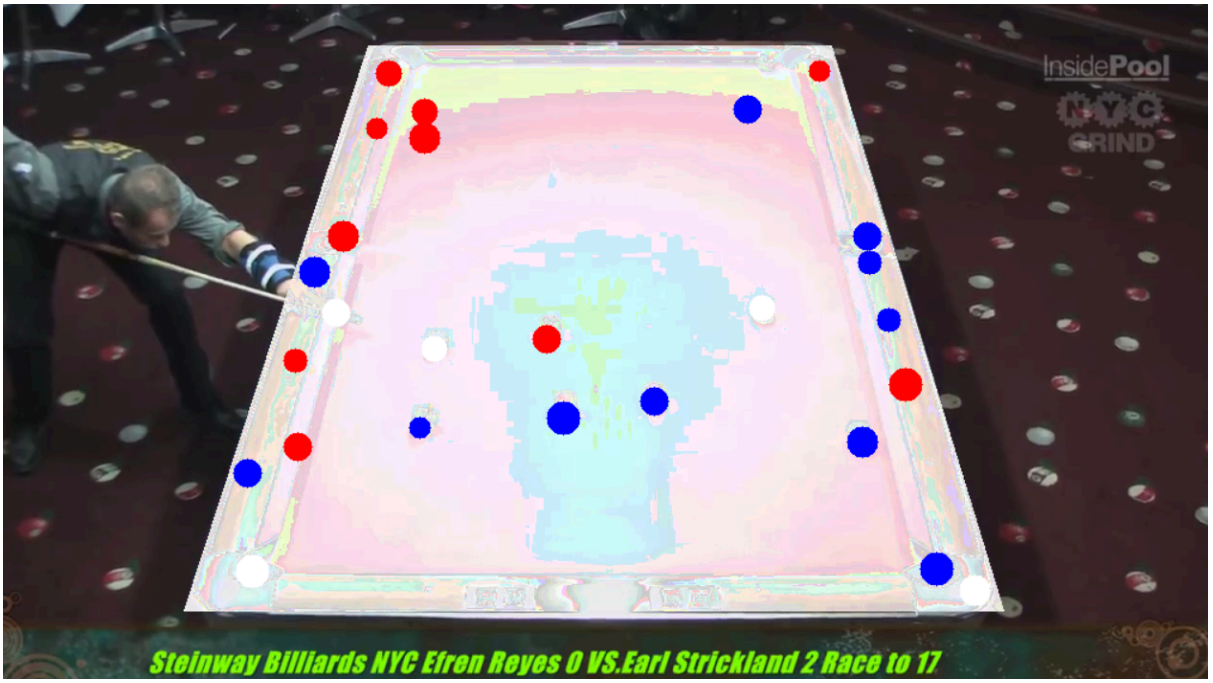
Final top view:

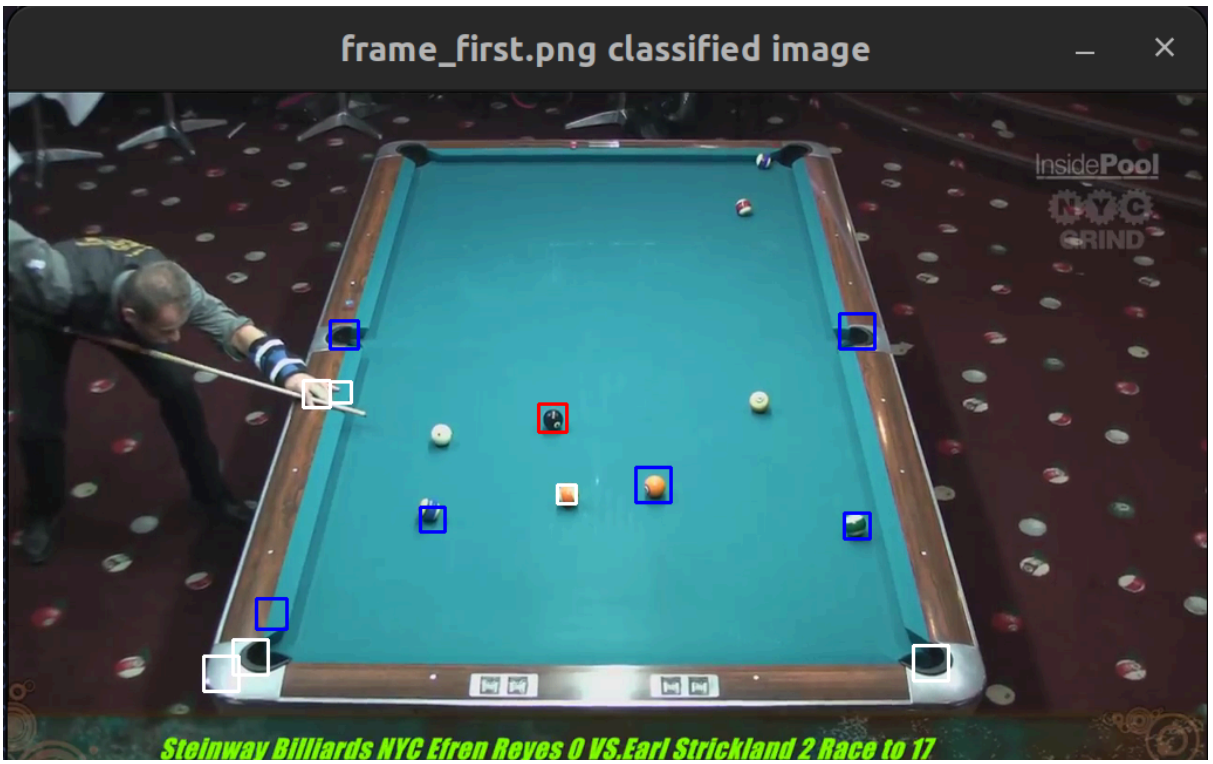


game4_clip1

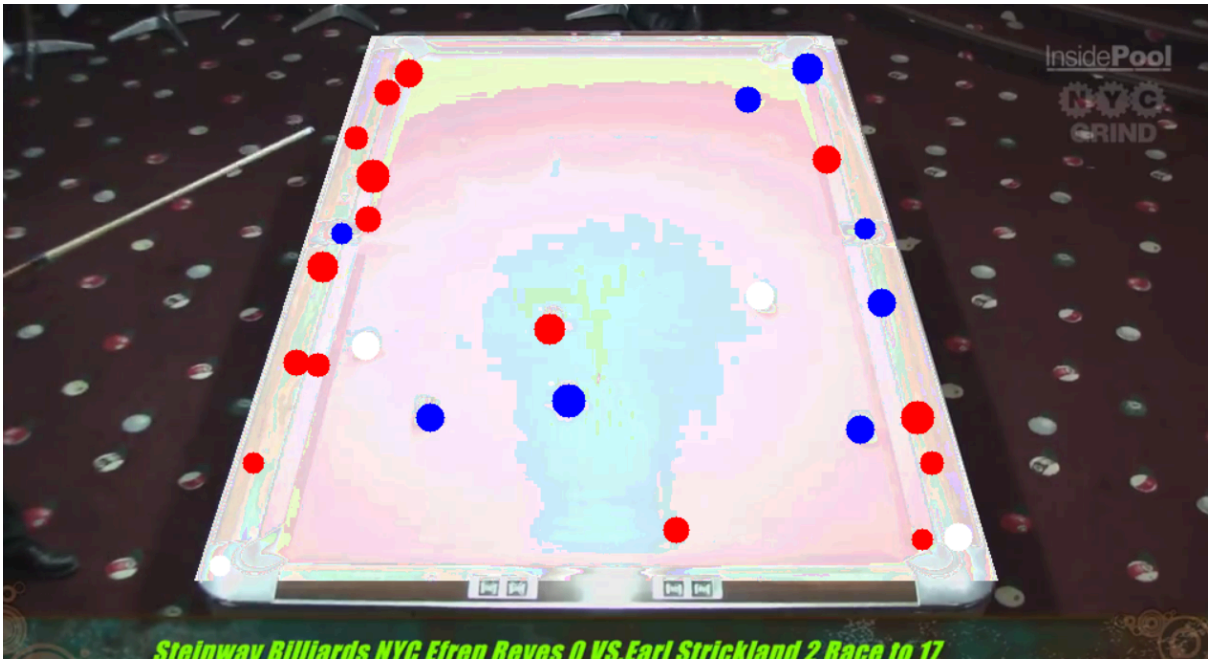
```
Mean IoU for image ../data/game4_clip1:
Class 0 IoU: 0.858838
Class 1 IoU: 0
Class 2 IoU: 0
Class 3 IoU: 0
Class 4 IoU: 0.114576
Class 5 IoU: 0.782902
Mean IoU: 0.292719
mAP of frame_first.png 0.0227273
Mean IoU for image ../data/game4_clip1:
Class 0 IoU: 0.863571
Class 1 IoU: 0.0489614
Class 2 IoU: 0
Class 3 IoU: 0
Class 4 IoU: 0.144458
Class 5 IoU: 0.794082
Mean IoU: 0.308512
mAP of frame_last.png 0.0454545
```

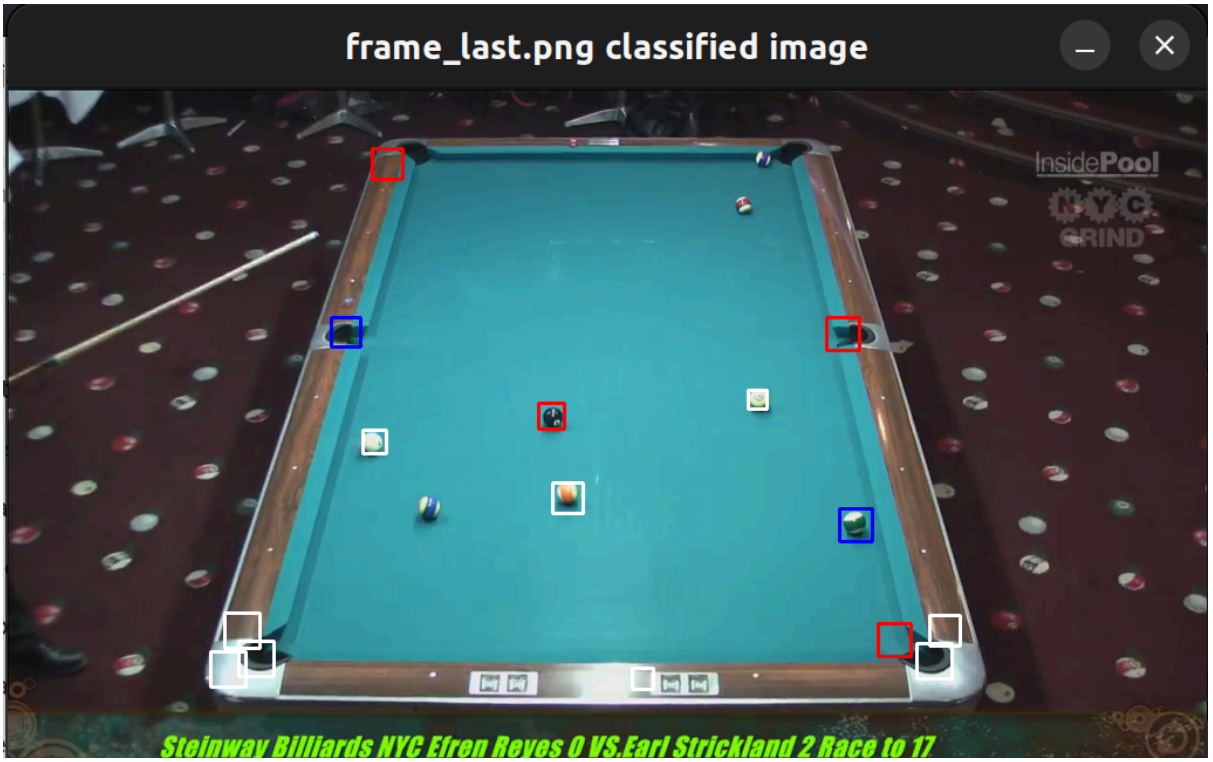
First frame:


InsidePool
NYC
GRIND
Steinway Billiards NYC Efren Reyes 0 VS.Earl Strickland 2 Race to 17

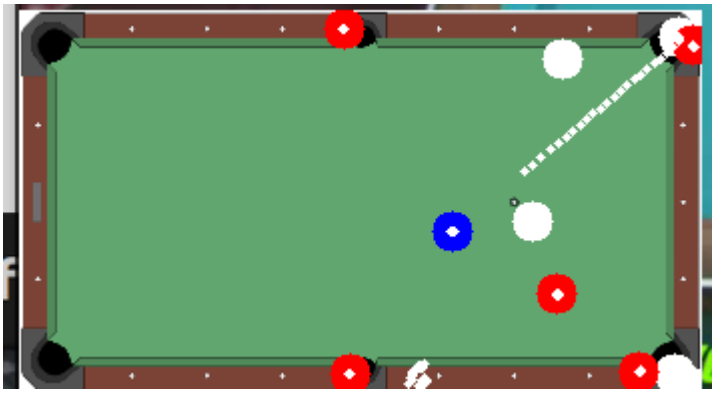

frame_first.png classified image
InsidePool
NYC
GRIND
Steinway Billiards NYC Efren Reyes 0 VS.Earl Strickland 2 Race to 17

Last frame:





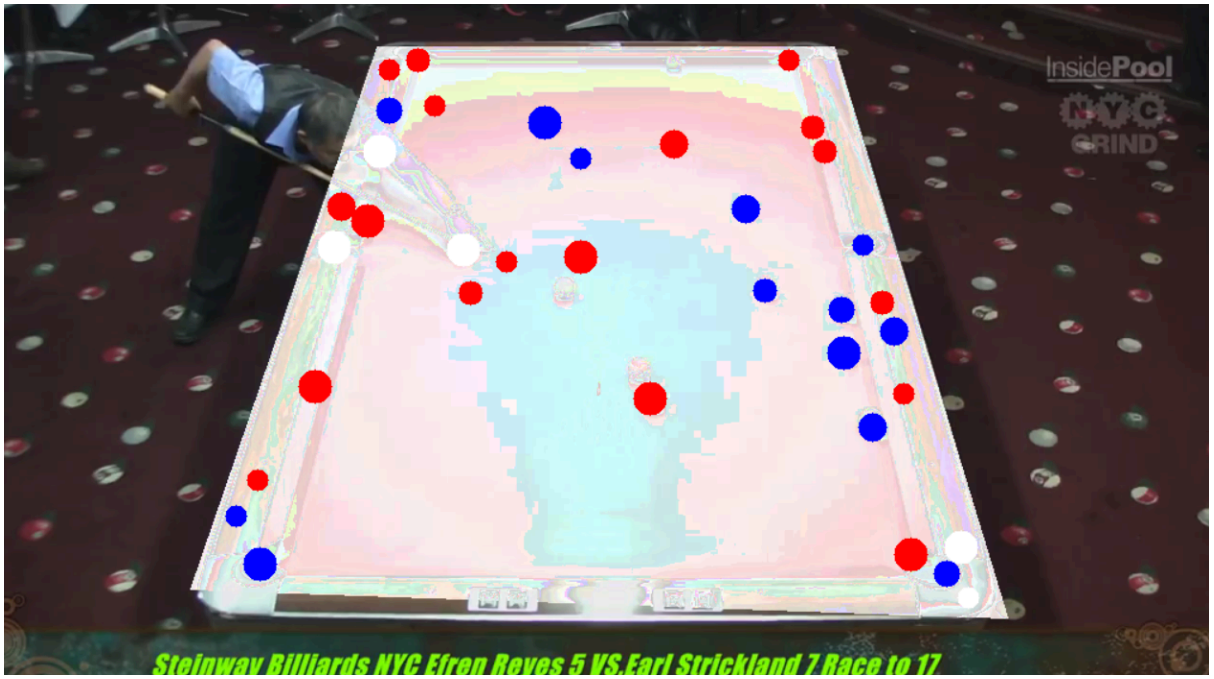
Final top view:

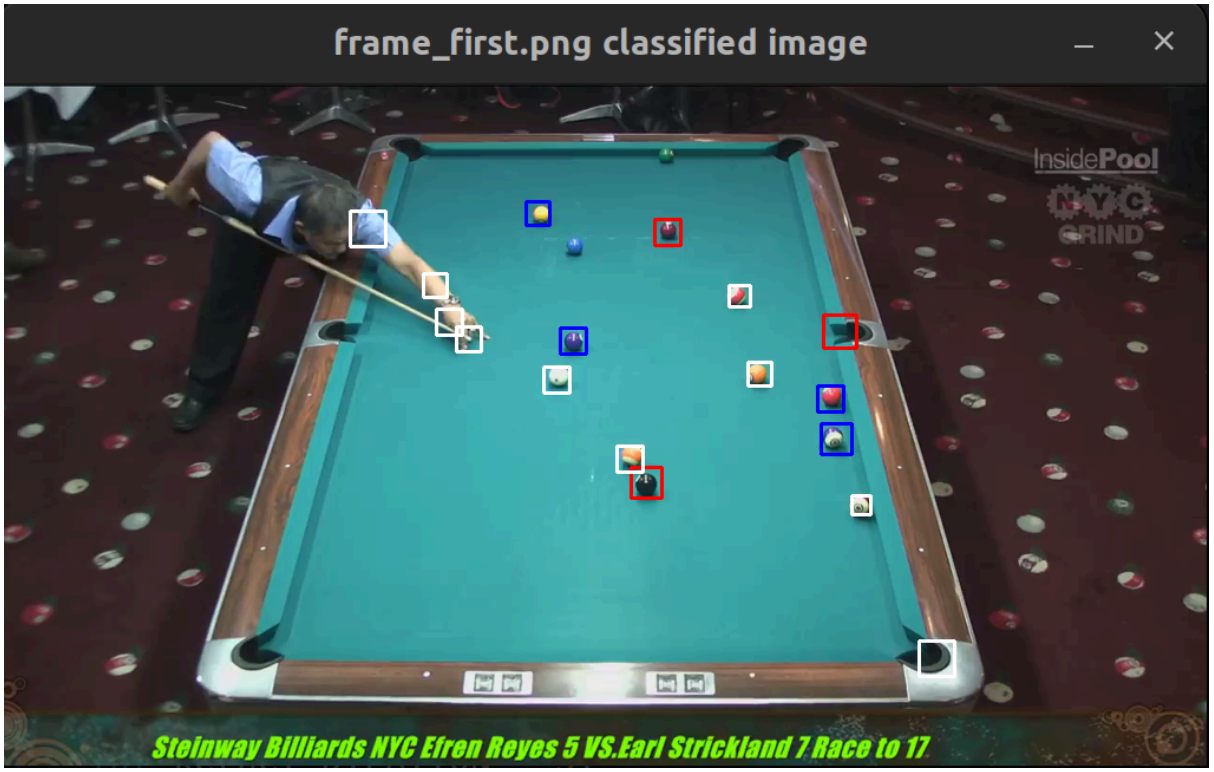


game4_clip2

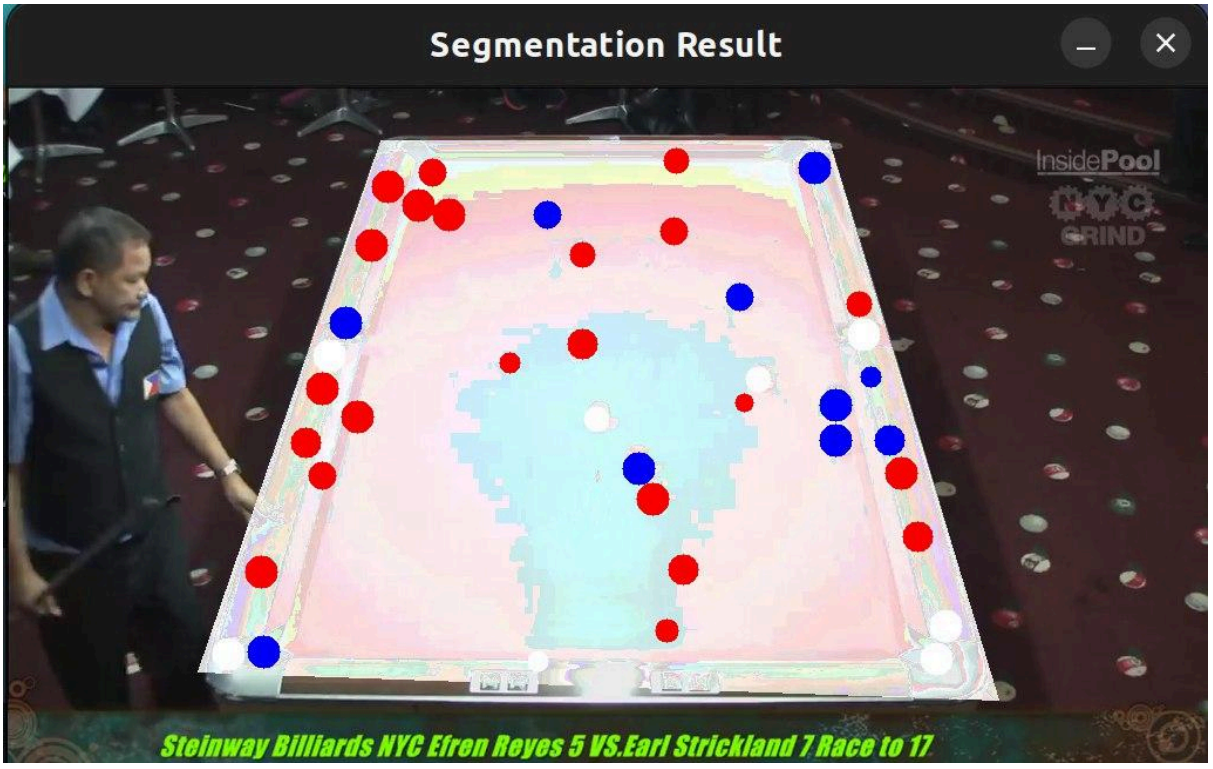
```
Mean IoU for image ../data/game4_clip2:
Class 0 IoU: 0.868167
Class 1 IoU: 0.0573748
Class 2 IoU: 0
Class 3 IoU: 0.0799263
Class 4 IoU: 0.121639
Class 5 IoU: 0.788381
Mean IoU: 0.319248
mAP of frame_first.png 0.103896
Mean IoU for image ../data/game4_clip2:
Class 0 IoU: 0.878974
Class 1 IoU: 0
Class 2 IoU: 0
Class 3 IoU: 0.0816087
Class 4 IoU: 0.187303
Class 5 IoU: 0.801134
Mean IoU: 0.324837
mAP of frame_last.png 0.0681818
```

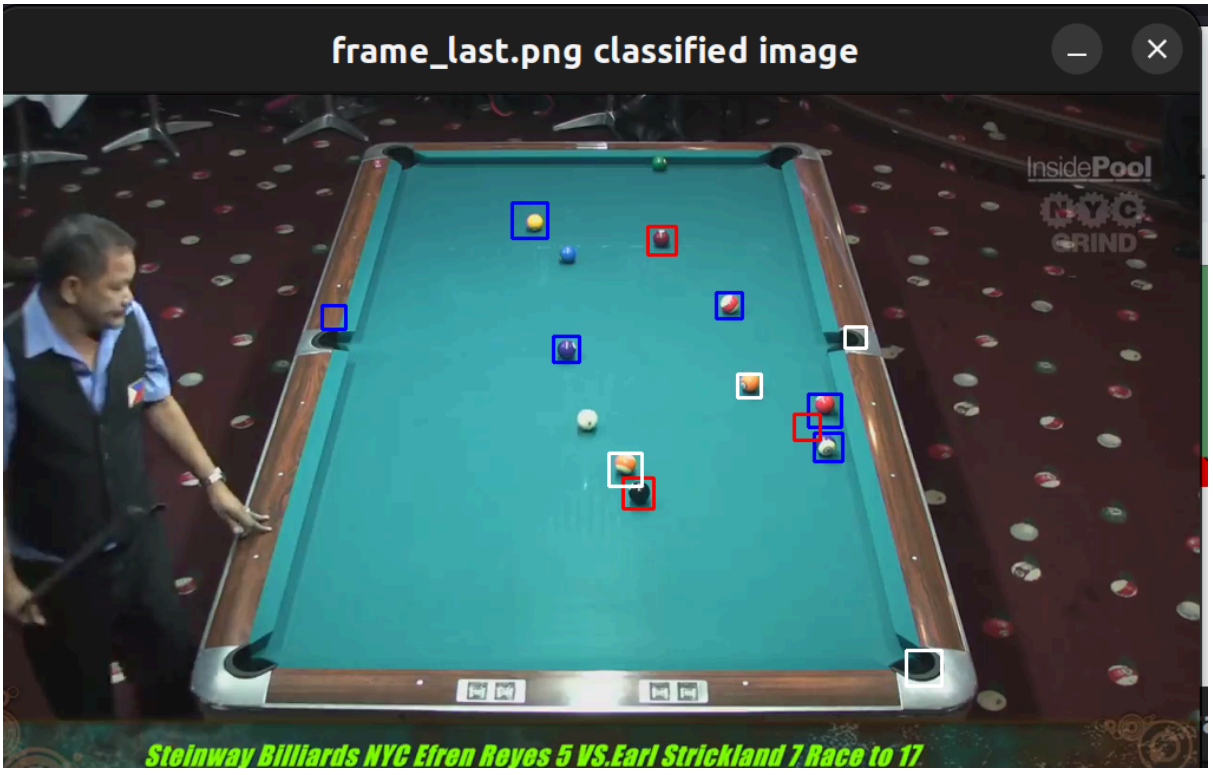
First frame:


InsidePool
NYC
GRIND
Steinway Billiards NYC Efren Reyes 5 VS.Earl Strickland 7 Race to 17

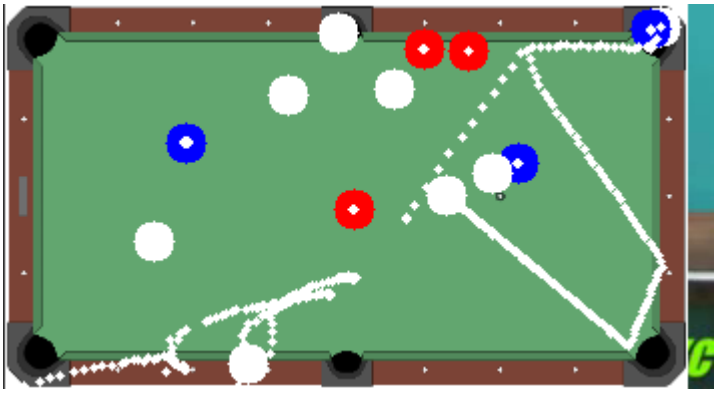

frame_first.png classified image
InsidePool
NYC
GRIND
Steinway Billiards NYC Efren Reyes 5 VS.Earl Strickland 7 Race to 17

Last frame:





Final top view:



Some final considerations

The program works on all the images of the dataset provided but some parts of the pipeline could be improved.
The field segmentation works well, for most images the IoU of the field is around 97/98% except the last game where it's around 90%, it works independently of camera perspective or illumination differences.
The ball detection and classification work well when the table is well and uniformly illuminated, for this reason they struggle where the illumination is not uniform or when a lot of noise is present like in the last game (game4).
The tracking works well even though CSRT can be slower than alternatives (like KCF) it provides a significant improvement in accuracy over other methods.

The first steps for future possible improvements to try to improve the pipeline would be actions that deal with false positives during ball detection and using more powerful techniques to classify the ball.

Hours spent on the project

Matteo De Gobbi: 71h 37m on field detection and segmentation, ball classification, ball tracking, integrating the code to work together, fixing bugs. Helped with the AP computation. (This time also includes the time for writing the report)
Alessandro Di Frenna: 90h on ball detection and segmentation and drawing the minimap
Mohammadali Jafari: 69h => mAP, mIoU, Segmentation Mask, Main, Creating the CMake, Setting up the Git and managing the branches on related tasks, Cleaning the code on the related tasks, and helping in the initial draft of ball classification.