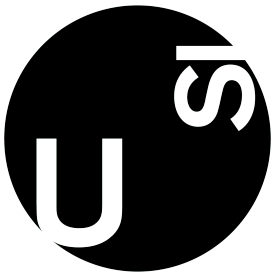
Università della Svizzera italiana



Faculty of Informatics

Computer Vision & Pattern Recogniion
(Spring 2024)

Final Course Project

Reconstruct Snooker Table and Balls from Side View

Submitted by:

Erifeoluwa Jamgbadi | jamgbe@usi.ch
Krunal Rathod | krunal.rathod@usi.ch
Riccardo Giacometti | riccardo.giacometti@usi.ch

Submitted on: June 11, 2024

June 11, 2024

Contents

1 Pre-processing Video (one side view frames only)

The task of preprocessing a video involves filtering frames to retain only those that match a specified reference frame, which shows a side view of a snooker table. The method relies on comparing each frame of the video to the reference frame based on pixel similarity. The function `preprocess_video` is designed to take three inputs: the path to the video file, the reference frame, and an optional output path for saving the filtered video. Initially, the video is read using `cv2.VideoCapture`, and the dimensions of the frames are obtained to define a threshold for pixel matching. If an output path is provided, a video writer is initialized to save the filtered frames to a new video file with the same frame rate and dimensions as the input video.

Figure 1: reference frame

The core of the function is a loop that processes each frame of the video. Each frame is read and converted to grayscale, as is the reference frame 1. The absolute difference between the grayscale versions of the current frame and the reference frame is calculated to identify matching pixels. If the number of matching pixels exceeds the threshold (0.5 in the code), the frame is considered a match. These frames are added to a list, displayed using ‘`cv2.imshow`, and written to the output video if an output path is specified.
After processing all frames, the video capture and writer objects are released, and all OpenCV windows are closed to free up resources. The function returns a list of filtered frames if an output path is not provided. An example usage of the function involves specifying the paths to the input video and reference frame, along with an optional output path. For instance, consider the following call to the function:

```
video_path = "WSC_sample.mp4"
reference_frame = cv2.imread("reference_image.png")
output_path = "filtered_video.mp4"
preprocess_video(video_path, reference_frame, output_path)
```

In this example, the function processes the video `WSC_sample.mp4`, comparing

each frame to the reference image `reference_image.png`. Frames that match the reference frame are saved to a new video file named `filtered_video.mp4`. The video preprocessing function filters frames based on their similarity to a given reference frame.

2 Reconstruct Camera Position

2.1 Get table dimension

To get the whole green area and further get the coordinates, we did the following steps:
- implemented a lower and upper bound for the colour green e.g. lowerBound = (36,0,0) and upperbound = (86, 255, 255).
- using these ranges, we were able to create a mask that would just shade in all the places in the image that had green and turn everything else black as shown in Figure 3.
- we used erosion and dilation to smooth the obtained image.
- then we found the corners of the table of the playing area using cv2.findContours.
- since there were lots of points obtained, we used cv2.approxPolyDP to get approximate the points, so we can just get 4 points.
- we then displayed these points on the green playing area to show the corners obtained in a visual way as shown in Figure 2. We got the mask of cushion(brown part) as shown in Figure 3 using a upper and lower bound threshold. We found the length and width of the playing area is 6.85m(length) 2.24(width). We then calculated the area which we got 15.344000000000001 meters.

Figure 2: table corner points of green area

- using the mask we were able to get the coordinates for the corners of the

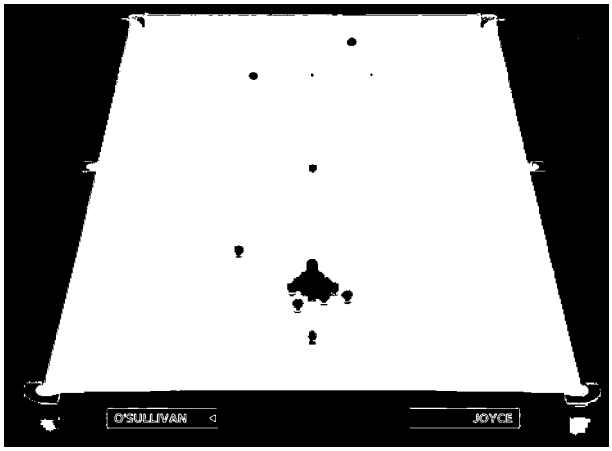


Figure 3: mask of playing area with brown cushion included

table of the brown part using cv2.findContours. One of the reasons we didn't use these coordinates for calculating the DLT in section 2,3 is because we noticed that the green edge coordinates and the ones of the brown one were really similar.

We then decided to use user click for getting the table positions, since the

Figure 4: final coordinates of edges of cushion

above method didn't work out precisely as we wanted it to.
Figure 6 shows the result of our manual implementation of Sobel and opencv

```
print(tableNCushionCorners)
```

```
[[ 375   53]
 [ 905   54]
 [ 309  278]
 [ 967  277]
 [ 243  619]
 [1041  617]
 [1077  642]
 [ 196  637]
 [ 349   34]
 [ 930   36]]
```

Figure 5: table position points

Hough transform to find lines between baize (green) and wood (brown). We adjusted the theshold and rho of the cv2.HoughLines, until we found the desired result for the lines.

2.2 Compute coordinates of the (yellow, green, brown, blue, pink, black) spots

For this part, we used 2 methods, since the first method obtained wasn't as satisfactory. In the first method, we got the lower and upper bound of all the colour balls we wanted to detect. Then we created a mask for each of the upper and lower bound using cv2.inRange and combined these mask using cv2.bitwise_or. Next, we used cv2.HoughCircles to identify the circles in the image based on the mask.

The difficulty we had with this part was that, by using the upper and lower bound range, it identified other objects that were the small colour therefore didn't turn it to black when only leaving the colour balls in the image. This resulted in some false things being identified as circles such as the letters in the plack thing on the nameboard.

The goal to detect the colour balls was achieved, but some other circles were detected too. We added all the coordinates obtained into a list. We then used our other method which was the mouse click event to obtain coordinates and we searched for the index of those coodinates obtained through the mouse click event with the ones in the full coordinates array we got previously. The matches were found and the coordinates obtained are as follows: [(548, 134), (728, 142), (644, 132), (642, 286), (640, 422), (642, 548)]. The colour ball detected results are shown in Figure 7. We tried another method to get the ball positions. We used mouse click events to get the points of the table corners and ball position.

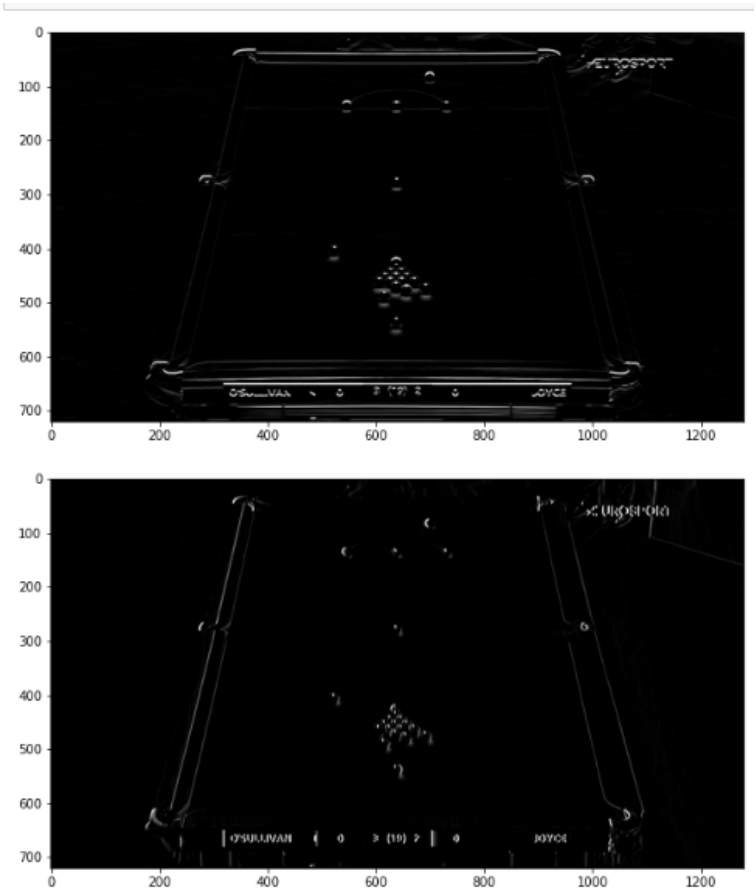


Figure 6: table position point using sobal and houghTranform

2.3 Use correspondence pairs to find camera matrix

We were not satisfied with the points we got for the table dimension and ball position, so we used the points given by the professor. We used those points to find the matrix A by implementing the DLT algorithm. After implementing the DLT algorithm to get matrix A, we applied svd to matrix A to get the camera position P. Figure 8 shows the result camera position P matrix, that we obtained.

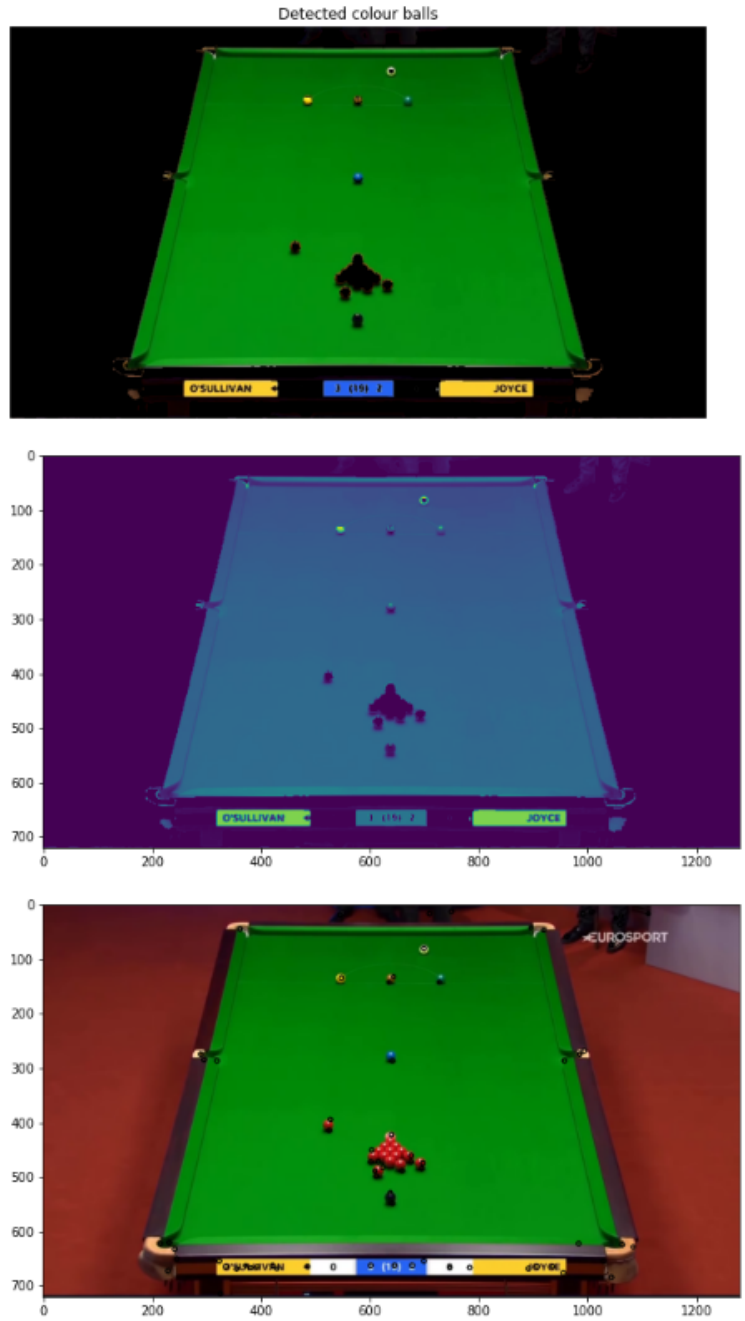


Figure 7: colour ball detected

```
[[ 3.53745529e+02  6.41016197e+01  6.40070509e+02]
 [ 1.03421751e-01 -1.24024868e+02  2.83516212e+02]
 [ 3.25120766e-04  1.00169796e-01  1.00000000e+00]]
```

Figure 8: camera position p

We then went further and used the matrix P from Figure 8 to calculate the camera calibration matrix K, the camera orientation R and the camera center C. The results gotten are shown in Figure 9.

```
Camera Calibration Matrix K:
[[-1.00000028e+00  2.92361247e-04 -1.15503517e-06]
 [-2.92362084e-04 -9.99999957e-01  8.07061983e-04]
 [-9.19081178e-07  8.07062286e-04  1.00000000e+00]]

Camera Orientation R:
[[ 0.44402436  0.08041537  0.80352585]
 [ 0.         -0.15570062  0.35563588]
 [ 0.          0.         -0.00154149]]

Camera Center C:
[-0.35678494 -0.0092431  -0.7721328 ]
```

Figure 9: camera calibration matrix krc found

3 Reconstruct Ball Positions

3.1 Detecting the red balls

In order to detect the red balls we have to convert the image from its BGR default representation to HSV. In addition, we also need to define two color ranges to filter out because red belongs to two different regions of the hue in HSV (0 to 10 and 160 to 180). The two filter masks are then merged by applying a bit-wise OR operation. This final mask is then applied to the image by performing a bit-wise AND operation. Also, the lower bounds for the saturation and value values had to be manually tuned to also remove the red carpet area. The final bounds for the masks are given in Table 1.
Figure 10 shows the result of applying the mask. It can be seen that the balls are close together so directly applying a Hough circle transform would not work so we need to do some further processing.

Table 1: HSV Bounds

Region and Bound	Hue	Saturation	Value
Mask 1 Lower Bound	0	165	165
Mask 1 Upper Bound	10	255	255
Mask 2 Lower Bound	160	165	165
Mask 2 Upper Bound	180	255	255

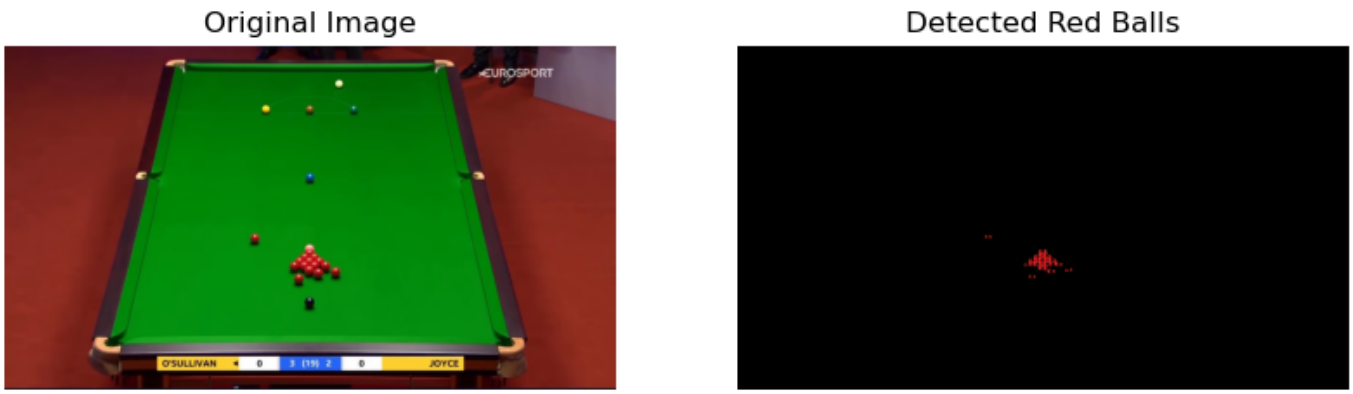


Figure 10: Filtered red color from original image

In order to slightly detach the red balls from each other, we convert the output from Figure 10 to a gray scale image (Figure 11) and then we binarise it with a threshold (Figure 12). After this, we apply the Hough circle transform to get the rough position of the red balls shown in Figure 13. From the Hough circles we can get the centres of them and therefore the red ball coordinates.

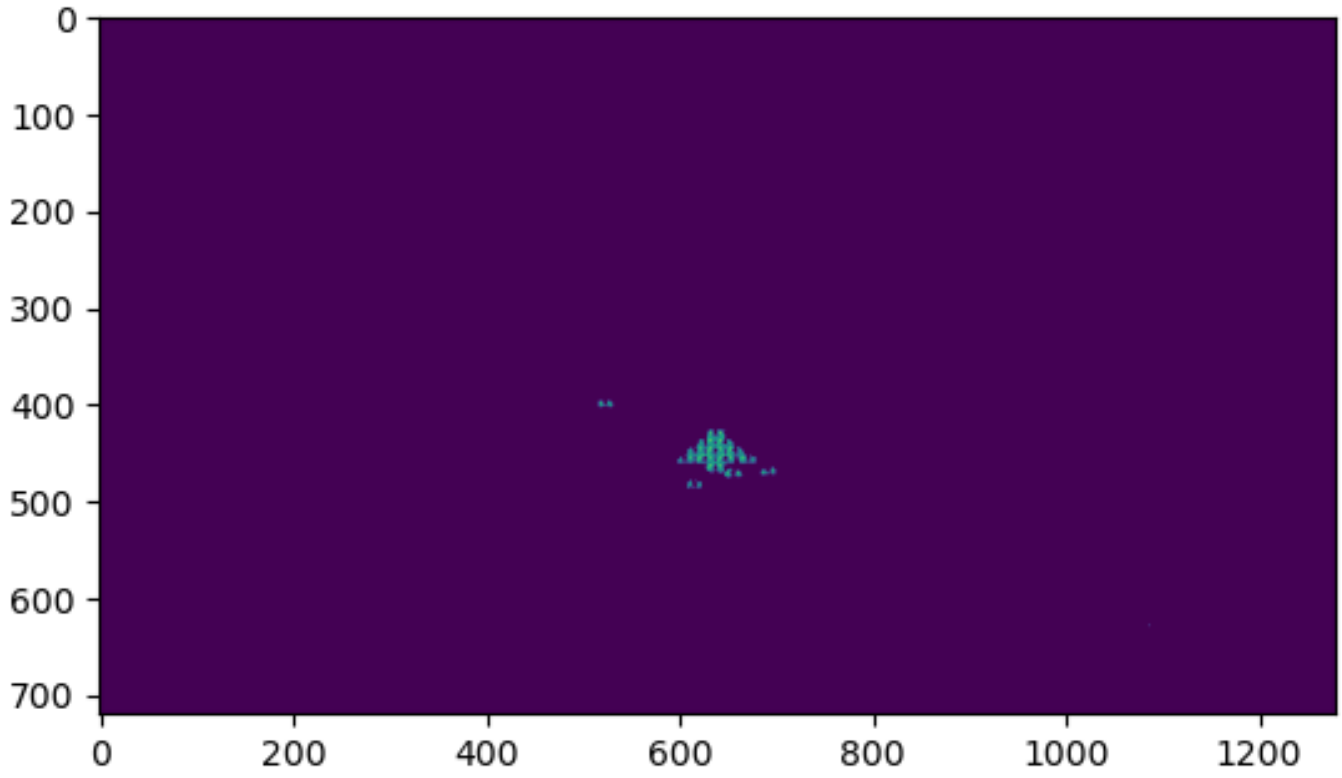


Figure 11: Grayscale from red ball detection

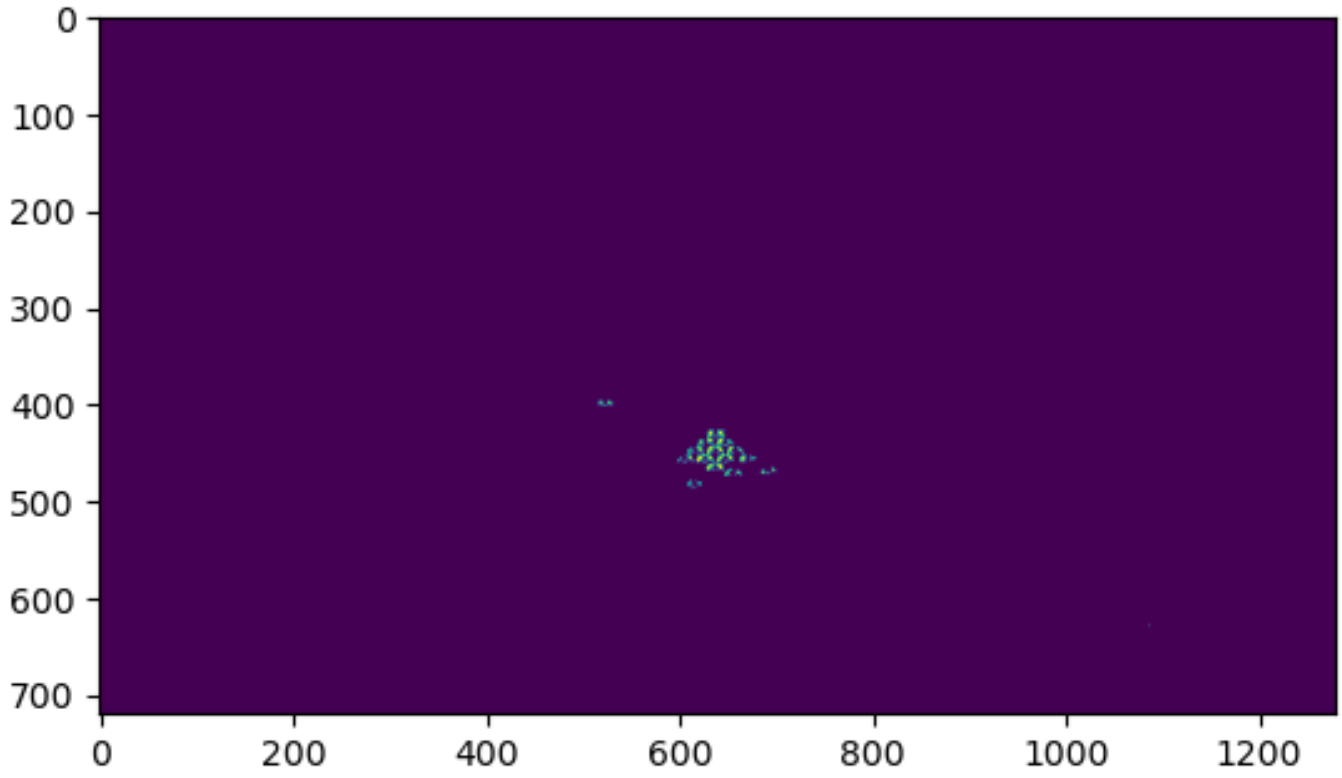


Figure 12: Binary image from gray scale image

Figure 13: Detected Red Balls

3.2 Projecting from Side View to Top View

To get the top view we use the corner coordinates from the side view and map them to arbitrary coordinates on the top view that follow the same aspect ratio between the width and length of the table. In order to map the corners we find the Homograph transformation by using Gaussian elimination to find the Homography matrix. From this we can apply the Homography matrix to the coordinates of the balls to get the top down view of them. The final result is shown in Figure 14.

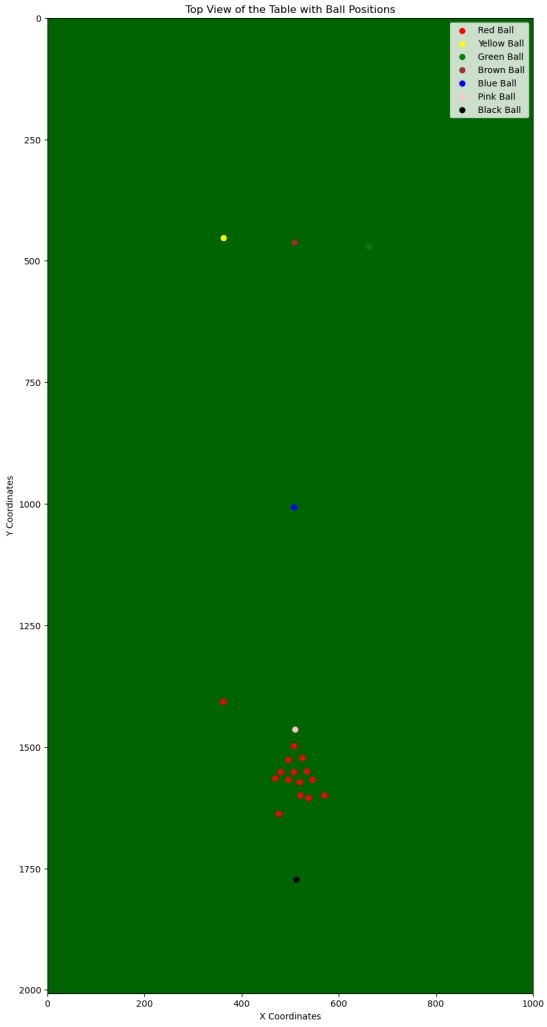


Figure 14: Top Down View of the Pool Table