IACV Project Report

Single View 8 Ball Pool Game Analysis with Deterministic Detectors and Continuity Based Error Correction

By: Mustafa Cagatay Sipahioglu

Person Code: 10899801

Enrollment Number: 220591

Table of Contents

Problem Statement

The objective of this project is to analyze a single-view video of a game of 8 Ball Pool.

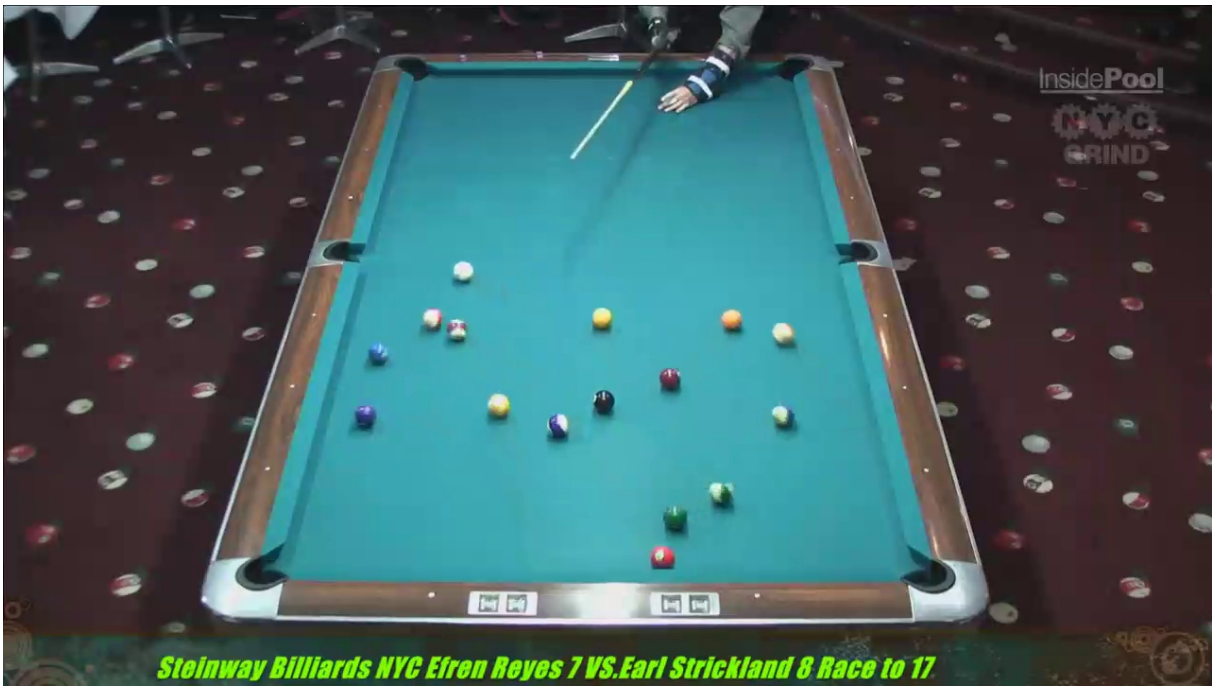


Figure 1: An Example Single-View 8 Ball Pool Game Frame

The game of 8 Ball Pool is simple and taking Figure 1 as reference it is easy to understand. There are 16 balls on a pool table. one cue ball (White ball), 7 solid colored balls (Each a different color, numbered through 1-7), one 8 Ball (Black solid colored ball), and 7 striped colored balls. (Each a different color, numbered through 9-15)

Two players are assigned to either the solid or the striped balls. The objective of each player is to first pot their 7 balls and only after that be the first to pot the 8 Ball. To pot a ball, the player must hit the cue ball with a cue (The stick) this cue ball is supposed to hit one of the player's objective balls and make it go into one of the holes in the table. (At 4 corners and 2 mid points of the longest edge of the table) If a player can pot a ball without any fouls they can play again.

Fouls include potting the cue ball, hitting an opponent's ball or the 8 ball with the cue ball before a player's own objective balls or hitting a ball off the table.

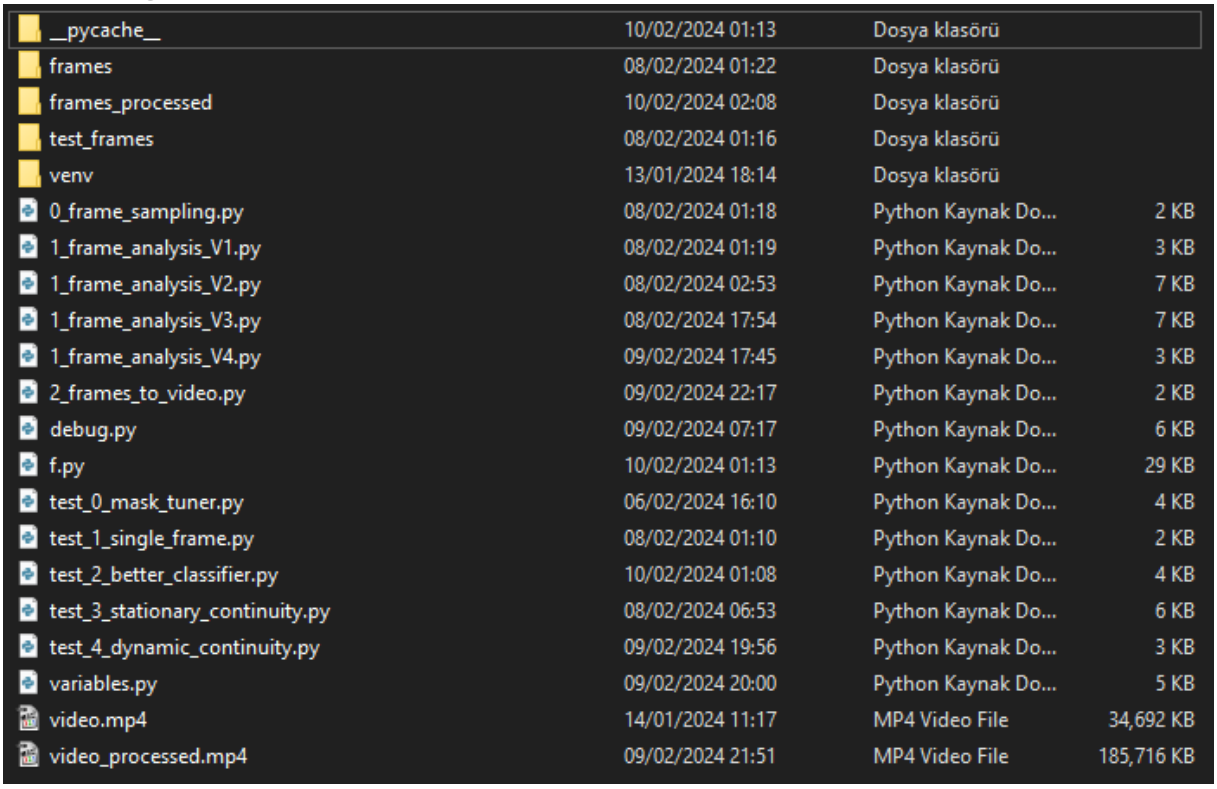
The game starts as the following: The first shot of the game is called the break, all 15 balls are put in a triangle formation on the table and one player strikes the cue ball to break this formation starting the game. After the break shot the first type of potted ball without any fouls is assigned as the objective balls of that player. (The other class of balls automatically become the objective of the other player.) However, these cases will be out of the scope of this project. (We won't be finding out which player has which class of objective balls, instead the two players will be regarded as the striped player and the solid player)

State of the Art

Above all, this project is an object detection and classification project. In theory if a computer could detect and classify each ball at each frame of the video perfectly there would be no reason to implement ball tracking and detection corrections.

Currently, really good detectors are available that use machine learning algorithms and will give nearly perfect results. However, this state of the art detectors usually tend to be very resource intensive to either run or develop in the first place. Therefore, this project implements a relatively simple detection software and instead finds the associations between each consequent frame of a video to track balls and correct detection mistakes.

Code Organization



__pycache__	10/02/2024 01:13	Dosya klasörü	
frames	08/02/2024 01:22	Dosya klasörü	
frames_processed	10/02/2024 02:08	Dosya klasörü	
test_frames	08/02/2024 01:16	Dosya klasörü	
venv	13/01/2024 18:14	Dosya klasörü	
0_frame_sampling.py	08/02/2024 01:18	Python Kaynak Do...	2 KB
1_frame_analysis_V1.py	08/02/2024 01:19	Python Kaynak Do...	3 KB
1_frame_analysis_V2.py	08/02/2024 02:53	Python Kaynak Do...	7 KB
1_frame_analysis_V3.py	08/02/2024 17:54	Python Kaynak Do...	7 KB
1_frame_analysis_V4.py	09/02/2024 17:45	Python Kaynak Do...	3 KB
2_frames_to_video.py	09/02/2024 22:17	Python Kaynak Do...	2 KB
debug.py	09/02/2024 07:17	Python Kaynak Do...	6 KB
f.py	10/02/2024 01:13	Python Kaynak Do...	29 KB
test_0_mask_tuner.py	06/02/2024 16:10	Python Kaynak Do...	4 KB
test_1_single_frame.py	08/02/2024 01:10	Python Kaynak Do...	2 KB
test_2_better_classifier.py	10/02/2024 01:08	Python Kaynak Do...	4 KB
test_3_stationary_continuity.py	08/02/2024 06:53	Python Kaynak Do...	6 KB
test_4_dynamic_continuity.py	09/02/2024 19:56	Python Kaynak Do...	3 KB
variables.py	09/02/2024 20:00	Python Kaynak Do...	5 KB
video.mp4	14/01/2024 11:17	MP4 Video File	34,692 KB
video_processed.mp4	09/02/2024 21:51	MP4 Video File	185,716 KB

Figure 2: Code Files

Looking at Figure 2 the codebase can be seen. These files and folders serve the following purposes:

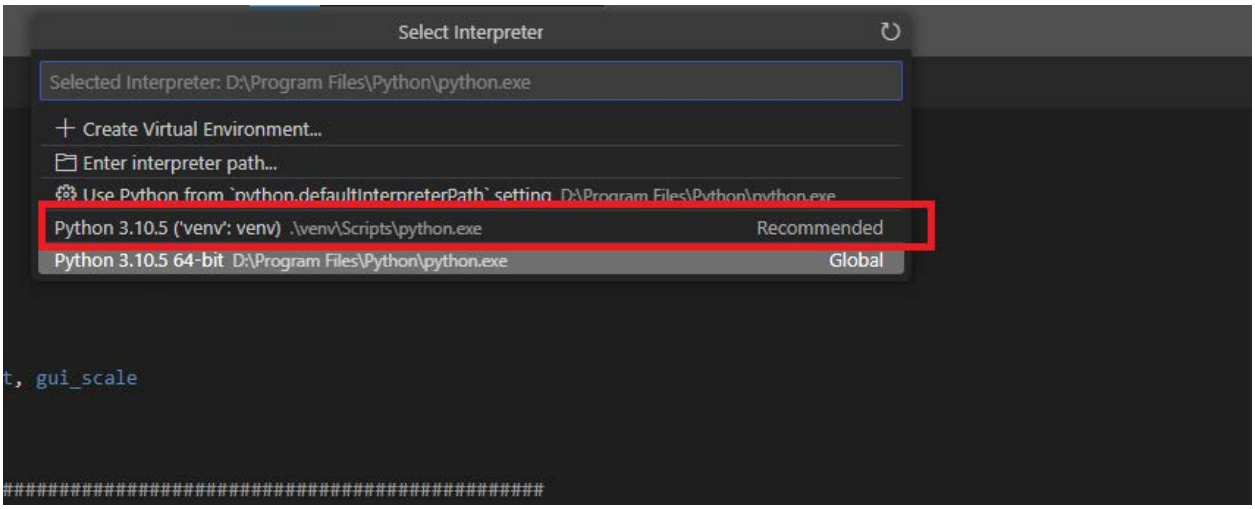
- **venv, _pycache_:** These folders hold the virtual environment this code can be run on. A virtual environment basically holds the required libraries to run the code so the reader doesn't have to install those libraries. The codebase uses OpenCv (cv2) and numpy libraries. If the reader doesn't want to install these libraries, how to activate the virtual environment will be discussed shortly.
- **video.mp4:** This is the unprocessed game video we start from.
- **debug.py, f.py, variables.py:** These are part of the main codebase they hold debugging functions for printing processed images, actual processing functions and the tunable variables respectively.
- **0_frame_sampling.py, frames: 0_frame_sampling.py** takes in the video.mp4 and fills the folder **frames** with the frames of this video sampled at the sampling_period. (sampling_period is defined in the **variables**) This is done as to not have to sample each frame of the video every time we want to do processing.
- **test_i…:** Files and folders with the test prefix are testing programs for the modules of the main code. These will be explored more in depth to understand the main code in the next sections.
- **1_frame_analysis_Vi, frames_processed:** This is the main processing code. (With version suffix) As of now you should be running **1_frame_analysis_V4.py**. Each version i simply corresponds to the methods in **test_i**... being applied to the whole video.
- **2_frames_to_video.py, video_processed:** After each frame is processed by the above code, **2_frames_to_video.py** stitches all of these processed frames back into a video: **video_processed**.

Activating a Virtual Environment (venv)

1. Open your code IDE (VS Code in my case) in the code folder.
2. Open a new terminal in your IDE, navigate to the code folder.
3. Run: .\venv\Scripts\activate
4. VS Code shows in green that a venv is activated.

```
PS D:\_Education\Database_Master's\Uni-11\Computer Vision\3-Project\Final Version> .\venv\Scripts\activate
(venv) PS D:\_Education\Database_Master's\Uni-11\Computer Vision\3-Project\Final Version>
```

5. Make sure you change the interpreter to the venv interpreter instead of the computer's globally installed interpreter.



Defining a Ball

The game state completely depends on the balls. Therefore, we need a software implementable description of a "ball". A ball is defined by the following fields:

Class

Cue Ball ('C'), 8 Ball ('8'), Solid ('O') or Striped ('||')

State

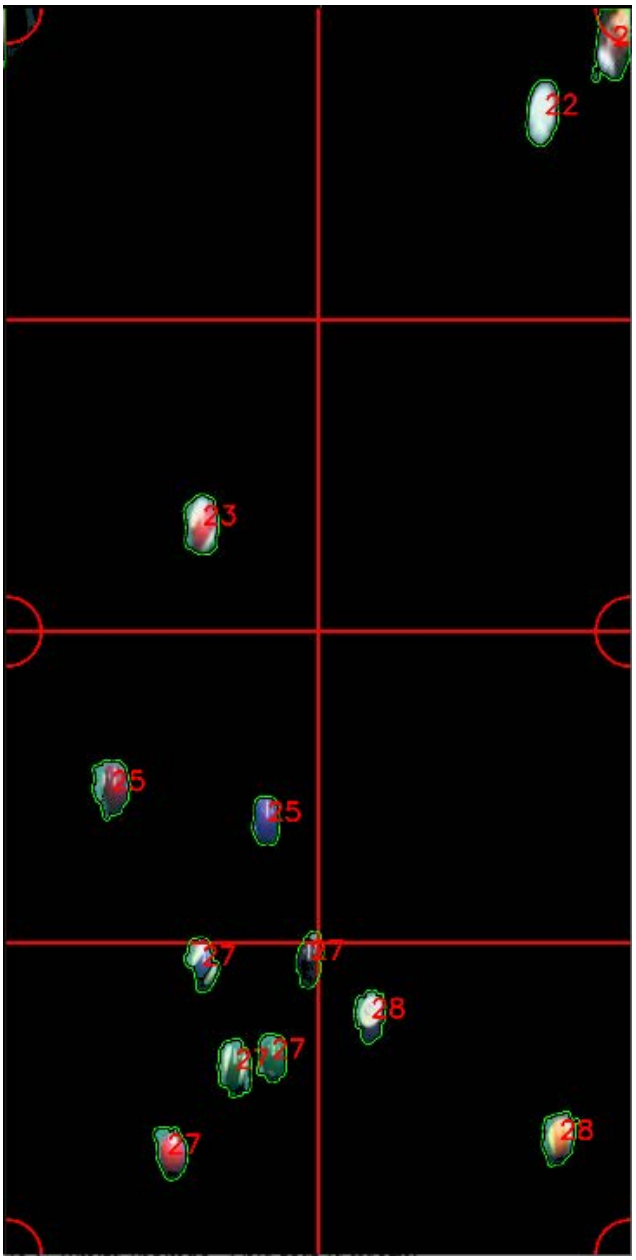


Figure 3: Table With State Sectors

State is a classification of the position of the ball. As you can see in Figure 3, the table is separated into 8 octants with states 21-28 (Starting from top left going to bottom right in reading order) and 6 special "solo sectors" that simply defines a ball very close to a hole. (Defined as circular regions around each hole, with states 11-16)

Center

Center of a ball is the center of the smallest enclosing circle drawn around the contour.

Contour

Contour refers to the OpenCV contour defining the ball. This is an array of points that define the border of the ball image.

Ball Image

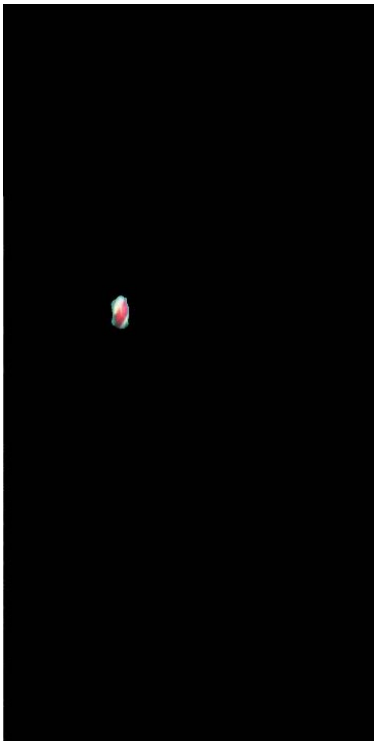


Figure 4: Example Ball Image

Ball image is the image of only that ball on the table as seen by an example in Figure 4. (The completely black table is part of the ball image, hence all pixels are black except the inside of the ball contour)

Luminance

Luminance is the average luminance value of the pixels of the ball image that are not dark. (All black pixels and pixels below a luminance threshold (Probably a shadow) are eliminated) This average luminance value is found by averaging the L values of the non-dark pixels of the ball image defined in the LAB color space.

Practically cue ball usually has the highest luminance while the 8 Ball has the lowest luminance.

Color Ratio

Color ratio is the ratio of colored pixels to white pixels of a ball image. In the HLS color space: White pixels are defined to have lightness over 220/255 (So not only pure white but all white-ish pixels) and a colored pixel is defined in the rest of the lightness space with saturation over 40/255.

Introduction

We have to process frames of the video (For example Figure 1) to detect balls on this video.

1. The first step we will always be doing is to first extract only the table image from the video.
2. Then we will rectify this image to look like the top view of the table (A rectangle).
3. Thirdly we will delete all pixels corresponding to the tablecloth of the table from the rectified table image. At the end of this we will be left with a rectangular (Top-view like) image of the table with only the balls in it.
4. Fourthly we will detect the contours in the image with just the balls. In theory all the ball contours should be detected.
5. And fifthly we will find the class, state and other associated variables of each ball (as defined in “Defining a Ball”) to create an array of balls.

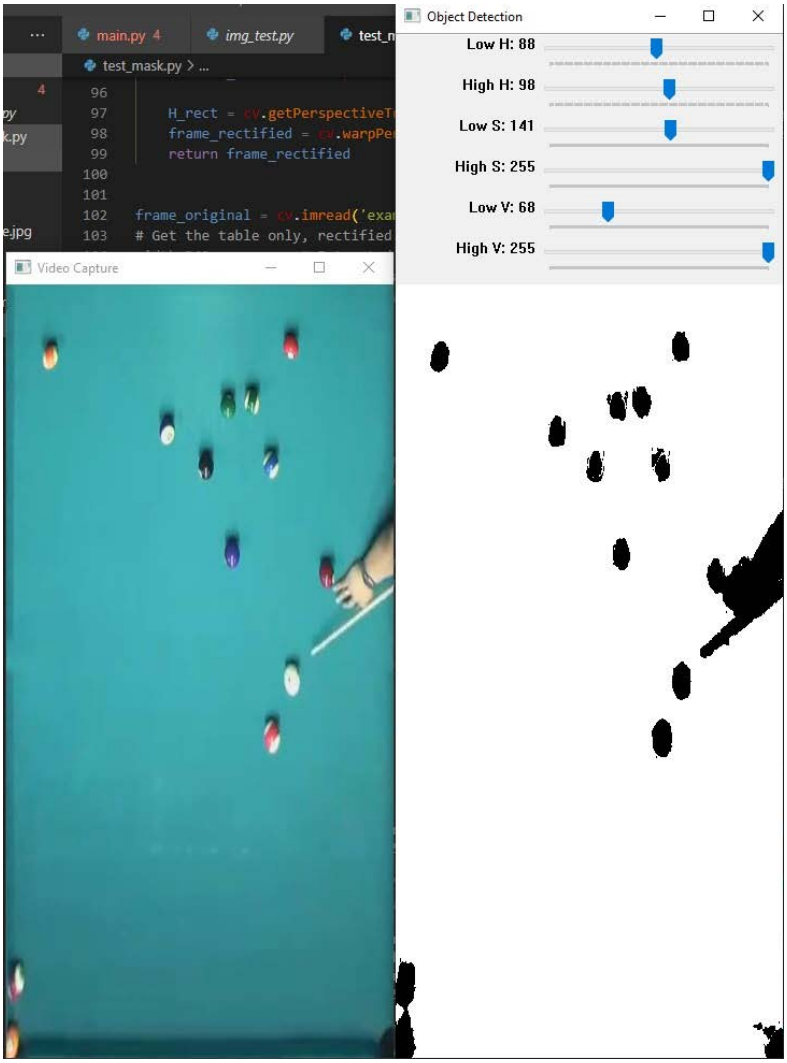


Figure 5: Extracted and rectified table image (Left) and Applied mask with HSV Sliders (Right)

Before analyzing **test_2_better_classifier,** it is beneficial to look at **test_0_mask_tuner**. **test_0_mask_tuner** is a simple piece of code that given an image it provides you with sliders for HSV color channels to see what a mask within these HSV limits include. This test is useful to tune the color interval for removing all the tablecloth-colored pixels from the image. An example masking procedure is seen in Figure 5.

Step 1: Detection and Classification (Version 1, 2)

To do the main analysis steps defined in "Introduction" we can simply look at **test_2_better_classifier.py. (test_1_single_frame.py** is a depreciated piece of code, it is completely same with test_2 except its ball classifier has a worse performance. Only does luminance classification, the reader will understand what this means in the following paragraphs.)

```
# Get the table only, rectified.
img_table=f.get_table(frame,img_width,img_height)
if dbg==1: debug.gui_image('1.Rectified Image',img_table,gui_scale)

# Remove the tablecloth, only the balls and other objects should be left.
img_reference=f.remove_tablecloth(img_table)
if dbg==1: debug.gui_image('2.Masked Image',img_reference,gui_scale)

# Detect the contours.
contours=f.detect_contours(img_reference)
if dbg==1: debug.gui_image_with_contours('3.Detected Contours',img_reference,gui_scale,contours)


# Detect balls with class and state.
balls=f.detect_balls_better(img_reference,contours)
if dbg==1: debug.gui_image_with_ball_classes('4.Detected Balls',img_reference,gui_scale,balls)
if dbg==1: debug.gui_image_with_ball_states('5.Image with State Sectors',img_reference,gui_scale,balls)
```

In **f.py** all the needed functions can be found. These functions can be summarized as the following:

- **get_table:** Given an original frame (Like Figure 1) and the corner coordinates of the table (defined in **variables.py**) this function returns the rectified image of just the table in the frame. (img_table)
- **remove_tablecloth:** Given img_table this function deletes all the pixels that are in the color ranges for the tablecloth of the pool table. This function also deletes all very dark pixels from the image. (Most of the images of the holes and very dark shadows) img_reference is returned.
- **detect_contours:** Given img_reference, detect contour uses OpenCV library's built in findContours method to find the contours in the image. Returns contours.
- **detect_balls_better:** Given all the contours detected on a table. This function filters the ball-like contours and then returns an array of balls with all the properties such as class, state, center etc.
    - Firstly, each contour's smallest enclosing circle and rectangle as well as that contour's area are found. The center and radius associated to this contour is found from the smallest enclosing circle and the aspect ratio of this contour is found from the smallest enclosing rectangle. Using the tuned parameters ball_radius_limits, ball_aspect_ratio_limits, ball_area_limits from **variables.py** the ball-like contours are filtered.
    - For the ball-like contours, their state (**find_state:** simply cross checks the center property of the ball with the sectors defined on the table), luminance (**calculate_luminance**: as defined in "Defining a Ball") and their luminance based class (**classify_ball_luminance:** Splits the luminance values into 5 regions using thresholds in the variables file. From lowest luminance to highest the classes are "Hole", "8 Ball", "Solid", "Striped", "Cue" ball. It is easy to understand that balls with more white in their images will have higher luminance

values. (This was the only classification technique used in **test_1_single_frame.py**, as seen below we have improved our classification technique))

- **‘**Hole’ class contours are eliminated. ball_image is calculated (**image_inside_contour:** Calculates the image left inside the ball contour.)
- Due to lighting, sometimes the luminance values of some striped and solid balls may be unexpectedly high/low. Therefore, luminance based classification was found to be good for classifying the 8 Ball, the Cue ball and Holes as well as classifying any ball if the lighting is very dim. (Sectors 27, 15 and 28, 16). So after the luminance based classification, striped and solid balls are reclassified using their color ratios. (**calculate_color_ratio**: As defined in “Defining a Ball” + **classify_ball_color_ratio:** Uses tuned limits to classify a ball as ‘8’, ‘C’, ‘O’ or ‘||’)
- Each detected, filtered, classified ball is now in a list called “balls”. Before returning this array of balls we finally do a correction of impossibilities such as multiple detected cue balls or 8 balls. (**resolve_ball_impossibilities**)
    - If there are multiple ‘C’ balls detected, only assign the highest luminance one as the ‘C’ ball, correct the others to be a striped ball. ‘||’
    - If there are multiple ‘8’ balls detected, only assign the lowest luminance one as the ‘8’ ball, correct the others to be a solid ball. ‘O’
    - If 16 balls were found there must exactly be 1 ‘C’, 1 ‘8’, 7 ‘O’, 7’||’ balls. The ‘C’ and ‘8’ balls are made sure to be singular in the previous step. Now we can do striped/solid correction as:
        - If there is an extra solid ball: Out of the 3 lowest color_ratio solid balls, the highest luminance one should actually be striped
        - If there is an extra striped ball: Out of the 3 highest color_ratio striped balls, the lowest luminance one should actually be solid.
    - This method can always correct multiple ‘C’ or ‘8’ balls. In the case of 16 balls it can only correct up to a single extra misclassified solid or striped ball.
- The array of balls is returned.

The results of analyzing a single frame using **test_2_better_classifier.py** can be seen in Figure 6 in the next page.

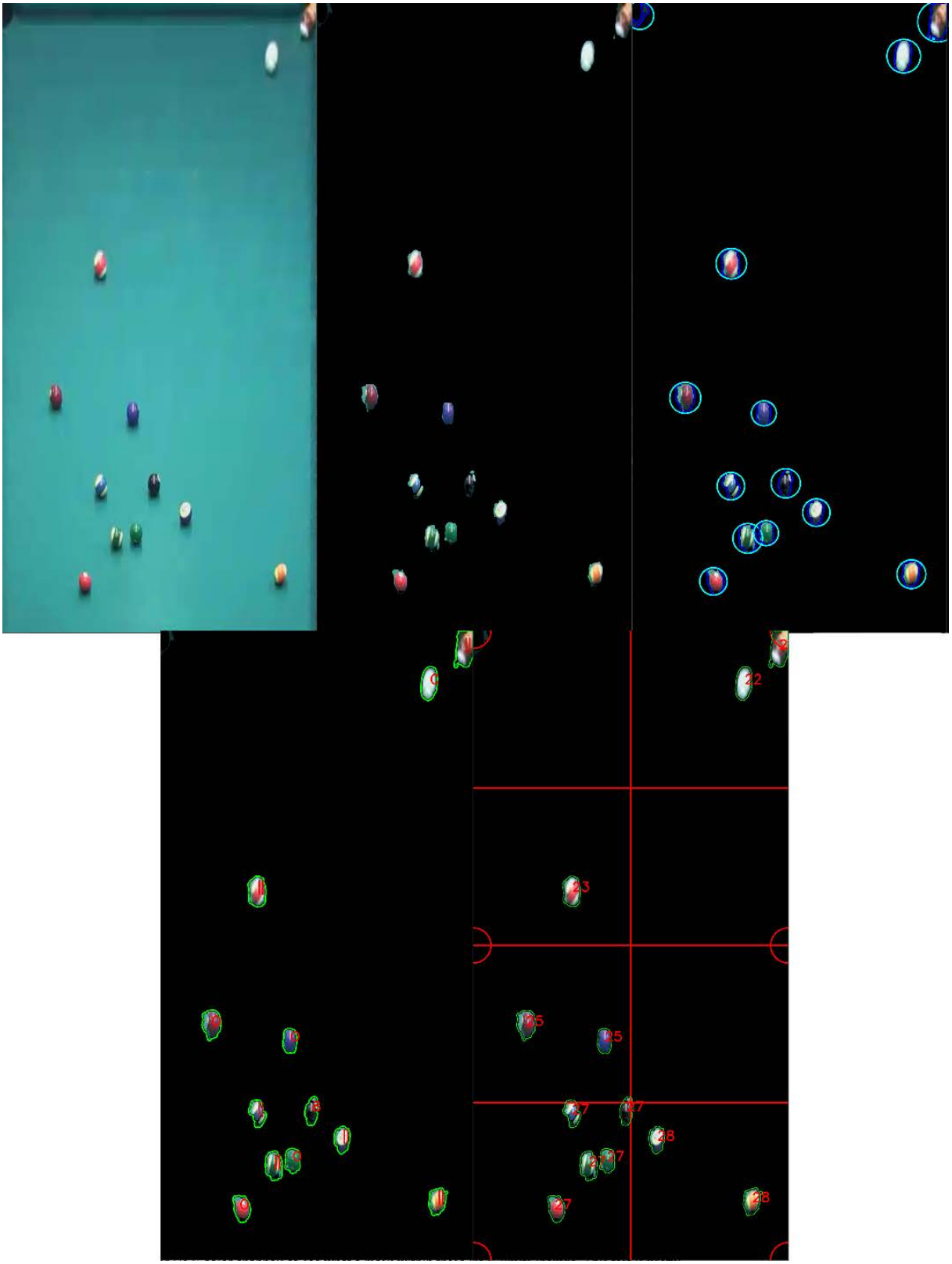


Figure 6: Rectified Table, Table Cloth Removed, Detected Contours and Smallest Enclosing Circles, Balls with Classes, Balls with Sectors

Step 2: Stationary Continuity (Version 3)

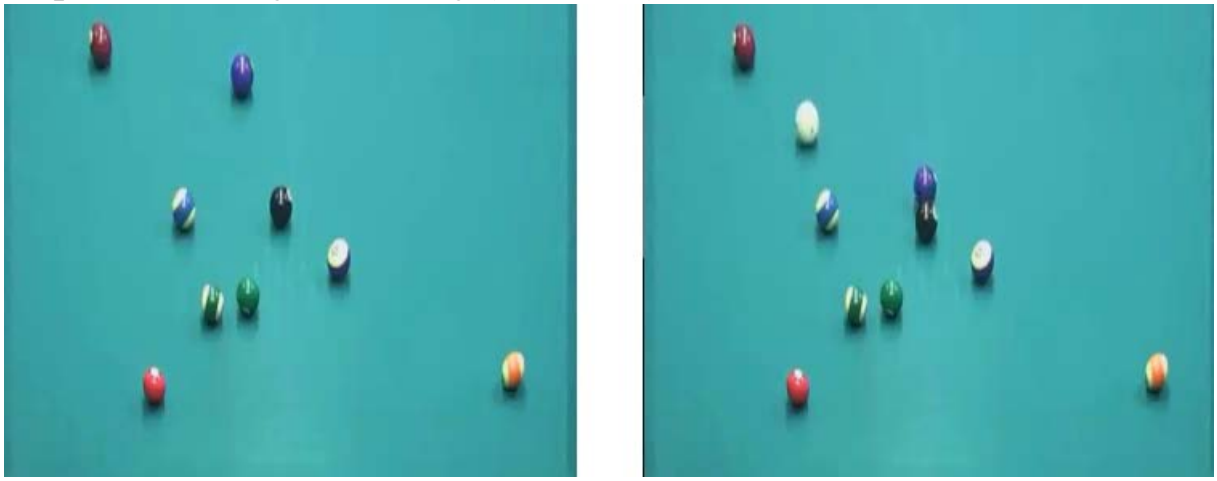


Figure 7: Stationary Continuity Test 2 Frames: Before and After

Now that we have a semi-reliable way of classifying and detecting balls we can build on top of this knowledge. Particularly this section focuses on how we can detect if a ball is left behind another ball. Looking at Figure 7, we can see that the solid purple ball approaches the 8 Ball and stops behind it while the 8 Ball just stays still.

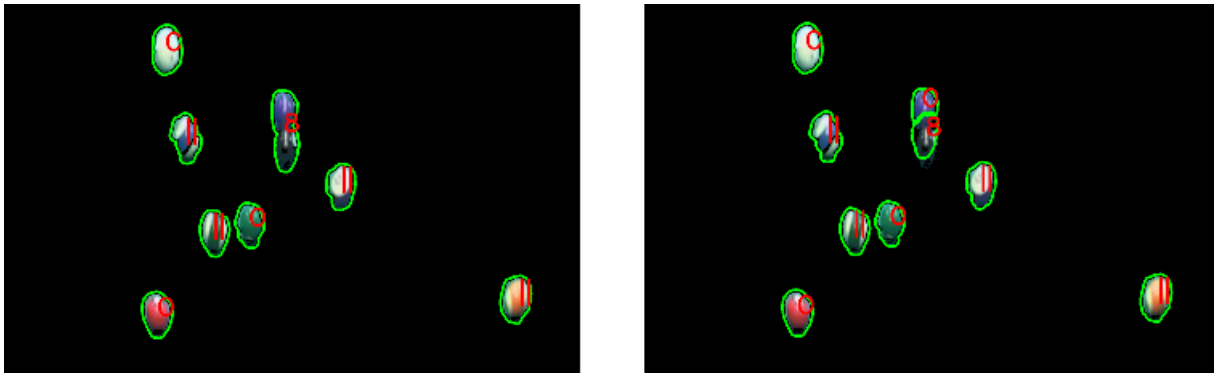


Figure 8: Ball Detection without and with Stationary Continuity Correction

Two really close balls will most probably be detected as a single contour and there are many cases where this contour will pass through the size filters and will be classified as a single ball. We would like to avoid 2 balls being detected as a single ball for as much as possible. Figure 8 shows how the solid purple ball and the 8 Ball would be detected and classified without and with stationary continuity being checked. We can look at **test_3_stationary_continuity.py** to understand how this is implemented.

- Taking 2 consequent frames before_frame and after_frame. We can first detect and classify all our balls normally as the previous section has explained.
- Then we can import the after_frame. First we can loop through the balls we had detected in the before_frame and calculate if each ball has moved using **did_ball_move**:
    - We know the image and the contour of the before_ball.
    - In the after-image we can find the image lying in the contour of the before_ball.
    - If the image lying in this contour is nearly completely black, then we know that the ball has moved away.

    - Otherwise, if the image lying in the contour of the before ball is the same in both the frames we can say that this ball has not moved. (We can calculate this by checking if the number of different pixels between each image is below a threshold.)
    - And most importantly if the difference between the two images are noticeable this means the ball has not moved but instead obstructed by a person moving in front of the camera. This works because if a ball has moved away the image inside the contour would be black and it would have returned the "moved" state before the program could execute up until this line. So now it means that there is another non-zero image in the contours that is not of the ball's, it must be due to a person with non-black pixels having come into the contour.
    - And a last important thing to note here is that a slightly moved ball will be returned as "moved" from this function. This is because balls roll and even a slight roll will make the difference between the two images very high.
    - Movement_state "moved=1", "stationary=0" or "obstructed=2" is returned.
- If a ball is obstructed it is simply added to the after_balls array holding the balls in the after frame to not lose track of it.
- If a ball is stationary an after_ball is calculated with the same position and contour as the before_ball. This contour is reclassified to be robust against slight lighting changes or errors. This after_ball is added to after_balls. Now remember at this point we have not yet done any detections on the after-image we have simply deducted the balls that didn't move and added them to the balls array of the after image. So what we can now do is we can delete all of the detected "stationary" balls from the after image. (Not the obstructed balls since the image inside their contours are not the images of balls)
- After this deletion now the after_image only contains the images of the moved balls. Therefore, now we run our detection and classification algorithm on the after image with the stationary balls subtracted. The detected balls will be the balls that have moved between the frames.
- We can add in the newly detected balls into the balls array of the after frame. (**resolve_add_new_balls**)
    - If the center of a newly detected ball is inside the contour of a before_ball this means that this newly detected ball is the slightly moved version of the before_ball so we can simply replace the old ball in the balls array with the new ball without appending it to the balls array to avoid duplicates.
- After adding/replacing all the balls, we should do one final pass to resolve any duplicate cue or 8 balls. (**resolve_ball_impossibilities**)
- And we finally have an array of balls in the after_frame. We can repeat this procedure by defining before_balls=after_balls and taking the next available frame.

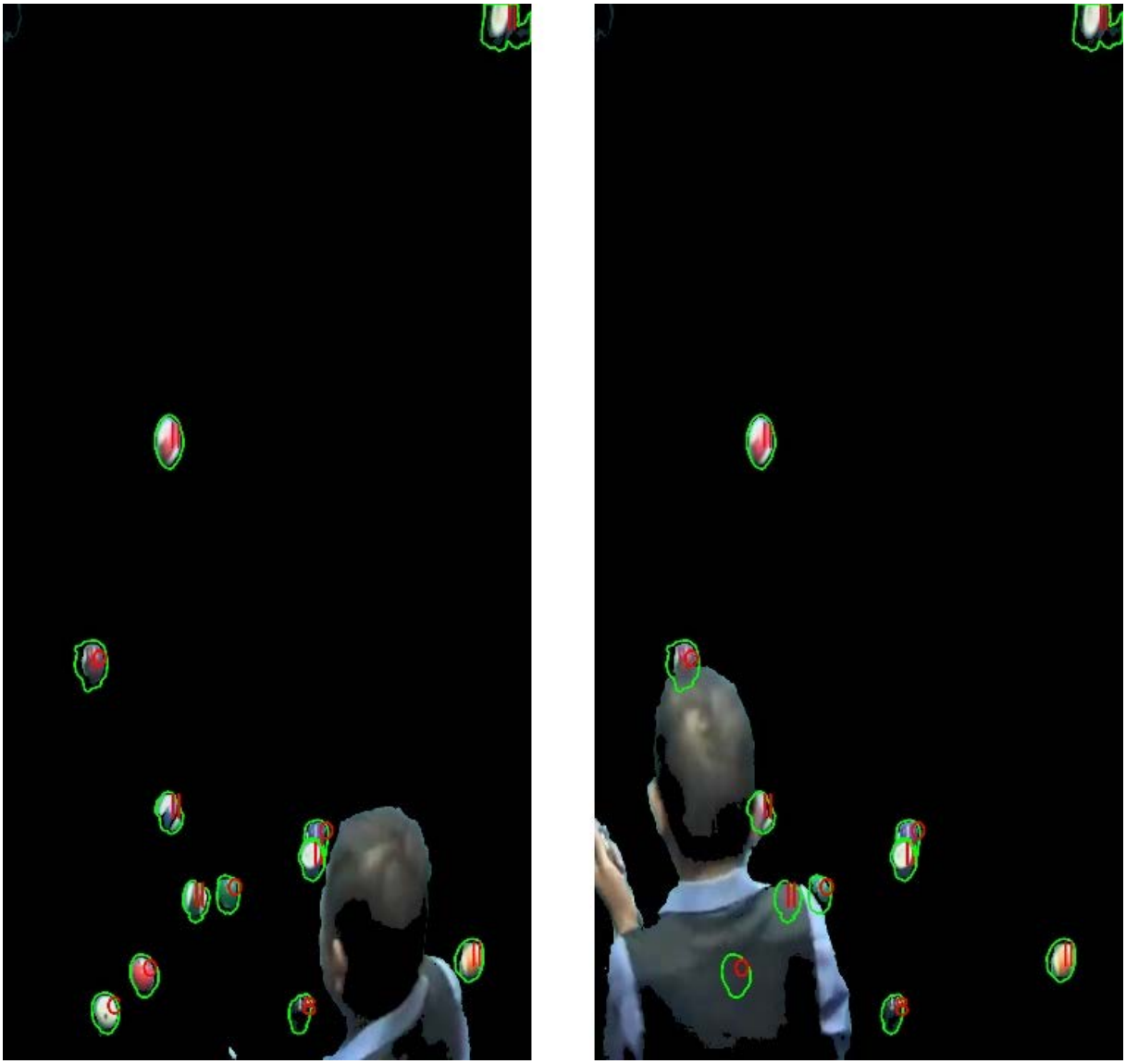


Figure 9: Obstruction Rejection in Action

As a result, the human rejection characteristics of this methodology can be seen in Figure 9.

IMPORTANT:

- The developed human (obstruction) rejection methodology was not carried over to Version 4 since it created problems in Version 4. It has been discussed here because the writer believes that the solution created here is valuable and could possibly be implemented in Version 4 after some refinement. (Currently problematic if a ball is hit very fast by another ball and the hitting ball is left in the position of the hit ball. The program detects the fast displaced ball as obstructed not moved)
- In the Version 4 Folder the code **test_3_stationary_continuity** does not do obstruction rejection. f.did_ball_move method was changed for the sake of Version 4, if the reader wants to access the version with obstruction rejection, the codebase of Version 3 in the folder “Old Versions” must be used completely.

Step 3: Dynamic Continuity (Version 4)

Going from Version 3 to Version 4 not all the features were carried through. Firstly, instead of remembering a ball obstructed by a person, it will be forgotten and redetected once the person moves away. This had to be done to increase the robustness of the methodology followed in this version. Secondly instead of storing all balls in a single array where many duplication, incorrect detection and sorting issues arise, the methodology of master balls was created and it will be discussed now. The program **test_4_dynamic_continuity** should be followed for this methodology.

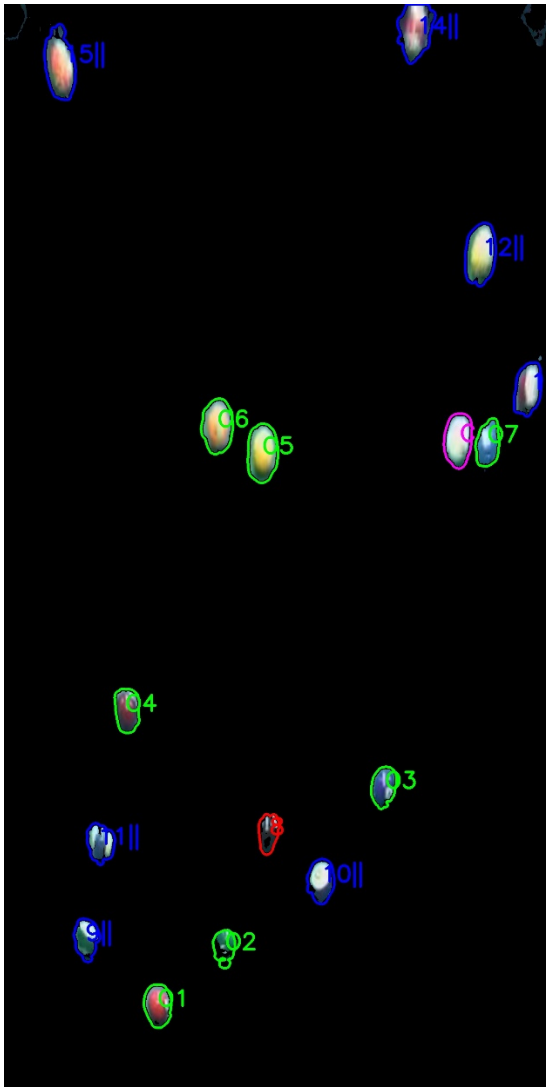


Figure 10: Each Ball on the Table Matched to a Master Ball

A master balls array is an array of 16 initially undetected balls. These balls will always have the classes of 1 Cue, 7 Solids, 1 8Ball and 7 Striped balls in this order of indexing. Master balls all start off with state=-1 (Undetected) and other fields empty. The function **update_master_balls** does everything for this program. It simply takes in the master_balls array and an image frame. It updates the master balls with the new information in the given frame and returns the updated master_balls array. Looking at the master_balls array, every information regarding the game in that frame of the video can be calculated. (Which balls are in game, which are potted, where are they etc.) Therefore, please focus on the **update_master_balls** function in file **f.py** now.

The idea is as follows, we have 16 balls in the game: cue ball + 15 numbered balls. Each master ball corresponds to directly that singular numbered ball. (For example imagine master_balls[5] is tracking Ball with Number 5 on it throughout the game) The only difference between a master_ball and tracking the actual ball number 5 is that the same class balls' order can change between themselves. **A master ball is a de facto way of following each individual ball separately.** Figure 10 better shows how each master ball looks. (For example: See that O4 corresponds to the Solid Ball Number 7. The class will be correct but the number is simply the index of the master ball. Master balls with index 1-7 can correspond to any solid ball with any real number since the real number of the balls are irrelevant to the game state)

1. Any ball that was known to be on the table in the last update must first be matched to a ball on the table. Create an array of master_ball_matched_flags with 16 elements. Any master_ball that was detected previously (not state -1) and not potted (not state 0) needs to be matched with a ball on the table. (For the master balls that need matching set their flags to 0, if not set their flags to -1. During our execution whenever a master ball is matched its flag will be set to 1.)
2. If a ball has not moved, set its matched_flag to 1, no update is needed to the ball data. This procedure is done nearly exactly the same as described in section Step 2: Stationary Continuity (Version 3) the only difference is that no recalculation of the ball class, image etc. is made. The old ball is simply assigned to be the new ball. This methodology works well if the real ball is classified correctly in its first classification, after that we will mostly track its position without having to reclassify it and it will have to be lost (undetected, obstructed) for its class to be recalculated and be reassigned a slot in the master balls.

   Out of the 16 slots, we have now matched some of the slots with the stationary balls.
3. We remove these stationary balls from the new_frame. We can now run our detection algorithm to find the moved balls. Our next step is to match each moved ball to the most likely master ball. (Another matched flag array is also created for the moved balls)
4. Taking each master and moved ball combination, if the center of the moved ball is inside one of the master balls' contours, this is that master ball's slightly moved detection. Update that master ball data with the new found moved ball data. Set the matched flags for the master and moved ball to 1.

   The slightly moved master balls are now matched to their new positions.
5. The cue ball is nearly always detected perfectly. If a cue ball is detected in the moved balls, simply update the cue ball slot (master_balls[0]) with the new data.

   If a cue ball is among the detected moved balls, update the cue ball slot.
6. If there are still moved and master balls that need matching: These are moved balls that have noticeably changed their positions. To make the best matching between the combinations of the found moved balls and last known position data of the master balls: Find the distance between each combination of moved and master ball. Then match the balls starting from the closest distance combination. (Given that the matching distance is not too big) Update and recheck the matched flags for each ball to avoid matching any moved or master ball twice. Also do not match any 'C' ball to 'O' or '8' balls and '8' ball to '||' or 'C' Balls. There could be slight mistakes in our classifications during execution but there will never be a mistake so dire that a solid ball will be classified to be the cue ball or a striped ball will be classified to be the 8 Ball.

Noticeably moved master balls are now updated if they were also detected in their new position.

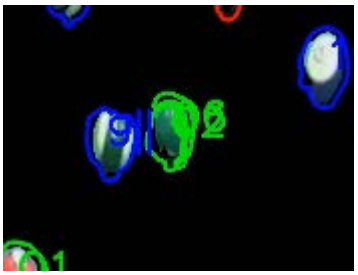
7. At this point if we are left with moved balls that still need matching: This means that the moved balls we have detected at this step is more than the master balls we already knew the existence of. This means that at this point these moved balls are newly detected master balls. There are 2 possibilities: Either a master ball was never detected (state -1) and this is its first detection or a master ball that was mistakenly thought to be potted (state 0). Either way we can assign each moved ball to the appropriate slot. (The slot where the class is the same as the moved ball class.)
   Unmatched moved balls are used to populate undetected master balls or correct mistakenly potted master balls.
8. At this point if we are left with master balls that still need matching instead: This means that a master ball is missing. There are 2 possibilities: Either the master ball was potted (If its last know position was a solo sector and now it is missing it was probably potted, change the state to 0. This condition is not robust on its own therefore the potted correction was implemented in the step before) or it simply went missing. (Undetected in this frame or obstructed. Change the state to -1 so that we will try to find it a match in the next iteration.)
   Unmatched master balls are updated to be potted or missing.
9. Lastly, assuming that the cue ball must always be in game. (No fouls are assumed) We should check if any of the master balls we have saved in another class actually could be the cue ball if its class was recalculated. Remember that we had assumed that whenever we calculate the class of the cue ball it will be classified correctly. This step has to be implemented because see that in steps 4,5 the slight movement of a ball case has higher priority than checking if there is a detected cue ball. When a cue ball strikes another ball really fast there is a possibility that between the 2 frames the striked ball just flies away and the cue ball actually ends up in the previous position of the striked ball. In such cases an actual cue ball image can replace the ball it had struck due to the “Slight Movement” condition. This step 9 remedies that past decision. (Remember that slight movement of a contour does not check if the before and after contours are the same class. This allows to detect a ball movement even though it gets mistakenly classified between 2 frames. (Physical movement was given priority over visual classification))
   Cue ball is made sure to be correctly matched.

Future Work

To improve the performance and robustness of this project the below topics can be explored.

- Using better image pre-processing techniques could improve the robustness of our classifier.
- Creating a better contour detector. This could solve the following possible problems:
    - During violent actions such as at the initial breaking of the balls many wrong classifications and detections may be made that won't be remedied easily.
    - Two or more balls together may not be able to be resolved to multiple separate balls.
    - If balls can't be detected a few frames before being potted, after being potted there is a possibility that they will be marked as undetected instead of potted.
- Creating a better contour classifier. (Technically if we had a perfect detection+classification system we wouldn't have to track the balls to keep track of the game state.) This could solve the following possible problems:
    - Currently balls are being classified as striped or solid but in reality they could be further classified to the exact number they are by looking at their colors. (If this could be implemented robustness could be increased noticeably)
- Implementing visual centers for contours. When a ball is left behind another ball or during slight movements a ball's contour will be detected like a crescent shape. (Concave) Yet we are still calculating the centers of concave contours from the center of the minimum enclosing circle. (A more popular alternative with a similar result would be calculating the centroid of a contour) However this might result in the contour center to be outside of the concave contour. As a result, a center might be detected to be in another ball's contour resulting in 1 ball to be defined as 2 as seen below. To prevent this, a visual center (A center that is guaranteed to be inside the concave contour's borders) should be calculated.

  This is actually an ongoing problematic topic in computer graphics. (Calculating the "visual" center of a concave polygon) In the literature this "visual center" is called as the pole of inaccessibility. This problem has procedural solutions such as the quad-trees method or makeshift methods using triangulation however to the writer's knowledge analytical solutions don't yet exist.



- This project does not implement class correction. This is not a problem if the detection and classification algorithms work well, however to increase robustness class correction can be implemented. (Currently, if a ball is tracked without interrupts the class will never be updated, so the classification made at the first detection has to be correct) A class correction algorithm can be developed. One idea could be to store the class history of a ball in the last k frames. And do corrections every once in a while by looking at the mode of the history.

Conclusion

As of Version 4 this software has the following capabilities:

- Extracting, rectifying and masking (Removing the tablecloth) an 8 Ball Pool Table.
- Detecting contours, filtering the ball-like contours.
- Classify these contours using their luminance and the ratio of colored to white-ish pixels they hold. (These classes are: Cue, 8, Solid or Striped Balls. A 5^{th} class called 'Hole' is used during the elimination of contours if the image of the holes are detected as contours)
- Differentiate and track each of the 16 balls separately during frames using only their contours.
    - Detect if that ball stayed stationary.
    - Detect if that ball moved only slightly.
    - Detect if that ball moved noticeably find its new position.
    - Detect if that ball was potted.
    - Correct itself if a ball was wrongfully detected to be potted.
    - Do these for all of the balls simultaneously.
    - Detect if a ball on a table is obstructed by a person, do not lose its position until the human moves away from the table. (Explored and implemented in Version 3, however depreciated in Version 4 due to robustness issues.)
- During frames where a ball may not be able to be detected, the self-correction mechanism mainly relies on marking that ball as undetected until it can be detected in a future frame.

In conclusion this project takes an 8 Ball Pool game video shot in a stationary single-view and demonstrates how a simple tuned deterministic vision model can be used to keep track of the game state using the continuity between frames and game logic.