POOL-AID: Detection and Identification of Pool Balls for the purpose of recommending shots to beginner players.

By Samuel Bauza, Brian Choi, Chuanquiao Huang and Olga Souverneva

Mentor: Ryan Kastner
6/8/2016
Spring 2016
Embedded Systems Design Project
University of California San Diego

1. ABSTRACT:

Pool or billiards is a popular pass time. However, as with many games, there is a barrier to entry for beginners who struggle to visualize critical shots and miss the opportunity to score critical points. POOL-AID addresses this problem by providing shot trajectory guidelines. POOL-AID consists of an algorithm used to detect billiard balls on a pool table, then calculate shots and overlay them on a rotated video. First openCV is used to isolate the table, then detect each pool ball. Table isolation is done by using color subtraction, blob detection and an affine transform. Ball detection is done by background subtraction, blob morphology and detection. We implement a color-based classifier with bipartite matching. Finally, physics calculations are used for simple shot / collision detection.

2. INTRODUCTION:

Our primary intention with POOL-AID is to guide beginner players by displaying shot and ball trajectories during a game of standard 8-ball pool. Due to the popularity of the game and technical challenge of pool, it is unsurprising that a number of similar applications have been presented in both published literature and the hobbyist community [1-5]. These make use of desktop as well as mobile platforms. However there does not exist an application that can be implemented on a compact, self contained, embedded device like an FPGA. Such a platform has the advantage of accelerating vision algorithms as well as being power efficient and portable enough to become a standalone embedded device in the future. Thus the secondary intention of POOL-AID is to present an implementation that could be implemented in full or in part on an FPGA.

The POOL-AID system design is consistent with existing work in that a standard, inexpensive video camera (GoPro Hero3) is mounted vertically above a pool table [1-2, 4-5]. The table takes up most of this frame and is assumed to be in focus and in frame. Using color subtraction, blob detection and an affine transform we are able to isolate the pool table and limit the frame to just the playing field . Then we subtract the background of the table canvas. The balls in the playing field are detected using blob detection and blob morphology. We classify the detected balls based on hue and value in the HSV color space. At this stage of the algorithm there is a high chance of false positive detection thus balls are classified as one of the 16 balls that could be in play or as false positives. To reduce error we perform several iterations of detection-classification. After the last round, we perform run a bipartite matching to get the most-likely classification of all the balls in the frame. Finally, physics calculations are used to determine shot trajectories and collisions that are displayed to the user using arrows overlaying the frame.

Our project differs from some of the existing work in the following:

1. We use standard pool balls with no custom tags or markings.
2. We attempt to detect and classify balls even if the playing field is obstructed by a player's hand or a cue stick.
3. Our application works for moving as well as stationary balls.

This makes the design more challenging in that our ball detection is not assisted by any tags. Standard pool colors are less differentiable than custom colors which makes color-based classification more difficult. Playing field obstructions can easily confuse the detection and classification algorithms. Since we are using an inexpensive camera for this project, the shutter speed is insufficient to control motion blur. Thus the moving balls are less defined in both shape and color which complicates both the detection and the classification.

Due to these challenges we limit the scope of our work in the following: The initial POOL-AID implementation is intended to work with a single pool table and set of balls of non-television colors. However we present ideas on how to modify our design to make it more versatile. POOL-AID is currently limited to standard 8-ball pool and would need to be extended to cover other types of pool like 9-ball. The output of POOL-AID is

video overlaid with shot trajectories, however a more sophisticated user interface can be added in the future. Additionally, we assume that the user can record several seconds of video before beginning a game to aid with background subtraction. Finally, we use COTS tools to remove the fisheye introduced from the GoPro from our footage and do not attempt our own implementation.

3. TECHNICAL INFORMATION:

3.1 CAMERA PLACEMENT

The next section will detail the methods used to produce these results, as well as methods that were tried, but did not produce as strong results. In its current state, the algorithm uses a video taken from approximately the perspective shown Fig. 4. Video framing requirements are as follows:

- The center of the image should be the pool table
- The table, including pockets, should be entirely in frame
- The edges of the pool table should be distinctly colored from the cloth of the table
- Camera must be perpendicular to table
- The long edge of the table should match the long edge of the frame, though this can be accounted for

3.2 TABLE ISOLATION

It is important for later stages of the algorithm to know the location of each corner pocket in the frame. It is also highly beneficial to have the table take up the entire image. To achieve this, the region of interest, or ROI, must be determined. This is done by:

1. Determining table color
2. Subtracting all other colors from image
3. Finding contours in remaining image
4. Selecting largest contour and fitting a rectangle to it
5. Using bounds of rectangle to rotate original image to fill frame

There are several methods of determining the table bounds. The first method attempted was to simply use the openCV function findcontours(*1) and select the biggest contour as the table. This provided generally jittery results and was often wildly inaccurate.

The next attempted method was to isolate the table color in the image, and use findcontours on the resulting image. There are several methods that may be used to determine table color:

1. Hard code the table color
2. Have a user manually select the table color
3. Assume the center of the image includes the table, and use that color
4. Take a histogram for each color in the image, and try to infer table color using peaks

In its current state, the color of the table is hard-coded, though in later iterations the colors of a certain region around the center of the frame will be averaged, and assumed to be the table. Removing all colors in the image not within a certain range of the determined table color, produces the mask in Fig. 5. Running findcontours in this mask was much more reliable and robust than running it on the initial image.

With the list of contours produced by the openCV function findcontours, the table contour is found by selecting the largest contour. This worked reliably for all footage captured for this project, and is expected to work reliably in most cases where the table is framed properly. Issues may arise if the carpet or background is a similar color to the table.

With the table contours selected, the bounds of the table can be determined by using one of two primary methods.

1. Use minEnclosingRectangle(*2) and simply use its bounds

2. Use a HoughLinesP to find lines in the image, after masking it with the mask produced above, cluster them based on proximity, and determine bounds using these clusters

Method 2 was not attempted this quarter, as method 1 was simpler and provided satisfactory results, despite non-ideal lighting conditions. If there is a strong lighting gradient across the table, or if the camera is not perpendicular to table, method 1 may be non-sufficient, in which case other groups(*3) have found success using method 2.

With the table bounds determined the original image can be rotated to have the table fill the frame., the openCV function getAffineTransform(*4) can be used to generate a warp matrix. This function takes as arguments three points in the original image, and 3 in the desired output image. The warp corners may change depending on the alignment of the table (vertical vs. horizontal). This warp matrix is then passed, along with the original image, to the openCV function warpAffine(*5). The resultant image is shown in Fig. 6.

3.3 Ball Detection

The next step in the algorithm is to determine with as much accuracy as possible, to location of potential pool balls. This is done by

1. Using an initialization period of at least 30 frames to average the table bounds produced by the method above, then holding bounds constant
2. Once the image is stable, take an average of the image for at least 10 frames to determine the table background
3. During regular play mode, subtract this background from the image
4. Threshold the image that remain to generate a mask
5. Find contours on this mask, and prune them by area/ circularity criterion
6. Use several iterations of blob morphology / findcontours to ensure all balls are detected

Other methods were tried to detect pool balls without an initialization period, to a lesser success.. HoughCircles worked well, though decreased in accuracy when balls were in motion, and often picked up large amounts of noise. FindContours similarly picked up large amounts of noise.

In its current state, an initialization period is needed to determine bounds of table as well as the background. This initialization period allowed us to generate much more accurate detection results than were possible without it. During this period, the table must be completely unoccluded. This includes players / cues leaning over the table, and pool balls / rack already on the table.

The first step of the initialization period is to average the table bounds. The calculation provides a jittery table, and averaging the results allowed for a smooth image to be produced. This assumes the camera does not move relative to the pool table, which for most cases will be a reasonable assumption. After determining the table bounds and rotating the image, take an average of the image over several frames. This will produce a reliable background image.

The first step in the regular play algorithm is to subtract the background. Simply subtracting the background produces an image that looks similar to the original, but is actually color shifted. To ensure this color shift does not affect ball identification, the subtracted image is used to create a mask, which is placed over the original image to generate Fig. 7. To the naked eye this appears to be a decent representation of the foreground, however it is color shifted. The mask is created by thresholding each individual color channel (RGB or HSV). Tweaking of the threshold value for each separate channel will be necessary to produce a decent image that does not exclude billiard balls similar in color to the pool table.

The second step is to look for contours in the image. To remove all contours that are too large to be a ball, contours that include too much of an area are excluded. Excluding contours with an extremely low circularity also

helps remove the pool cue. The result is good when no players / cues are occluding the table, and no balls are in close proximity to each other, however it fails when these criterion aren't met.

To account for balls in close proximity to each other, we use a loop that repeatedly applies a morphological operation to the mask. By doing several iterations, the algorithm had better accuracy in picking up both difficult to detect balls (black, striped blue / violet) as well as adjacent bright balls (red/orange). This did come at a significant time cost, so other methods should be examined. See Fig. 8 and 9 for the mask before and after this iterative process.

This process did have an issue with occluding players, particularly when wearing a shirt color similar to the table felt. To account for this, it may be beneficial to erase all contours which touch the edge of the frame, as these can only be occluding objects and not the billiard balls we are looking for.

An image of each potential ball plotted on the frame is in Fig. 10.

3.4 BALL IDENTIFICATION:

Threshold-based classifier:

Standard 8-ball is played with 16 balls. The balls can be differentiated by color and by the presence of a stripe. Since the colors are easily distinguishable to the human eye, our initial approach was to just use thresholds for determining the color and presence or absence of a stripe. Classification consisted of the following simple algorithm:

1. Determine the min and the max threshold for each of the nine colors using the training data. We used mean +/- standard deviation of the distribution for each channel with some manual adjustment to make the categories exclusive.
2. Determine the min and max white-to-other color ratio for the presence of a stripe. We used mean +/- 2 standard deviations of training data.
3. For each ball patch in the frame, for each color, sum the number of pixels between the min and max thresholds.
4. Find the color with the highest count. If the count is below a minimum, output false positive.
5. Compute the ratio of white-to-other color.
    a. If ratio > max ratio for striped balls, output cue ball
    b. If ratio within min and max thresholds for striped ball:
        i. If color black, select next highest color, and repeat.
        ii. Else output striped color.
    c. Output color.

To develop the classifier we selected 50 frames from our recorded video footage that contained most of the balls and were at least 20 frames apart. These were manually classified for a total of 850 image patches sized 8x8 pixels to 16x16 pixels. We split the frames randomly into a training set of 24 frames and a test set of 26 frames. The high ratio of test to train images is necessary as some of the balls are in motion for only a small portion of the footage and we wanted to ensure representative testing. We chose to split by frame as opposed to by individual ball as we wanted to be able to apply a matching algorithm, described later. A selection of balls in the test set is shown in Fig. 11.

We tested such a classifier in RGB and HSV space with no noticeable change in accuracy. However we found that while RGB classification required all three channels, in the HSV space only the Hue and Value channels were needed. This type of classifier achieved 72.2% accuracy on the test set with most of the errors coming from incorrectly classifying striped balls, Fig. 12. For example, all the red-striped balls (11) were incorrectly classified as solid red (3).

Max Matching Classifier development:

The threshold-based classifier yielded low accuracy. A primary reason for this is that when the balls are in motion, it is difficult for even a human observer to say if a shape is the striped-yellow ball (9) without looking at the other balls in play. Thus the problem is equivalent to weighted bipartite matching problem between the detected centers and the ball classifiers. This insight lead us to trying to find the most-likely or the lowest-cost matching for all the centers in the frame. The algorithm had the following key steps:

1. Determine the probability density function (pdf) for the training set data. For the black ball (8) we used Value whereas for the other balls we used Hue. Note for the reddish balls the Hue needs to be transformed to range from -0.5 to 0.5 (-180° to 180°) as the Hue wraps between 0 and 1 (0° and 360°). Note that striped balls consist of white pixels and other colors. See Fig. 13-14. We used kernel density estimation (KDE) with the epanechnikov kernel.
2. For each ball patch in the frame, for each label, estimate the probability that the ball distribution is the same as the training distribution.
3. For the frame, compute least cost matching between the detected balls and labels. We used the hungarian/munkres algorithm with $O(n^3)$ running time because of ease of implementation and small number of nodes in the graph.

When tested on the same test set as the threshold-based classifier this classifier achieved 98.64% accuracy. Most of these errors were due to incorrect classification of false positives (ghosts) and striped balls, Fig. 15.

While we were unable to implement and benchmark the algorithm on an FPGA we measured the runtime of our C++ implementation on CPU (Windows 10, Intel Core i7). Our original implementation of Step 2 used an open source version of pdf estimation for each ball patch. However this ran slowly, and we found that by using a histogram alone we could significantly improve running time with no noticeable effect on accuracy. To further optimize the runtime, we used multiple threads to exploit the parallelism of the cost computation. 3 Threads yielded optimal running time of 13.2ms. This is below the running time required to achieve 40fps.
The running time with the max possible threads (1 per detected ball) is worse due likely due to thread spawning, heap access, and scheduling overheads, see Table 1.

Table 1

Classifier running time on CPU per video frame with 17 balls detected in frame (i.e. 1 false positive)

	Original			**No pdf calculation**		
Threads	**1**	**3**	**17**	**1**	**3**	**17**
Time/Frame	**8.276s**	**5.99s**	**6.458s**	**18.072 ms**	**13.181ms**	**26.335 ms**

Future improvements:

While a color based classifier was the simplest to implement, it is the least tolerant of variation in lighting and differences in pool ball sets and tables from different manufacturers. While max matching helps address the problem, it breaks down when multiple balls are blurred, badly lit, or absent as well as when a large number of false positives are present. This created many integration issues with the iterative ball detection. We have thus thought of the following improvements to improve accuracy and versatility. One observation was that classification probability drops exponentially with distance from previous location. Thus we introduce a distance correction to the probability, $P_t = P_{ball} + Ke^{(\frac{-d}{\mu})}$, where d represents the distance from the ball's location at t-1, μ is determined from the training data, and K is a weight set experimentally. In our initial implementation we use (x,y) location (0,0) for balls

absent in the previous frame. If we implement tracking of scored balls, the pocket location can be used instead. Prototyping this correction in Matlab showed improved accuracy on our test set of 99.5%. Tracking of scored balls would also allow us to improve our matching by eliminating scored labels for the duration of the game. An additional future feature we have considered is use of a calibration algorithm while the balls are racked and before the game begins. This would help POOL-AID be tolerant of pool ball sets from different manufacturers however it would not help with irregularity of lighting in different areas of the playing field or throughout the game. Ideally we would like to move away from a color classifier to a feature based one in the future. It may be possible to use of a Viola and Jones-like detector for detecting striped, black, and cue balls on grayscale images of the ball patches. Such a classifier combines an Adaboost classifier with Haar-like features. However Haar features are not rotation invariant and may not be suited for round/spinning objects. The pool balls also exhibit multiple modalities, not only with rotation, but with motion blur, which complicates data set labeling and classifier design.

3.5 SHOT RECOMMENDATION:

Shot recommendation is meant to help pool players to learn the game. In order to help a player to better understand the game, Pool-Aid visualizes shot paths, with which a player can easily tell at what angle to hit a ball to score.

Key challenges for a pool game player are:

1. The angle to hit the cue ball in order to make a target ball roll into a pocket.
2. Where the balls will land at and after a collision.
3. Selecting the sequence of balls to hit to maximize the outcome of the game.

Pool-Aids aim to solve these problems for pool player by using the output of ball detection and identification to generate a shot recommendation.

Shot recommendation implementation:

Shot recommendation is premised on ball locations, pocket locations, and physics. Consider the following game scenario involving a collision between balls “A” and “B” (Fig. 1)::

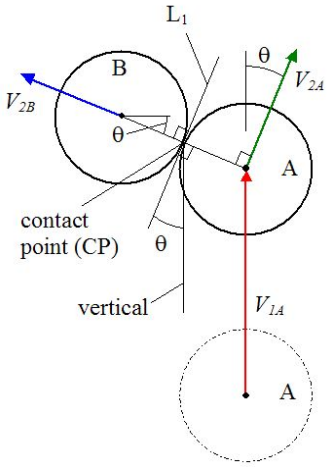

Figure 1

The blue arrow indicates a path that we want the ball “B” to go (Presumably the blue line is pointing at a pocket). We need to know where the initial cue ball location, “A” dashed, ball “B,” and pocket are. We rely on the detection and identification of the pool table to output these locations. Then we apply calculation based on the locations to find the contact point, $\Psi(x, y)$, where the collision takes place and the angle Θ that cause the collision take place. With $\Psi(x, y)$ and Θ, we can render the red line above for the pool game. This process solves the first challenge of “what is the angle we must hit the cue ball at.”

Since the process of shot recommendation runs before a real shot takes place, we don't know what is the velocity of the cue ball so to predict where it lands will be challenging. But shot recommendation is always focus on making pockets after a shot. In other words, our goal of taking a shot is to hit ball "B" into a pocket. Therefore we can estimate where the balls will land at after collision based on the data we have. First select a possible pocket on the table that ball "B" can go in. Calculate the range of velocity ΔV_{2B}(min, max) for ball "B" that can reach the pocket.

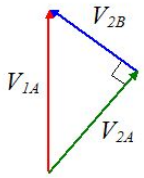

Figure 2

Now we have the estimated velocity of ball "B." By applying some mathematics we can get the estimate values of velocity V_{2A}. V_{1A} is the main impact of the calculation which depends on the player . Thus we add an uncertainty variable ω into the calculation. Then we are able to get a set (noted as δ) of possible locations of where the cue ball will land at after collision. However, if we provide this location set to a player, that will be hard for the player to understand and rendering will be difficult. Hence we take the mean of all the location inside the set and output the mean value to the renderer as our final decision. This process solves the second challenge of determining "where the balls will land at and after a collision."

Figure 3
Demonstration of the shot recommendation overlaid on the table

Future improvement

While Fig. 3 demonstrates that we have addressed the first and second challenge, we want to solve the last challenge of "selecting the sequence of balls to hit to maximize the outcome of the game" in the future. Our idea for an algorithm is to calculate the set δ of cue ball based on every hittable ball on the table and merge all the set δ's to a power set (noted as Σ). For each location in Σ, we make a filter to select the one that is closest to a hittable ball which is the closest to any hittable pockets on the table. Then we manipulate the set with the filter to get the best shot (noted as β) and output it . Finally, we output the data of β to renderer to display the shot path from cue ball to ball β and the aftermath location of collision.

3.6 HARDWARE ACCELERATION:

Several video processing functions would greatly benefit from acceleration. As part of the project an attempt was made to use an FPGA to accelerate portions of the algorithm, though this was ultimately unsuccessful. Several notes regarding the use of FPGA tools such as Vivado are included here.

The following is a rough description of how to generate and upload a video processing structure onto an FPGA.

1. Write HLS code using C/C++, or use existing openCV HLS libraries
2. Use Vivado HLS to generate an IP core, which can be imagined as a hardware block
3. Use a separate program to connect your custom IP core to the processor
4. Program the processor using more accessible C/C++ code, though in a way that integrates your block
5. Upload everything onto the board and processor.

Our group encountered numerous issues throughout this process, many tutorials are specific to a particular version of Vivado, and many structures require unique licenses that we did not have access to. If you are using HLS design tools for the first time, expect the process to take several months.

Ultimately we were able to output solid bars through the HDMI port (Fig. 16-17), though we were not able to load images / video onto the board in time for our final submission dates.

4. MILESTONES:

We started the quarter aiming to achieve high detection and classification accuracy, design shot recommendations, and implement our algorithms on the FPGA. However while we succeeded in design milestones we did not finish our FPGA related milestones as they ended up being more technically challenging and took longer than the time allowed.

The following milestones were successfully completed in this project:

1. Collect an hour of pool footage and create test and training data sets.
2. Dynamic Pool Table Identification.
3. Pool table subtraction.
4. Accurate Pool Ball Detection
5. Pool ball classification 95% accurate and at 40fps.
6. Convert all code to C/C++.
7. Shot physics/trajectory calculation.

We ended up modifying and or not completing the following milestones:

1. End-to-end pipeline integration. While we converted all our code to C++, integrating the iterative detection and integration posed some design challenges and we have several unresolved integration issues.
2. Libraries Compiled using FPGA synthesizable code. While we could not complete this for all our code, we synthesized a function calling an openCV HLS version of the openCV method used in our project.
3. Benchmark performance on the FPGA.As we did not finish loading our code to the FPGA, we did some benchmarking on CPU instead.
4. Stream video/ images to the FPGA. This milestone was modified to showing color bars from the FPGA.
5. Full video capture/processing independent of PC. This was not completed as it was dependant on earlier milestones listed above.

5. CONCLUSION:

POOL-AID demonstrates how object detection, physics, and geometry can be combined to provide helpful shot recommendations. While our current application runs only on a PC, it can be modified to make us of a FPGA in the future. With the following modifications our project can be relevant and helpful to players with varying amount of skill.

1. Support variation of lighting during the pool game.
2. Support different pool ball sets and tables.
3. Keep track of pool balls that were shot into pool pockets for the duration of the game.
4. Make use of a non-color-based classifier.
5. Recommend not a single shot, but the first of a sequence of balls to hit to maximize the outcome of the game.
6. Use HLS to accelerate object detection algorithms.

9. REFERENCES:

(1) Lars Bo Larsen, Rene B. Jensen, Kasper L. Jensen, and Søren Larsen. Development of an automatic pool trainer. In Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology, ACE '05, pages 83–87, New York, NY, USA, 2005. ACM.

(2) Lars Bo Larsen,Peter M. Jensen, Kenneth Kammersgaard, and Lars Kromann. "The Automated Pool Trainer - a Multi Modal System for Learning the Game of Pool." Intelligent Multi Media, Computing and Communications : Technologies and Applications of the Future. IEEE Computer Society Press. 2001, pages 90-96. accessed online June 2 2016 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.982&rep=rep1&type=pdf.

(3) "Detecting billiard balls with OpenCV." http://stackoverflow.com/questions/8162234/detecting-billiard-balls-with-opencv

(4) Jesper Baekdahl and Simon Have. Detection and Identification of Pool Balls using Computer Vision. 2011. accessed online June 2 2016 http://vbn.aau.dk/ws/files/52663081/main.pdf

(5) "Hacking on Side Projects: The Pool Ball Tracker." accessed online June 2 2016 http://vbn.aau.dk/ws/files/52663081/main.pdf

(6) openCV findContours documentation: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#findcontours

(7) openCV minEnclosingRectangle documentation: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=minarearect#minarearect

(8) Group using Hough Lines to detect pool table https://pranjalv.com/poolvision.pdf

(9) The physics of Billiards: http://www.real-world-physics-problems.com/physics-of-billiards.html

Figure 4.
Perspective of footage that is sent to the algorithm

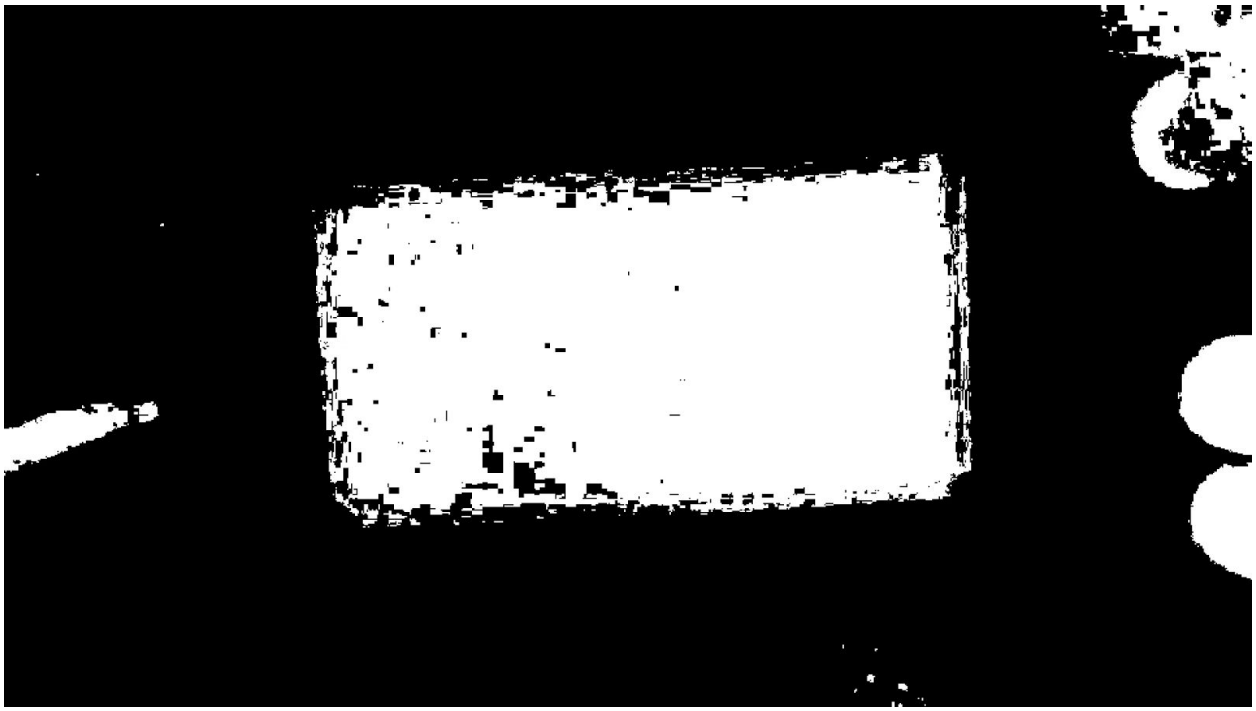

Figure 5
Masked image of table after table color is isolated

Figure 6
Table image after it has been warped to fill the frame

.

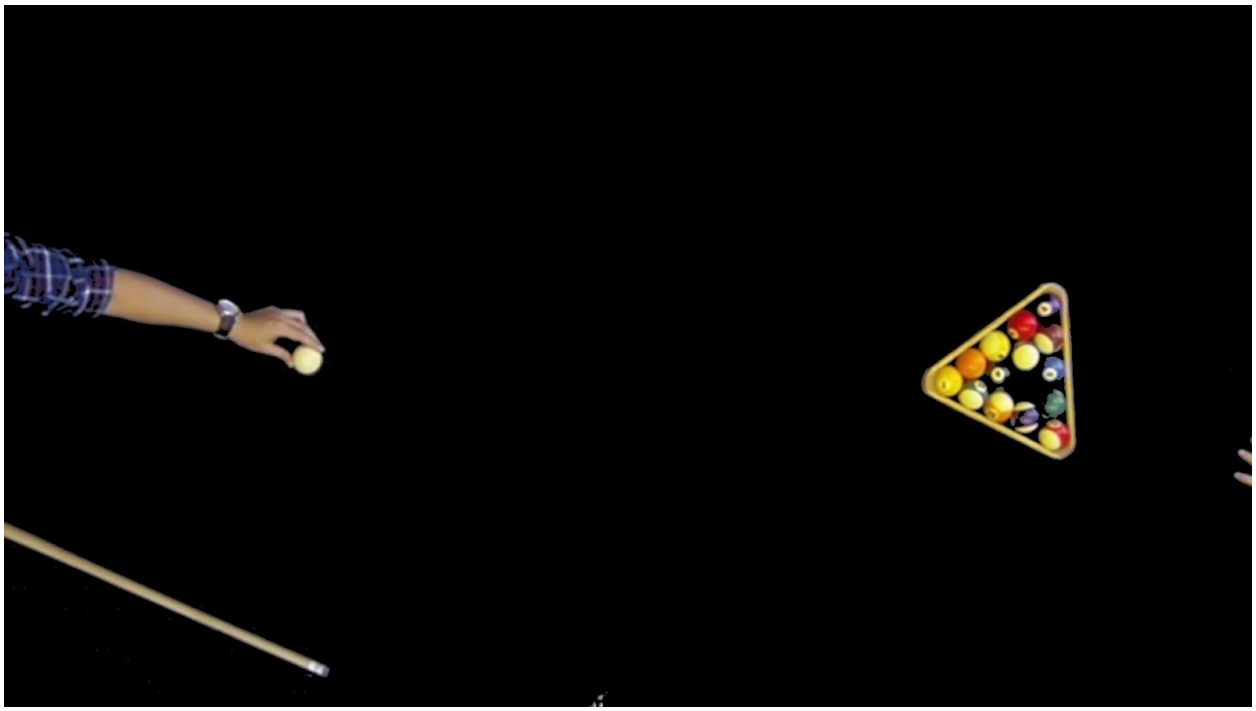

Figure 7
Image after background is masked out

Figure 8
Mask before morphological operations

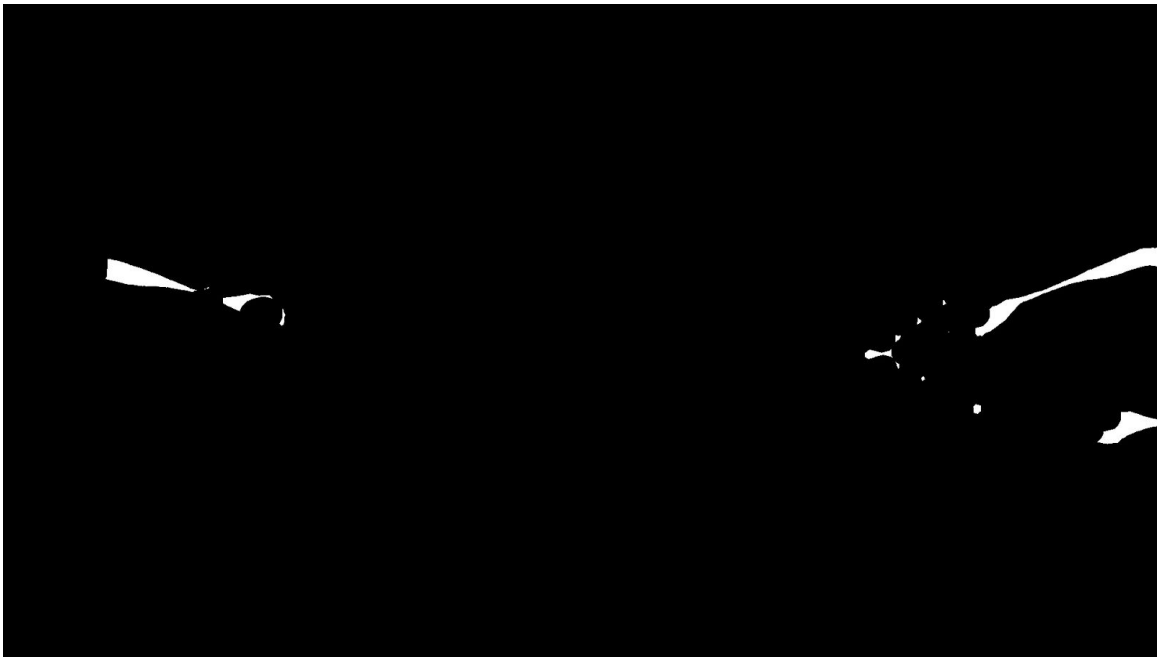

Figure 9
Image after morphological operations

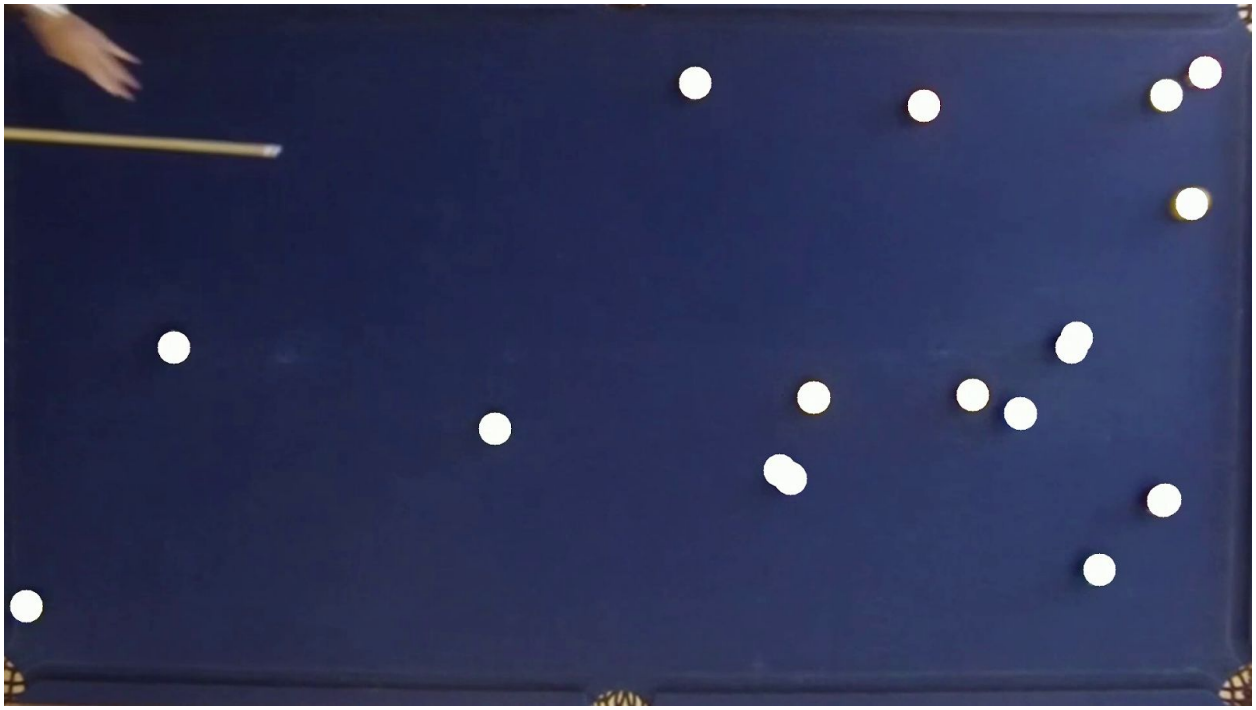

Figure 10
Image once each potential ball is displayed over table image

Figure 11
Selection of pool ball test images. From left-to-right: false positives, balls 1-15, and cue ball.

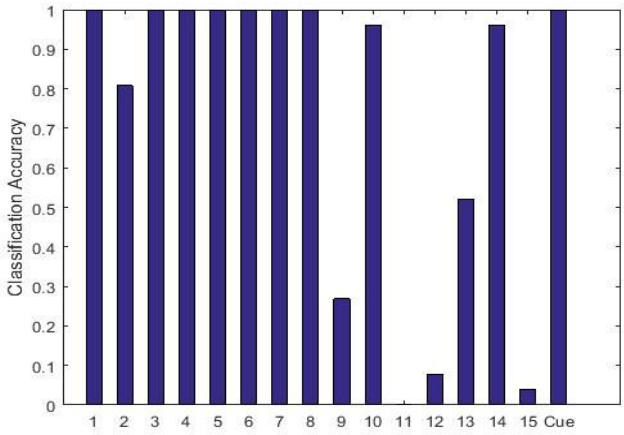

Figure 12. Test classification accuracy of the threshold-based classifier.

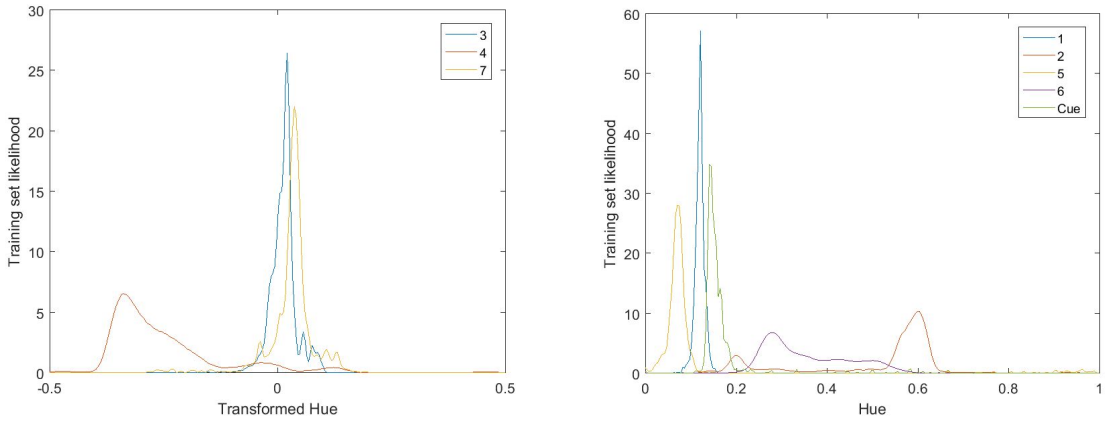

Figure 13

Pdf for reddish (left) and non-red (right) balls. Note the small overlap between yellow (1) and cue ball and the large overlap between the red (3) and maroon (7) balls. Since a matching algorithm is used, this is acceptable.

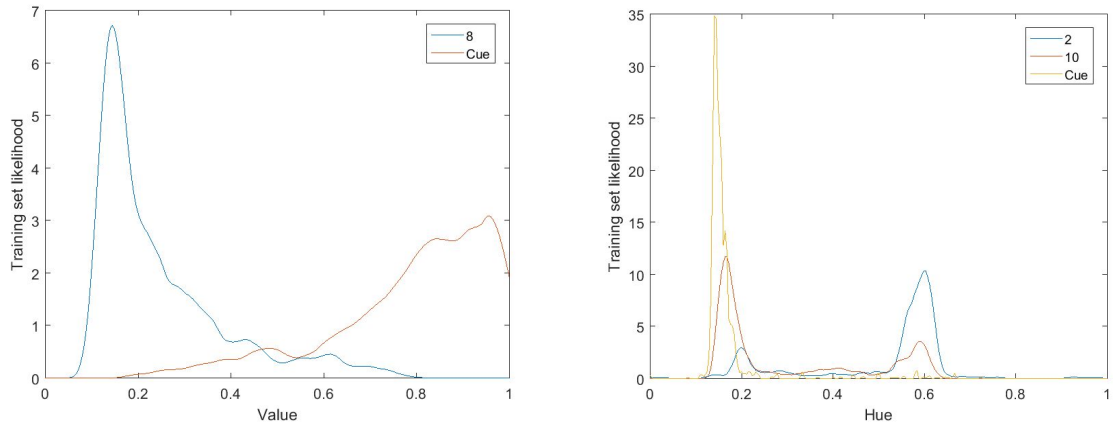

Figure 14

Special cases for the black and striped balls. Left, the black ball is not distinguishable by hue thus the value channel is used. Right, pdf for the blue-striped ball (10). Striped ball pdf overlaps both the solid (2) and the cue distributions.

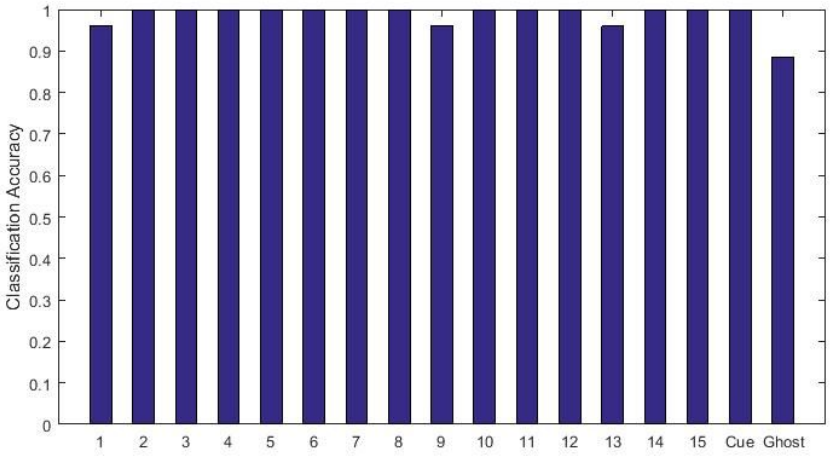

Figure 15

Test classification accuracy of the Max Matching Classifier classifier.

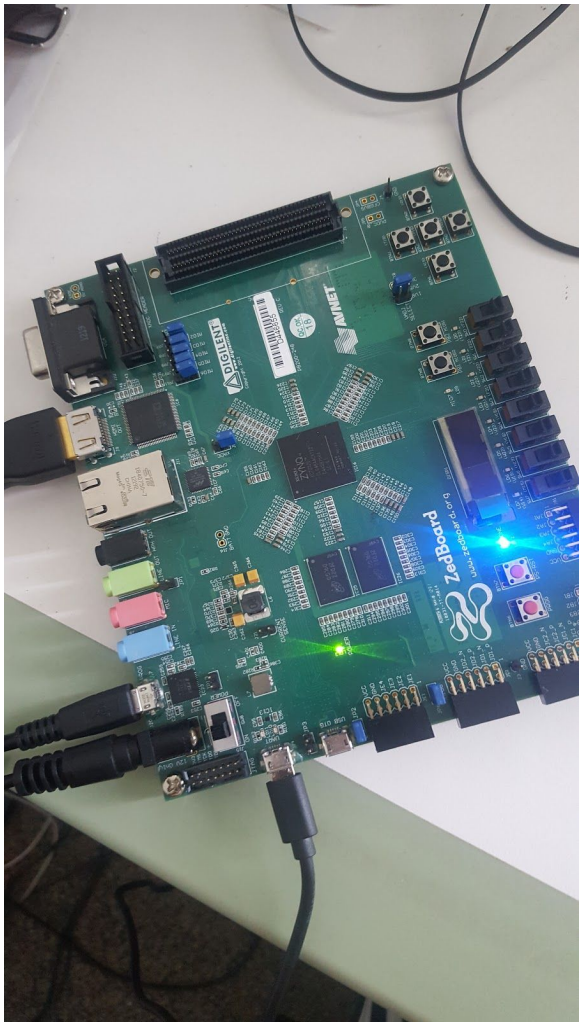

Figure 16

Figure 17