Board Game - Gameplay Detection Project

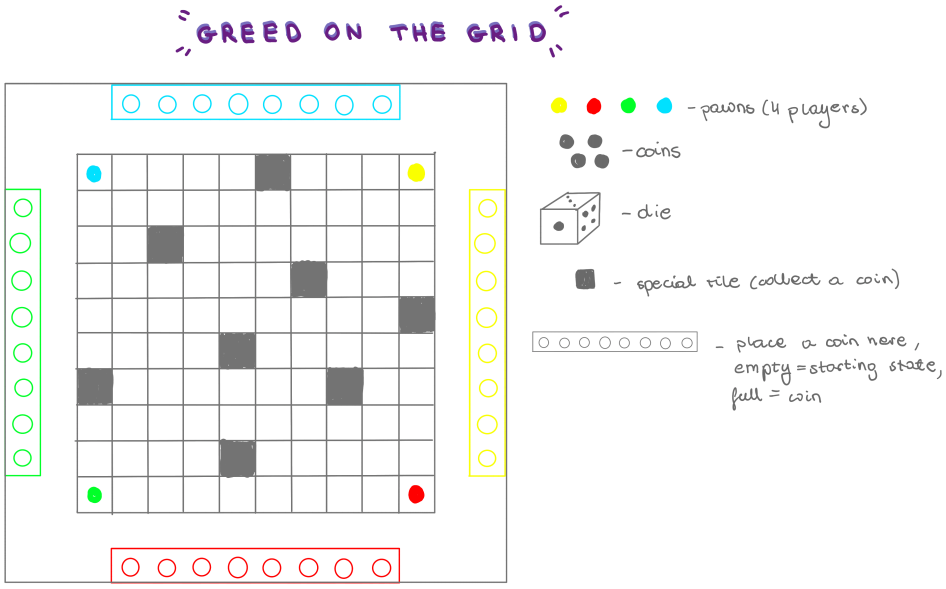
Authors

1. Julia
2. Marcel

1. Selcted Board Game

1.1 Description

"Greed on the Grid" is a competitive board game where 4 players race to collect coins on a grid-based board. Each player starts in a designated corner with a pawn and takes turns rolling a die to determine how many steps they can move in four possible directions: up, down, left, or right. Landing on a special tile allows a player to collect a coin, but they cannot use the same special tile twice consecutively. The game ends when a player collects 8 coins, filling their coin slots, and is declared the winner.



RULES:

1) A pawn can move in four directions: up, down, right, left.
2) Rolling a die determines the number of steps
3) If a pawn lands on a special tile, a player may collect one coin.
4) A player cannot land on the same special tile twice in a row.
5) If a player collects 8 coins, they win.

1.2 In-game items

There are three in-game items:

- pawns (four: red, yellow, green, blue),
- coins,
- dice.

1.3 Game events

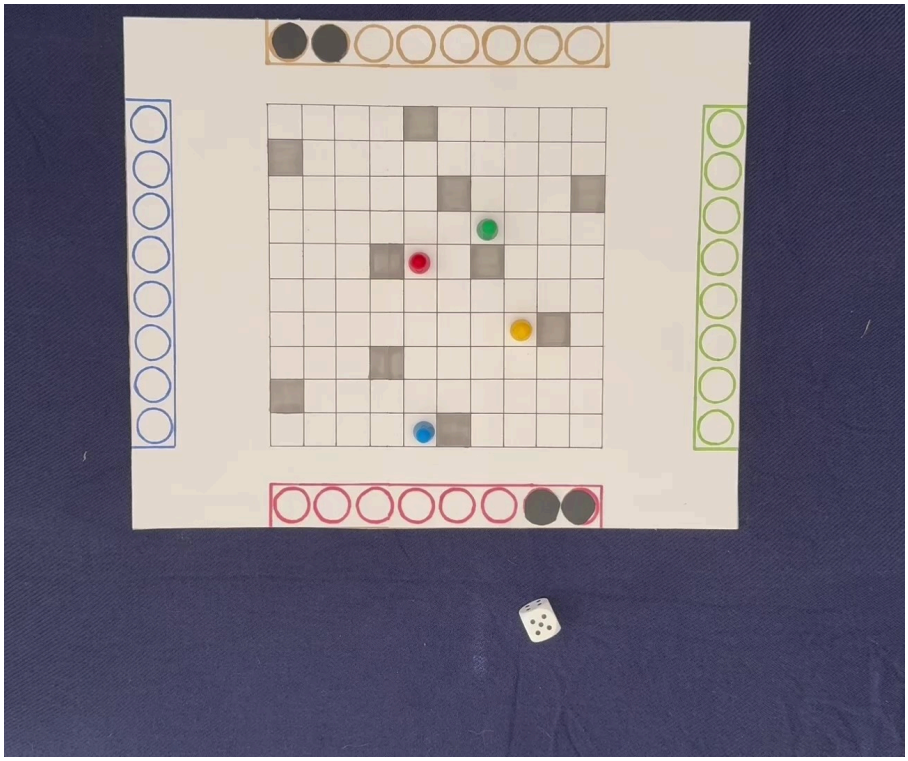
The game events include:

- **Starting the Game**: All 4 pawns are placed in their respective corners, and the coin slots are empty.
- **Ending the Game**: The game concludes when a player collects 8 coins, filling their coin slots.
- **Rolling a Die**: Determines the number of steps a player can move during their turn.
- **Moving the Pawn**: Players can move their pawn in one of four directions: up, down, left, or right.
- **Landing on a Special Tile**: If a pawn lands on a special tile, the player has the option to collect a coin.
- **Collecting a Coin**: Players collect a coin upon landing on a special tile and place it in their coin slot.

2. Dataset Description

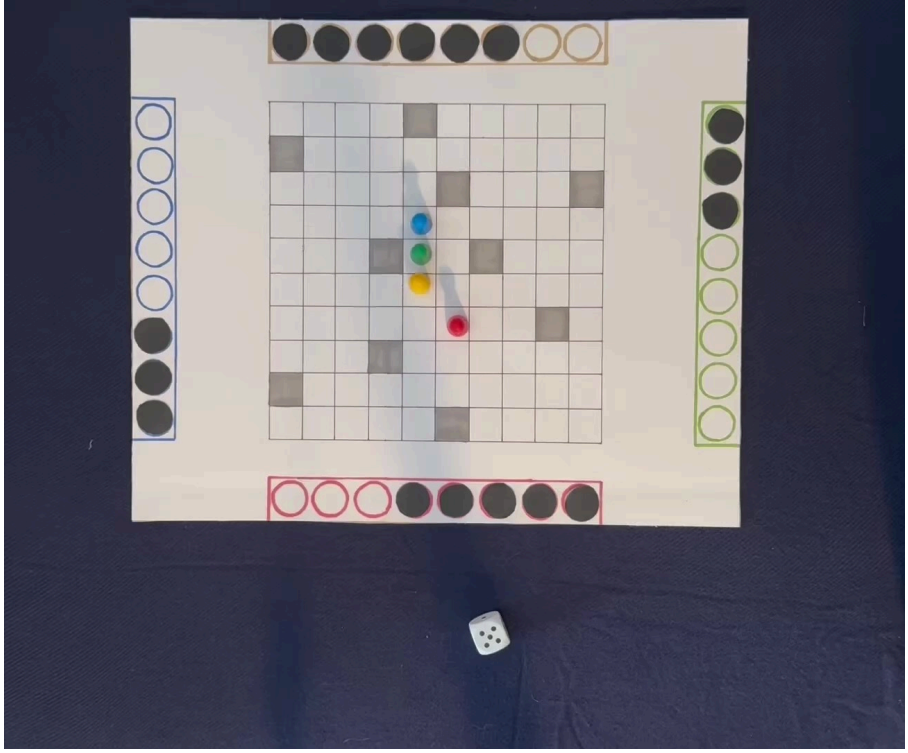
2.1 Dataset - difficulty: EASY

The videos were recorded from a bird's-eye view, with ideal lighting conditions provided by overhead light. The camera was positioned to include only the board and dice, ensuring a clear view. The player's hand rarely obstructed the game pieces, allowing for unobstructed visibility throughout the recordings.



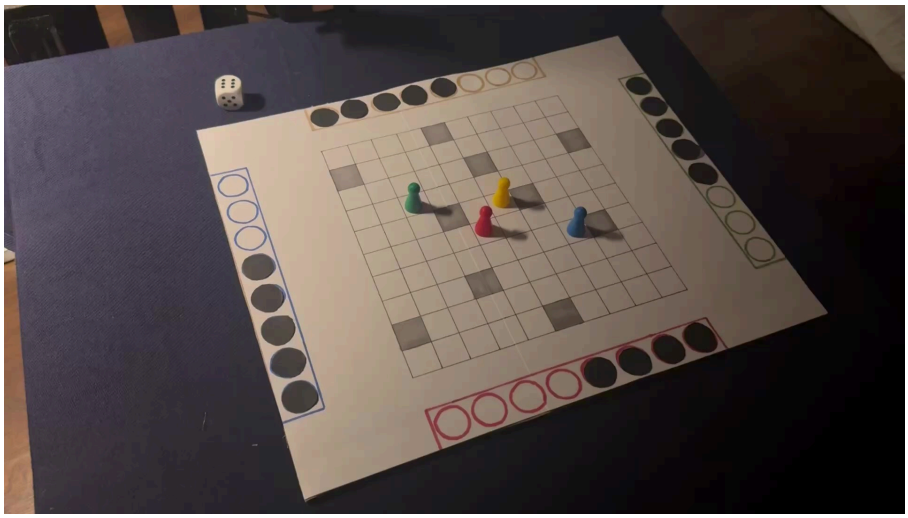
2.2 Dataset - difficulty: MEDIUM

The videos were captured from a bird's-eye view with the camera focused on the board and dice. However, varying lighting conditions were introduced, including shadows, reflections, and natural light sources. The player's hand occasionally covered the game pieces, resulting in moments where the dice and board were partially obscured.



2.3 Dataset - difficulty: HARD

The videos featured different filming angles, including perspectives that deviated from the bird's-eye view. Shadows, reflections, and frequent camera movement or shaking were present, adding to the complexity. The player's hand often covered the game pieces while interacting with them, with camera movement ranging from slight shifts to significant shaking.



3. Experimentation

3.1 Coin Detection

a) First iteration of tech

Our initial approach to coin detection relied on two basic observations: the **coins are circular and have a dark grayish color**. Based on this, we determined that the task was to identify how many dark circles were present in each player's vault.

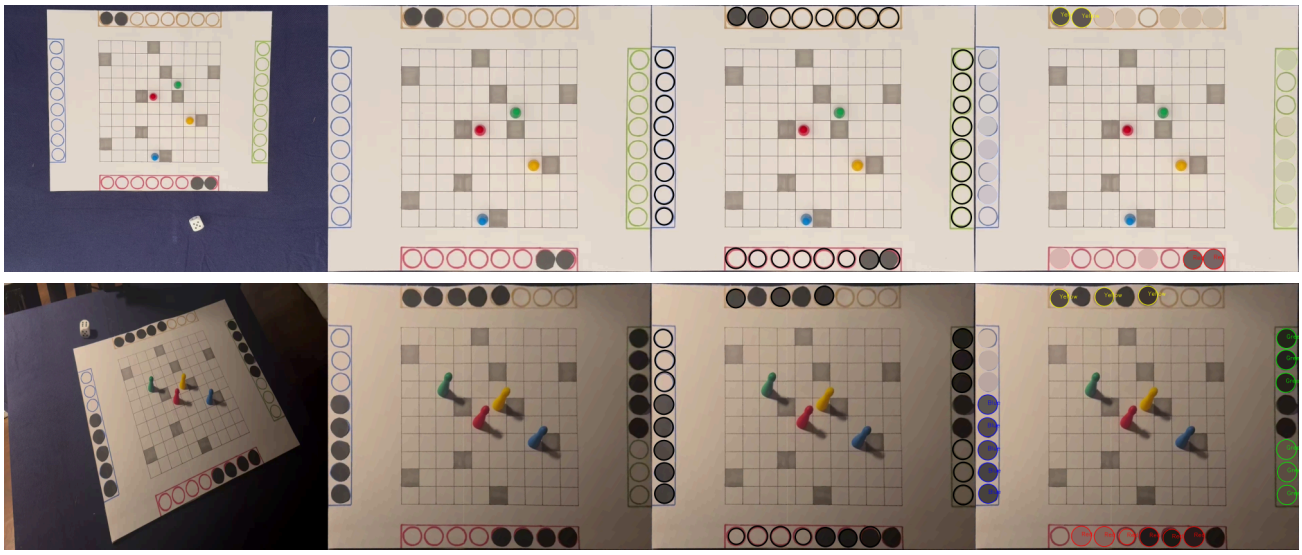
To achieve this, we first extracted the warped view of the board and converted it into a blurred grayscale image. Then, we used the **HoughCircles** function with carefully chosen parameters to detect circles on the preprocessed board.

For each detected circle, we calculated the mean color within its borders and compared it against a predefined threshold value of (150, 150, 150). Any circle with a **mean color below this threshold** was classified as a coin. Finally, we identified which player owned each coin based on its position.

The images below illustrate the **algorithm's different stages**: the input frame, the extracted board, the detected circles, and the annotated circles filled with their mean color and assigned to the respective players. Examples include one easy frame and one hard frame.

This approach performed well for easy images with ideal lighting conditions. Moreover, having an initial overhead view ensured that circles remained undistorted during warping, making their detection nearly flawless. However, problems emerged when tested on more difficult examples. In some cases, warping caused circles to become so distorted that they were no longer recognizable by the HoughCircles algorithm. More importantly, the method was prone to false positives. Relying solely on mean color for thresholding meant that some circles were incorrectly classified as coins when dark shadows were cast on them.

Ultimately, we concluded that this approach needed to be revised.



b) Second iteration

The second iteration of our coin detection solution aimed to address the issues identified in the previous approach—specifically, improving the **detection of deformed circles** and avoiding **false detection of shadows** as coins. The first issue was resolved relatively easily by adjusting the parameters of the HoughCircles function to be more tolerant of "almost circular" shapes. While this adjustment resulted in detecting more circles than necessary, a simple restriction to circles located within each player's vault effectively eliminated this as a problem.

To tackle the issue of shadows being misclassified as coins, we shifted our perspective: instead of viewing coins as "dark circles," we defined them as "high-contrast circles relative to the background." Our new approach focused on assessing the **contrast between each detected circle and its immediate neighborhood**.

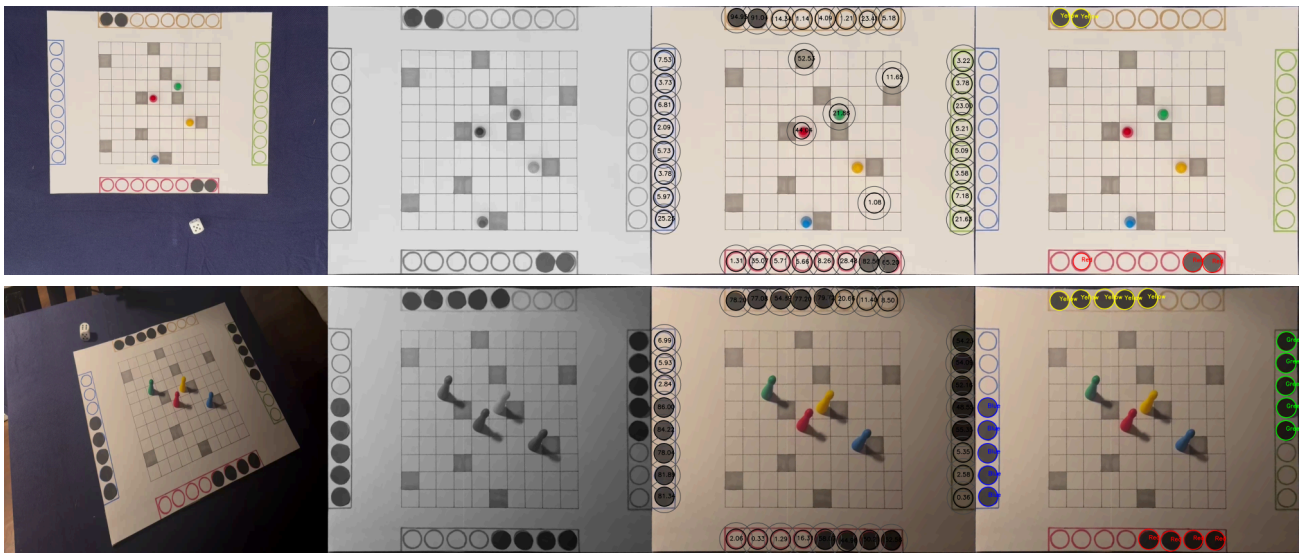
For the calculations, we utilized a **grayscale** image, as it effectively highlights the lightness or darkness of different areas. For each detected circle, we computed the mean grayscale value within its borders and the mean value for a slightly larger circle centered at the same point. The contrast value was then determined by taking the absolute difference between these two means. To classify a circle as a coin, we compared its contrast value against a threshold determined through manual analysis of false positives.

Below are the results, shown in sequence: the input frame, the blurred grayscale image of the warped board, detected circles with their immediate neighborhoods marked and annotated with contrast values, and the final circles classified as coins.

This approach performed better on hard examples than the previous one. Shadows were no longer misclassified as coins, and **all circles were correctly detected**, even when deformed. However, the method still made errors, such as misclassifying one circle in an easy example. With a threshold of 35, an empty circle that had a contrast value of 35.07 was incorrectly classified as a coin. On closer inspection, this occurred because the detected circle excluded the "border" of the coin's area, which instead appeared in the neighborhood, inflating the contrast value.

Unfortunately, in other test cases, valid coins had contrast values lower than 35 due to shadows. Simply **increasing the threshold would not solve the problem**.

While this approach provided a strong foundation for our final solution, it required further refinement. Specifically, adjustments were needed for the contrast threshold value, the size of the neighborhood, and techniques to equalize lighting conditions.



3.2 Pawn Detection

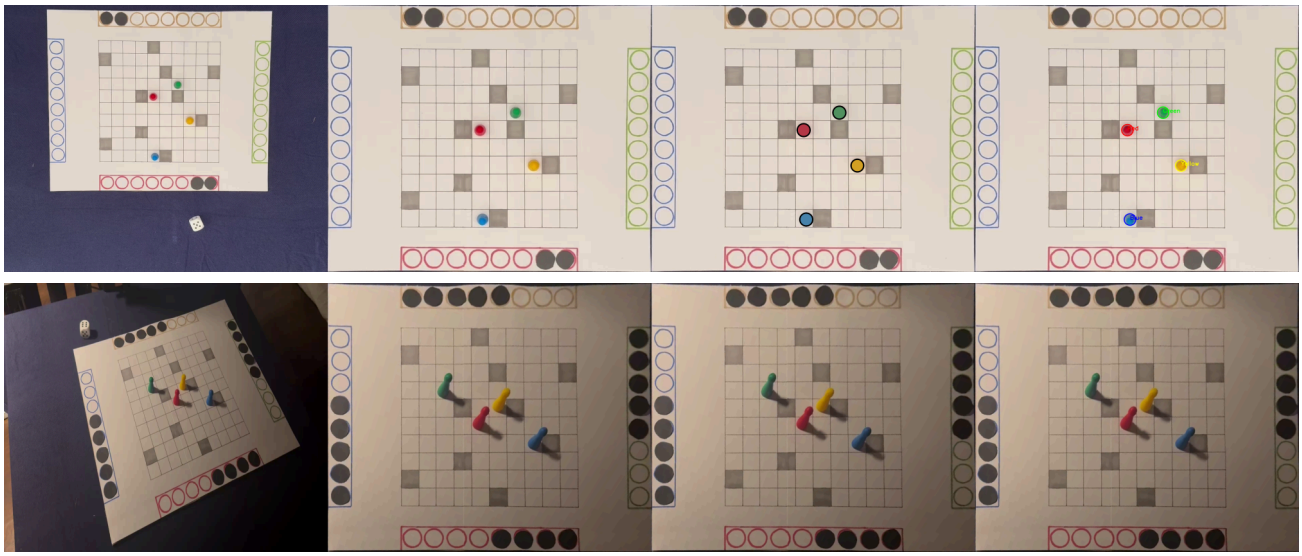
a) First iteration of tech

Our initial approach to pawn detection was **overly simplistic** in its concept. We assumed that, after unwarping the frame into a board view, each pawn could be approximated as a circle. Based on this assumption, we attempted to **detect circle**s on the board using the HoughCircles function, with the minimum and maximum radius parameters adjusted to include the expected sizes of pawns while ignoring coins. For each detected circle, we calculated the **mean RGB color** within its borders and compared it to the expected color characteristics of pawns of different colors. The positions of the pawns on the grid were determined by comparing the coordinates of each circle's center to the upper-left corner of the grid.

The results below show the different stages of the algorithm: the input frame, the extracted and unwarped board, circles detected and filled with their mean color, and the classification of the circles based on their color.

While this approach performed reasonably well for straightforward examples with an overhead view, it **failed entirely for angled** or challenging frames. The assumption that pawns are circular, while simplistic, held true for unwarped boards of frames taken from a top-down perspective. Unfortunately,

unwarping frames captured at an angle caused the pawns to appear distorted—far from circular—rendering this method ineffective for such cases.



b) Second iteration

The second iteration of our pawn detection algorithm was the last to utilize the HoughCircles function. Before abandoning the idea of approximating pawns as circles, we aimed to determine how much improvement could be achieved with this approach. Several changes were made to the initial version. First, we adjusted the circle detection parameters to allow for detecting **"almost circles"** hoping this would capture parts of angled pawns as circles. While this adjustment led to a significant increase in false positives, we mitigated some computational costs by restricting analysis to circles found within the **grid boundaries**.

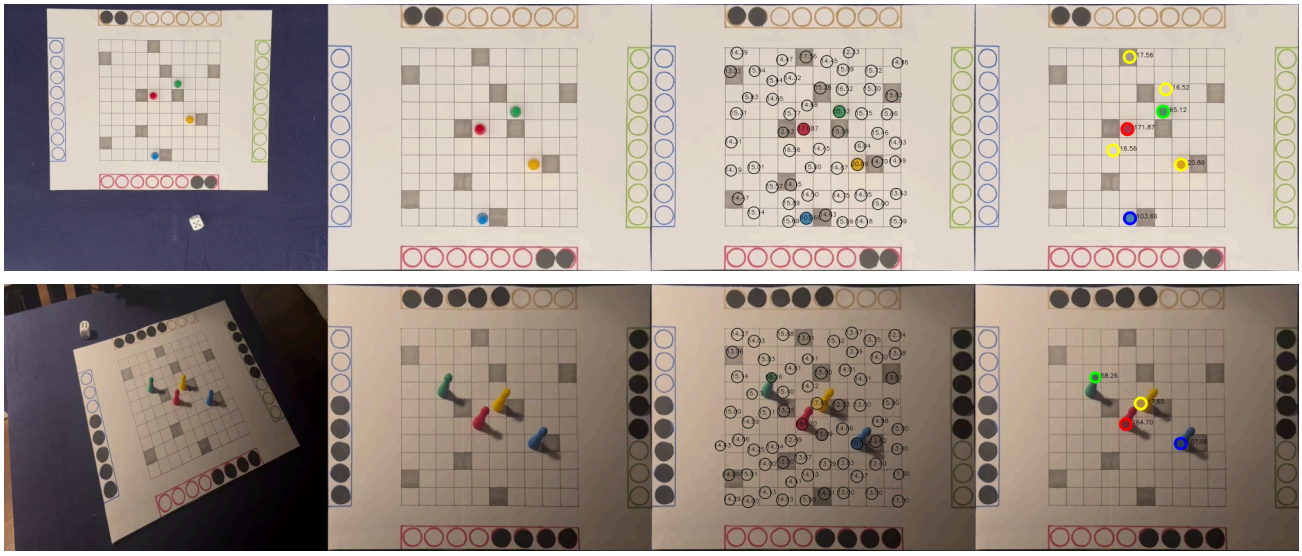
Instead of calculating the mean RGB value, we switched to computing the **mean hue**, expecting this to simplify the process of setting color thresholds for pawn classification.

The results of this iteration are shown below, with stages including the input frame, the unwarped board view, all detected circles within the board and their corresponding hue values, and finally, the circles classified as pawns.

As observed, this approach was somewhat successful for the hard example frame, though the position of a detected green circle would complicate identifying the correct location of the green pawn. Unfortunately, for easier frames, many **false positives** appeared, particularly for yellow values. What seemed like a straightforward thresholding adjustment proved more challenging upon closer examination.

After analyzing multiple frames, we found it **impossible to establish a threshold** that consistently detected all correct pawns while excluding false positives. Often, false positives exhibited similar hue values to actual pawns due to specific lighting conditions. We also explored incorporating saturation and value from the HSV color space, as well as the LAB color space, but these attempts were unsuccessful.

At this stage, we decided to pursue a more nuanced approach to overcome these limitations.



c) Third iteration

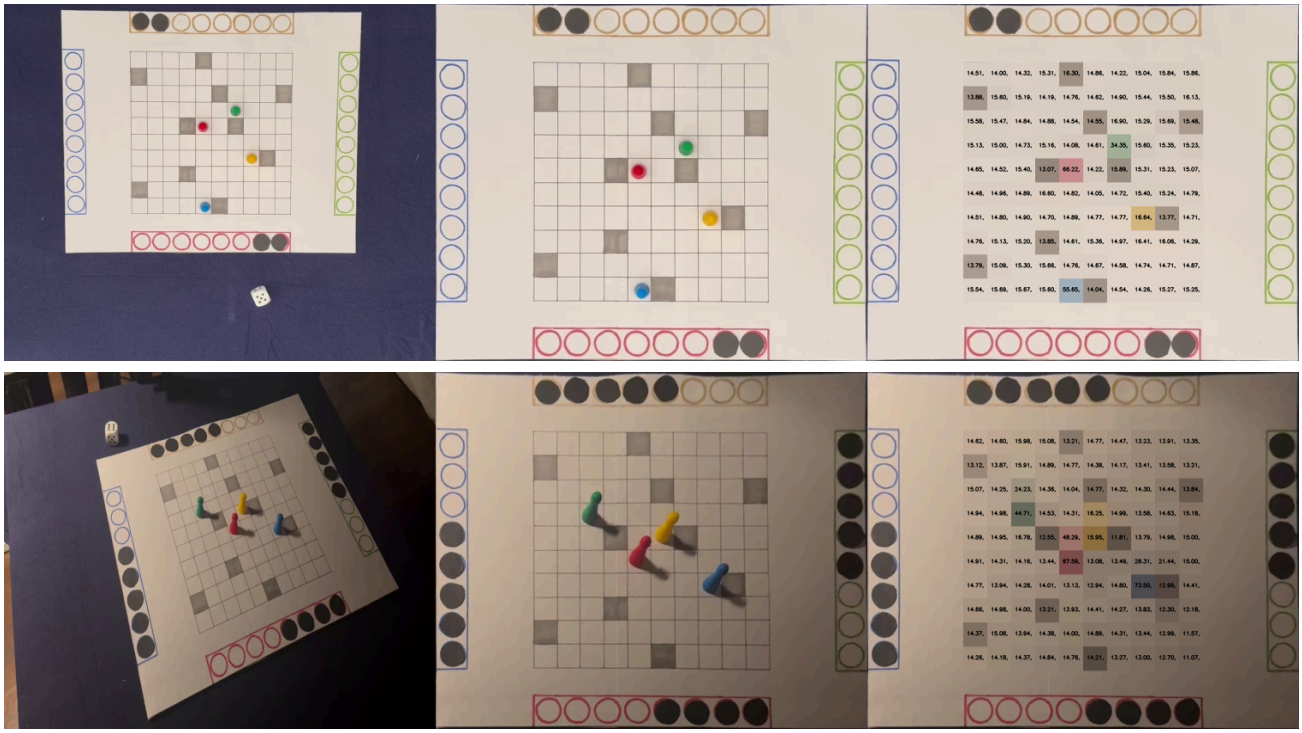
Our third attempt at this technology involved directly identifying the tile on which each pawn was located based on its color. We hypothesized that calculating the mean color value for each tile and selecting the **"most red" tile** for the red pawn, "most yellow" for the yellow pawn, and so on, might yield promising results.

The analysis of this approach is shown below. The first image depicts the frame to be analyzed, the second shows the unwarped board, and the third illustrates the mean hue values of each tile.

At first glance, this method seemed like an interesting idea. It is relatively **easy for a human** to visually identify the "most red" tile, even on an angled board. However, determining this programmatically using a distance metric proved far more challenging. Accounting for varying conditions made it **difficult to define a reliable color template** for matching tiles. We experimented with RGB, HSV, and LAB color spaces, including all three channels, but calculating a consistent mean color proved nearly impossible.

Another significant concern was how this method would behave if something obstructed the view of the board. For example, if a hand covered a tile, would it then be incorrectly classified as the "most yellow" tile? Such errors could lead to a situation where a pawn was **misidentified in the wrong location**, which was less desirable than failing to detect an obstructed pawn altogether.

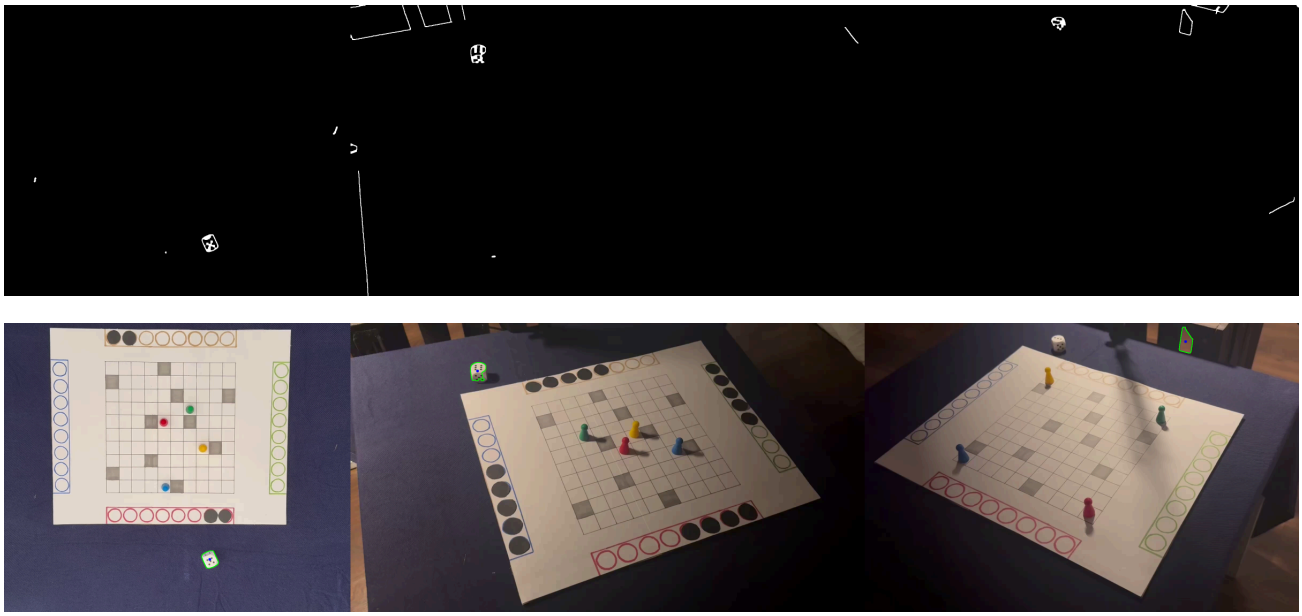
Given these challenges, we abandoned this idea at the stage of visualizing the grid with calculated color values, without proceeding to classify the pawns based on this method.



3.3 Dice Detection

This approach aimed to detect dice in video frames by isolating the board's corners and **analyzing regions outside the board**. It began by identifying the board area using **edge detection** and **contour analysis**, approximating the board's boundary as a quadrilateral. If board corners were not pre-defined, the function determined them dynamically by finding the largest rectangular contour in the frame. A mask was then applied to **exclude the board region** and focus solely on external areas, where dice might appear. Contours in the excluded regions were analyzed based on their size to filter **potential dice candidates**. Once a single contour matched the size criteria, its center was calculated using image moments, and the dice's position was returned.

While effective in some cases, this method struggled with hard difficulty videos where dice appeared at challenging angles or lighting conditions. The algorithm often misidentified other regions (e.g., reflections, edges, or noise) as dice due to the complexity of distinguishing dice contours from background elements. This inconsistency, especially in harder scenarios, led to its abandonment in favor of more robust detection strategies.



4. Used Techinques

4.1 Board Detection

In [1]:

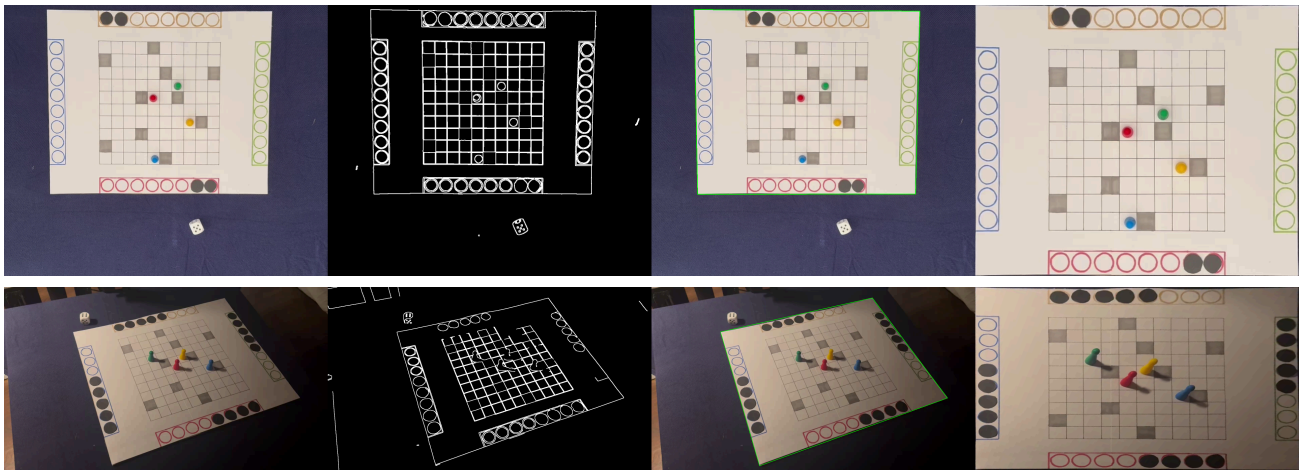
```python
def sort_corners(corners):
    corners = np.array(corners)

    sorted_by_y = sorted(corners, key=lambda x: x[1])
    top_points = sorted(sorted_by_y[:2], key=lambda x: x[0])  # Top points sorted by x
    bottom_points = sorted(sorted_by_y[2:], key=lambda x: x[0])  # Bottom points sorted by x

    top_left, top_right = top_points
    bottom_left, bottom_right = bottom_points

    return top_left, top_right, bottom_right, bottom_left


def detect_board(state):
    gray = cv2.cvtColor(state, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 0)
    edges = cv2.Canny(blur, 50, 150)

    kernel = np.ones((3, 3), np.uint8)
    edges = cv2.dilate(edges, kernel, iterations=1) #Make the edges thicker


    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    largest_rect = None
    max_area = 0

    state_copy = state.copy()
    for contour in contours:
        # Approximate the contour to a polygon
        epsilon = 0.02 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)

        # Check if the polygon has 4 points
        if len(approx) == 4:
            area = cv2.contourArea(contour)
            cv2.drawContours(state_copy, [approx], -1, (0, 255, 0), 2)
            if area > max_area: # Largest area should be the board
                largest_rect = approx
                max_area = area

    if largest_rect is not None:

        largest_rect_points = largest_rect.reshape(-1, 2)

        n= 70
        width = 36 * n
        height = 29.7 * n

        dst_points = np.array([[0, 0],
                               [width - 1, 0],
                               [width - 1, height - 1],
                               [0, height - 1]], dtype="float32")

        top_left, top_right, bottom_right, bottom_left = sort_corners(largest_rect_points)

        # Compute the perspective transformation matrix
        M = cv2.getPerspectiveTransform(np.array([top_left,
                                                  top_right,
                                                  bottom_right,
                                                  bottom_left], dtype="float32"),
```

```
                                    dst_points)

        return M, width, height

    return None, None, None
```

To simplify calculations and detections, we needed to ensure a **consistent view of the board**. This required an approach to detect the board within a frame and warp the image so that the board appeared as if viewed from above, even when the original frame was taken at an angle.

We began by converting the frame to a blurred grayscale image, which was then processed using the Canny edge detection algorithm. The resulting edges were dilated to make them thicker and more easily identifiable. Next, we performed **contour detection**, focusing only on polygons with four points, as these could indicate rectangular shapes like the board. We assumed that the **largest contour** fitting these criteria corresponded to the board.

Once the board's location was identified in the original frame, the next step was to calculate the **perspective transformation matrix**. To achieve accurate results, we analyzed the real-life measurements of the board and applied a scaling factor *n*. Using the destination points derived from these measurements, we mapped each detected corner of the board to its corresponding transformed point, obtaining the perspective transformation matrix.



4.2 Coin Detection

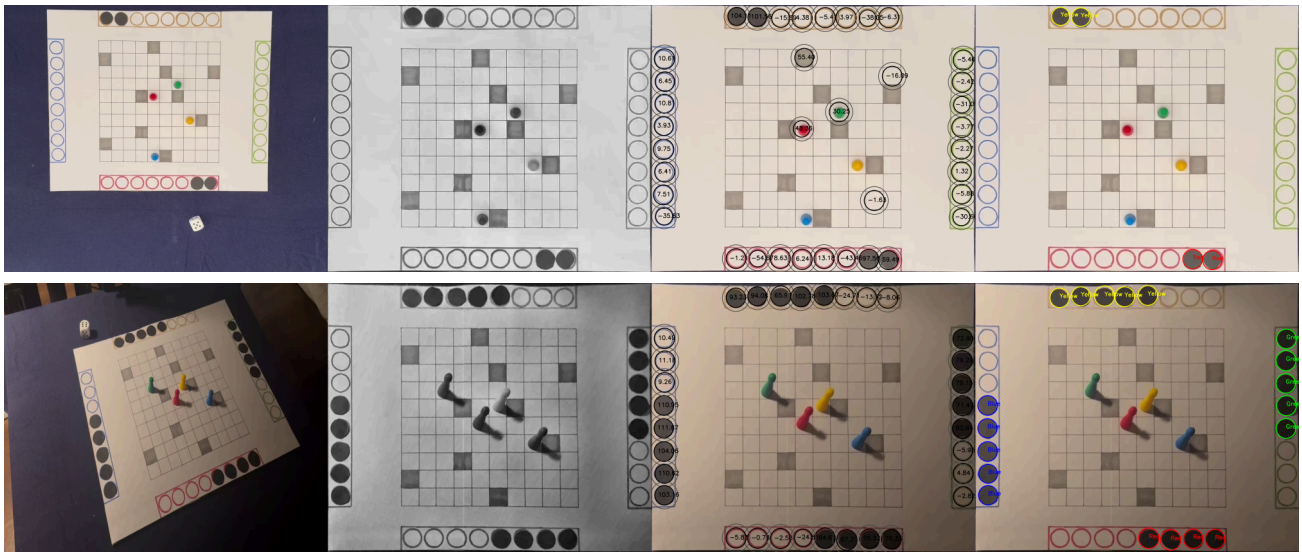
In [2]:
```
def get_score(board):
    gray = cv2.cvtColor(board, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 0)

    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    clahe_img = clahe.apply(gray) #Obtain grayscale image with equalized histogram

    margin = 2*2*70

    circles = cv2.HoughCircles(blur,
                               cv2.HOUGH_GRADIENT,
                               dp=1,
                               minDist=50,
                               param1=8,
                               param2=25,
                               minRadius=62,
                               maxRadius=80)

    green_points = 0
    red_points = 0
    yellow_points = 0
```

```
    blue_points = 0

    green_circles=[]
    red_circles=[]
    yellow_circles=[]
    blue_circles=[]

    board_copy = board.copy()

    if circles is not None:
        circles = np.round(circles[0, :]).astype("int")
        for (x, y, r) in circles:
            mask = np.zeros_like(clahe_img, dtype=np.uint8)
            cv2.circle(mask, (x, y), r, 255, -1)
            masked = cv2.bitwise_and(board, board, mask=mask)

            #Calculate the mean brightness of the circle
            circle_brightness = cv2.mean(clahe_img, mask=mask)[0]

            outer_radius = r + 30
            large_mask = np.zeros_like(clahe_img, dtype=np.uint8)
            cv2.circle(large_mask, (x, y), outer_radius, 255, -1)

            #Calculate the mean brightness of the neighborhood
            surrounding_mask = cv2.subtract(large_mask, mask)
            background_brightness = cv2.mean(clahe_img, mask=surrounding_mask)[0]

            #Calculate the contrast between the circle and the background
            contrast = background_brightness - circle_brightness

            if contrast > 57: #If the contrast between the circle and the background is high
                if x < margin:
                    blue_points += 1
                    blue_circles.append((x, y, r))

                elif x >= board.shape[1] - margin:
                    green_points += 1
                    green_circles.append((x, y, r))

                elif y < margin:
                    yellow_points += 1
                    yellow_circles.append((x, y, r))

                elif y >= board.shape[0] - margin:
                    red_points += 1
                    red_circles.append((x, y, r))

    return (green_points, green_circles), (red_points, red_circles), (yellow_points, yellow_c
```

The final version of our coin detection system builds logically upon the previous attempts. By addressing the identified issues, we developed a simple yet effective solution. We used **CLAHE** to equalize lighting conditions, reduced the size of the analyzed neighborhood, and **increased the contrast threshold** to improve accuracy.

In the final pipeline, we began by extracting the board from the input frame and calculating both its blurred grayscale image for circle detection and a CLAHE-enhanced grayscale image for contrast calculations. For each detected circle, we compared its mean lightness to the mean grayscale value of a slightly larger circle with a radius increased by 30 pixels. We also removed the use of the absolute difference to focus exclusively on detecting dark spots on a light background, avoiding the reverse scenario.

Each contrast value was then compared to a threshold of 57, with circles exceeding this threshold classified as coins. Finally, we determined the ownership of each coin based on its location on the board.



4.3 Pawns detection

In [3]:
```python
def calculate_centroid(contour):
    M = cv2.moments(contour)
    if M["m00"] != 0:
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
        return (cX, cY)
    return None


def calculate_base(contour):
    lowest_point = max(contour, key=lambda x: x[0][1])
    lowest_point[0][1]-=60
    return tuple(lowest_point[0])


def get_position(board):
    corners = board_corners(board)

    centroid = np.mean(corners, axis=0)
    scale_factor = 1

    expanded_corners = (corners - centroid) * scale_factor + centroid

    mask = np.zeros_like(board, dtype=np.uint8)
    roi_corners = np.array(expanded_corners, dtype=np.int32)
    cv2.fillPoly(mask, [roi_corners], (255, 255, 255))

    masked = cv2.bitwise_and(board, mask)

    hsv = cv2.cvtColor(masked, cv2.COLOR_BGR2HSV)

    color_ranges = {
    "red": [(150, 100, 100), (178.5, 255, 255)],
    "yellow": [(15, 138, 128), (30, 238, 255)],
    "green": [(30, 25, 63), (80, 200, 200)],
    "blue": [(80, 60, 60), (150, 255, 255)],
    }

    masks = {}
    for color, (lower, upper) in color_ranges.items(): #Get a mask for each color
        lower = np.array(lower, dtype="uint8")
```

```
    upper = np.array(upper, dtype="uint8")
    masks[color] = cv2.inRange(hsv, lower, upper)

combined_mask = np.zeros_like(masks["red"])

for mask in masks.values():
    combined_mask = cv2.bitwise_or(combined_mask, mask) #Combine all masks

combined_mask = cv2.dilate(combined_mask, np.ones((3, 3), np.uint8), iterations=3)

segmented = cv2.bitwise_and(masked, masked, mask=combined_mask)

edges = cv2.Canny(combined_mask, 50, 150) #Detect edges

contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
convex_hulls = [cv2.convexHull(cnt) for cnt in contours]
#Filter out small contours
filtered_hulls = [cnt for cnt in convex_hulls if cv2.contourArea(cnt) > 5000]

centroids = [calculate_centroid(cnt) for cnt in filtered_hulls]
threshold = 100
merged_countours = []
for i, cnt in enumerate(filtered_hulls): #Merge contours that are close to each other
    centroid = centroids[i]
    if centroid is None:
        continue
    merged = False
    for merged_cnt in merged_countours:
        if (calculate_centroid(merged_cnt) and
            math.dist(centroid, calculate_centroid(merged_cnt)) < threshold):

            merged_cnt = np.concatenate([merged_cnt, cnt], axis=0)
            merged = True
            break

    if not merged:
        merged_countours.append(cnt)

filtered_hulls = merged_countours

board_copy = board.copy()
cv2.drawContours(board_copy, filtered_hulls, -1, (0, 255, 0), 2)
for cnt in filtered_hulls:
    cv2.putText(board_copy,
                f"{cv2.contourArea(cnt):.2f}",
                calculate_base(cnt),
                cv2.FONT_HERSHEY_SIMPLEX,
                1,
                (0, 0, 0),
                2)

green_position = None
red_position = None
yellow_position = None
blue_position = None

for cnt in filtered_hulls: #For each contour, calculate the base and the mean color
    if cnt is not None:
        base = calculate_base(cnt)
        cv2.circle(board_copy, base, 10, (0, 0, 255), -1)
        mean_colors = []

        for color in color_ranges.keys():
            color_mask = masks[color]
            mask = np.zeros(color_mask.shape[:2], dtype=np.uint8)
            cv2.drawContours(mask, [cnt], -1, 255, -1)
```

```
            mean_color = cv2.mean(color_mask, mask=mask)
            mean_colors.append(mean_color[0])

        #Get the color with the highest mean value of overlap
        max_indx = np.argmax(mean_colors)
        if max_indx == 0:
            red_position = base
        elif max_indx == 1:
            yellow_position = base
        elif max_indx == 2:
            green_position = base
        elif max_indx == 3:
            blue_position = base

return green_position, red_position, yellow_position, blue_position
```

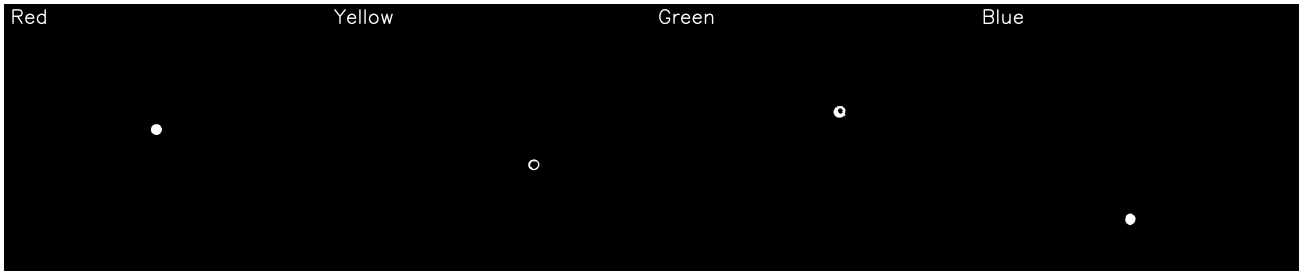
The final version of our pawn detection system takes a drastic departure from previous experiments, as none of them yielded satisfactory results. Our key realization was that the **pawns have significantly different colors** compared to the board, making color the primary feature to focus on rather than shape.

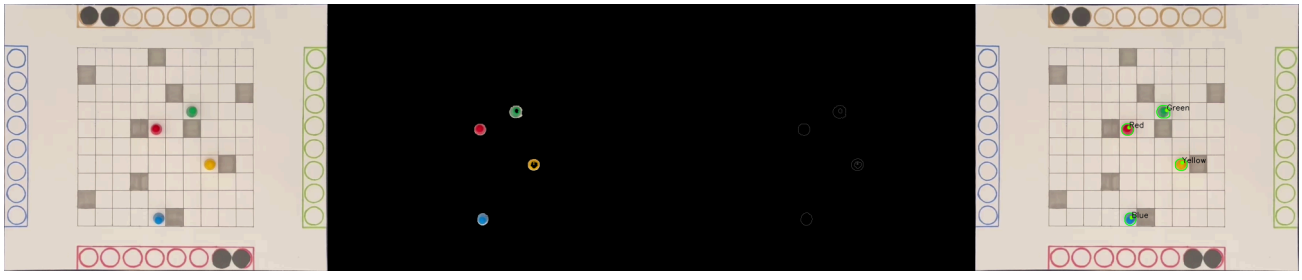
We hypothesized that defining specific HSV color ranges could allow us to **isolate the pawns**. After setting initial HSV boundaries, we used each range to create binary masks that captured only the areas of the image within the thresholds for each color. The results of this step are shown below, with four binary masks for each pawn color highlighting the regions of the board that match the desired ranges.

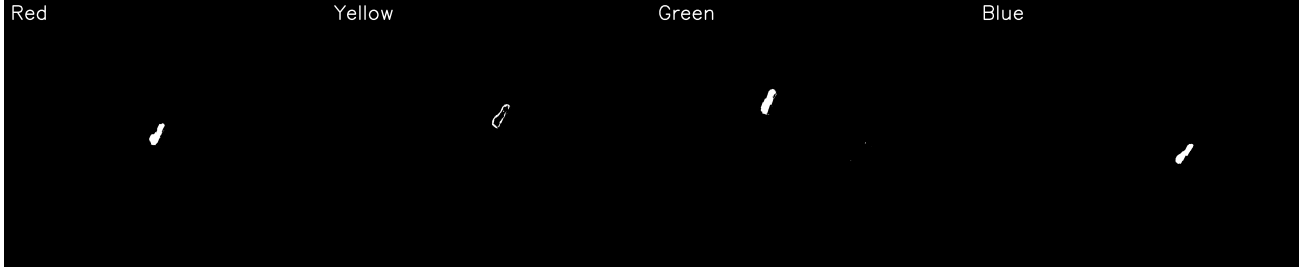
Once the masks were generated, we combined them into a single image and applied the **Canny edge detection** algorithm to identify edges. From these edges, we calculated contours, aiming to **outline each pawn**. To reduce noise, we restricted the area of detected contours, excluding small artifacts or excessively large shadows.

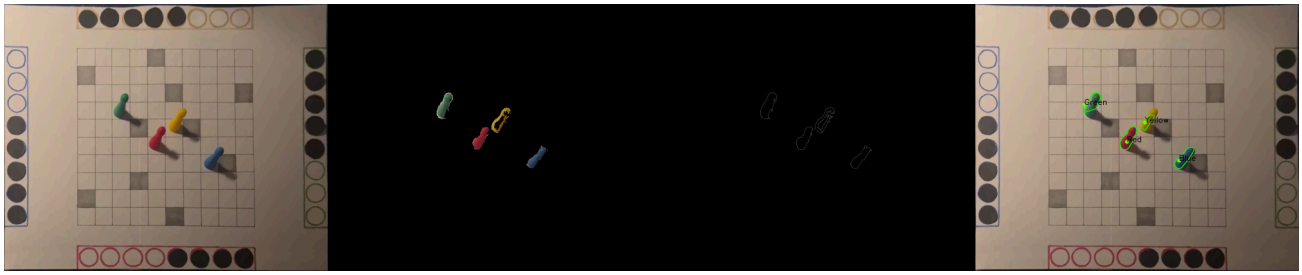
Through experimentation, we observed that a single pawn was sometimes split into multiple contours due to holes in the masks. To address this, we introduced a merging step to **connect nearby contours into a single unified shape**. To determine the color of each detected contour, we calculated the overlap between the contour and the masks for each color, assigning the color with the **highest overlap** to the pawn. This approach minimized the risk of false classification caused by small patches of noise in the image.

Finally, the position of each pawn was calculated by identifying the **lowest point** of the detected contour and slightly adjusting it upward to account for minor inaccuracies in the contours.





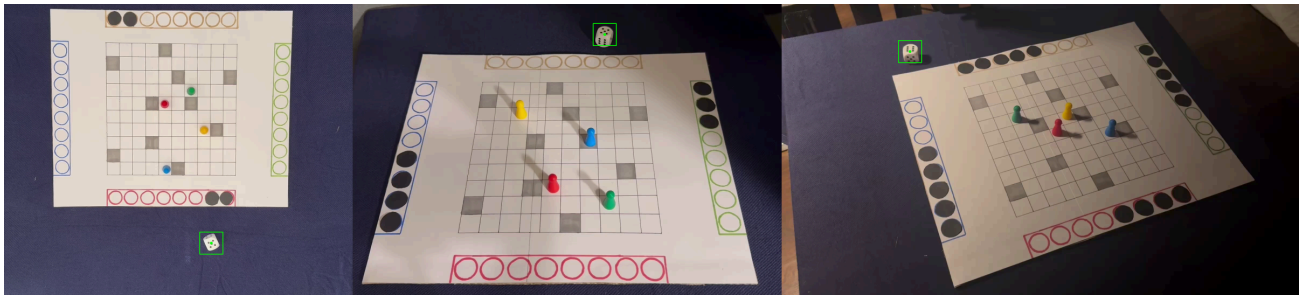




4.4 Dice Detection

In [4]:
```python
def detect_dice(frame, template):
    # template is provided as a read grayscale image
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    res = cv2.matchTemplate(frame_gray, template, cv2.TM_CCOEFF_NORMED)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

    # calculate the center of the dice and the bounding box
    top_left = max_loc
    bottom_right = (top_left[0] + template.shape[1],
                    top_left[1] + template.shape[0])

    dice_center = (top_left[0] + template.shape[1] // 2,
                   top_left[1] + template.shape[0] // 2)

    # return the center of the dice and the bounding box coords
    return dice_center, top_left, bottom_right
```

The `detect_dice` function identifies the location of a dice in a given frame using template matching. It converts the input frame to grayscale and uses the `cv2.matchTemplate` method with a provided grayscale dice template to find the best match within the frame. The function calculates the coordinates of the top-left and bottom-right corners of the bounding box surrounding the detected dice, as well as its center point. These coordinates are determined based on the highest matching score obtained from the template matching operation. Finally, it returns the dice's center position, along with the bounding box's top-left and bottom-right coordinates, enabling precise localization of the dice within the image.



4.5 Counting Die's Pips

In [5]:
```python
def count_dice_pips(frame, dice_center):

    # crop the image to get the dice top region
    top_left = (dice_center[0] - 25, dice_center[1] - 25)
    bottom_right = (dice_center[0] + 25, dice_center[1] + 25)
    crop_top_left = (max(0, top_left[0]), max(0, top_left[1]))
    crop_bottom_right = (min(frame.shape[1], bottom_right[0]),
                         min(frame.shape[0], bottom_right[1]))

    # extract the cropped region of the dice
    only_dice = frame[crop_top_left[1]:crop_bottom_right[1],
                      crop_top_left[0]:crop_bottom_right[0]]

    only_dice = cv2.resize(only_dice, (100, 100))
    only_dice = cv2.GaussianBlur(only_dice, (3,3), 0)
    only_dice_gray = cv2.cvtColor(only_dice, cv2.COLOR_BGR2GRAY)

    # hough circle transform to detect the dice pips
    circles = cv2.HoughCircles(only_dice_gray,
                               cv2.HOUGH_GRADIENT,
                               dp=1,
                               minDist=10,
                               param1=50,
                               param2=12,
                               minRadius=3,
                               maxRadius=10)
    if circles is not None:
        pips = len(circles[0])
        if pips > 6:
            pips = 6
    else:
        pips = 0

    return pips
```

The `count_dice_pips` function determines the number of pips (dots) on a dice in a given frame, based on its center coordinates. It first defines a **square region around the dice**, ensuring the crop is within the image boundaries, and extracts this region for further processing. The cropped region is **resized**, **blurred**, and **converted to grayscale** to enhance the visibility of the pips. Using the **Hough Circle Transform**, the function detects circular features in the grayscale image, which correspond to the dice pips. If circles are detected, their count is capped at six (since standard dice have up to six pips); otherwise, the pip count is set to zero. Finally, the function returns the pip count as an integer, representing the dice's face value.

As shown in the examples below, the function performs exceptionally well for recordings with easy and medium difficulty levels. However, it encounters challenges with hard difficulty, where the die is displayed at an angle. This perspective makes it difficult to distinguish the top face of the die from its sides. Despite exploring various approaches to address this issue, it proved challenging to develop a method that effectively isolates the top face without compromising the detection accuracy for the other recordings.

4.6 Dice Tracking

To track the dice in the recording, we began by detecting the dice using the previously mentioned dice detection technique. The center point of the detected dice was selected as the tracking point, and we applied the **Optical Flow** technique to track this point across frames. If a suitable matching point was detected in the subsequent frame, we updated the tracking point accordingly. If not, we redetected the dice to maintain accurate tracking.

To account for camera shake, we distinguished between significant and minor movements of the dice. If the movement was minor, we attributed it to camera shake and retained the previous tracking point. However, when the movement was significant, we displayed a message indicating that the dice were being rolled. Additionally, we implemented periodic dice re-detection to ensure accurate tracking and prevent the tracker from drifting off the dice. When a die changed its position, we recalculated the number of pips to reflect the updated state.

5. Effectiveness for each dataset

We successfully implemented detection for several key game elements, including:

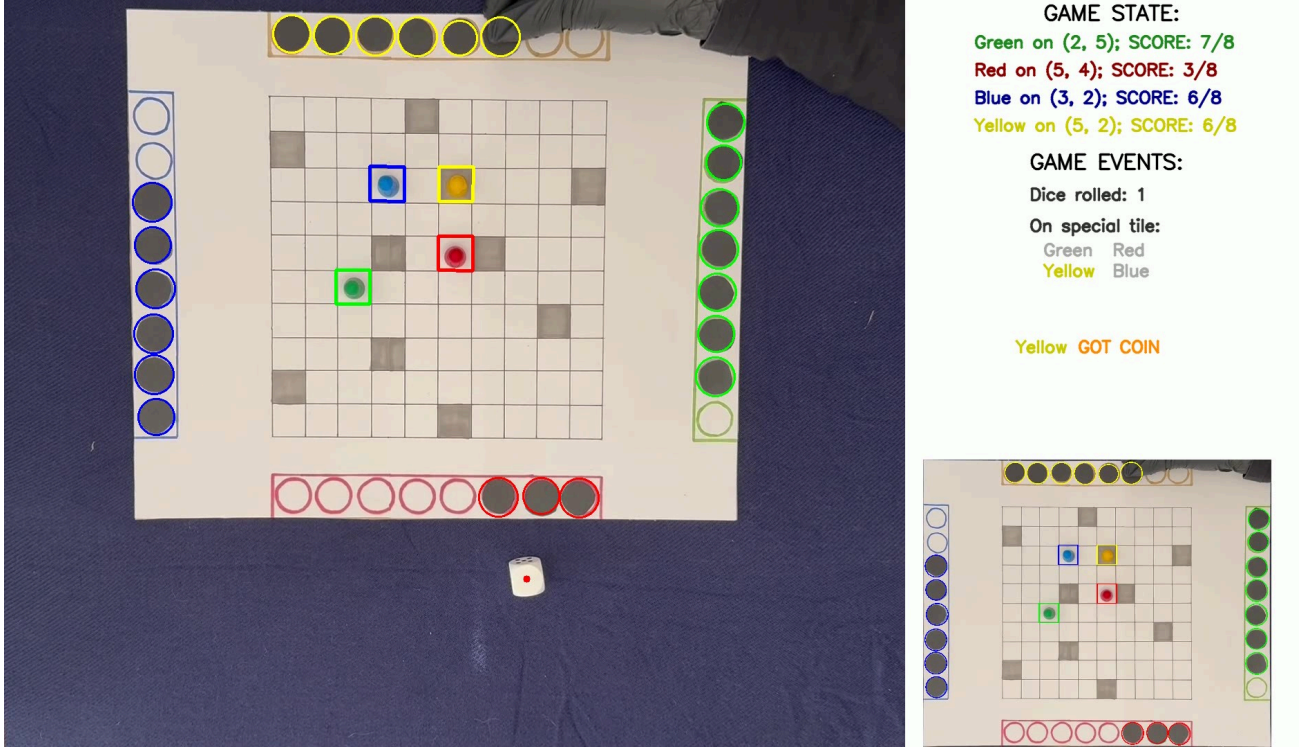
- **Board Detection**: Identifying the game board and extracting it to track other elements.
- **Coin Detection**: Detecting the coins in players' vaults and accurately counting them.
- **Pawn Detection**: Identifying pawns on the grid and distinguishing them based on their colors.
- **Dice Detection**: Locating the dice in the recording using a template matching technique.
- **Dice Pips Detection**: Determining the number of pips rolled on the dice.

Additionally, we achieved accurate identification and tracking of game events, such as:

- **Players' Current Possessions**: Counting the coins in players' vaults and displaying their scores in real-time.
- **Score Updates**: Displaying an on-screen message whenever a player gains a coin.
- **Player's Position**: Tracking the position of each pawn on the grid and displaying the corresponding grid tile coordinates continuously.
- **Rolling the Dice**: Detecting significant motion of the dice and displaying a message indicating the dice are being rolled.
- **Game Start**: Recognizing the starting configuration of the game and displaying a "Game starts now" message.
- **End of the Game (Winner Announcement)**: Identifying when a player collects 8 coins to fill their vault and displaying a message indicating the winner.
- **Special Tile**: Recognizing when a player lands on a special tile and indicating which player is currently placed on that tile.

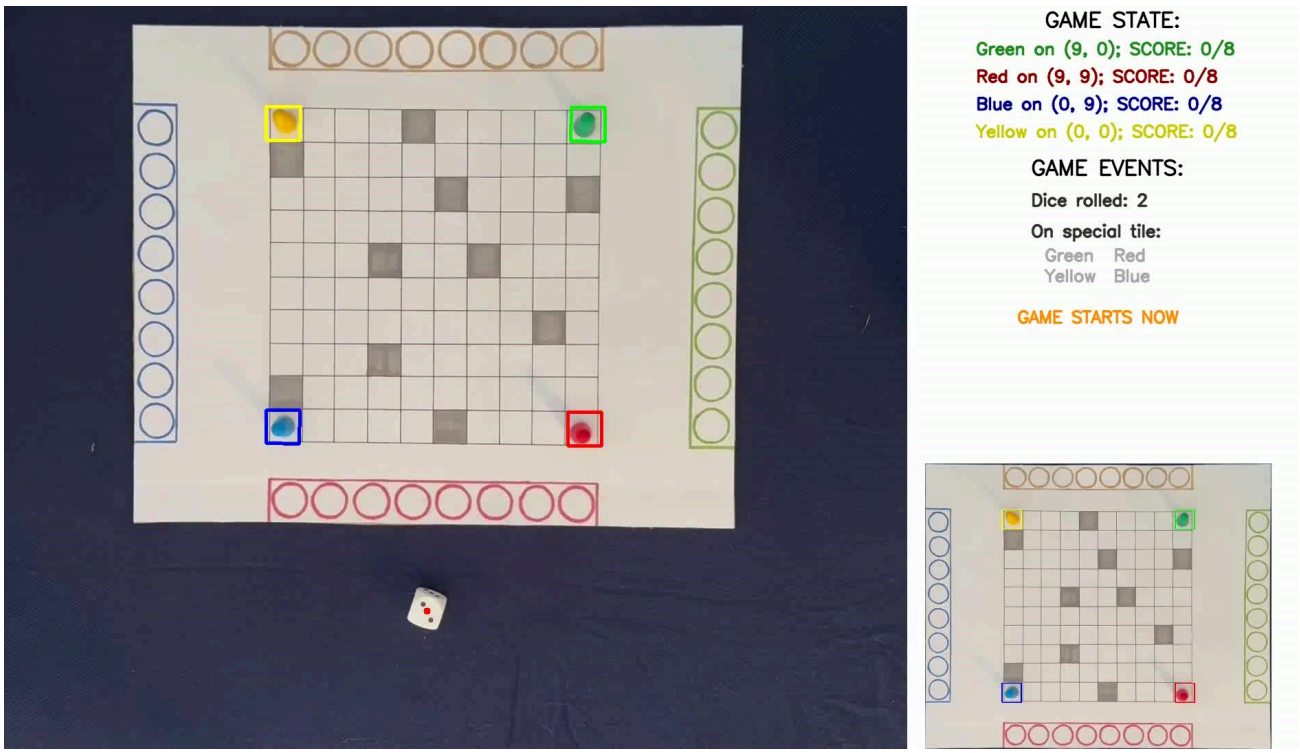
Finally, we implemented a method for **tracking the dice** throughout the recordings.

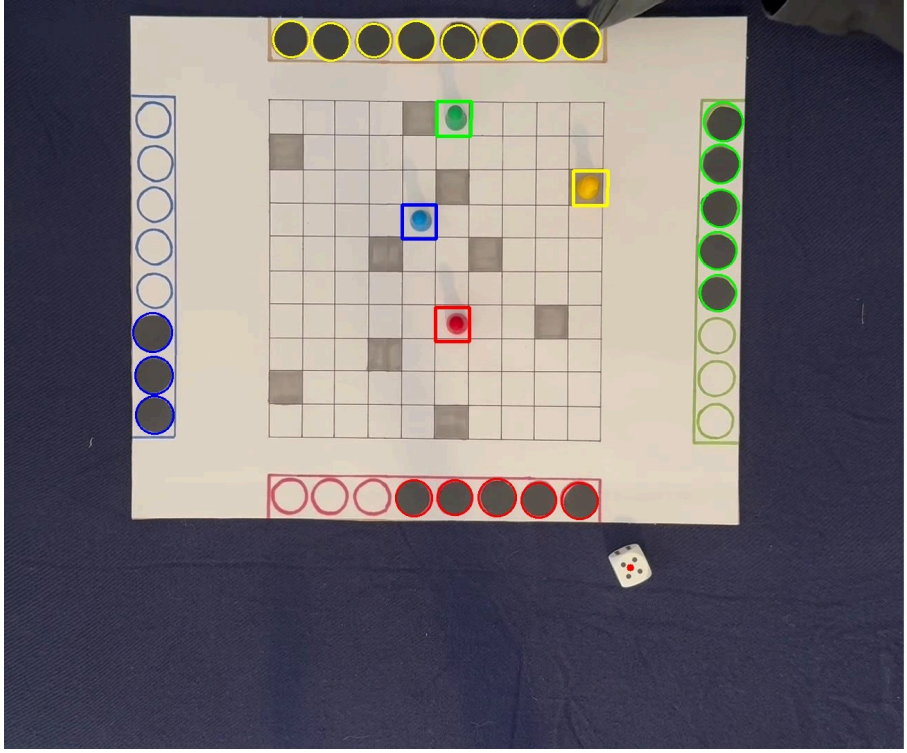
5.1 Effectiveness for Dataset: EASY

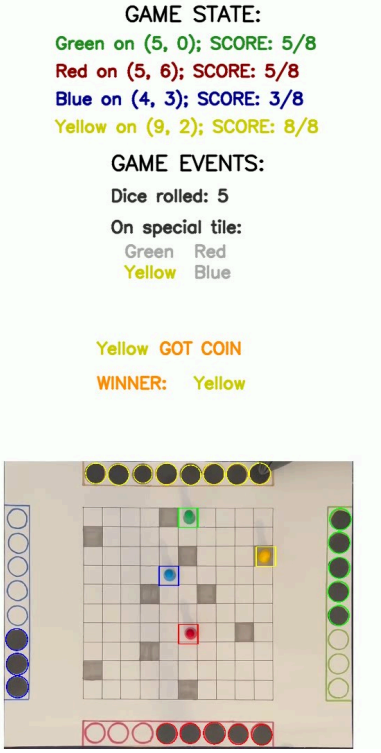


Our approach was highly effective for recordings of easy difficulty. Occasionally, when certain game pieces were covered by the player's arm, temporary inaccuracies occurred in the game status. However, these quickly corrected themselves as soon as the board was uncovered. The information panel on the right side displayed all game-related information, including an extracted view of the board, providing a clear and comprehensive overview of the game's state.

5.2 Effectiveness for Dataset: MEDIUM

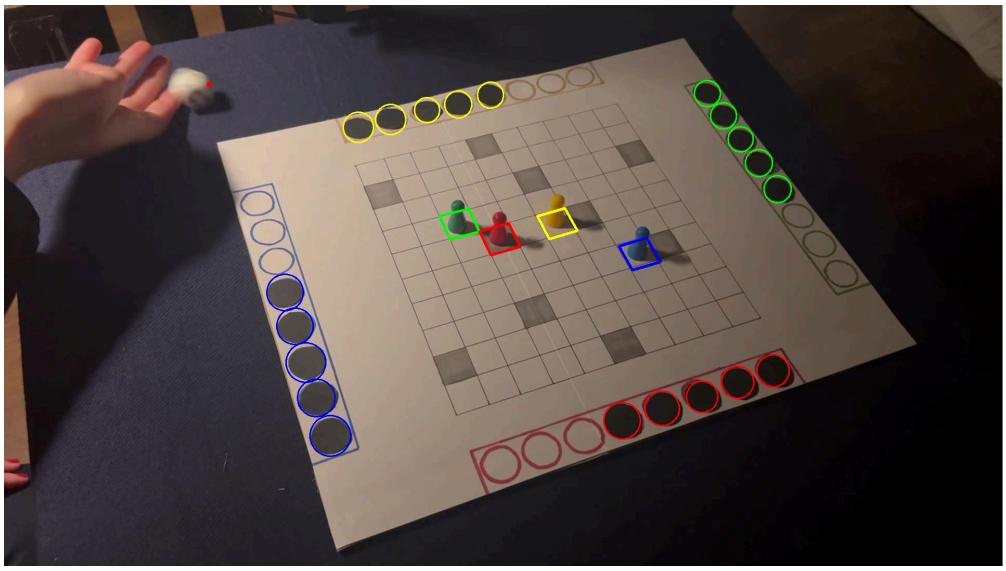


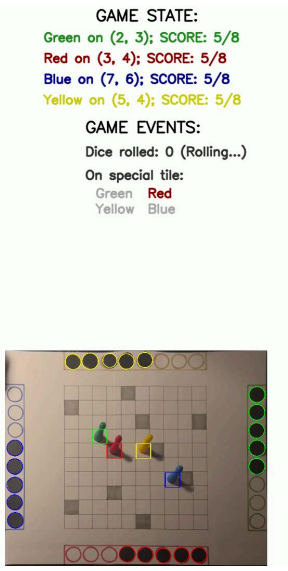




The approach performed similarly for medium difficulty recordings. While the varying lighting conditions and occasional obstructions introduced some challenges, the overall system maintained accurate detection and tracking, handling the dynamic scenarios effectively.

5.3 Effectiveness for Dataset: HARD





For hard difficulty recordings, the results were slightly less reliable. Camera movement occasionally caused the dice tracking to falsely detect rolling when the dice was stationary. Additionally, dice pip counting often produced incorrect results due to the prominent visibility of the dice's sides when viewed at an angle. Frequent occlusions by the player's hand led to intermittent inconsistencies in tracking the board and game pieces, making these scenarios more challenging for the system.

6. Analysis and Conclusion

Throughout this project, we explored a variety of computer vision techniques to analyze and track elements of a board game. Our approach successfully detected many game elements and displayed gameplay status effectively. By working with a diverse dataset that ranged in difficulty from easy to hard, we developed solutions that performed well in many scenarios. However, the challenges presented by

the dataset—such as adapting to diverse recording conditions, including varying lighting, camera angles, and movement—highlighted areas for improvement.

Specific difficulties included inefficiencies in counting dice pips when the dice were viewed at steep angles. Hand obstructions and camera movement caused temporary inaccuracies in tracking game components, which also impacted overall performance. Similarly, our board detection struggled when obstacles such as an arm obscured the board. To address this, we relied on previously detected board corners and used them consistently throughout the recording, assuming that the board position would not change significantly. These limitations represent opportunities for further refinement and enhancement of our methods.

Despite these challenges, we achieved several significant successes. For example, when recordings captured the board at an angle, we used perspective transformations to ensure that the board and game components remained recognizable. Iterative refinements to algorithms, such as using contrast-based methods for coin detection, minimized false positives caused by shadows or reflections. Furthermore, implementing a re-detection mechanism for dice tracking allowed us to recover from temporary loss of tracking due to obstructions or sudden movements.

Our success was rooted in combining multiple computer vision techniques—edge detection, contour analysis, color space analysis, template matching, and others. This integration of diverse methods was critical in achieving robust performance across various conditions. While there is room for improvement, particularly in handling difficult scenarios, the project demonstrated the potential of combining computer vision methods to analyze complex, dynamic recordings effectively.