Computer Vision & Pattern Recognition Course Project

Prof. Kai Hormann
Università della Svizzera italiana

TA Nikki Zimmerman

Students *Fabian Gobet Yassine Oueslati Henrique Gil*

Tasks

1. **Pre-process Video:**
    - Filter the video to retain only the frames showing the side view.
    - Use correlation or distance metrics to compare frames and identify the desired frames.
    - Determine a threshold to separate wanted from unwanted frames.
2. **Reconstruct Camera Position:**
    - Obtain the dimensions of the table and the positions of the balls from references like Wikipedia.
    - Compute the coordinates of the ball spots (yellow, green, brown, blue, pink, black).
    - Measure and verify the playing area dimensions in the image.
    - Compute the coordinates of the playing area or green area corners using Sobel gradients and Hough transform to find lines between the baize (green) and wood (brown).
3. **Reconstruct Ball Positions:**
    - Identify a frame where all balls are on their spots.
    - Use Sobel gradients and Hough transform to find the corners of the green area.
    - Use the positions of the ball markers to compute the coordinates of the ball centers.
    - Ensure the image points of the ball markers have the same cross ratio as the corresponding world points.

Steps

Pre-process Video

- Take a reference frame (F) that shows the side view.
- Compute correlation or distance metrics between (F) and other frames to identify similar frames.

- Determine a threshold to filter out unwanted frames.

Reconstruct Camera

- Measure the table dimensions and ball positions.
- Compute the coordinates of the playing area or green area corners.
- Use the DLT algorithm to find the camera matrix (P).
- Decompose (P) to obtain the calibration matrix (K), rotation matrix (R), and camera center (C).
- Optionally, use non-linear optimization to minimize geometric error and refine the camera parameters.

Reconstruct Ball Positions

- Utilize the reflections on the balls caused by overhead lights to estimate ball positions.
- Given the image coordinates of the highlight, compute the ball positions.
- Detect the highlights in the image by finding the centers of the white reflections within the red regions of the balls.
- Use correlation with a prototypical image of a red ball to find the average position of the white pixels.

Implementation

1. **Color Space Transformations:**
    - Convert images between BGR and HSI color spaces.
    - Generate grayscale images from BGR images with optional custom weights.
2. **Histogram Analysis:**
    - Calculate the frequency distribution of color intensities for BGR and HSI color channels.
    - Plot the frequency distributions for visual analysis.
3. **Edge Detection:**
    - Apply the Canny edge detection algorithm to identify edges in images.
    - Use Sobel operators and non-maximum suppression for edge thinning.
4. **Camera Calibration:**
    - Perform camera calibration using Direct Linear Transform (DLT) to obtain the projection matrix, camera center, calibration matrix, rotation matrix, and translation vector.
    - Recover 3D coordinates from 2D projections.
5. **3D Point Calculation:**
    - Calculate intersection points and quotas to map 2D image points to 3D space.
    - Correct 3D coordinates based on light source and camera position.

6. **Pixel Analysis:**
    - Identify specific pixels in an image based on hue values and defined regions of interest.

In [ ]:
```
import cv2
import matplotlib.pyplot as plt
import numpy as np
from tqdm.notebook import tqdm
import math
from scipy.ndimage import convolve
from scipy.linalg import null_space
import pickle
```

Functions

- **get_bgr_frequencies(image)**: Calculates the frequency of each color intensity level (0-255) for the blue, green, and red channels in a given image.
- **get_hsi_frequencies(image)**: Calculates the frequency of each hue (0-360), saturation (0-1), and intensity (0-1) level in a given image in HSI color space.
- **plot_rgb_freq(freqs, save=False, save_path_name=None)**: Plots the frequency distributions for the red, green, and blue channels using data from `get_bgr_frequencies()`. Optionally saves the plot to a file.
- **plot_hsi_freq(freqs, save=False, save_path_name=None)**: Plots the frequency distributions for hue, saturation, and intensity using data from `get_hsi_frequencies()`. Optionally saves the plot to a file.
- **histo_mse(hist1, histo2)**: Calculates the mean squared error between two histograms of BGR frequencies, providing a measure of similarity between two images.
- **bgr_to_hsi(bgr_image)**: Converts an image from BGR color space to HSI color space.
- **hsi_to_bgr(hsi_image)**: Converts an image from HSI color space to BGR color space.
- **show_bgr_image(image, grayscale=False)**: Displays a BGR image using Matplotlib, with an option to display in grayscale.
- **get_gray_scale(bgr_image, custom_weights=None)**: Converts a BGR image to grayscale, with optional custom weights for the blue, green, and red channels.
- **is_frame_close(true_frame, test_frame, avg_threshold=1)**: Determines if the mean absolute difference between two frames is below a given threshold, returning a boolean and the mean difference.

- **apply_gaussian_filter(img, sigma=1, a=1)**: Applies a Gaussian filter to a grayscale image to reduce noise and smooth the image.
- **threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.09)**: Applies thresholding to an image to create a binary image, identifying weak and strong edges.
- **hysteresis(img, weak, strong=255)**: Applies hysteresis to a binary image to identify and retain strong edges.
- **sobel_operate(image)**: Applies the Sobel operator to an image to detect edges, returning the gradient magnitude and direction.
- **non_max_suppression(G, theta)**: Applies non-maximum suppression to thin edges in an image, using gradient magnitude and direction.
- **canny_edge_detection(img, sigma=1, a=1, lowThresholdRatio=0.05, highThresholdRatio=0.09)**: Implements the Canny edge detection algorithm, combining grayscale conversion, Gaussian filtering, Sobel operator, non-maximum suppression, thresholding, and hysteresis.
- **hough_line_space(edge_image)**: Performs Hough Line Transform to detect lines in an edge-detected image, returning the Hough space, theta map, and rho map.
- **binarized_thresholding(gray_image, threshold)**: Applies binarized thresholding to a grayscale image, creating a binary image based on a threshold value.
- **plot_hough_lines(image, lines, theta_map, rho_map, save_path=None, indices=None)**: Plots detected Hough lines on an image and optionally saves the image to a file.
- **Normalization(nd, x)**: Normalizes the given points in 2D or 3D space.
- **DLTcalib(nd, xyz, uv)**: Computes the Direct Linear Transform (DLT) calibration matrix from 3D to 2D point correspondences.
- **CameraCalibration(xyz, uv)**: Performs camera calibration to obtain the projection matrix, camera center, calibration matrix, rotation matrix, and translation vector.
- **find_intersection(X_0, X_1, quota)**: Finds the intersection point of a line with a plane at a given quota.
- **get_3d_point_at_quota_given_2d(H, H_nullspace, x, quota)**: Recovers the 3D coordinates of a point given its 2D coordinates and a quota value.
- **get_quota(x, y, H)**: Calculates the quota value for a given 2D point based on the calibration matrix.

- **get_normal_and_intersection(ball, light, camera)**: Corrects the 3D coordinates of a ball and calculates the normal vector and intersection point with a plane.
- **find_bottom_pixel(image, x_min, x_max, y_min, y_max, hue_min, hue_max)**: Finds the bottom pixel of a ball within a specified region and hue range.

In [ ]:
```
def get_bgr_frequencies(image):
    """
    Calculates the frequency of each color intensity level (0-255) for th

    Parameters:
    image (np.ndarray): A 2D array of pixels, where each pixel is represe
                        Each integer should be in the range 0-255, repres

    Returns:
    dict: A dictionary containing three keys: 'red', 'green', and 'blue'.
          The value at index i in these lists represents the frequency of
          - hist['red'][i]: Frequency of red intensity level i
          - hist['green'][i]: Frequency of green intensity level i
          - hist['blue'][i]: Frequency of blue intensity level i

    Example:
    >>> image = np.array([
    ...     [[0, 0, 0], [255, 255, 255], [127, 127, 127]],
    ...     [[0, 0, 0], [255, 0, 0], [0, 255, 0]],
    ...     [[0, 0, 255], [255, 255, 0], [0, 255, 255]]
    ... ])
    >>> histogram = get_bgr_frequencies(image)
    >>> print(histogram['red'][255])   # Output: 2 (2 pixels have red int
    >>> print(histogram['green'][255]) # Output: 4 (4 pixels have green i
    >>> print(histogram['blue'][255])  # Output: 3 (3 pixels have blue in
    """
    hist = {'red': [0] * 256, 'green': [0] * 256, 'blue': [0] * 256}
    for row in image:
        for pixel in row:
            hist['red'][pixel[2]] += 1
            hist['green'][pixel[1]] += 1
            hist['blue'][pixel[0]] += 1
    return hist

def get_hsi_frequencies(image):
    """
    Calculates the frequency of each hue (0-359), saturation (0-1), and i

    Parameters:
    image (np.ndarray): A 2D array of pixels, where each pixel is represe
                        H represents the hue angle in degrees (0-359), S

    Returns:
    dict: A dictionary containing three keys: 'hue', 'saturation', and 'i
          - hist['hue'][i]: Frequency of hue angle i (0-359)
          - hist['saturation'][i]: Frequency of saturation level i (0-100
          - hist['intensity'][i]: Frequency of intensity level i (0-100,

    Example:
    >>> image = np.array([
```

```
    ...     [[0, 0, 0], [360, 1, 1], [180, 0.5, 0.5]],
    ...     [[0, 0, 0], [360, 0, 0], [0, 1, 0]],
    ...     [[0, 0, 1], [360, 1, 0], [0, 1, 1]]
    ... ])
    >>> histogram = get_hsi_frequencies(image)
    >>> print(histogram['hue'][0])    # Output: 4 (4 pixels have hue angl
    >>> print(histogram['saturation'][100]) # Output: 3 (3 pixels have sa
    >>> print(histogram['intensity'][100])  # Output: 3 (3 pixels have in
    """

    hist = {'hue': [0] * 360, 'saturation': [0] * 101, 'intensity': [0] *
    for row in image:
        for pixel in row:
            hist['hue'][int(pixel[0])-1] += 1
            hist['saturation'][int(pixel[1] * 101)-1] += 1
            hist['intensity'][int(pixel[2] * 101)-1] += 1
    return hist

def plot_rgb_freq(freqs, save=False, save_path_name=None):
    """
    Plots the frequency distributions of the red, green, and blue channel

    Parameters:
    freqs (dict): A dictionary containing the frequency distributions for
    save (bool): If True, save the plot to a file.
    save_path_name (str): The path where the plot should be saved if save

    Example:
    >>> freqs = {'red': [0]*256, 'green': [0]*256, 'blue': [0]*256}
    >>> freqs['red'][255] = 10
    >>> plot_rgb_freq(freqs, save=True, save_path_name='rgb_freq.png')
    """
    plt.plot(freqs['red'], color='red',label='Red')
    plt.plot(freqs['green'], color='green',label='Green')
    plt.plot(freqs['blue'], color='blue',label='Blue')
    plt.show()
    if save:
        assert save_path_name is not None, "Please provide a name for the
        plt.savefig(save_path_name)


def plot_hsi_freq(freqs, save=False, save_path_name=None):
    """
    Plots the frequency distributions of the hue, saturation, and intensi

    Parameters:
    freqs (dict): A dictionary containing the frequency distributions for
    save (bool): If True, save the plot to a file.
    save_path_name (str): The path where the plot should be saved if save

    Example:
    >>> freqs = {'hue': [0]*360, 'saturation': [0]*101, 'intensity': [0]*
    >>> freqs['hue'][180] = 10
    >>> plot_hsi_freq(freqs, save=True, save_path_name='hsi_freq.png')
    """
    fig = plt.figure(figsize=(10, 10))
    gs = fig.add_gridspec(2, 2)
    ax1 = fig.add_subplot(gs[0, :])
    ax2 = fig.add_subplot(gs[1, 0])
    ax3 = fig.add_subplot(gs[1, 1])
```

```
    cm_hue = plt.get_cmap('hsv')
    for i in range(256):
        ax1.bar(i, freqs['hue'][i], color=cm_hue(i/360))
    ax1.set_title('Hue')
    ax1.set_xticks(np.arange(0, 360, 18))

    for i in range(101):
        ax2.bar(i, freqs['saturation'][i], color='black')
        ax3.bar(i, freqs['intensity'][i], color='black')

    ax2.set_title('Saturation')
    ax2.set_xlabel('Saturation in 100 bins')
    ax2.set_ylabel('Frequency')

    ax3.set_title('Intensity')
    ax3.set_xlabel('Intensity in 100 bins')
    ax3.set_ylabel('Frequency')

    plt.show()
    if save:
        assert save_path_name is not None, "Please provide a name for the
        plt.savefig(save_path_name)
    plt.close()

def histo_mse(hist1, histo2):
    """
    Calculates the mean squared error (MSE) between two histograms of BGR

    Parameters:
    hist1 (dict): A dictionary containing the frequency distributions for
    histo2 (dict): A dictionary containing the frequency distributions fo

    Returns:
    float: The calculated mean squared error (MSE) between the two histog

    Example:
    >>> hist1 = {'red': [0]*256, 'green': [0]*256, 'blue': [0]*256}
    >>> histo2 = {'red': [0]*256, 'green': [0]*256, 'blue': [0]*256}
    >>> hist1['red'][255] = 10
    >>> histo2['red'][255] = 20
    >>> mse = histo_mse(hist1, histo2)
    >>> print(mse)  # Output: 10.0
    """
    mse = 0
    for i in range(256):
        mse += (hist1['red'][i] - histo2['red'][i])**2
        mse += (hist1['green'][i] - histo2['green'][i])**2
        mse += (hist1['blue'][i] - histo2['blue'][i])**2
    return np.sqrt(mse)

def bgr_to_hsi(bgr_image):
    """
    Converts an image from BGR color space to HSI color space.

    Parameters:
    bgr_image (np.ndarray): Input image in BGR format.

    Returns:
    np.ndarray: Output image in HSI format.
```

```
    Example:
    >>> bgr_image = np.array([[[0, 0, 255], [0, 255, 0], [255, 0, 0]]], d
    >>> hsi_image = bgr_to_hsi(bgr_image)
    >>> print(hsi_image.shape)  # Output: (1, 3, 3)
    """

    with np.errstate(divide='ignore', invalid='ignore'):
        B,G,R = np.squeeze(bgr_image[...,0]), np.squeeze(bgr_image[...,1]
        sum_rgb = R + G + B
        b,g,r = B/255, G/255, R/255

        I = (b + g + r) / 3
        indexes = sum_rgb != 0
        ri = r[indexes]
        gi = g[indexes]
        bi = b[indexes]

        H = np.zeros_like(I)
        H[indexes] = np.arccos(0.5 * (2*ri-gi-bi) / np.sqrt((ri-gi)**2 +
        H[(R == G) & (G == B)] = 0
        H = np.nan_to_num(H)
        bg_indexes = b>g
        H[bg_indexes] = 2*np.pi - H[bg_indexes]
        H = np.degrees(H)

        S = np.zeros_like(I)
        S[indexes] = (1 - np.minimum(np.minimum(ri,gi),bi)/I[indexes])

        return np.dstack((H,S,I))


def hsi_to_bgr(hsi_image):
    """
    Converts an image from HSI color space to BGR color space.

    Parameters:
    hsi_image (np.ndarray): Input image in HSI format.

    Returns:
    np.ndarray: Output image in BGR format.

    Example:
    >>> hsi_image = np.array([[[0, 1, 1], [120, 1, 1], [240, 1, 1]]], dty
    >>> bgr_image = hsi_to_bgr(hsi_image)
    >>> print(bgr_image.shape)  # Output: (1, 3, 3)
    """
    with np.errstate(divide='ignore', invalid='ignore'):
        H,S,I = np.squeeze(hsi_image[...,0]), np.squeeze(hsi_image[...,1]
        R,G,B = np.zeros_like(I), np.zeros_like(I), np.zeros_like(I)

        ind120 = H < 120
        ind240 = (H >= 120) & (H < 240)
        ind360 = (H >= 240)

        H = np.radians(H)

        B[ind120] = (I*(1-S))[ind120]
        R[ind120] = (I*(1+(S*np.cos(H)/np.cos(np.pi/3-H))))[ind120]
        G[ind120] = (3*I-(R+B))[ind120]
```

```
        H[ind240] = (H-np.radians(120))[ind240]

        R[ind240] = (I*(1-S))[ind240]
        G[ind240] = (I*(1+S*np.cos(H)/np.cos(np.pi/3-H)))[ind240]

        H[ind360] = (H-np.radians(240))[ind360]

        G[ind360] = (I*(1-S))[ind360]
        B[ind360] = (I*(1+S*np.cos(H)/np.cos(np.pi/3-H)))[ind360]
        R[ind360] = (3*I-(G+B))[ind360]

        R, G, B = np.clip(R, 0, 1) * 255, np.clip(G, 0, 1) * 255, np.clip
        return np.dstack((B,G,R)).astype('uint8')


def show_bgr_image(image, grayscale=False):
    """
    Displays a BGR image using Matplotlib.

    Parameters:
    image (np.ndarray): Input image in BGR format.
    grayscale (bool): If True, display the image in grayscale.

    Example:
    >>> image = np.array([[[0, 0, 255], [0, 255, 0], [255, 0, 0]]], dtype
    >>> show_bgr_image(image)
    """
    if grayscale:
        plt.imshow(image, cmap='gray')
    else:
        plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

def get_gray_scale(bgr_image, custom_weights=None):
    """
    Converts a BGR image to grayscale.

    Parameters:
    bgr_image (np.ndarray): Input image in BGR format.
    custom_weights (list of float, optional): Custom weights for the B, G
        If not provided, the default weights [0.299, 0.587, 0.114] are us

    Returns:
    np.ndarray: Grayscale image.

    Example:
    >>> bgr_image = np.array([[[0, 0, 255], [0, 255, 0], [255, 0, 0]]], d
    >>> gray_image = get_gray_scale(bgr_image)
    >>> print(gray_image.shape)  # Output: (1, 3)
    """
    if custom_weights is not None:
        return np.dot(bgr_image[...,:3], custom_weights)
    return np.dot(bgr_image[...,:3], [0.299, 0.587, 0.114])

def is_frame_close(true_frame, test_frame, avg_threshold=1):
    """
    Checks if the mean of the absolute difference between two frames is b

    Parameters:
    true_frame (np.ndarray): The reference frame.
```

```
    test_frame (np.ndarray): The frame to be tested.
    avg_threshold (float): The threshold for the mean absolute difference

    Returns:
    tuple: A tuple containing a boolean indicating whether the frames are

    Example:
    >>> frame1 = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> frame2 = np.array([[0, 0, 0], [255, 255, 254]], dtype=np.uint8)
    >>> result, diff = is_frame_close(frame1, frame2)
    >>> print(result)  # Output: True
    """
    diff = np.abs(true_frame - test_frame)
    return np.mean(diff) < avg_threshold, np.mean(diff)

def apply_gaussian_filter(img, sigma=1, a=1):
    """
    Applies a Gaussian filter to a grayscale image.

    Parameters:
    img (np.ndarray): Input grayscale image.
    sigma (float): Standard deviation for Gaussian kernel.
    a (int): Kernel size parameter.

    Returns:
    np.ndarray: Blurred image.

    Example:
    >>> img = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> blurred_img = apply_gaussian_filter(img, sigma=1, a=1)
    >>> print(blurred_img.shape)  # Output: (2, 3)
    """
    M, N = img.shape
    b = a
    m = 2*a+1
    n = 2*b+1

    sigma = 1
    g = np.zeros((m))
    for s in range(-a,a+1):
        g[s+a] = math.exp(-s*s/2/(sigma*sigma))
    c = g/np.sum(g)
    r = c

    A1 = np.copy(img)
    for i in range(0,M):
        for j in range(0,N):
            I = 0
            for s in range(-a,a+1):
                if (i+s < 0):
                    I = I + c[s+a] * img[0,j]
                elif (i+s > M-1):
                    I = I + c[s+a] * img[M-1,j]
                else:
                    I = I + c[s+a] * img[i+s,j]
            A1[i,j] = round(I)

    B1 = np.copy(A1)
    for i in range(0,M):
        for j in range(0,N):
```

```
            I = 0
            for t in range(-b,b+1):
                if (j+t < 0):
                    I = I + r[t+b] * A1[i,0]
                elif (j+t > N-1):
                    I = I + r[t+b] * A1[i,N-1]
                else:
                    I = I + r[t+b] * A1[i,j+t]
            B1[i,j] = round(I)

    return B1


def threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.09):
    """
    Applies a threshold to an image to create a binary image.

    Parameters:
    img (np.ndarray): Input image.
    lowThresholdRatio (float): Low threshold ratio.
    highThresholdRatio (float): High threshold ratio.

    Returns:
    tuple: A tuple containing the binary image, weak pixel value, and str

    Example:
    >>> img = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> res, weak, strong = threshold(img, lowThresholdRatio=0.05, highTh
    >>> print(res.shape)  # Output: (2, 3)
    """
    highThreshold = img.max() * highThresholdRatio
    lowThreshold = highThreshold * lowThresholdRatio

    M, N = img.shape
    res = np.zeros((M,N), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i, strong_j = np.where(img >= highThreshold)
    zeros_i, zeros_j = np.where(img < lowThreshold)

    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThresho

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    return (res, weak, strong)

def hysteresis(img, weak, strong=255):
    """
    Applies hysteresis to an image to identify strong edges.

    Hysteresis - the phenomenon in which the value of a physical property
        lags behind changes in the effect causing it.

    Parameters:
    img (np.ndarray): Input binary image.
    weak (int): Value representing weak edges.
    strong (int): Value representing strong edges.
```

```
    Returns:
    np.ndarray: Image with strong edges.

    Example:
    >>> img = np.array([[0, 25, 0], [255, 0, 25]], dtype=np.int32)
    >>> result = hysteresis(img, weak=25, strong=255)
    >>> print(result.shape)  # Output: (2, 3)
    """
    M, N = img.shape

    for i in range(1, M-1):
        for j in range(1, N-1):
            if (img[i, j] == weak):
                if (
                    (img[i+1, j-1] == strong) or (img[i+1, j] == strong)
                    (img[i+1, j+1] == strong) or (img[i, j-1] == strong)
                    (img[i, j+1] == strong) or (img[i-1, j-1] == strong)
                    (img[i-1, j] == strong) or (img[i-1, j+1] == strong)
                ):
                    img[i, j] = strong
                else:
                    img[i, j] = 0
    return img

def sobel_operate(image):
    """
    Applies the Sobel operator to an image to detect edges.

    Parameters:
    image (np.ndarray): Input grayscale image.

    Returns:
    tuple: A tuple containing the gradient magnitude and gradient directi

    Example:
    >>> img = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> G, theta = sobel_operate(img)
    >>> print(G.shape)  # Output: (2, 3)
    """
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = convolve(image, Kx)
    Iy = convolve(image, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255

    theta = np.arctan2(Iy, Ix)

    return G, theta

def non_max_suppression(G, theta):
    """
    Applies non-maximum suppression to an image to thin edges.

    Parameters:
    G (np.ndarray): Gradient magnitude image.
    theta (np.ndarray): Gradient direction image.
```

```
    Returns:
    np.ndarray: Thinned edge image.

    Example:
    >>> G = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> theta = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.float32)
    >>> result = non_max_suppression(G, theta)
    >>> print(result.shape)  # Output: (2, 3)
    """
    M,N = G.shape
    Z = np.zeros((M,N), dtype=np.int32)
    angle = theta * 180. / np.pi
    angle[angle < 0] += 180

    for i in range(1,M-1):
        for j in range(1,N-1):
            q = 255
            r = 255

            if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                r = G[i, j-1]
                q = G[i, j+1]

            elif (22.5 <= angle[i,j] < 67.5):
                r = G[i-1, j+1]
                q = G[i+1, j-1]

            elif (67.5 <= angle[i,j] < 112.5):
                r = G[i-1, j]
                q = G[i+1, j]

            elif (112.5 <= angle[i,j] < 157.5):
                r = G[i+1, j+1]
                q = G[i-1, j-1]

            if (G[i,j] >= q) and (G[i,j] >= r):
                Z[i,j] = G[i,j]
            else:
                Z[i,j] = 0
    return Z


def canny_edge_detection(img, sigma=1, a=1, lowThresholdRatio=0.05, highT
    """
    Applies the Canny edge detection algorithm to an image.

    Parameters:
    img (np.ndarray): Input image.
    sigma (float): Standard deviation for Gaussian kernel.
    a (int): Kernel size parameter.
    lowThresholdRatio (float): Low threshold ratio.
    highThresholdRatio (float): High threshold ratio.

    Returns:
    np.ndarray: Edge-detected image.

    Example:
    >>> img = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint8)
    >>> edges = canny_edge_detection(img, sigma=1, a=1, lowThresholdRatio
```

```
    >>> print(edges.shape)  # Output: (2, 3)
    """
    gray_scale_img = get_gray_scale(img)
    gray_scale_img = apply_gaussian_filter(gray_scale_img, sigma, a)
    G, theta = sobel_operate(gray_scale_img)
    Z = non_max_suppression(G, theta)
    (res, weak, strong) = threshold(Z, lowThresholdRatio, highThresholdRa
    img_final = hysteresis(res, weak, strong)
    return img_final


def hough_line_space(edge_image):
    """
    Performs Hough Line Transform to detect lines in an edge-detected ima

    Parameters:
    edge_image (np.ndarray): Input edge-detected image.

    Returns:
    tuple: A tuple containing the Hough space, theta map, and rho map.

    Example:
    >>> edge_image = np.array([[0, 0, 0], [255, 255, 255]], dtype=np.uint
    >>> hough_space, theta_map, rho_map = hough_line_space(edge_image)
    >>> print(hough_space.shape)  # Output: (180, 284)
    """
    M,N = edge_image.shape

    rho_max = int(np.hypot(M, N))
    cap = 2*rho_max
    theta = np.linspace(-90, 90, cap, endpoint=True)
    theta_map = {i:np.radians(t) for i, t in enumerate(theta)}
    rho_map = {r:i for i, r in enumerate(np.arange(-rho_max, rho_max+1, 1
    hough_space = np.zeros((len(theta), 2*rho_max), dtype=np.uint8)

    for i in tqdm(range(M)):
        for j in range(N):
            if edge_image[i,j] > 0:
                for k,t in enumerate(theta):
                    r = i*np.cos(np.radians(t)) + j*np.sin(np.radians(t))
                    r = np.round(r).astype(int)
                    hough_space[k, rho_map[r]] += 1

    rho_map = {i:r for r,i in rho_map.items()}

    return hough_space, theta_map, rho_map


def binarized_thresholding(gray_image, threshold):
    """
    Applies binarized thresholding to a grayscale image.

    Parameters:
    gray_image (np.ndarray): Input grayscale image.
    threshold (int): Threshold value.

    Returns:
    np.ndarray: Binarized image.

    Example:
```

```
    >>> gray_image = np.array([[0, 127, 255]], dtype=np.uint8)
    >>> binarized_img = binarized_thresholding(gray_image, 128)
    >>> print(binarized_img)  # Output: [[  0   0 255]]
    """
    img = np.zeros_like(gray_image)
    filt = gray_image < threshold
    img[filt] = 0
    img[~filt] = 255
    return img


def plot_hough_lines(image, lines, theta_map, rho_map, save_path=None, in
    """
    Plots the Hough lines on an image.

    Parameters:
    image (np.ndarray): Input image.
    lines (list): List of lines to plot, where each line is represented a
    theta_map (dict): Dictionary mapping theta indices to theta values.
    rho_map (dict): Dictionary mapping rho indices to rho values.
    save_path (str, optional): Path to save the plotted image. If None, t
    indices (list, optional): List of indices for the lines.

    Returns:
    np.ndarray: Image with plotted Hough lines.

    Example:
    >>> image = np.array([[[0, 0, 255], [0, 255, 0], [255, 0, 0]]], dtype
    >>> lines = [(0, 0)]
    >>> theta_map = {0: 0}
    >>> rho_map = {0: 0}
    >>> result = plot_hough_lines(image, lines, theta_map, rho_map)
    >>> print(result.shape)  # Output: (1, 3, 3)
    """
    Gn = image.copy()
    for i, (ti,ri) in enumerate(lines):
        t = theta_map[ti]
        r = rho_map[ri]

        b = np.cos(t)
        a = np.sin(t)

        x0 = a*r
        y0 = b*r

        x1 = int(x0 + 10000*(-b))
        y1 = int(y0 + 10000*(a))
        x2 = int(x0 - 10000*(-b))
        y2 = int(y0 - 10000*(a))
        cv2.line(Gn, (x1, y1), (x2, y2), (255, 0, 0), 3)

        if indices is not None:
            cv2.putText(Gn, f'({ti},{ri})', (10, 30), cv2.FONT_HERSHEY_SI

    if save_path is not None:
        cv2.imwrite(save_path, Gn)
    return Gn

def compute_pre_mask(theta_map=None, rho_map=None):
    """
```

```
    Computes a pre-mask for ball detection based on Hough lines.

    Parameters:
    theta_map (dict): A dictionary mapping indices to theta values. Defau
    rho_map (dict): A dictionary mapping indices to rho values. Default i

    Returns:
    numpy.ndarray: The computed pre-mask.

    Example:
    >>> pre_mask = compute_pre_mask()
    """

    if theta_map is None:
        theta_map = pickle.load(open('./data/theta_map.pkl', 'rb'))
    if rho_map is None:
        rho_map = pickle.load(open('./data/rho_map.pkl', 'rb'))

    lines = {
        'top':[1471,1510],
        'bottom':[1472,2092],
        'left':[2712, 1829],
        'right':[213, 581]
    }

    for k in lines.keys():
        lines[k] = [theta_map[lines[k][0]], rho_map[lines[k][1]]]


    image = cv2.imread('./data/balls_in_position.png')

    a = lambda x,y: x>y
    b = lambda x,y: x<y
    l = [a,b,a,b]

    mask = np.ones_like(image[...,0])

    for i,line in enumerate(lines.values()):
        theta, rho = line
        a = np.sin(theta)
        b = np.cos(theta)
        x = lambda y: -b*y/a + rho/a

        for xi in range(image.shape[1]):
            for yi in range(image.shape[0]):
                if l[i](x(yi), xi):
                    mask[yi,xi] = 0
    i,j = 33,356
    mask[i:i+25,j:j+25] = 0
    i,j = 33,900
    mask[i:i+25,j:j+25] = 0

    return mask

def get_components(mask):
    """
    Retrieves connected components from a binary mask using depth-first s

    Parameters:
    mask (numpy.ndarray): The binary mask containing the components.
```

```
    Returns:
    list: A list of components, where each component is represented as a

    Example:
    >>> import numpy as np
    >>> mask = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]], dtype=bool)
    >>> components = get_components(mask)
    """

    rows, cols = mask.shape
    visited = np.zeros_like(mask, dtype=bool)
    components = []

    def dfs(row, col):
        stack = [(row, col)]
        component = []

        while stack:
            curr_row, curr_col = stack.pop()
            if curr_row < 0 or curr_row >= rows or curr_col < 0 or curr_c
                continue

            visited[curr_row, curr_col] = True
            component.append((curr_row, curr_col))

            neighbors = [(curr_row + 1, curr_col), (curr_row - 1, curr_co
                          (curr_row, curr_col + 1), (curr_row, curr_col
            for neighbor in neighbors:
                if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols an
                    stack.append(neighbor)

        return component

    for pixel in np.argwhere(mask):
        if not visited[tuple(pixel)]:
            component = dfs(pixel[0], pixel[1])
            components.append(component)

    return components

def get_balls_position_and_color(image, pre_mask, intensity_threshold=0.6
    """
    Retrieves the positions and colors of balls from an image and a preco

    Parameters:
    image (numpy.ndarray): The input image.
    pre_mask (numpy.ndarray): The precomputed mask indicating ball region
    intensity_threshold (float): The intensity threshold for filtering ou
    component_threshold (tuple): A tuple containing the lower and upper t

    Returns:
    list: A list of tuples, each containing the position and color of a b

    Example:
    >>> import numpy as np
    >>> image = np.random.randint(0, 256, size=(100, 100, 3))
    >>> pre_mask = np.random.randint(0, 2, size=(100, 100))
    >>> balls_poses_colors = get_balls_position_and_color(image, pre_mask
    """
```

```
    img = image.copy()
    mask = pre_mask.copy()
    img[mask == 0] = [0,0,0]
    hsi_image = bgr_to_hsi(img)

    hue = hsi_image[...,0]
    table = (hue > 90) & (hue < 130)

    mask[table] = 0
    img[mask == 0] = [0,0,0]
    sample_img = img.copy()

    intensity = hsi_image[...,2]
    table = intensity < intensity_threshold
    mask[table] = 0
    img[mask == 0] = [0,0,0]

    components = [c for c in get_components(mask) if len(c) > component_t
    if len(components) > 22:
        # get the 22 largest components
        components = sorted(components, key=lambda x: len(x), reverse=Tru

    avg_points = []
    for c in components:
        c = np.array(c)
        avg_points.append(np.mean(c, axis=0))

    result = []

    for p in avg_points:
        # sample a region around the point
        x,y = int(p[1]), int(p[0])
        region = sample_img[y-4:y+7,x-4:x+6]
        avg_color = np.mean(region, axis=(0,1))
        result.append((p, avg_color))

    return result

def find_intersection(X_0, X_1, quota):
    """
    Finds the intersection point of a line segment defined by two points

    Parameters:
    X_0 (numpy.ndarray): The first point of the line segment.
    X_1 (numpy.ndarray): The second point of the line segment.
    quota (float): The quota value.

    Returns:
    numpy.ndarray: The intersection point.

    Example:
    >>> import numpy as np
    >>> X_0 = np.array([0, 0, 0])
    >>> X_1 = np.array([1, 1, 1])
    >>> intersection_point = find_intersection(X_0, X_1, 0.5)
    """

    t = (quota - X_0[2]) / (X_1[2] - X_0[2])
    p = X_0 + t * (X_1 - X_0)
```

```
    return p

def get_3d_point_at_quota_given_2d(H,H_nullspace, x, quota):
    """
    Computes the 3D point at a given quota along a line in 3D space defin

    Parameters:
    H_nullspace (numpy.ndarray): The nullspace of the homography matrix.
    x (numpy.ndarray): The 2D point in homogeneous coordinates.
    quota (float): The quota value.

    Returns:
    numpy.ndarray: The 3D point at the given quota.

    Example:
    >>> import numpy as np
    >>> H_nullspace = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1]]
    >>> x = np.array([1, 1, 1])
    >>> quota = 0.5
    >>> point_3d = get_3d_point_at_quota_given_2d(H_nullspace, x, quota)
    """

    X = np.linalg.lstsq(H, x, rcond=None)[0]
    H_nullspace = H_nullspace.reshape(4)
    X_0 = X / X[-1]
    X_0 = X_0[:-1]
    X_1 = X + H_nullspace
    X_1 = X_1 / X_1[-1]
    X_1 = X_1[:-1]
    return find_intersection(X_0, X_1, quota)

def referance_mapping(point):
    """
    Maps a point from one reference frame to another.
    The destination referance system is relative to the snooker-table-top

    Parameters:
    point (tuple): The point coordinates (x, y) in the original reference

    Returns:
    tuple: The mapped point coordinates (p1, p2) in the new reference fra

    Example:
    >>> mapped_point = reference_mapping((100, 100))
    """

    x, y = point
    p1 = int((x*156/0.889)+200)
    p2 = int((-y*312/1.778)+355)
    return (p1, p2)

def generate_simulation_frame(background_image, ball_poses_colors):
    """
    Generates a simulation frame with circles representing ball positions

    Parameters:
    background_image (numpy.ndarray): The background image.
    ball_poses_colors (list): A list of tuples containing ball positions

    Returns:
```

```
    numpy.ndarray: The simulation frame with circles drawn.

    Example:
    >>> import numpy as np
    >>> background_image = np.zeros((500, 500, 3), dtype=np.uint8)
    >>> ball_poses_colors = [((100, 100), (0, 255, 0)), ((200, 200), (0,
    >>> simulation_frame = generate_simulation_frame(background_image, ba
    """

    for position, color in ball_poses_colors:
        cv2.circle(background_image, referance_mapping(position), 5, colo
    return background_image

def Normalization(nd, x):
    """
    Normalizes the given points in 2D or 3D space.

    Parameters:
    nd (int): Number of dimensions (2 or 3).
    x (np.ndarray): Points to be normalized.

    Returns:
    tuple: Transformation matrix and normalized points.

    Example:
    >>> x = np.array([[1, 2], [3, 4], [5, 6]])
    >>> Tr, x_norm = Normalization(2, x)
    >>> print(Tr)
    >>> print(x_norm)
    """
    x = np.asarray(x)
    m, s = np.mean(x, 0), np.std(x)
    if nd == 2:
        Tr = np.array([[s, 0, m[0]], [0, s, m[1]], [0, 0, 1]])
    else:
        Tr = np.array([[s, 0, 0, m[0]], [0, s, 0, m[1]], [0, 0, s, m[2]],

    Tr = np.linalg.inv(Tr)
    x = np.dot( Tr, np.concatenate( (x.T, np.ones((1,x.shape[0]))) ) )
    x = x[0:nd, :].T

    return Tr, x


def DLTcalib(nd, xyz, uv):
    """
    Computes the Direct Linear Transform (DLT) calibration matrix.

    Parameters:
    nd (int): Number of dimensions (3 for 3D).
    xyz (np.ndarray): 3D points.
    uv (np.ndarray): Corresponding 2D points.

    Returns:
    np.ndarray: Calibration matrix.

    Example:
    >>> xyz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    >>> uv = np.array([[1, 2], [3, 4], [5, 6]])
    >>> H = DLTcalib(3, xyz, uv)
```

```
    >>> print(H)
    """
    xyz = np.asarray(xyz)
    uv = np.asarray(uv)

    n = xyz.shape[0]

    Txyz, xyzn = Normalization(nd, xyz)
    Tuv, uvn = Normalization(2, uv)

    A = []

    for i in range(n):
        x, y, z = xyzn[i, 0], xyzn[i, 1], xyzn[i, 2]
        u, v = uvn[i, 0], uvn[i, 1]
        A.append( [x, y, z, 1, 0, 0, 0, 0, -u * x, -u * y, -u * z, -u] )
        A.append( [0, 0, 0, 0, x, y, z, 1, -v * x, -v * y, -v * z, -v] )

    A = np.asarray(A)
    U, S, V = np.linalg.svd(A)

    L = V[-1, :] / V[-1, -1]

    H = L.reshape(3, nd + 1)
    H = np.dot( np.dot( np.linalg.pinv(Tuv), H ), Txyz )

    H = H / H[-1, -1]

    return H

def CameraCalibration(xyz,uv):
    """
    Performs camera calibration to obtain the projection matrix, camera c

    Parameters:
    xyz (np.ndarray): 3D points.
    uv (np.ndarray): Corresponding 2D points.

    Returns:
    tuple: Projection matrix, camera center, calibration matrix, rotation

    Example:
    >>> xyz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    >>> uv = np.array([[1, 2], [3, 4], [5, 6]])
    >>> H, C, K, R, t = CameraCalibration(xyz, uv)
    >>> print(H)
    >>> print(C)
    >>> print(K)
    >>> print(R)
    >>> print(t)
    """
    nd = 3
    H = DLTcalib(nd, xyz, uv)
    M = H[:, :3]
    R, K = np.linalg.qr(M)
    C = np.dot(np.linalg.inv(-M), H[:, 3])
    t = np.dot(-R, C)

    return H ,C,K, R, t
```

```
def get_quota(x, y, H):
    """
    Calculates the quota value for a given 2D point.

    Parameters:
    x (float): X-coordinate of the 2D point.
    y (float): Y-coordinate of the 2D point.
    H (np.ndarray): Calibration matrix.

    Returns:
    float: Quota value.

    Example:
    >>> H = np.eye(3)
    >>> quota = get_quota(1, 2, H)
    >>> print(quota)
    """
    ball = np.dot(H, [x, y, 1])
    ball = ball/ball[2]
    return ((ball[1]-2)/194 +0.5)*0.0525

def get_normal_and_intersection(ball, light, camera):
    """
    Corrects the 3D coordinates of a ball and calculates the normal vecto

    Parameters:
    ball (np.ndarray): 3D coordinates of the ball.
    light (np.ndarray): Light source coordinates.
    camera (np.ndarray): Camera coordinates.

    Returns:
    tuple: Normal vector and intersection point.

    Example:
    >>> ball = np.array([1, 1, 1])
    >>> light = np.array([0, 0, 6])
    >>> camera = np.array([2, 2, 2])
    >>> N, B = get_normal_and_intersection(ball, light, camera)
    >>> print(N)
    >>> print(B)
    """
    N = (light - ball + camera - ball)/2
    N = N/N[2]
    B = ball + N*(0.02525 - ball[2])
    return N, B

def find_bottom_pixel(image, x_min, x_max, y_min, y_max, hue_min, hue_max

    """
    Finds the bottom pixel of a ball within a specified region and hue ra

    Parameters:
    image (np.ndarray): Input image in BGR format.
    x_min (int): Minimum x-coordinate for the region of interest.
    x_max (int): Maximum x-coordinate for the region of interest.
    y_min (int): Minimum y-coordinate for the region of interest.
    y_max (int): Maximum y-coordinate for the region of interest.
    hue_min (float): Minimum hue value for the range.
    hue_max (float): Maximum hue value for the range.
```

```
    Returns:
    tuple: Coordinates of the bottom pixel (max_x, max_y).

    Example:
    >>> image = np.zeros((500, 500, 3), dtype=np.uint8)
    >>> max_x, max_y = find_bottom_pixel(image, 100, 400, 100, 400, 100,
    >>> print(max_x, max_y)
    """
    Gn = image.copy()

    hsi_image = bgr_to_hsi(Gn)
    hue = hsi_image[...,0]
    table = (hue > hue_min) & (hue < hue_max)
    filtered = np.ones((Gn.shape[0],Gn.shape[1]), dtype=np.uint8)

    filtered[table] = 0
    filtered[:x_min,:] = 0
    filtered[x_max:,:] = 0
    filtered[:,:y_min] = 0
    filtered[:,y_max:] = 0

    non_black = np.argwhere(filtered == 1)
    max_y = np.max(non_black[:,0])
    non_black = non_black[non_black[:,0] == max_y]
    max_x = np.max(non_black[:,1])
    non_black = non_black[non_black[:,1] == max_x]

    return max_x, max_y


def find_intersection_point(line1, line2):
    """
    Finds the intersection point of two lines given in polar coordinates.

    Parameters:
    line1 (tuple): A tuple (theta1, rho1) representing the first line in
    line2 (tuple): A tuple (theta2, rho2) representing the second line in

    Returns:
    np.ndarray: The (x, y) coordinates of the intersection point.

    Example:
    >>> line1 = (np.pi/4, 1)
    >>> line2 = (np.pi/3, 1)
    >>> point = find_intersection_point(line1, line2)
    >>> print(point)
    """
    t1, r1 = line1
    t2, r2 = line2
    A = np.array([[np.sin(t1), np.cos(t1)], [np.sin(t2), np.cos(t2)]])
    b = np.array([r1, r2])
    return np.linalg.solve(A, b)


def construct_A_2d(pts_src, pts_dst):
    """
    Constructs the matrix A used in computing a homography from 2D point

    Parameters:
    pts_src (list of tuple): A list of (x, y) tuples representing source
```

```
    pts_dst (list of tuple): A list of (u, v) tuples representing destina

    Returns:
    np.ndarray: The constructed matrix A.

    Example:
    >>> pts_src = [(1, 2), (3, 4), (5, 6)]
    >>> pts_dst = [(7, 8), (9, 10), (11, 12)]
    >>> A = construct_A_2d(pts_src, pts_dst)
    >>> print(A)
    """
    A = []
    for i in range(len(pts_src)):
        x, y = pts_src[i][0], pts_src[i][1]
        u, v = pts_dst[i][0], pts_dst[i][1]
        A.append([-x, -y, -1, 0, 0, 0, x*u, y*u, u])
        A.append([0, 0, 0, -x, -y, -1, x*v, y*v, v])
    return np.array(A)


def compute_homography_2d(A):
    """
    Computes the homography matrix from the matrix A using Singular Value

    Parameters:
    A (np.ndarray): The matrix A constructed from point correspondences.

    Returns:
    np.ndarray: The computed homography matrix H.

    Example:
    >>> A = np.random.rand(8, 9)
    >>> H = compute_homography_2d(A)
    >>> print(H)
    """
    U, S, Vh = np.linalg.svd(A)
    L = Vh[-1, :] / Vh[-1, -1]
    H = L.reshape(3, 3)
    return H


def warp_perspective_2d(image, H, dst_size):
    """
    Warps an image to a new perspective using a given homography matrix.

    Parameters:
    image (np.ndarray): The input image to be warped.
    H (np.ndarray): The homography matrix.
    dst_size (tuple): The size of the output image (width, height).

    Returns:
    np.ndarray: The warped image.

    Example:
    >>> image = np.random.rand(100, 100, 3)
    >>> H = np.eye(3)
    >>> dst_size = (200, 200)
    >>> warped_image = warp_perspective_2d(image, H, dst_size)
    >>> print(warped_image.shape)
    """
```

```
    H_inv = np.linalg.inv(H)
    dst_img = np.zeros((dst_size[1], dst_size[0], 3), dtype=image.dtype)
    width, height = dst_size

    for i in range(width):
        for j in range(height):
            src_pt = np.dot(H_inv, [i, j, 1])
            src_pt /= src_pt[2]
            x, y = int(src_pt[0]), int(src_pt[1])
            if 0 <= x < image.shape[1] and 0 <= y < image.shape[0]:
                dst_img[j, i] = image[y, x]

    return dst_img
```

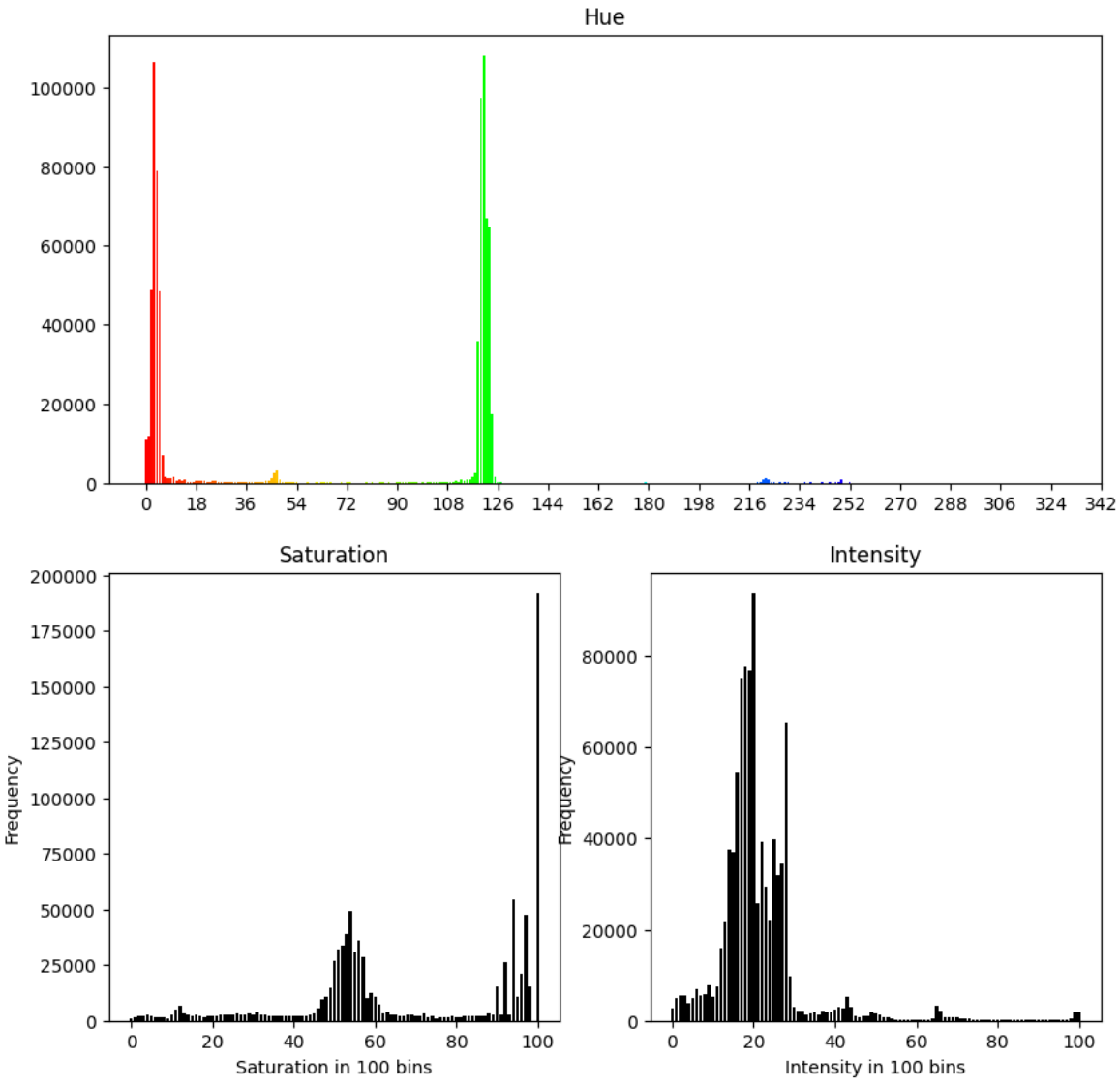
1. Video Filtering

Conversion to HSI and plotting of frequencies regarding Hue, Saturation and Intensity

In [ ]:
```
image_o = cv2.imread('./data/WSC sample.png')
image = image_o.copy()
show_bgr_image(image)
```

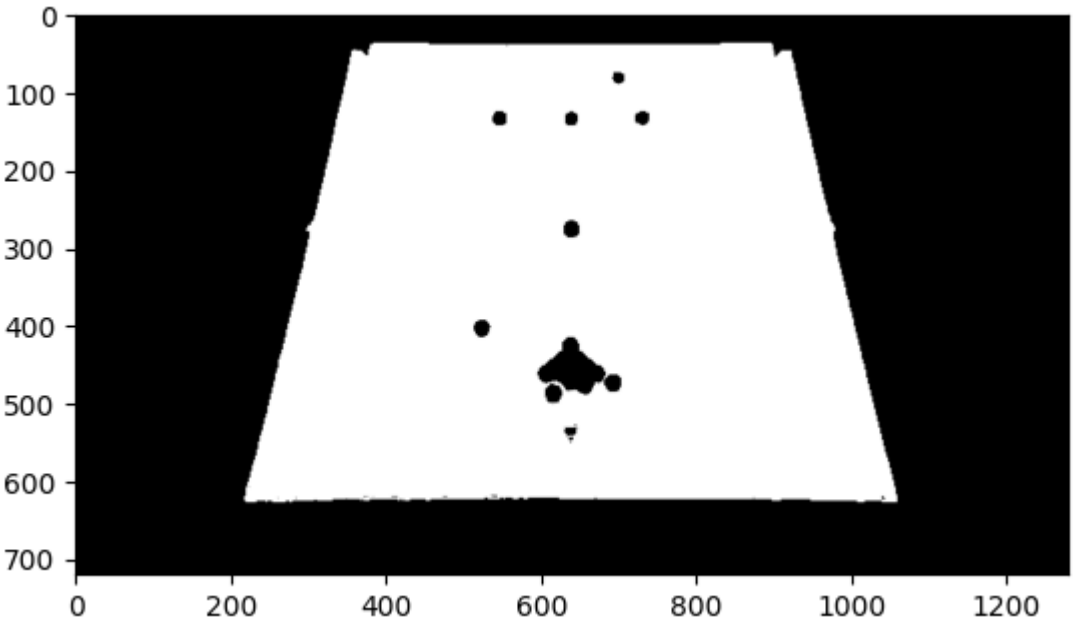
In [ ]:
```
show_bgr_image(hsi_to_bgr(bgr_to_hsi(image)))
```

In [ ]:
```
hsi_image = bgr_to_hsi(image)
hsi_freqs = get_hsi_frequencies(hsi_image)
plot_hsi_freq(hsi_freqs)
```



In [ ]:
```
hue = hsi_image[...,0]
table = (hue > 100) & (hue < 130)
```

```
filtered = np.zeros_like(image)
filtered[table] = [255,255,255]
show_bgr_image(filtered)
```



Getting some stats about the video

In [ ]:
```
video = cv2.VideoCapture('./data/WSC.mp4')
frame_count = video.get(cv2.CAP_PROP_FRAME_COUNT)
frame_rate = video.get(cv2.CAP_PROP_FPS)
video_duration = frame_count / frame_rate
print(f"The video duration is {video_duration} seconds.")
# convert to hour, minutes seconds
video_duration = int(video_duration)
hours = video_duration // 3600
minutes = (video_duration % 3600) // 60
seconds = video_duration % 60
print(f"The video duration is {hours} hours, {minutes} minutes and {secon
video.release()
```

```
The video duration is 12262.6 seconds.
The video duration is 3 hours, 24 minutes and 22 seconds.
```

In [ ]:
```
interesting_frames_timestamps = {
    0: {'hour': 0, 'minute': 16, 'second': 56 },
    1: {'hour': 0, 'minute': 17, 'second': 31 },
    2: {'hour': 0, 'minute': 28, 'second': 40 },
    3: {'hour': 0, 'minute': 29, 'second': 8 },
    4: {'hour': 0, 'minute': 29, 'second': 39 },
    5: {'hour': 0, 'minute': 29, 'second': 42 },
    6: {'hour': 0, 'minute': 59, 'second': 18 },
    7: {'hour': 2, 'minute': 51, 'second': 40 },
    8: {'hour': 2, 'minute': 39, 'second': 27 },
    9: {'hour': 0, 'minute': 0, 'second': 0 },
    10: {'hour': 0, 'minute': 13, 'second': 26 }
}

video = cv2.VideoCapture('./data/WSC.mp4')
for i, timestamp in interesting_frames_timestamps.items():
    video.set(cv2.CAP_PROP_POS_MSEC, (timestamp['hour']*3600 + timestamp[
```

```
    ret, frame = video.read()
    cv2.imwrite(f'./data/interesting_frames/{i}.png', frame)
video.release()
```

In [ ]:
```
image_filtered = image.copy()
image_filtered[~table] = [0,0,0]

dist = {}
for i in range(len(interesting_frames_timestamps)):
    test_frame = cv2.imread(f'./data/interesting_frames/{i}.png')
    test_frame[~table] = [0,0,0]
    bool, dist[i] = is_frame_close(get_gray_scale(image_filtered), get_gr
    cv2.putText(test_frame, f"Error: {dist[i]}", (10, 30), cv2.FONT_HERSH
    cv2.imwrite(f'./data/interesting_frames/{i}_filtered.png', test_frame

for i,v in dist.items():
    print(f"Frame {i} -> distance {v}")
```

```
Frame 0 -> distance 1.332555802951388
Frame 1 -> distance 0.9002221549479167
Frame 2 -> distance 2.7958595800781243
Frame 3 -> distance 0.6331101085069442
Frame 4 -> distance 14.877442995876734
Frame 5 -> distance 1.8299508083767362
Frame 6 -> distance 5.02634072265625
Frame 7 -> distance 4.076061618923611
Frame 8 -> distance 8.848121379123267
Frame 9 -> distance 28.011986188151056
Frame 10 -> distance 1.4629001139322915
```

A good threshold seems to be 1.0, hence we filter the whole video using our mask the that difference threshold

In [ ]:
```
video = cv2.VideoCapture('./data/WSC.mp4')
frame_width = int(video.get(3))
frame_height = int(video.get(4))
fps = video.get(cv2.CAP_PROP_FPS)
out = cv2.VideoWriter('./data/WSC_filtered.mp4', cv2.VideoWriter_fourcc(*
true_table = ~table
filtered_gray_image = get_gray_scale(image_filtered)
frame_count = video.get(cv2.CAP_PROP_FRAME_COUNT)


for i in tqdm(range(int(frame_count))):
    ret, frame = video.read()
    if ret:
        test_frame = frame.copy()
        test_frame[true_table] = [0,0,0]
        test_frame = get_gray_scale(test_frame)
        cond, diff = is_frame_close(filtered_gray_image, test_frame, avg_
        if cond:
            out.write(frame)
    else:
        break
```

```
video.release()
out.release()
```

```
 0%|          | 0/367878 [00:00<?, ?it/s]
```

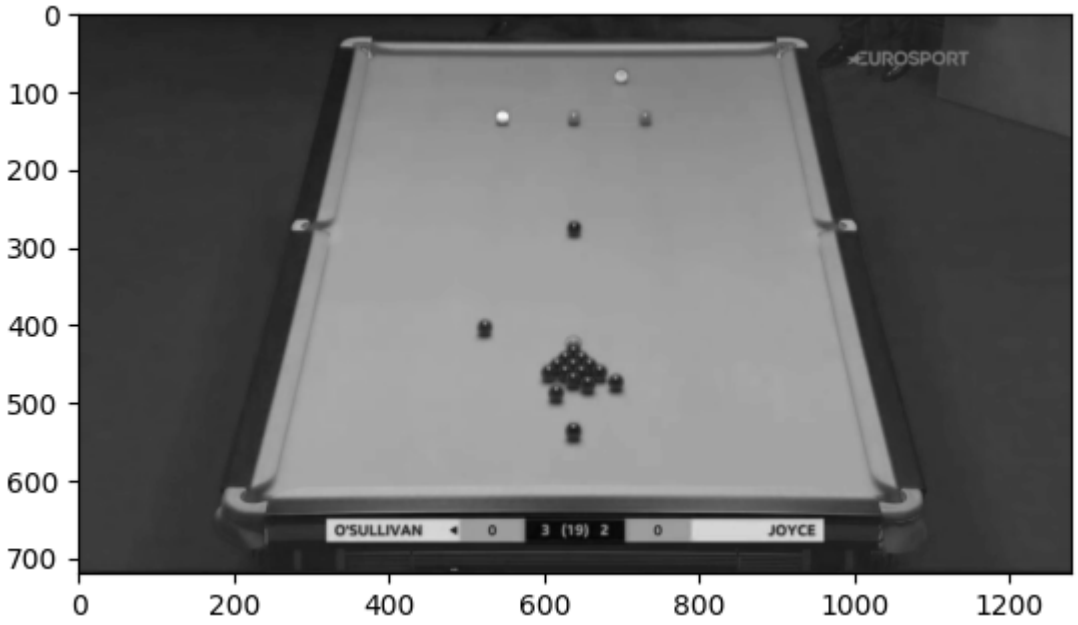
Getting the frame suggested in the slides, in timestamp 2:11:52, and saving it.

In [ ]:
```
video = cv2.VideoCapture('./data/WSC.mp4')
video.set(cv2.CAP_PROP_POS_MSEC, (2*3600 + 11*60 + 52)*1000)
ret, frame = video.read()
cv2.imwrite(f'./data/balls_in_position.png', frame)
show_bgr_image(frame)
```

2. Camera Reconstruction

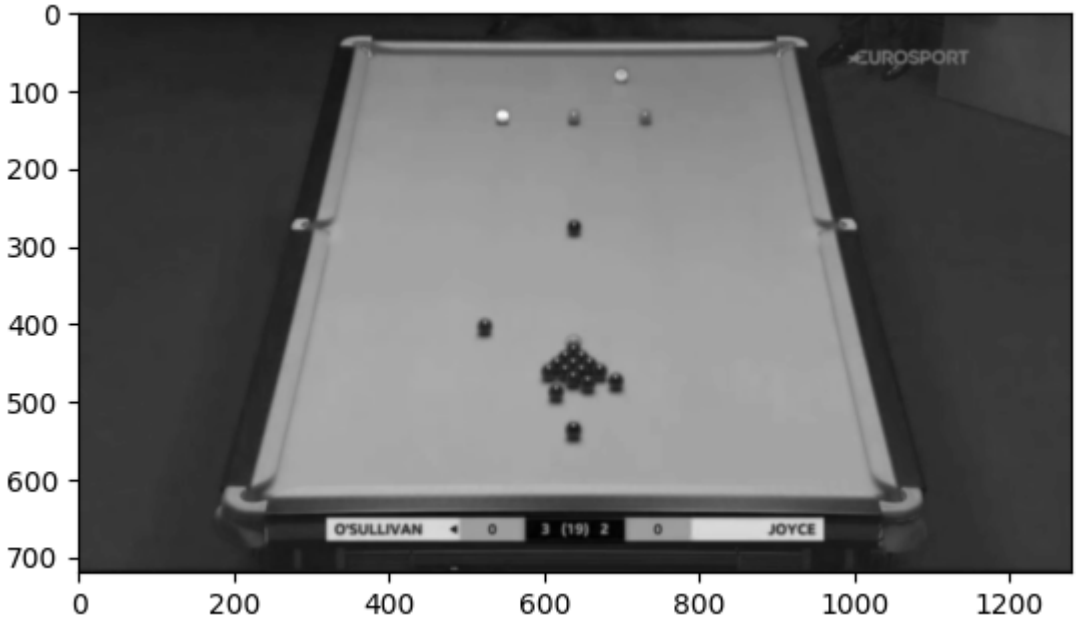
Here we applied a custom version of grayscaling to bring out the contrast between the green and the brown of the table, which denote lines we are looking for

In [ ]:
```
image_o = cv2.imread('./data/WSC sample.png')
image = image_o.copy()
weights = np.array([-0.5, 1, 0])
gray_image = get_gray_scale(image, weights)
show_bgr_image(gray_image, grayscale=True)
```

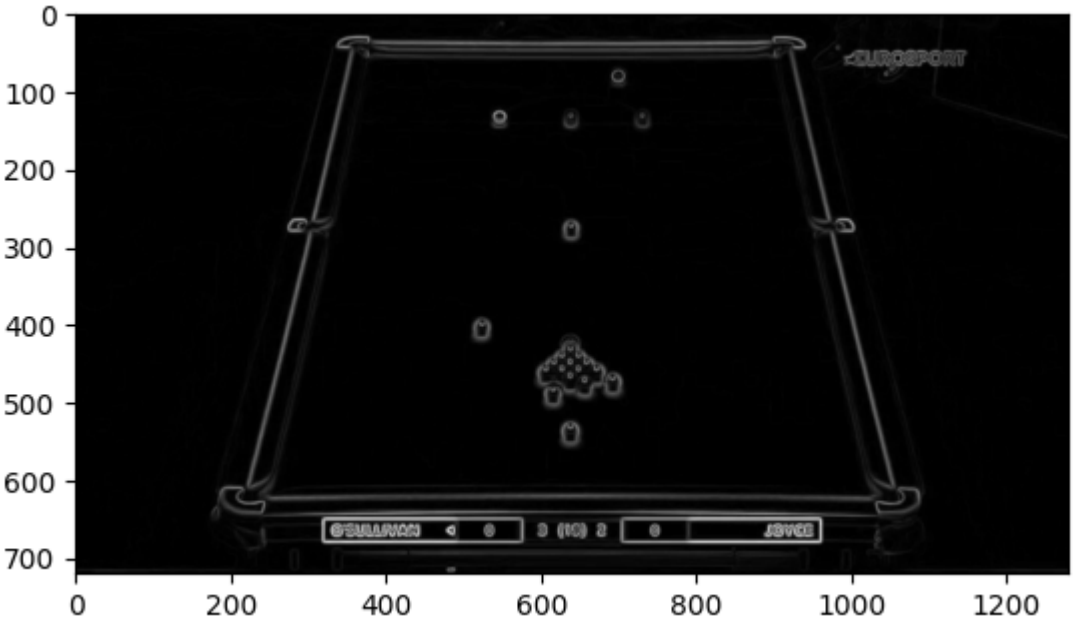


Next, we defined a sigma value of 30 and a box of width and height 10 for the Gaussian blur, so we could minimize the noise once we compute the lines

In [ ]:
```
sigma=30
a=10

gray_scale_img = apply_gaussian_filter(gray_image, sigma, a)
show_bgr_image(gray_scale_img, grayscale=True)
```

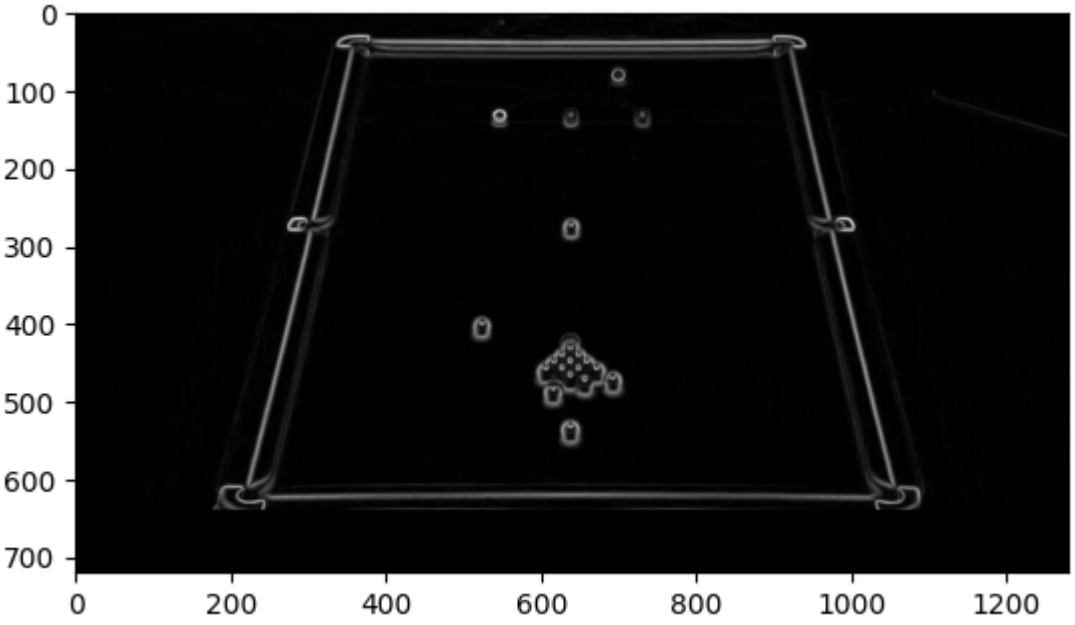


In [ ]:
```
G, theta = sobel_operate(gray_scale_img)
show_bgr_image(G, grayscale=True)
```



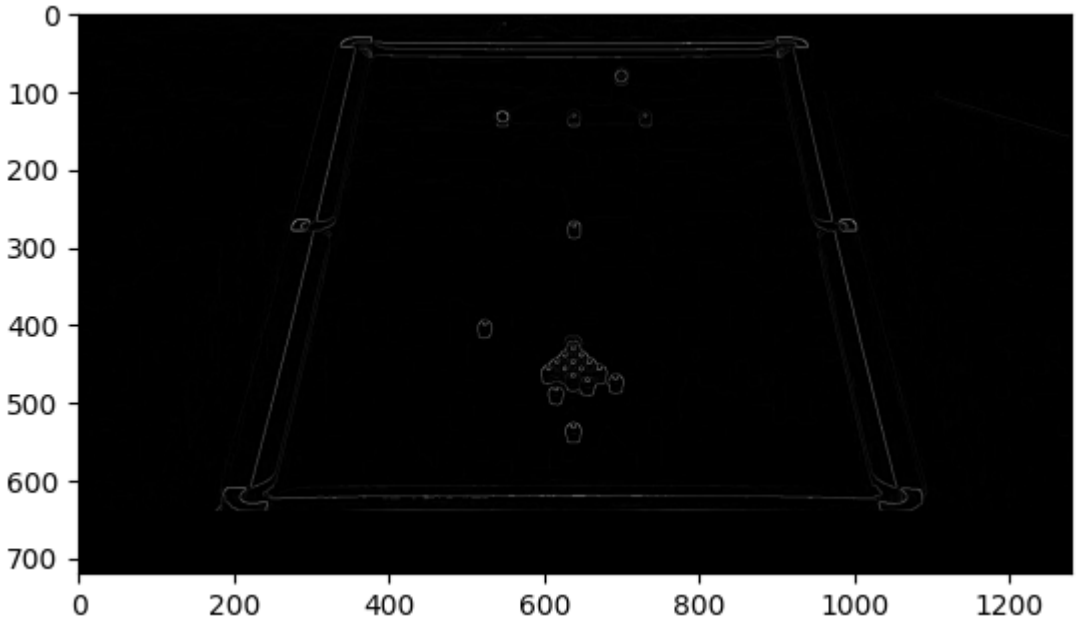
Before moving on, we decided to blacken out the bottom pixels where the name and scores of the players appear to further reduce noise.

In [ ]:
```
G[640:,:] = 0
G[0:100,950:] = 0
theta[640:,:] = 0
theta[0:100,950:] = 0
show_bgr_image(G, grayscale=True)
```



Having blured the image, we proceeded to apply non maxima supression

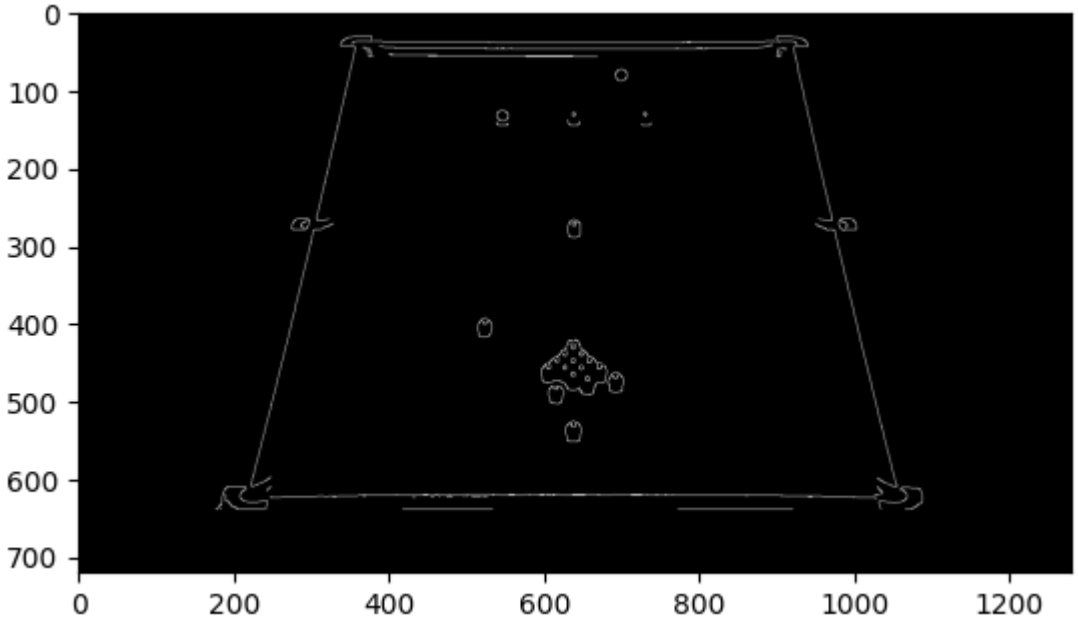
In [ ]:
```
Z = non_max_suppression(G, theta)
show_bgr_image(Z, grayscale=True)
```



For threshholding, we found that a lower upper threshold of 0.1 and 0.3 provided the best results to get the lines we were looking for.

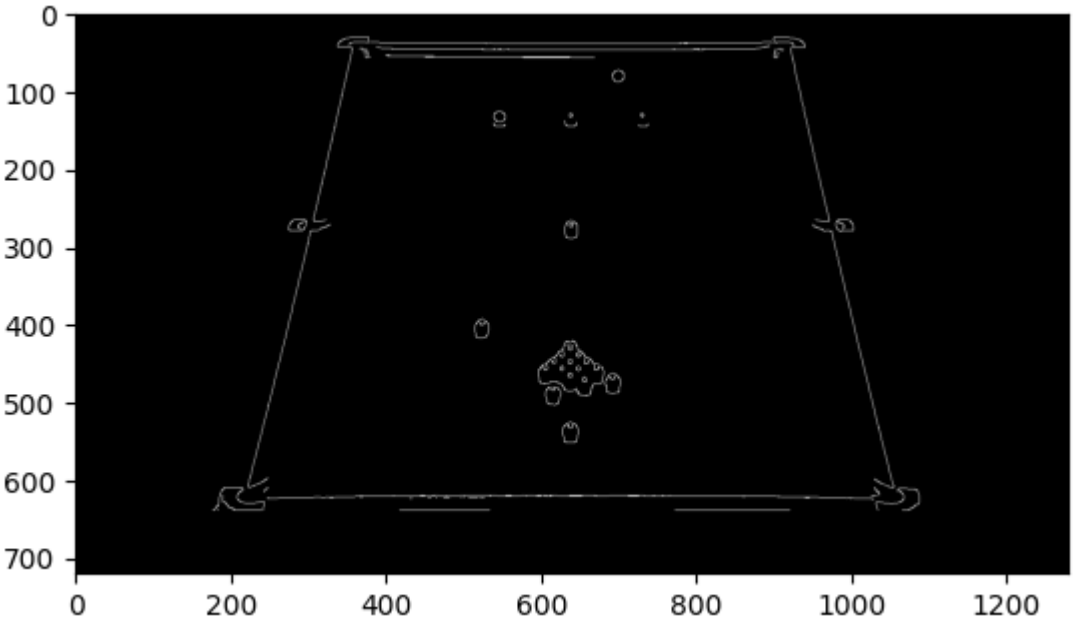
With the resulting image from thresholding we applied hysteresis

In [ ]:
```
lowThresholdRatio=0.1
highThresholdRatio=0.3

(res, weak, strong) = threshold(Z, lowThresholdRatio, highThresholdRatio)
img_final = hysteresis(res, weak, strong)
show_bgr_image(img_final, grayscale=True)
```



Finally, we applied a binarization with a filter of 50 intensity to scale the values to either 0 or 255

In [ ]:
```
binary_image = binarized_thresholding(img_final, 50)
show_bgr_image(binary_image, grayscale=True)
```

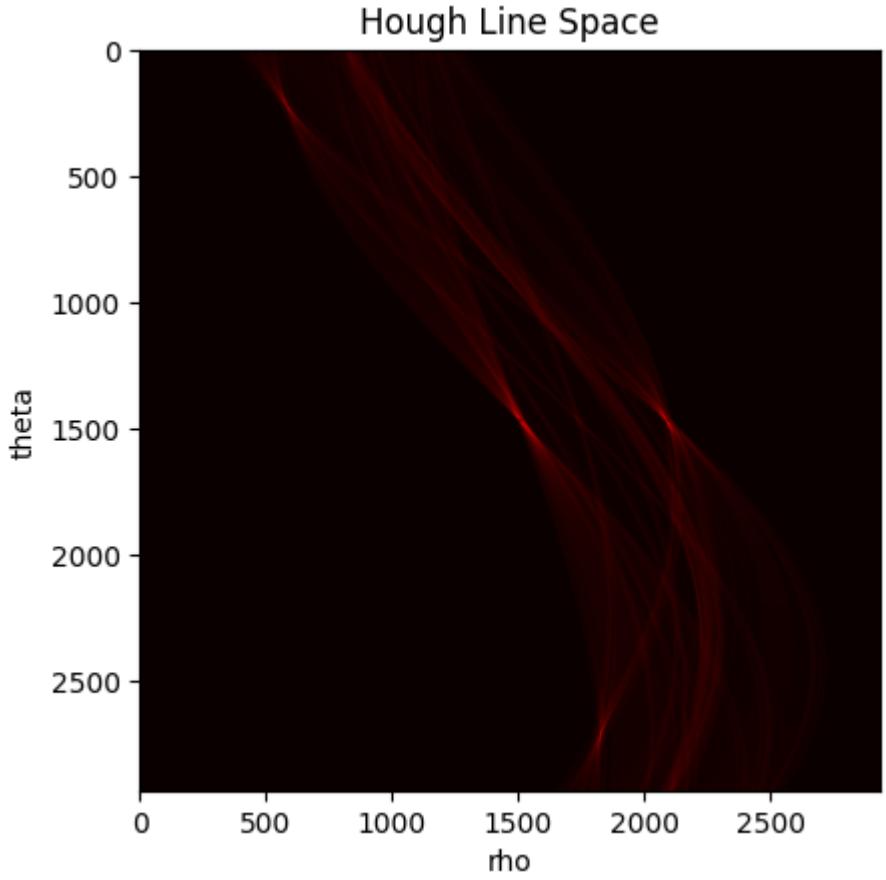


We are now in condition of applying a hough space transform to retrieve the line equations and later on filter them out with some lower bound limit of votes. We also retrieve a theta mapping and rho mapping that will be handy to later plot the lines and assess which are the ones we are looking for.

In [ ]:
```
space, theta_map, rho_map = hough_line_space(binary_image)
```

```
  0%|          | 0/720 [00:00<?, ?it/s]
```

In [ ]:
```
plt.imshow(space, cmap='hot')
plt.xlabel('rho')
plt.ylabel('theta')
plt.title('Hough Line Space')
plt.show()
```



In order to use the computed maps and space later, we decided that it would be best to save them, in case the kernel gets reset.

In [ ]:
```
with open('./data/hough_space.pkl', 'wb') as f:
    pickle.dump(space, f)

with open('./data/theta_map.pkl', 'wb') as f:
    pickle.dump(theta_map, f)

with open('./data/rho_map.pkl', 'wb') as f:
    pickle.dump(rho_map, f)
```

In [ ]:
```
theta_map = pickle.load(open('./data/theta_map.pkl', 'rb'))
rho_map = pickle.load(open('./data/rho_map.pkl', 'rb'))
space = pickle.load(open('./data/hough_space.pkl', 'rb'))
```

We then found that selecting lines that had a votes above 200 provided the best lines, so we filtered the respective space. The result ended being 76 total lines.

In [ ]:
```
votes_lower_limist = 200
max_args = np.argwhere(space > votes_lower_limist)
print(len(max_args))
```

```
76
```

For each of the lines, we drew them in the image, labeled the image and saved it to analyze which ones are most suitable.

In [ ]:
```
lines = []

for i,j in max_args:
    lines.append((i,j))

for i,line in enumerate(lines):
    save_path = f'./data/hough/hough_lines_{max_args[i][0]}_{max_args[i][
    plot_hough_lines(image, [line], theta_map, rho_map, save_path=save_pa
```

The most suitable lines we found can be seen below

In [ ]:
```
lines = {
    'top':[1471,1510],
    'bottom':[1472,2092],
    'left':[2712, 1829],
    'right':[213, 581]
}


fig, axs = plt.subplots(2, 2, figsize=(10, 10))
for i, (k, v) in enumerate(lines.items()):
    img = cv2.imread(f'./data/hough/hough_lines_{v[0]}_{v[1]}.png')
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    axs[i//2, i%2].imshow(img)
    axs[i//2, i%2].set_title(k)
    axs[i//2, i%2].axis('off')
plt.show()
```

top

bottom

left

right

Given the lines, we computed the pair wise intersection between them to get the points we were looking for

In [ ]:
```
for k in lines.keys():
    lines[k] = [theta_map[lines[k][0]], rho_map[lines[k][1]]]

top_left = find_intersection_point(lines['top'], lines['left'])
top_right = find_intersection_point(lines['top'], lines['right'])
bottom_left = find_intersection_point(lines['bottom'], lines['left'])
bottom_right = find_intersection_point(lines['bottom'], lines['right'])

print(f'Top left: {top_left}')
print(f'Top right: {top_right}')
print(f'Bottom left: {bottom_left}')
print(f'Bottom right: {bottom_right}')

img = cv2.cvtColor(image_o.copy(), cv2.COLOR_BGR2RGB)
fig, ax = plt.subplots()
ax.imshow(img)
ax.scatter([top_left[0], top_right[0], bottom_right[0], bottom_left[0]],
plt.show()
```

```
Top left: [361.64364174  40.64544018]
Top right: [919.50948027  38.55546268]
Bottom left: [219.94856423 622.94779295]
Bottom right: [1054.17207249  618.92951417]
```

Next, our goal is to find the bottom pixels of the brown and blue ball. For that we applied a simple framing and HSI masking for each of the balls.

In [ ]:
```
image = cv2.imread('./data/balls_in_position.png')
max_x, max_y = find_bottom_pixel(image, 250, 300, 610, 800, 100, 144)
max_x_top, max_y_top = find_bottom_pixel(image, 100, 200, 600, 700, 70, n

print(f'The bottom blue pixel is at ({max_x}, {max_y})')
print(f'The bottom brown pixel is at ({max_x_top},{max_y_top})')

image = cv2.imread('./data/balls_in_position.png')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
fig, ax = plt.subplots()
ax.imshow(image)
#plot both points with radius 5
ax.scatter(max_x, max_y, c='r', s=10)
ax.scatter(max_x_top, max_y_top, c='r', s=10)
plt.show()
```

```
The bottom blue pixel is at (641, 287)
The bottom brown pixel is at (637,143)
```

We now have 6 points, we can therefore compute a planar homography or a 3D to 2D homography.

In [ ]:
```
uv = np.array([
    [361.64364174, 40.64544018],   # top left
    [919.50948027, 38.55546268],   # top right
    [219.94856423, 622.94779295],  # bottom left
    [1054.17207249, 618.92951417], # bottom right
    [641, 143],                    # bottom brown ball
    [641, 287]                     # bottom blue ball
])


xyz = np.array([
    [-0.9398, 1.8353, 0.0381],
    [0.9398, 1.8353, 0.0381],
    [-0.9398, -1.8353, 0.0381],
    [0.9398, -1.8353, 0.0381],
    [0, 1.0475, 0.0],
    [0,0,0]
])

Gn = cv2.imread('./data/balls_in_position.png')
for i in range(uv.shape[0]):
    cv2.circle(Gn, (int(uv[i, 0]), int(uv[i, 1])), 8, (0, 0, 255), -1)

show_bgr_image(Gn)
```

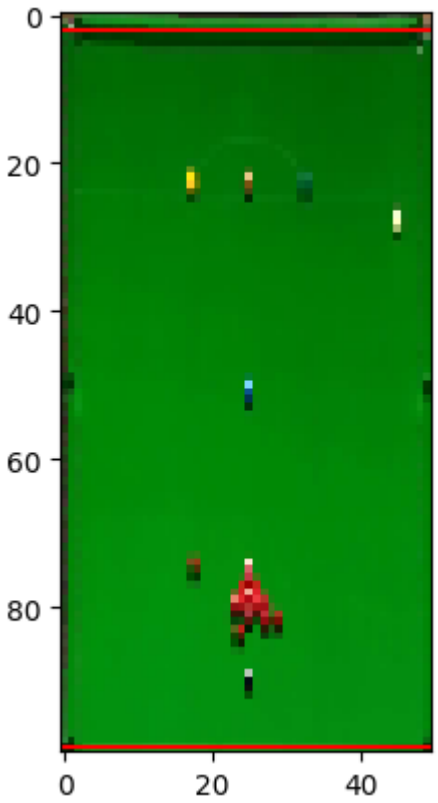
Our goal is to find the light reflection on top of each ball to then estimate the center of the ball knowing the camera and light positions in the 3D map. But we know also that the reflection on the ball tends to be higher on it if the ball is closer to the camera, and close to the middle of the ball if far from the camera.

In order to compute the height, that varies between 0.5 and 1 factor of the ball height, we implement a function that retrieves the respective quota given a 2D pixel. To achieve this, we can map the pixel into the planar transformation as a top view, and then retrieve the x coordinate, which denotes the position of the ball relative to the length of the table.

In [ ]:
```
table_ratio = 3.569/1.778
img_width = 50
img_height = round(img_width * table_ratio)
print(f"Image width: {img_width}, Image height: {img_height}")

pts_src = np.array([
    [361.64364174, 40.64544018],   # top left
    [919.50948027, 38.55546268],   # top right
    [219.94856423, 622.94779295],  # bottom left
    [1054.17207249, 618.92951417], # bottom right
])

pts_dst = np.array([
    [0, 0],
    [img_width, 0],
    [0, img_height],
    [img_width, img_height],
])

A = construct_A_2d(pts_src, pts_dst)
H_2d = compute_homography_2d(A)
top_view = warp_perspective_2d(image, H_2d, (img_width, img_height))
```

```
plt.imshow(top_view)
plt.axhline(y=2, color='r', linestyle='-')
plt.axhline(y=99, color='r', linestyle='-')
plt.show()
```

```
Image width: 50, Image height: 100
```



By analysing the aboeve image, we can clearly see that the transformed x coordinate can be fed into a function that maps pixels in range from 2 to 99, to 0.5 and 1, respectively.

We are now in condition of creating a 3D map of our table and balls using a 3D to 2D homography. Furthermore, given a 3D homogenous coordinate X, we know that x = HX, where x is a 2D homogenous coordinate. Therefore, if we want to compute the line of points that map into x, we can use the nullspace of the homography and a particular solution of the system (one of the points)! In other orders, once we can compute that 3D line of points, we can then intersect it with the z = quota plane to retrieve a really good estimate of the surface reflection point on the ball (accounting for the homography errors).

In [ ]:
```
H,C,K,R,t = CameraCalibration(xyz,uv)
nullspace = null_space(H).T.squeeze()

print("the Transformation: \n",H)
print("\nThe camera position:\n ",C)
print("\nThe camera matrix:\n",K)
print("\nThe rotation matrix:\n",R)
```

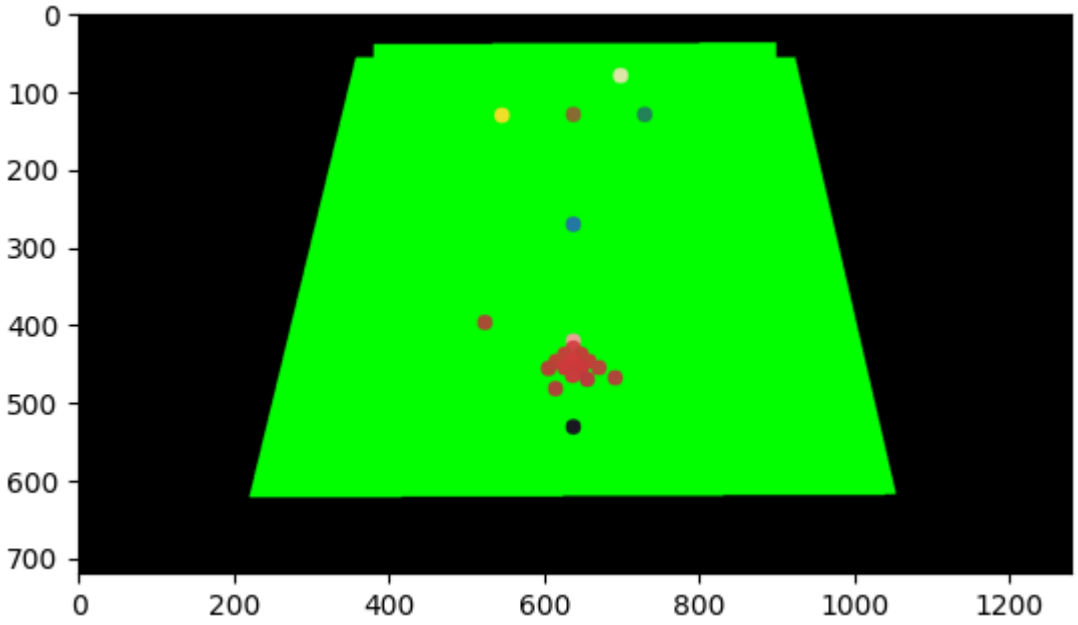
```
print("\nThe translation matrix:\n",t)
print("\nThe null space of the camera matrix:\n",nullspace)
```

```
the Transformation:
 [[ 3.54499144e+02  6.96779244e+01 -1.11536445e+02  6.40651017e+02]
 [-1.28647804e+00 -1.22077843e+02 -4.12549240e+02  2.87004845e+02]
 [ 6.94704528e-04  1.07661599e-01 -1.22907048e-01  1.00000000e+00]]

The camera position:
  [ 0.2510911  -6.35116758  2.57428341]

The camera matrix:
 [[-354.50147832  -70.12048356  110.03857912]
 [   0.          121.82422741  412.95101728]
 [   0.            0.           -0.48716936]]

The rotation matrix:
 [[-9.99993415e-01 -3.62897825e-03  1.24334675e-06]
 [ 3.62897793e-03 -9.99993026e-01  8.82618730e-04]
 [-1.95966610e-06  8.82617430e-04  9.99999610e-01]]

The translation matrix:
 [ 0.22803799 -6.3543066  -2.56867626]

The null space of the camera matrix:
 [-0.03623157  0.91645139 -0.37146014 -0.14429652]
```

as a first step, we get all the reflection points on the balls using region and HSI filters, and also the respective ball color by sampling around the reflection points. The image below does not represent the true position of the balls, since we are mapping the reflections, but it serves as a good indicator that we are on the right path. The actual center of the ball shall be computed later on.

In [ ]:
```
pre_mask = compute_pre_mask()
img = cv2.imread('./data/WSC sample.png')
bpos_color = get_balls_position_and_color(img, pre_mask, intensity_thresh

# create an image like img full of green
plot_img = np.zeros_like(img)
plot_img[:] = (0, 255, 0)
pre_mask = compute_pre_mask()
plot_img[pre_mask == 0] = (0,0,0)

for i, (pos, color) in enumerate(bpos_color):
    x, y = int(pos[1]), int(pos[0])
    cv2.circle(plot_img, (x, y), 10, color, -1)

print(len(bpos_color))
show_bgr_image(plot_img)
```

```
22
```



Since we know the position of the camera, the position of the light and the reflection point on the surface of the ball, using to vectors centered in the reflection point and pointing torwards the camera and the light, we can then find the coplanar vector that stands in the middle of these two. The latter vector is the normal of the plane tangent to the reflection point of the ball. In other words, we can normalize if and invert its direction with a magnitude equal to the radius of thr ball to get the center.

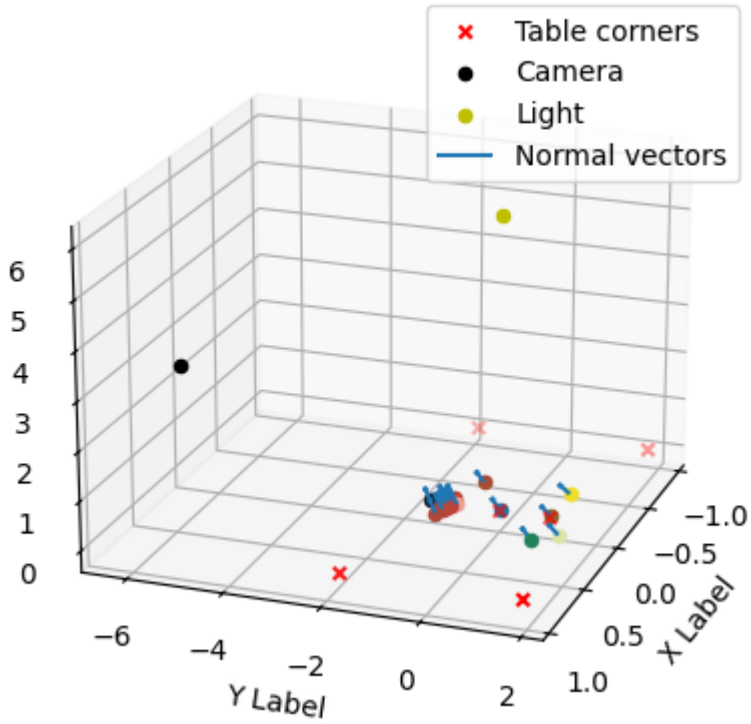
In [ ]:
```
balls_3d = []
normal_vectors = []
light = np.array([0, 0, 6])


for ball in bpos_color:
    y, x = ball[0]
    quota = get_quota(x, y, H_2d)

    ball = get_3d_point_at_quota_given_2d(H,nullspace, [x, y, 1], quota)

    N, B = get_normal_and_intersection(ball, light, C)
    balls_3d.append(B)
    normal_vectors.append(N)
```

For proof of concept, we mapped in a 3D world the balls we found and the vectors normal to the plane tangent to the reflection point on the surface of the ball.

In [ ]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xyz[:, 0], xyz[:, 1], xyz[:, 2], c='r', marker='x')
ax.scatter(C[0], C[1], C[2], c='black', marker='o')
ax.scatter(0,0,6, c='y', marker='o')
```

```
for i in range(len(balls_3d)):
    color = np.array([bpos_color[i][1][2],bpos_color[i][1][1],bpos_color[
    ax.quiver(balls_3d[i][0], balls_3d[i][1], balls_3d[i][2], normal_vect
    ax.scatter(balls_3d[i][0], balls_3d[i][1], balls_3d[i][2], c = color/

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
ax.view_init(elev=20, azim=20)
ax.legend(['Table corners', 'Camera', 'Light', 'Normal vectors'])
plt.show()
```

```
/var/folders/4h/m4cl822x663crm6w17f4x4fh0000gn/T/ipykernel_44349/126925316
1.py:10: UserWarning: *c* argument looks like a single numeric RGB or RGBA
sequence, which should be avoided as value-mapping will have precedence in
case its length matches with *x* & *y*.  Please use the *color* keyword-ar
gument or provide a 2D array with a single row if you intend to specify th
e same RGB or RGBA value for all points.
  ax.scatter(balls_3d[i][0], balls_3d[i][1], balls_3d[i][2], c = color/25
5, marker='o')
```



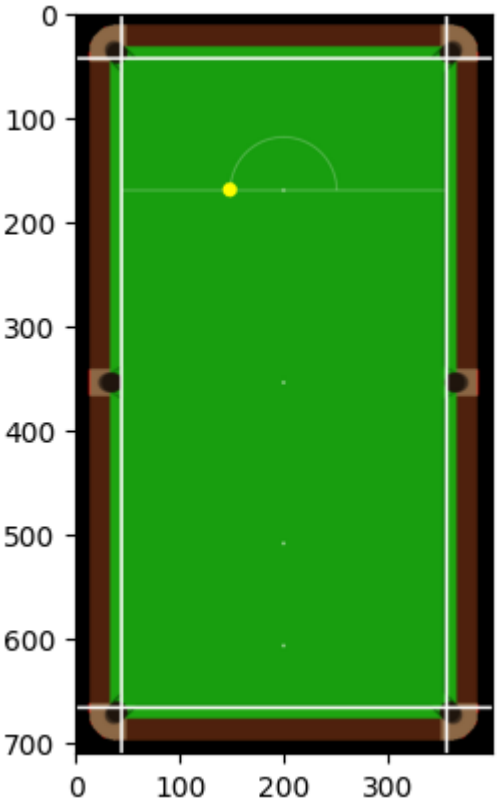
3. Generating a top view with the estimated center position of the balls.

To achieve this last step, we generated an image of the snooker table with the correct porpotions and performed a small study so we could map the balls from the xy plane of the 3D model into the image itselt.

In [ ]:
```
img = cv2.imread('./Data/snooker-table-top-view.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
height, width = img.shape[:2]
```

```
x1 = 44
cv2.line(img, (x1, 0), (x1, 711), (255, 255, 255), 2)
x2 = width-x1
cv2.line(img, (x2, 0), (x2, 711), (255, 255, 255), 2)


y1 = 44
cv2.line(img, (0, y1), (width, y1), (255, 255, 255), 2)
y2 = height-y1
cv2.line(img, (0, y2), (width, y2), (255, 255, 255), 2)


yellow = referance_mapping((-0.29235669, 1.05393455))
cv2.circle(img, yellow, 7, (255, 255, 0), -1)


plt.figure()
plt.imshow(img)
```

Out[ ]: <matplotlib.image.AxesImage at 0x16dd20d30>



We can see with the above lines study that our referance_mapping function effectively maps the yellow ball, value retrieved from the previous experiment, into the correct place. We can now test for all of the balls for a final proof.

In [ ]:
```
ball_poses_colors = []
for i in range(len(balls_3d)):
    ball_poses_colors.append((balls_3d[i][:-1], bpos_color[i][1]))

img = cv2.imread('./data/snooker-table-top-view.png')
img2 = generate_simulation_frame(img.copy(), ball_poses_colors)
show_bgr_image(img2)
```

Finally, with our previous processed video, we can apply all the techniques to generated a real time simulation of the game with the actual ball positions.

In [ ]:
```
video = cv2.VideoCapture('./data/WSC_filtered.mp4')
img = cv2.imread('./data/snooker-table-top-view.png')

frame_width = int(img.shape[1])
frame_height = int(img.shape[0])
fps = video.get(cv2.CAP_PROP_FPS)

out = cv2.VideoWriter('./data/WSC_filtered_simulation.mp4', cv2.VideoWrit
frame_count = video.get(cv2.CAP_PROP_FRAME_COUNT)

pre_mask = compute_pre_mask()

for i in tqdm(range(int(frame_count))):
    ret, frame = video.read()
    if ret:
        test_frame = frame.copy()
        bpos_color = get_balls_position_and_color(test_frame, pre_mask, i
        balls_3d = []
        for ball in bpos_color:
            y, x = ball[0]
            quota = get_quota(x, y, H_2d)
            ball = get_3d_point_at_quota_given_2d(H,nullspace, [x, y, 1],
            _, B = get_normal_and_intersection(ball, light, C)
            balls_3d.append(B)
        ball_poses_colors = []
        for i in range(len(balls_3d)):
            ball_poses_colors.append((balls_3d[i][:-1], bpos_color[i][1])
        img2 = generate_simulation_frame(img.copy(), ball_poses_colors)
        out.write(img2)
```

```
    else:
        break

video.release()
out.release()
```

```
 0%|          | 0/124406 [00:00<?, ?it/s]
```

Here we'll combine both videos in one single video side-by-side

In [ ]:
```
video1 = cv2.VideoCapture('./data/WSC_filtered.mp4')
video2 = cv2.VideoCapture('./data/WSC_filtered_simulation.mp4')

frame_width = int(video1.get(3))
frame_height = int(video1.get(4))
fps = video1.get(cv2.CAP_PROP_FPS)

out = cv2.VideoWriter('./data/WSC_filtered_combined.mp4', cv2.VideoWriter
frame_count = video1.get(cv2.CAP_PROP_FRAME_COUNT)

for i in tqdm(range(int(frame_count))):
    ret1, frame1 = video1.read()
    ret2, frame2 = video2.read()
    if ret1 and ret2:
        frame = np.hstack((frame1, np.vstack((frame2, np.zeros((10, frame
        out.write(frame)
    else:
        break

video1.release()
video2.release()
out.release()
```

```
 0%|          | 0/124406 [00:00<?, ?it/s]
```

https://youtu.be/iHuhpiC64j0