Dice Recognition

Cihan SARI

1 Introduction

The objective of this project is to determine dice count and values on the image using OpenCV and basic image processing tools. Although image dataset consist of color images, as dice are black or white and background is actually gray, all the images are used as gray-scale.

2 Steps

White dice;

- Segment dice bodies
- Segment dice circles inside a bounding box
- Count circles per dice body and add to results

Black dice;

- Preprocessing
- Segment dice circles inside a bounding box
- Group dice circles into dice bodies
- Treat each group as a dice, count circles per group and add to results

2.1 White Dice

Dices may come up very close to one another. Seperating two dices just by using their dice circle locations leads to miscalculations, as there are cases when a dice circle is closer to another circle from another dice. Therefore, I wanted to start by segmenting the dice bodies. I have made a few attempts to segment white dice as:

- Constant thresholding (>200 intensity value)
- Otsu thresholding
- Canny edge detection

However, I was not happy with the individual results. Therefore I ended up using dynamic thresholding followed by an edge detection:

2.1.1 Segment dice bodies

- Expected dice body is 40x40 pixels (calculated from samples). Therefore, to smooth out dice body I have used a mask with size 120x120 (due to nyquist theorem) to average each pixel by their neighbors.
- I have used a dynamic thresholding to filter out noise from the image: $x_{th} = 120, I_m =$ smoothed image

$$I_c(i,j) = \begin{cases} I_o(i,j) & \text{if} \quad I_o(i,j) - I_m(i,j) \geq x_{th} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

- Used canny edge detector on I_c:
  $h_{low} = 5 \quad h_{high} = 40 \quad s_{aperture} = 5$ and then calculated contours.
- To eliminate overlaps or partial segmentations, each contour is converted to their convex hull.

- Each dice body candidate is then checked by their area and rectangularity.

  Rectangularity is measured by the area ratio between the a contour and its smallest enclosing rectangle.

$$rect_{contour} = \frac{area_{smallestrectangle}}{area_{contour}} \tag{2}$$

  All candidates that satisfy following conditions are considered as dice body:

$$\begin{array}{l} diceBodyMaxAreaThreshold = 2000 \ > \ area_{bc} \ > \ diceBodyMinAreaThreshold = 350 \\ rectangularity_{bc} \ > \ diceBodyMinRectangularity \ = \ 0.5 \end{array} \tag{3}$$

2.1.2 Segment dice circles inside a bounding box

- Dice body image (I_{db}) is extracted from original image (I_o).
- Any pixel inside the image with intensity below 150 is picked (I_{bin}).
- Connected component analysis on I_{bin} to get circle candidates.
- Each circle candidate is then checked by their area and circularity.

  Circularity is measured by the area ratio between the a contour and its smallest enclosing circle.

$$rect_{contour} = \frac{area_{smallestcircle}}{area_{contour}} \tag{4}$$

- All candidates that satisfy following conditions are considered as dice circle:

$$\begin{array}{l} diceCircleMaxAreaThreshold = 100 \ > \ area_{cc} \ > \ diceCircleMinAreaThreshold = 10 \\ circularity_{cc} \ > \ diceCircleMinCircularity \ = \ 0.4 \end{array} \tag{5}$$

- Hence, calculated all the **white** dice bodies and their corresponding circles inside (which represent dice values).

2.2 Black Dice

Black dice are much more problematic than white dice due to their bodies' very low contrast from background. I have made a few different approaches to segment black dice;

- Inverting the image and running the same procedure as white dice
- Testing out different dynamic threshold values and masks followed by canny edge detector with various parameters

However, none of them provided good results. All had serious issues to seperate dice body from its own shade. As total number of black dice in the dataset sample was low and there were not two such dice next to each other, I have jumped directly to segmentation of all the remaining dice circles and to use their locations to group them up.

2.2.1 Segment bright dice circles from image

- All the dice circle candidates are calculated by thresholding the original image and then using connected component analysis:

$$I_{bin}(i,j) = \begin{cases} 1 & \text{if} \quad I_o(i,j) < 150 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

- All candidates that do not satisfy conditions from Eq 5 as removed.
- Remaining circles are used to generate another binary image to avoid overlapping.

2.2.2 Segment dice circles inside a bounding box

- Dice circles image is dilated by a square element (7x7) to generate different dice bodies.
- 2.1.2 is used with this difference:

  Rest of the pixels are set to 0 and any pixel inside the image with intensity above 150 is picked I_{bin}.

2.3 Results

All the white and black dice are located, and their circles are segmented. Each dice is then checked for how many circles they have to recognize the value.

3 Assumptions

For dice recognition to be successful:

- Main assumption is to have black or white dices on a gray background. Changing background or dice color could cause decline in performance.
- Eq: 3 and Eq: 5 should be satisfied. Therefore, large changes in dice shape or size are not acceptable.
- Dynamic thresholding (Eq: 1) requires at least 120 intensity difference between white dice and their immediate, averaged surroundings to segment white dice from background, black dice and each other. Therefore, high illumination change is not acceptable.
- Black dice body is **not** segmented. Therefore, dice circle extraction for black dice is very susceptible to noise. Circularity and area limitations are used to filter the noise out, however, there maybe cases where this is not enough.
- Most important of all, black dice circles are grouped together with a simple binary dilation operation. If two or more black dice appear close to each other, this algorithm would not be able to seperate different black dice.

All these assumptions proved correct on partial dataset. However, on whole dataset, one or more of these assumptions may prove false. I have tried to loosen limits as much as I could without affecting the performance. Still, new data may require to add or replace some of these assumptions.

4 Otsu Thresholding

Otsu thresholding is performed by seperating the image to background and foreground for all the intensity values, and measuring spread of the pixel levels for each intensity value. The aim is to minimize the spread of the pixel values from both background and foreground.

4.1 Within Class Variance

Let ω, μ, σ^2 be class weight, class mean and class variance, respectively. For 8 bit image I_o, H represents histogram array, N is total number of pixels in the image and t is the intensity level for which Within Class Variance is calculated. We first calcuate ω, μ and σ^2:

$$\omega_b = \frac{\sum_{i=0}^{t} H(i)}{N} \qquad \omega_f = \frac{\sum_{i=t+1}^{255} H(i)}{N} \tag{7}$$

$$\mu_b = \frac{\sum_{i=0}^{t} H(i)\ i}{N_b} \qquad \mu_f = \frac{\sum_{i=t+1}^{255} H(i)\ i}{N_f} \tag{8}$$

$$\sigma_b^2 = \frac{\sum_{i=0}^{t} (i-\mu_b)^2 H(i)}{N_b} \qquad \sigma_f^2 = \frac{\sum_{i=t+1}^{255} (i-\mu_f)^2 H(i)}{N_f} \tag{9}$$

Then Within Class Variance is calculated as:

$$\sigma_W^2 = \mu_b \sigma_b^2 + \mu_f \sigma_f^2 \tag{10}$$

4.2 Between Class Variance

Calculating σ^2 requires two iterations, therefore is a slow approach. By a bit of manipulation, Between Class Variance (σ_B^2) is far quicker due to its simplicity. Otsu shows that minimizing the Within Class Variance is the same as maximizing Between Class Variance.

$$\sigma_B^2 = \sigma^2 - \sigma W^2 = \omega_b \omega_f (\sigma_b \sigma_w)^2 \tag{11}$$