REPORT

Melih Kağan Özçelik

PART 1:

In the first game of Part 1, to find the maximum numbered dice I followed these steps in order:

1) Take a screenshot, convert it to the grayscale and apply threshold

2) Find contours using cv2.findContours

3) Loop through contours by checking the heights and widths of bounding rectangles

4) Find 3 rectangle corresponds to the 3 dice

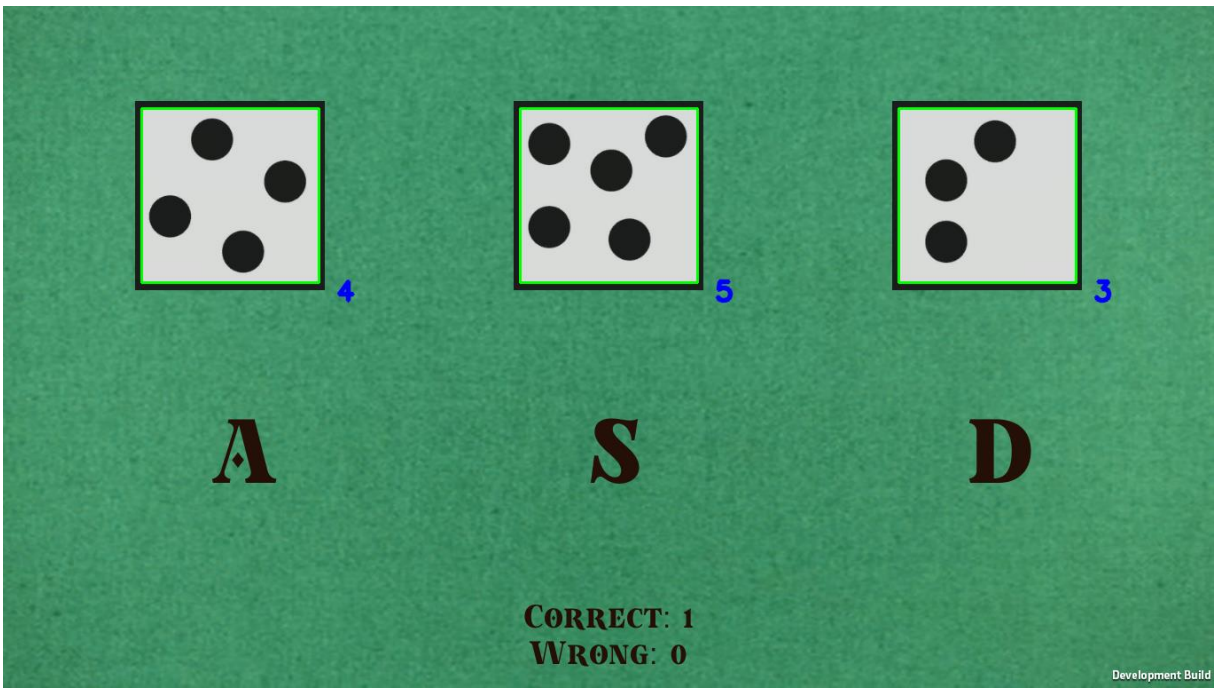
5) Crop these 3 rectangles as an image and apply blur

6) Use cv2. HoughCircles to count number of circles

7) Use pyautogui to select correct dice

8) Return to the Step 1 if pyautogui FailSafeException is not triggered

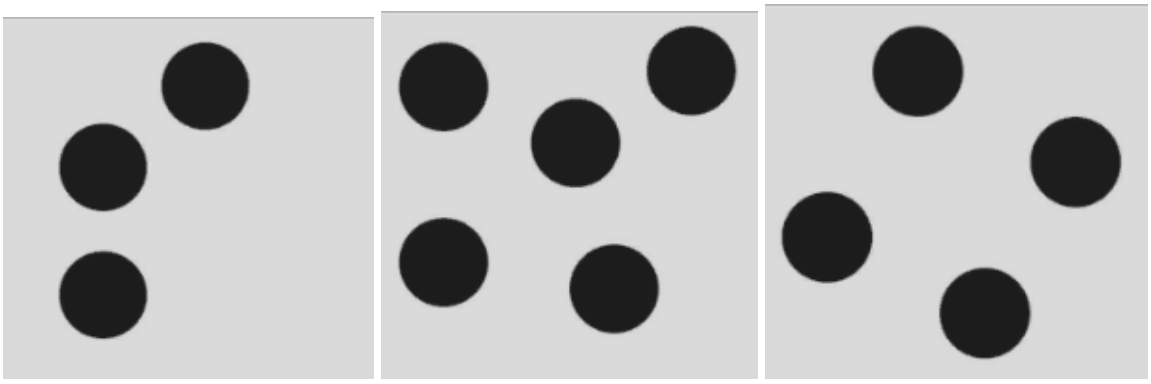
Example Result:



Green borders drawn by using =cv2.rectangle(screenshot,(x,y),(x+w,y+h),(0,255,0),2)

Number of circles =cv2.putText(screenshot, str(num_of_circles), (x+w+20, y+h+20), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 3, cv2.LINE_AA)

Cropped Images of Above Example:



This solution worked with %100 success rate from 0 to 200 correct answers.

In the second game of Part 1, to find the maximum numbered dice I needed to change some of the steps of first game. In this part, I follow this method:

1) Take a screenshot, convert it to the grayscale and apply threshold

2) Find contours using cv2.findContours

3) While looping through contours check:

- The area of the cv2.minAreaRect if it is above 20.000(hardcoded)
- The width and the height of Bounding Rectangle (w,h <280)
- If the center of Enclosing Circle is close to saved centers, this means do not include more than one face from one dice.

4) For every founded dice face, order box points of minarearect

5) Apply perspective transform to the dice face to look like it bounding rectangle as a regular shape

6) Form cropped image from transformed face and blur it

7) Use cv2. HoughCircles to count number of circles

8) Ignore this steps and start over if:

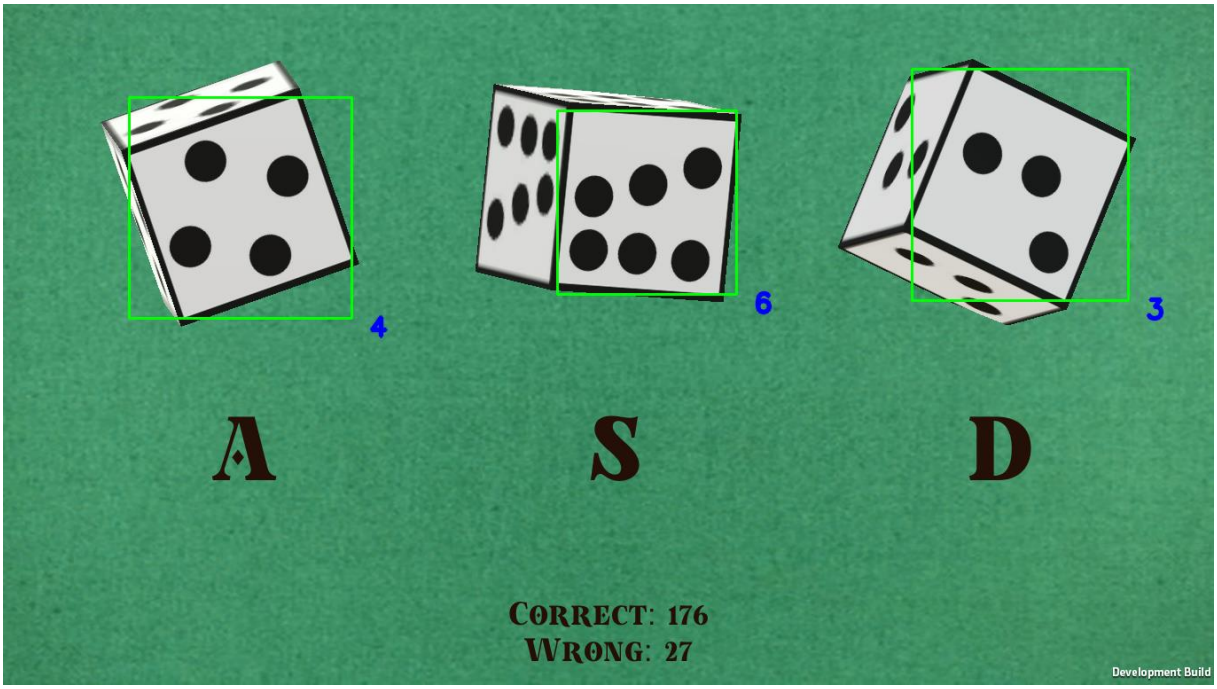
- Cv2.houghcircle cannot find circles one of the faces
- Number of founded dice faces is smaller than 3

9) Find and locate the maximum dice using max_dice_list

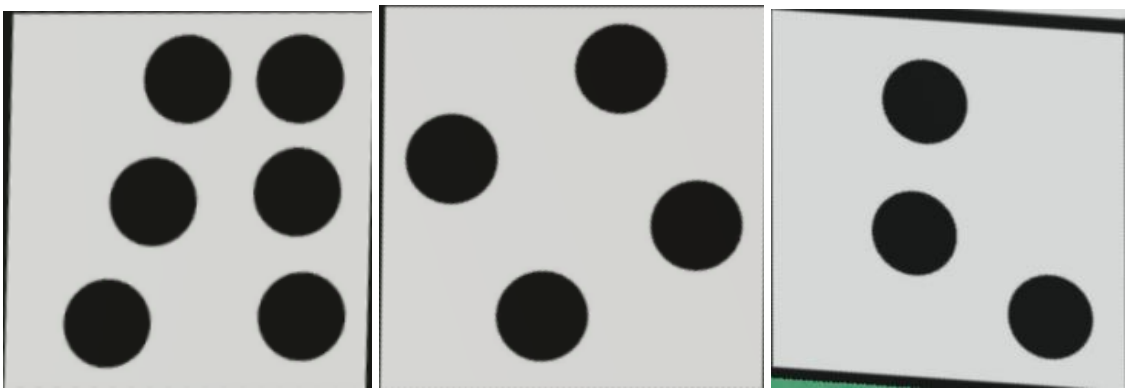
10) Use pyautogui to select correct dice

11) Return to the Step 1 if pyautogui FailSafeException is not triggered

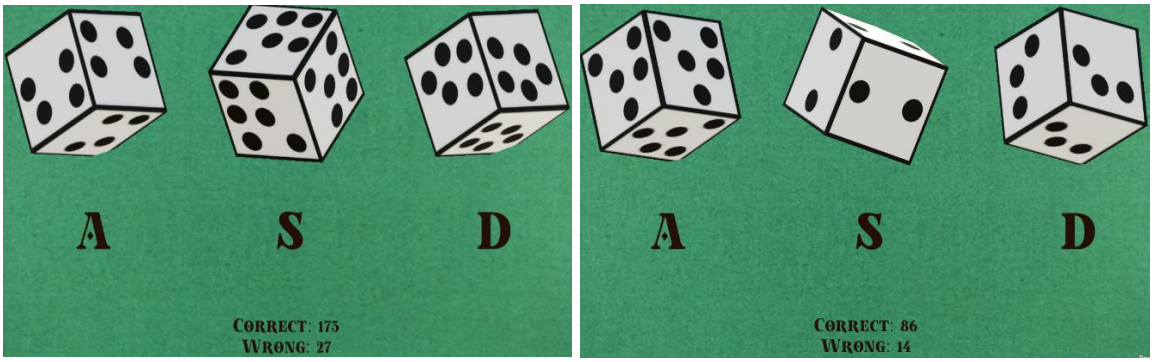
Example Result:



Transformed faces:

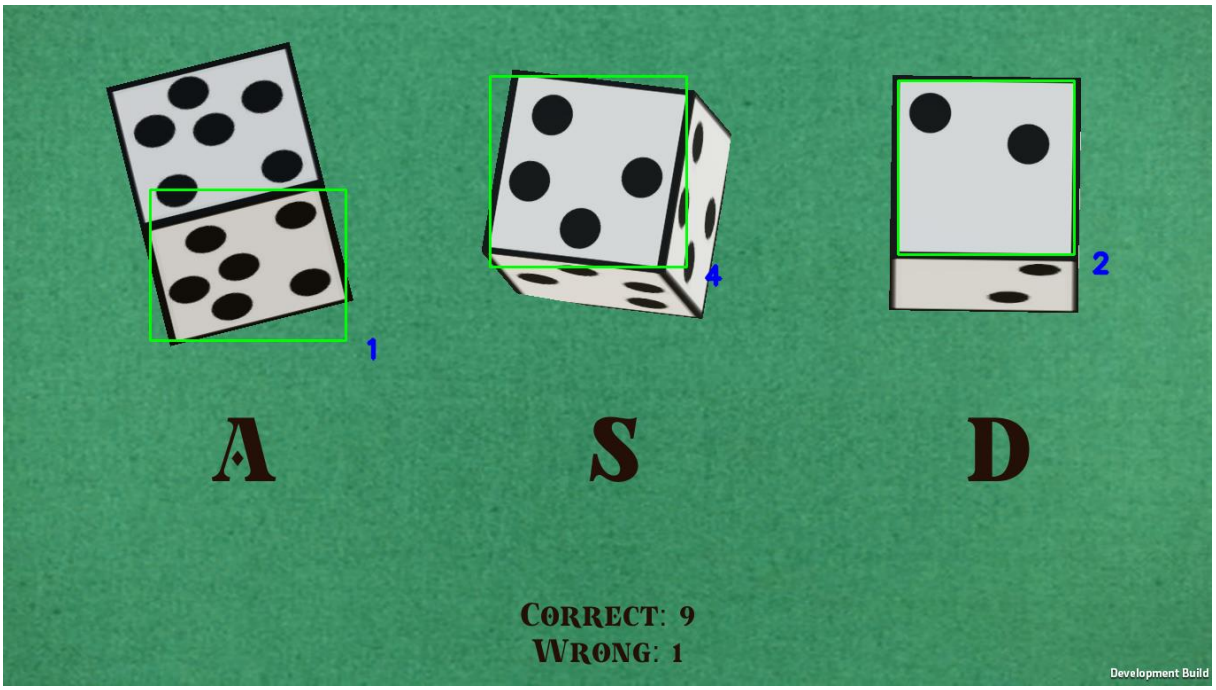


Success Rate:

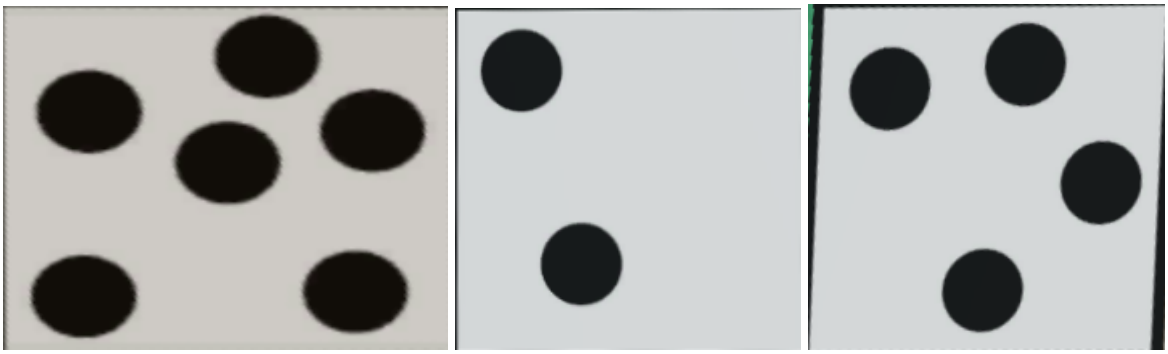


Success rate is approximately %86. Wrong cases usually caused by the ineffective transformed images. Houghcircle gives the number of circles as 1 some of the cases. If the assumption of there will be no dice that numbered one count as true, the results of 1 numbered faces can be ignored too. This assumption can increase success rate above to %90. Example is shown below.

Failed Detection:



Transformed Faces of Failed Detection:



PART 2:

In the second part, to solve mined grid problem, I started to design my solution by deciding how to determine shocked or normal face. To solve that face detection problem, I used dlib with 68 facial landmarks. I predefined normal and shocked faces and got these faces landmarks.

Then, I created a Face Class choosing some facial features like eyebrow length, jawline length.

```
#calculate the length of facial shapes like eyebrows, nose, jawline
def calc_length(points,start,end):
    length = 0
    for i in range(start, end):
        p1 = [ points.part(i).x , points.part(i).y ]
        p2 = [ points.part(i+1).x , points.part(i+1).y ]
        distance = math.sqrt( ((p1[0]-p2[0])**2)+((p1[1]-p2[1])**2) )
        length = length + distance
    return int(length)
#face class for comparing facial features
class Face:
  def __init__(self, points):
    self.left_eyebrow = calc_length(points,17,21)
    self.right_eyebrow = calc_length(points,22,26)
    self.nose_bridge = calc_length(points, 27,30)
    self.jawline = calc_length(points,0,16)
    self.lower_nose = calc_length(points,31,35)
```

Then, I created dictionaries for normal and shocked faces using Face Class attributes.

```
f_normal = Face(points_normal)
f_shocked = Face(points_shocked)
dic_normal = {}
for attr,value in vars(f_normal).items():
    dic_normal[attr] = value
dic_shocked = {}
for attr,value in vars(f_shocked).items():
    dic_shocked[attr] = value
```

Using is_shocked function, I counted the matching attributes (facial features) between real-time face obtained by screenshot from the game and predefined faces. According to the matches, I determine whether the face is shocked or normal.

```
def is_shocked(image):
    image = cv2.cvtColor( image , cv2.COLOR_BGR2GRAY)
    rectangles = detector( image )
    points = predictor( image , rectangles[0] )
    f_curr = Face(points)
    count_normal = 0
    count_shocked = 0
    for attr,value in vars(f_curr).items():
        n = dic_normal[attr]
        s = dic_shocked[attr]
        if ( abs(n-value) > abs(s-value) ):
            count_shocked = count_shocked + 1
        else:
            count_normal = count_normal +1
    if ( count_normal > count_shocked ):
        return False
    return True
```

Before the start trying moves , I locate the our main character dino on the map for once, because location of the character does not changes during the game. I assume that the character will always start from white grid. Otherwise, different threshold value should be applied to locate character. However this possibility is not implemented in the solution.

To makes moves on the grid, I used try_move function which returns true if the target move is possible and made successfully. Try_move function controls:

- End of the grid by checking pixel value (255,111,114) same as background color
- The face by taking screenshots after each keystroke and calling is_shocked function
- If the character enter the targer grid by keeping starting grid color and comparing with the target grid.

To improve try_move function I always tried to keep the character in the center of the grid after move is completed. To provide that ability I spent a lot of time to determine specific keydown times to each direction. However at only one specific grid the shocked face never turns into a normal face therefore the program stucks in that grid. I uploaded the video record

of this situation. To save bot from that situation I used my hands to move character to the bottom grid of that bugged grid. After that bot could be able to reach end of the maze.

In the main loop, while trying moves I kept the last move for backtracking and add every successful move to the path.

Here is the flow of main loop:

- Try all moves except last move
- If there is no successful move go back to the last successful grid by performing opposite of the last move
    - Try possible moves except last move (don't turn back to same place)
- If character is close to end of the map check it and break the main loop

Finally the flow of the try_move:

- Get screenshot of the game
- Determine the control points of grid crossing and end of maze
- Check whether the move targets the out of maze, if so return false
- Press keys according to the move, keep amount of the key strokes
- Check regularly the face after each tiny move
- If face turns into shocked face stop and try go back to center of the grid, return false
- If the edge of grid successfully crossed keep moving to the center of the target grid, return true