Documentation: Mathable Score Calculator

Introduction

This computer vision solution addresses the task of analyzing 200 images from 4 Mathable games, each consisting of 50 moves. During a round, players can make up to 7 moves, forming equations on the board to earn points. A round ends if no more moves are possible.

The objective is to:

- Task 1: Identify the position of the newly added piece on the board.
- Task 2: Recognize the number on the newly added piece.
- Task 3: Compute the score for each round corresponding to a player by summing earned points based on valid equations formed by the new piece placement in each move of the round.

This document outlines the methodology, key implementation steps, and the core algorithms for the tasks.

Preprocessing strategy in the solution

In order to tackle all the tasks, the first step is to extract the game board from the images.

The first step involves preprocessing the image.

I used the **blue channel** because, in this context, the board has a stronger contrast compared to the table when compared to the red or green channels. This makes the board more distinguishable which helps isolate the object.

Next, I applied **blurring** and **sharpening** filters to enhance the features for better contour detection. I used **median blur** to reduce salt-and-pepper noise, as it preserves edges better than averaging filters by smoothing out isolated high-intensity pixels while maintaining the sharpness of the board. I also used **Gaussian blur** to reduce Gaussian noise (random variations in intensity), further smoothing the image and reducing high-frequency details that could interfere with thresholding.

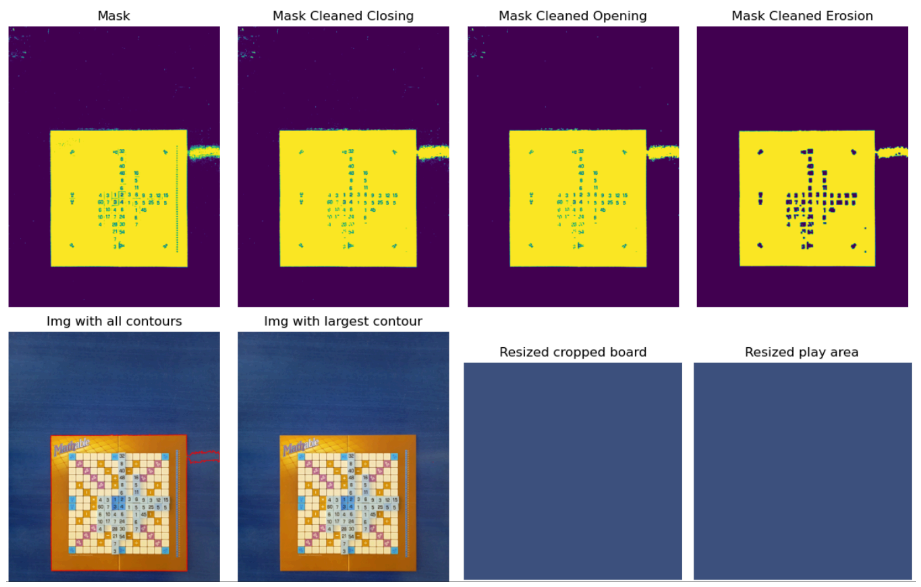
I then applied **edge-enhancing techniques**, such as **cv.addWeighted**, to emphasize boundaries. This was done by blending the median blurred image with the negative of the Gaussian-blurred image).

Then, since the object has uniform intensity, I used **global thresholding** (`cv.threshold`) to create a binary mask. After thresholding, I applied **morphological operations** to refine the mask further. I used a **5x5 ones kernel** with the `cv.MORPH_CLOSE` method to clean the internal areas of the mask (fill small holes), and then the `cv.MORPH_OPEN` method to clean the external parts. Additionally, I applied **erosion** to shrink any noise captured in the mask that doesn't belong to the board. See docs.

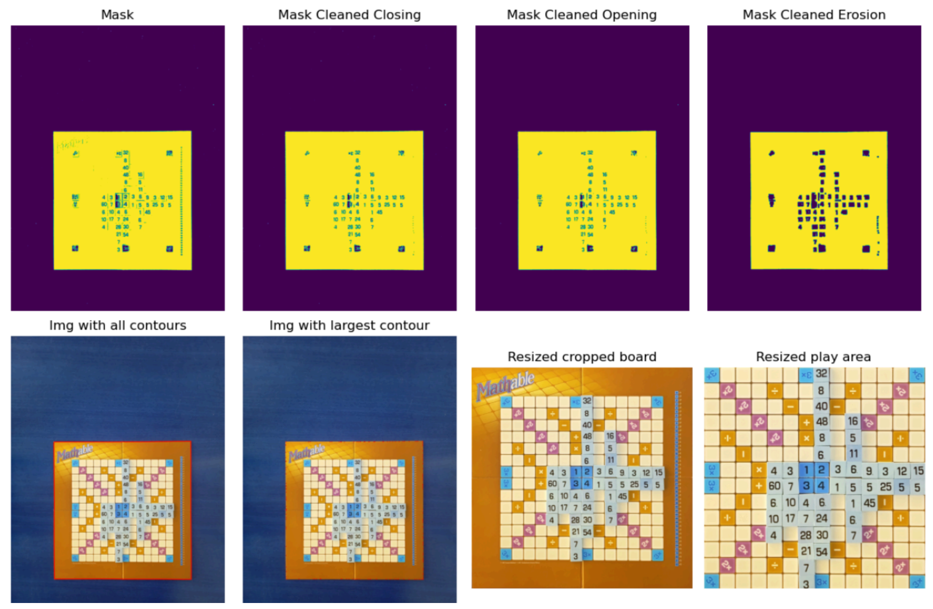
Next, I used **cv.findContours** to detect the contours in the cleaned mask (`mask_cleaned3`). I sorted the contours by area in descending order, assuming that the largest contour would be the game board. I also performed a check to validate that the detected contour has a square-like aspect ratio, ensuring it's the board and not some other object.

I encountered an **issue** in a few images where the lighting was interpreted as a contour. To address this, I increased the thresholding value from 30 to 80, which helped eliminate unwanted noise, as shown in the improved images below.

Binary Thresholding Applied: 50 to 255



Binary Thresholding Applied: 80 to 255



Once the board was detected, I used **cv.boundingRect(board_contour)** to crop it. Then, I resized the cropped board to a consistent width of 1900 pixels and hardcoded the area of interest (playable region) to focus on for the next tasks.

Since the **camera position and the board's position are fixed and consistent**, **perspective transformation** (`cv.getPerspectiveTransform`) was unnecessary, as the board's orientation remains stable across images.

Task1: Identify the position of the newly added piece on the board

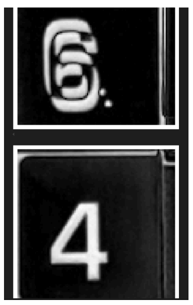
To detect the position of the newly added piece on the board, I utilized the previous and current board images. Both images were first transformed to **grayscale**. Since I had already resized the playable area of the board to a consistent 1400px by 1400px, and the board itself is 14x14, I **cropped each individual tile from both the previous and current boards**.

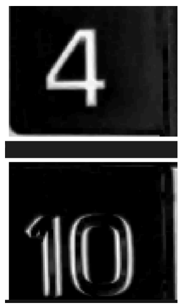
Next, I applied the **absolute difference** between corresponding tiles from the two images. The reason for using the absolute difference is that it highlights the pixel-wise differences between the two images, making it easier to detect any changes or newly added pieces. By comparing the two tiles pixel-by-pixel, I could quantify how much the current image deviates from the previous one.

To identify the **most significant change**, I stored the maximum difference found across the entire matrix in a max variable by computing the mean of the difference (using `np.mean`) from the result of the absolute difference operation (`diffabs`). This allowed me to track the most prominent differences and detect the newly added piece effectively.

This approach **worked for 199 out of 200 images**. However, in one particular case, a tile was slightly moved in the current board compared to the previous one. Even though both images contained the number 6, the difference between the tiles was higher due to the slight displacement. Upon inspecting the case, I noticed that the difference image contained both the internal and external contours of the number, whereas for the correct newly added piece, the number was a filled contour. **To solve this issue**, I applied **erosion** to shrink the white pixels in the difference image. This helped remove the external contours while preserving the filled contours of the number, effectively resolving the issue of false differences caused by minor shifts in tile positions. This solution is demonstrated in the images below.

Before erosion: **After erosion:**





Task2: Recognize the number on the newly added piece

To obtain the piece templates, I manually cropped individual images from the provided "imagini_auxiliare" folder. Each cropped image was selected to contain **exactly** the content of a specific template, resulting in a total of **46 templates** you can find in “templates” folder.

Additionally, for some templates, I intentionally left a couple of rows or columns of pixels at the top, bottom, left, or right edges of the cropped images. This padding was added to enhance the performance of the template matching algorithm, allowing it to more easily distinguish between similar templates. For example, this approach helped to differentiate between visually similar templates such as the number 1 and 11 etc.

To solve the second task, I first transformed all the templates and the cropped tile image into grayscale.
For the actual matching process, I used the **template matching** technique, which involves sliding the template image over the cropped tile image and calculating how similar the template is to each region of the tile image. The function **cv.matchTemplate()** was used for this, along with the **TM_CCOEFF_NORMED** method. This method compares the template with the image at each location by calculating the normalized correlation coefficient. It produces a similarity score for each position, where higher values indicate a better match.

Task 3: Compute the score for each round corresponding to a player

To solve the third task, which involves computing the score for a new piece added to the board, I used a method that evaluates the current board state, the position and value of newly added piecer, checks for valid equations, and computes the score based on these equations.

For each new piece added, I check the four possible equations around the new piece: two vertical equations (one above and one below the piece) and two horizontal equations (one to the left and one to the right). The results from these checks are stored as tuples, which contain whether the equation is valid and the operator used.

After evaluating the equations, I filter the valid equations and, if the tile where the piece was placed is an operator, I ensure that the operator matches the operator of the valid equations to comply with the rules of the game.

The score is then calculated by counting the number of valid equations, multiplying this count by the value of the new piece. Finally, if the tile where the piece was placed is a multiplier, the score is adjusted by multiplying it with the corresponding multiplier value.