Project report for:
Rubik's Cube Faces Recognition

LM − 32, a.a. 2023/2024

Andrea Ceron[1], Ivan Donà[2]
[1] andrea.ceron@studenti.unitn.it
[2] ivan.dona@studenti.unitn.it

Abstract

This paper describes a method for identifying images of a Rubik's cube that show three of its faces. These images can be either created on a computer or taken with a mobile phone camera. The first step in processing these images is to convert them into grayscale pictures with 256 shades of gray. They are then further simplified into binary images, which only use black and white (0 and 255).

The main technique used involves finding the largest semi-circular shape that is not touching the edges of the image, and use it as a Region Of Interest to then locate the cube's external and central corners. After this, the three faces are projected onto a flat surface to begin the color detection process accurately. To further improve the accuracy, the brightness and saturation of the faces will also be normalized.

One of the key uses of this technology is to help colorblind people solve Rubik's cubes. By detecting the colors of the cube automatically, colorblind individuals can also experience the fun and challenge of solving a Rubik's cube.

Introduction

Impact of Rubik's cube color detection on colorblind individuals

For colorblind individuals, this puzzle presents a unique set of challenges, as the game's core mechanics revolve around color recognition. This is where advancements in Rubik's cube color detection technology can make a significant difference. By enabling the identification of cube colors through computational means, this technology offers colorblind individuals an opportunity to engage with the Rubik's cube in a way that was previously inaccessible to them. While it may not revolutionize their lives entirely, it certainly adds a new dimension to their recreational options and allows them to experience the joy and challenge of solving a Rubik's cube, an activity that many others take for granted.

Current state of Rubik's cube color detection technology

In the realm of Rubik's cube color detection, the majority of existing technologies focus on analyzing a single face of the cube. This approach, while effective, offers a limited perspective as needs many images to be taken to capture the complete configurations of the cube. Moreover, the technologies that require only two images often rely heavily on Neural Networks for corner recognition and color detection, which, while powerful, require significant computational resources and complex programming.

Our project takes a different approach by combining the simplicity of single-face detection methods (the lack of Neural Networks) with the complexity of analyzing three cube faces simultaneously. This approach not only lowers the computational demands but also makes possible to develop an easily configurable and explainable Rubik's cube solving algorithm.

In conclusion, our project stands out in the current landscape of Rubik's cube color detection technologies by representing a novel method that balances the simplicity of single-face detection with the comprehensive analysis typically associated with Neural Network-based solutions.

Algorithm introduction

Overview

The algorithm starts with pre-processing steps that aim to standardize the input images. These steps involve transforming images into a square format and adjusting them to a uniform resolution. The following phase involves identifying the Region Of Interest (ROI), which essentially indicates the approximate location of the Rubik's cube within the image. This is achieved through a mix of thresholding and blurring techniques.

Once the ROI is determined, the Rubik's cube is isolated from the rest of the image. Its edges are then approximated using straight lines, and the intersections of these lines define the cube's corners. Thanks to those points the central corner of the cube is also located. The final step involves projecting the faces of the cube onto a flat surface, to which a normalization process is applied for as accurately as possible identify the colors of each of the nine squares on the Rubik's cube's three faces, in different saturation and lighting configurations.

Blurring

In image processing, Gaussian and mean kernels are used for smoothing and noise reduction, but they have distinct effects and applications.

- The Gaussian kernel is based on the Gaussian distribution and provides a smoother effect. It gives more weight to pixels near the center of the kernel and less to those farther away. This weighted averaging ensures that edges and transitions in the image are preserved while reducing noise, making it particularly effective for tasks like edge detection where maintaining structural integrity is crucial.

- On the other hand, the mean kernel applies a simple average across the pixels in the kernel's area. Each pixel within the kernel contributes equally to the final value. This uniform averaging results in a more pronounced blurring effect, which can effectively smooth out textures and patterns in an image.

Edge enhancement

The edge enhancement is accomplished applying a Gaussian blur (the mean kernel would not preserve the edges) to a channel using `cv2.GaussianBlur`, effectively reducing high-frequency components.

Then the algorithm performs a high-pass filter operation by subtracting the blurred channel from the original one. This step is done using `cv2.subtract`. The high-pass filter allows the edges and fine details in the image to be more pronounced since the blurring subtraction has reduced the impact of smaller, less significant variations.

Finally, the resultant image is enhanced for better visibility and edge accentuation through the `cv2.convertScaleAbs` function. This function scales, increase the contrast of the edges via the parameter `alpha`, calculates the absolute values, and converts the result to 8-bit.

The `alpha` value was selected to strike a balance between effective edge detection and minimal increase in noise presence. This is important because enhancing the edges can also inadvertently accentuate the noise.

Polygon approximation

The `cv2.approxPolyDP` function in OpenCV is used for simplifying the shape of a contour by reducing the number of points it contains. To the function is also passed a True boolean value since in our case we consider closed hexagonal-like contours. The key parameter in this function is epsilon, which determines the approximation accuracy.

By setting epsilon as a proportion of the contour's arc length, the approximation becomes adaptive to the contour size. This means larger contours can tolerate a larger absolute deviation, while smaller contours require a more precise approximation. The arc length is calculated via `cv2.arcLength`, by here too passing a True boolean value since in our case we consider closed hexagonal-like contours).

Circle approximation

This method is designed to determine whether a given contour closely approximates a circle, used to determine which contour is more likely the Rubik's cube.

- **Contour approximation:** The function first uses `cv2.approxPolyDP()` to simplify the contour. This function reduces the number of points in the contour, making it easier to analyze. The `epsilon_factor` parameter controls how much the contour is simplified - a smaller epsilon retains more detail, and was taken empirically.
- **Circle area calculation:** To assess the circularity, the function computes what the area of a circle would be if it had the same perimeter as the contour. The formula used is Area $= \frac{\text{Perimeter}^2}{4\pi}$.
- **Roundness ratio:** The roundness of the contour is evaluated by comparing the actual contour area to the calculated circular area. This is the ratio $\frac{\texttt{area}}{\texttt{area\_of\_circle}}$. A perfect circle would have a ratio of 1, as its area would match the calculated circle area exactly.
- **Threshold comparison:** Finally, the function checks if the roundness ratio exceeds a certain empirically taken threshold `roundness_threshold`. If it does, the contour is considered to closely resemble a circle.

Contour filtering

From a binary image are extracted the contours via the `cv2.findContours` function, and it then iterates through all found contours, computing each one's area using `cv2.contourArea`.

The biggest contour that is approximable to a circle over a certain threshold is then copied to a new image with a black background, and will become the Region Of Interest of the Rubik's cube.

Intersections filtering

The primary goal is to retain the intersections of lines that are neither nearly parallel to each other nor too distant from the convex contour, determined by a set

of thresholds. This is accomplished in three steps:

1. The `angular_difference` function calculates the minimal angular difference between two lines, each represented by an angle in polar coordinates. This function accounts for the circular nature of angles, ensuring accurate comparisons. For example, it recognizes that the angular difference between 355 degrees and 5 degrees is 10 degrees, not 350 degrees.
2. The `find_intersections` function is then employed if the intersection angle exceeds the threshold. It computes the coordinates where lines intersect, with each line represented in polar coordinates (rho, theta) as in the Hough Transform.
3. Lastly, intersections close to the convex contour (as per the threshold) are identified. The `cv2.pointPolygonTest` function accomplishes this by measuring the distance from each intersection point to the contour. If the distance is within the threshold, as indicated by a True boolean attribute, the intersection is deemed valid and retained.

Hough line transform

The `cv2.HoughLines` function in the algorithm is used for the detection of straight lines without specifically identifying their endpoints. This approach is distinct from the probabilistic Hough Transform (`cv2.HoughLinesP`), which provides endpoints of lines and allows for more direct control over line detection parameters.

`cv2.HoughLines` is particularly effective for identifying the straight lines of objects, including in our case the Rubik's cube. It operates based on a standard parameter space, detecting lines through a voting mechanism in a finite space of possible line orientations and distances. The key parameters for this function include:

- The minimum width of a series of pixels to be recognized as a line (rho).
- The precision level for detecting the angle of lines (theta).
- The minimum number of votes required for a line to be acknowledged.

Perspective analysis

This method allows for accurately locating the central point across various heights and angles at which the photo might be taken. The implementation composed of three main steps:

1. Extending the cube's top right and bottom left edges to infinity, and identifying their convergence point. Then is created a new line, that intersects this convergence point and the cube's top left corner.
2. Similarly, the bottom right and top left edges are extended to identify another convergence point, from which a second line is created, which intersects this convergence point and the cube's top right corner.
3. The central point of the Rubik's cube is then determined by the calculating intersection of these two newly created lines.

Implementation of algorithms and procedures

Input image assumptions

The input images for the algorithm need to meet certain criteria to ensure accurate Rubik's cube color detection:

- **Standard 3x3 Rubik's cube:** The images should feature a standard 3x3 Rubik's cube with the typical color scheme. This ensures consistency in color recognition across different images.
- **No colored manufacturer logo:** Ideally, the Rubik's cube in the image should not have a colored manufacturer's logo. Said colored logos can lead to incorrect color detections, as the algorithm might misinterpret the logo as part of the cube's color pattern.
- **No excessive angle shots:** The images should be taken straight on, with no excessive inclination or tilt. This means the camera or phone used to take the photo should be directly facing the cube, possibly without also being too distant from it.
- **Cube separation from edges:** It is important that the Rubik's cube is fully within the frame and not touching the edges of the image. Additionally, there should be no objects (like cables) connecting the cube to the edges of the image. This separation is crucial for the algorithm to correctly identify and isolate the cube from the rest of the image.

Figure 1: An ideal photo of a Rubik's cube

Image preprocessing

- **Image set to 300x300:** This standardization involves extending the border pixels of the shorter sides to create a square shape, and then by resizing the newly squared image to the desired resolution. This process is needed because

a higher resolution not only increases computational demands for subsequent steps but also introduces a considerable amount of irrelevant information. Excessive details, like noise patterns, can interfere with and complicate the processes that follow.

- **Edge enhancement:** Then the algorithm enhances the edges in the image using a high-pass filter, which is applied individually to each of the three primary color channels - red, green, and blue. After processing these channels separately, their results are combined to form a full BGR image (blue, green, red) [4].
- **Monochromatic conversion:** This image is converted in a gray image, consisting of an single channel ranging from 0 to 255 (from pure black to pure white).
- **Adaptive thresholding:** After the conversion is applied the `cv2.adaptiveThreshold` function. The function calculates the threshold for a pixel based on a small region around it, so it's ideal in situation like in the Rubik's cube where a face may receive more light than the others.

  The function employs a Gaussian threshold instead of an mean threshold, as the aim is to emphasize the edges. To achieve this, a relatively large pixel neighborhood size is used. Using a lower value would amplify the noise, while an even bigger value would result in thinner borders, diminishing their prominence.
- **Removal of unnecessary information:** Finally a Gaussian blur is applied to decrease the impact of the low intensity noise from the image, accentuated by the previous step, by giving it a more brighter value.

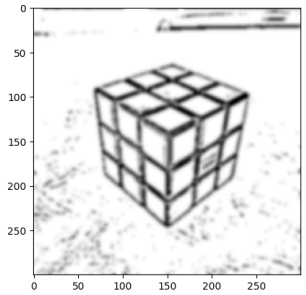

Figure 2: The Gaussian blur maintains the cube's edges and lowers the noise in the image

Separation from fine noise

- **Normalization:** After applying Gaussian smoothing to the image, the algorithm normalizes the smoothed image to have pixel values ranging from 0 to 255. This normalization is a preparatory step for the subsequent binary thresholding.
- **Binary Thresholding:** The normalized image is then subjected to inverted binary thresholding. This technique is designed to highlight the edges of the Rubik's cube, previously assigned a dark gray value in the Gaussian-smoothed image, by setting them white. Conversely, the rest of the elements, such as background noise (which appears as light gray), are transformed to black. For example, subtle details like wood grain, which might show up as light gray post Gaussian smoothing, are rendered black, thus effectively eliminating such noise from the image.

  This step is critical because the `cv2.findContours` function interprets white as the foreground and black as the background. If the background were white, it would be mistakenly identified as an object, leading to incorrect contour detection.

  In contrast to `cv2.adaptiveThreshold`, where the threshold varies across the image, this method employs a uniform threshold across the entire image. This threshold has been determined empirically to ensure optimal differentiation between the Rubik's cube and its background for accurate contour detection.
- **Contours Detection:** The algorithm then applies the OpenCV's `cv2.findContours` function to identify all potential contours (borders) in the binary image, including those of the Rubik's cube and any residual noise.

  The function employs the `cv2.RETR_LIST` parameter, as it's crucial to locate the Rubik's cube positioned at the center of the image. This approach ensures that we do not limit our search to just the outermost contours, allowing for a more accurate identification of the cube within the image.

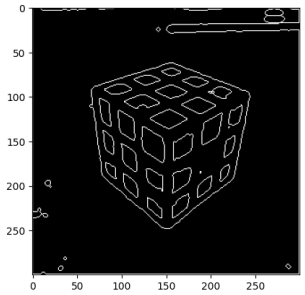

Figure 3: It remain in the picture only the cube, other objects and the strong noise

Separation from rough noise

- **Drawing black lines on image edges:** The algorithm begins this phase by drawing very thin black lines (2 pixel wide) along all four edges of the image. This step sets the stage for the subsequent filling process.
- **Filling with white from edges:** Starting at the bottom left corner within the boundaries of

the newly created black lines, the algorithm proceeds to fill the rest of the image with white.

This method of initiating the fill from the black lines permits to have a wide coverage, ensuring that the fill extends from all potential border pixels of the image. This approach is particularly effective in cases where an object near the border divides the image into subregions, a scenario where starting the fill from a single location would have been insufficient.

Conceptually this action tries to change all of the background to white. This is important since most of the residual rough noise consists of white spots on a black background, so this step effectively removes them. Only objects with a hollow structure, such as the Rubik's cube, other items present in the image, or very intense noise patterns (which must be quite large and thus have a white border with at least a black interior pixel) will remain visible in the image.

This happens because the white fill does not penetrate the interiors of these objects, as they are encircled by white borders. As a result, the Rubik's cube (along with other objects and significant noise) will still be visible, distinct from the background [1].

- **Re-Smoothing on enlarged image:** The algorithm inverts and extends the image to 500x500 and applies another round of smoothing. The enlargement is done to ensure a uniform transition at the edges of the image, preventing any sudden changes that might occur due to noise or edges of the Rubik's cube near the image end. The inversion process turns the objects white, aligning with the convention that white signifies the foreground and black denotes the background.

  Meanwhile, a mean kernel smoothing process is applied to remove the smaller black-bordered squares, which represent the individual squares on a Rubik's cube face. This is done by assigning these squares a value greater than zero. Notably, the Gaussian method is not employed here, as the focus is solely on entirely filling the Rubik's cube's area rather than maintaining the sharpness of its edges.
- **Thresholding and noise reduction:** The algorithm implements a simple thresholding technique, where any pixel exceeding a empirically determined brightness threshold is turned white, while the rest are set to black. This method effectively fills the area representing the Rubik's cube and simultaneously reduces the amount of residual noise in the image.

Rubik's cube *Region Of Interest*

- **Isolating the Rubik's cube:** Then, the algorithm identifies the Rubik's cube location by assuming that is the largest white shape against the black background, which approximates a circle beyond a set threshold. Consequently, it retains only this object, discarding all others. Any smaller white shape, which is approximable to a circle over the threshold, is treated as residual noise.

  The rationale for using a circular approximation threshold is to distinguish the cube from larger objects in the image, such as a chair in the background. While these objects might be larger than a Rubik's cube, the cube's rough hexagonal shape is more similar to a circle compared to other objects like said chairs or even bottles. This step ensures that the algorithm focuses exclusively on the Rubik's cube location, now clearly separated from other elements in the image [4].
- **Expanding the Rubik's cube location:** The final step in this phase involves enlarging the shape of the Rubik's cube approximated location. This is achieved with a dilatation technique, which effectively extends the white part of the image into the black areas. This step is crucial for smoothing out any irregularities along the edges of the cube. It specifically targets pixel-deep indentations or flaws, which could affect the precision of subsequent steps.

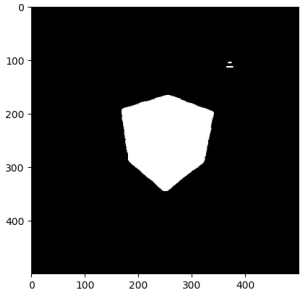

Figure 4: The aforementioned steps maintained the cube, objects and extreme noise

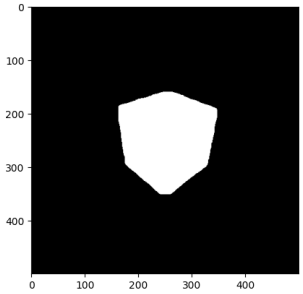

Figure 5: It remain in the picture only the ROI, without extreme noise or other objects

Rubik's cube isolation from image

- **Projection on to the original image:** The first step consists of projecting the ROI in to the

expanded original image using a `bitwise_and` operation. In this process, the black background is retained in the new image, while the white interior of the ROI acts as a transparent layer, allowing the Rubik's cube in the original image to be visible.

- **Adaptive thresholding:** Following its conversion to a monochromatic format, the algorithm applies adaptive thresholding to the isolated image of the cube. This approach differs from the first adaptive thresholding, that was utilized in the preprocessing step, and is key to both reduce the noise and produce a high-quality binary representation of the Rubik's cube.

  This applied adaptive thresholding, even when used with identical parameters of the previous one, excels in distinguishing the cube from surrounding noise, such as the wood grain of a table on which the cube is placed.

  The effectiveness of this separation is caused by the vicinity of the ROI's pure black border with the cube's edges, which are black too (but not as dark as the ROI). Consequently, background noise previously visible around the cube in the initial thresholding is almost totally eliminated, turning this space to pure black instead of the cube, ROI borders and occasional noise pixels which are set to pure white.
- **ROI border removal:** For the following, the algorithm extracts contours from the binary image using the `cv2.findContours` function. By also employing `cv2.RETR_EXTERNAL` as the parameter, the function only retrieves the outermost contour, which corresponds to the ROI's border in this scenario, excluding the cube and the noise.

  The algorithm then performs a black fill starting from one of the pixels on the outermost contour (the ROI border). This process effectively leaves only the Rubik's cube and the occasional noise pixels in the image.
- **Cropping to original size:** After having isolated the cube, the algorithm crops the image back to its original, standardized 300x300 size. This reduction in dimensions is crucial as it significantly facilitates the following computational processes.
- **Dilatation for gap closure:** For the final step, the algorithm ensure the Rubik's cube is perceived as a single unified object, by applying dilatation. This step is aimed at closing any gaps in the cube's binary representation, which might have resulted from the thresholding process.

Rubik's cube edges retrieval

- **Convex hull creation:** The algorithm uses the `cv2.convexHull` function to create a convex hull around the Rubik's cube, detected as the contour with the highest area (and thus ignoring the remaining noise). This convex hull is a contour that tightly wraps around the cube. The importance of this function lies in its ability to ignore small gaps along the cube's borders, ensuring that the resulting shape is a seamless hexagon.

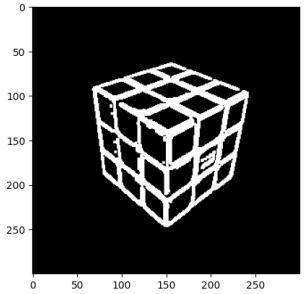

Figure 6: It remain in the picture only the cube, with a very high quality border

- **Assigning lines to edges:** The algorithm then aims to assign distinct lines for each edge of the hexagonal shape created by the convex hull, with the `cv2.HoughLines` function, designed to identify lines within the convex contour. The threshold, a parameter of the function, is dependent on the dimensions of the cube since the number of intersections needed to recognize a line changes according to the resolution. The other parameters are instead static, but like the threshold empirically determined.

  Intentionally, these parameters generate multiple overlapping lines for each edge of the cube. This redundancy is a strategic choice, as it's better to later filter out excess convergence points than to risk runtime errors due to potentially missing lines on any of the cube's edges.

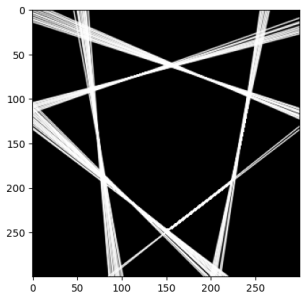

Figure 7: Image showing the lines that are assigned to each cube's edges

Rubik's cube corner retrieval

- **Generation of the corners:** The algorithm filters intersections to pinpoint the locations of the cube's corners. It locally averages these points, derived from a subset of all potential edge approximations, to determine the actual coordi-

nates of the corners. This is executed in two steps:

The DBSCAN algorithm clusters points that are densely grouped within a specified threshold, and labels as outliers those that are in low-density areas. This threshold must not be set too high, because then there would be a risk of merging distinct corners. The minimum number of points required to form a cluster is also considered, aiding in differentiating between actual corner clusters and noise.

Following this, the labels assigned by DBSCAN are used to identify members of each cluster. The centroid of each cluster is then calculated by averaging the coordinates of all the points in the cluster.

After determining all intersection points, the algorithm returns the identified corners of the Rubik's cube.

- **Sorting the corners:** The algorithm sorts the identified corners of the Rubik's cube. This sorting is done vertically (top to bottom) and horizontally (left to right) across the image. Assuming the image is taken within an acceptable inclination, this sorting determines the positional relationship of each corner with the Rubik's cube structure.
- **Calculating the central point:** The algorithm proceeds to calculate the perspective from which the photo was taken. This step is crucial for determining the position of the central point, which varies based on the cube's orientation relative to the camera. For instance, a photo taken from a higher angle (making the upper face appear taller) will position the central point lower, meanwhile a photo taken more level with the cube (making the upper face appear flatter) will place the central point higher [2].

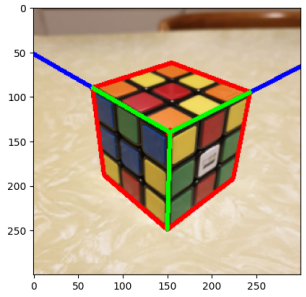

Figure 8: In red: the border, in yellow: the center separation, in blue: the perspective calculation lines

Face projection and color detection

- **Separating the three faces:** With the central and external corners located, the algorithm now focuses on identifying the three visible faces of the Rubik's cube. These faces are then projected onto three separate square images. This projection removes the perspective distortion present in the original photo, laying out each face flat, which greatly facilitates the subsequent color detection process.
- **Conversion to HSV format:** The algorithm converts the three faces of the Rubik's cube from the BGR (Blue, Green, Red) format to HSV (Hue, Saturation, Value). This conversion is needed to take account for varying environmental conditions, such as low illumination or limited color recognition by the camera, which would render difficult a BGR only color detection.

  This is possible since the color information is independent of the saturation and brightness value. Still, to reliably recognize the color white, a normalization of the saturation and value components of the HSV is utilized [3].
- **Most common hue in each square:** For each small square on the cube's faces, the algorithm calculates the most common hue, saturation and value. This is done using a circular mask, which isolates the square from the rest of the image, by coloring everything that sits outside the circle to pure black (also called a ROI).

  The algorithm executes this process in the function `most_common_hue`. Within this function, it assesses whether the average saturation falls below a specified threshold and the average brightness exceeds another threshold. This step is crucial for determining if the color being identified is white, as white does not fall within the hue range.

  If these conditions are met, indicating the presence of white, the function returns a negative value as a flag, otherwise it proceeds to return the actual average hue value.
- **Hue-based color identification:** The algorithm employs the `hue_to_color_name_and_bgr` function to assign both a BGR value and a text label to the identified colors. For white, the function provides a straightforward output configuration, meanwhile for other colors, it examines the specific hue range in which the color sits and then returns the corresponding text label and BGR value pair.
- **Scaling and displaying the results:** Once the algorithm has processed all nine squares of each Rubik's cube face, it enlarges the images of these faces to enhance visibility. This scaling is essential as the text indicating the color labels would otherwise be too small and difficult to read.

  Following this, the algorithm overlays black-bordered transparent circular markers, representing the Regions of Interest, along with the corresponding identified color values (shown as small colored circles) and textual color labels.

This final step effectively showcases the results of the entire process.

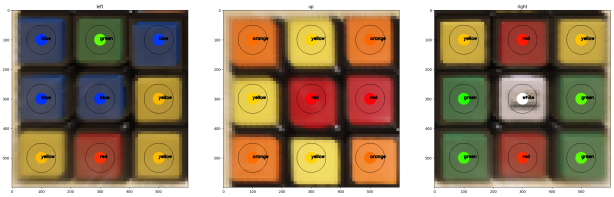

Figure 9: From the left to right, we have the leftmost, upper and rightmost face color identification

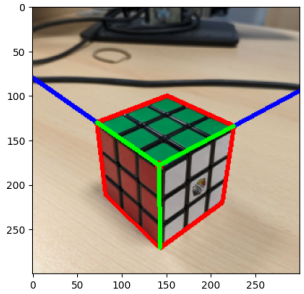

Figure 11: The cube gets detected, despite the table's noise pattern

Experiment's Results

Positive aspects

- **Good separation from clutter:** The algorithm demonstrated its capability to identify and analyze Rubik's cubes in images where multiple objects were present. This achievement is attributed to its ability to select objects based on size and their resemblance to a sphere.

  Thanks to this feature, the algorithm is versatile in processing various types of images. It can handle not only digital (clip-art) images but also real photographs with complex backgrounds. This adaptability means that there is no requirement for the background of the photos to be completely empty, allowing for a wider range of image conditions to be effectively analyzed.

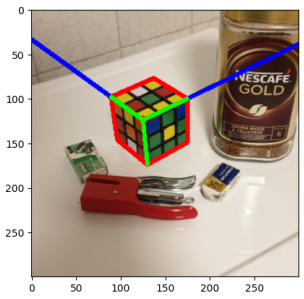

Figure 10: The cube gets detected, despite other objects being present

- **Good separation from noise:** The algorithm demonstrated a good reliability in identifying and analyzing Rubik's cubes within images in the presence of noise patterns such as the grain of a wooden table. This capability was specifically achieved through a ROI focalized adaptive thresholding.

  This last step, and the techniques utilized to generate the ROI, collectively enabled the algorithm to effectively distinguish the Rubik's cube from the background in a binary image, ensuring accurate separation despite the presence of distracting patterns or noise in the environment. (Figure 11)

- **Implementation technique:** The development of this color detector, which does not rely on Neural Networks, has been an excellent medium in deepening our knowledge of various functions within the OpenCV 2 library.

  Opting for this approach also streamlined the computational demands. Moreover, it highlights the effectiveness of traditional image processing techniques in solving complex tasks, since it shows good accuracy in this type of projects even without resorting to Machine Learning algorithms for optimizing coefficients.
- **Adaptability to different cube designs:** The algorithm, since it does not use Neural Networks, and uses as the detection method the edges of the cube instead, can easily be adapted to recognize the color configuration of Rubik's cubes of various designs, like triangular shaped or square shaped with a higher amount of internal cubes. This is not possible with NN since it would require the complete re-training of the model.
- **Precision in color detection:** The algorithm's ability to normalize the faces of the Rubik's cube significantly improves color recognition accuracy. This precision is observable in both digital images, which benefit from ideal lighting and saturation, and in real-world pictures, where normalization makes a substantial impact.

  Additionally, the strategy of averaging the hue, saturation and brightness over a specific area proves effective even when cube faces are not perfectly aligned. This averaging ensures that the correct number is identified, provided the majority of the area is still within the correct target region, thus compensating for any minor overlaps with adjacent squares. (Figure 9)
- **Precision in central corner detection:** The algorithm accurately identifies the central corner through a method based on geometric derivation, thus avoiding the use of coefficient-dependent formulas and making possible to have highly precise results without the need for calibration. Consequently, the algorithm can effectively process images with varying elevations and angles,

as long as the Rubik's cube is not excessively rotated. (Figure 12)

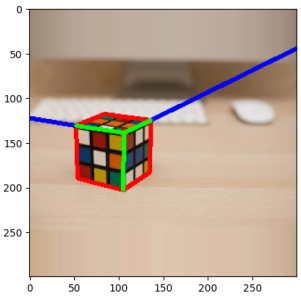

Figure 12: Faces' detection with a different photo angle

Limitations and Challenges

- **Edge detection failures:** In cases where the Rubik's cube edges were not visible enough (or not present, like in the case of border-less Rubik's cubes), or touching the image edges (directly or for example via a strong shadow and thus indirectly), the algorithm failed to correctly identify the cube. This limitation is due to the reliance on the filling method, which requires the cube to be isolated within the image.
- **Perspective issues:** Excessive inclination or rotation of the cube in the photos led to difficulties in estimating the corners correct order and the cube's central point, crucial for accurate face recognition.
- **Corner issues:** Having overly smooth corners on a Rubik's cube makes tricky to determine the lines to assign to each edge via the `cv2.HoughLines` function.
- **Background color:** Having a black colored background makes difficult estimating the black colored edge of the cube.

Conclusion

This project has successfully developed an algorithm capable of recognizing and analyzing the color configurations of three faces of a Rubik's cube simultaneously, under a variety of photographic conditions. Demonstrating a unique approach that does not depend on neural networks, the algorithm highlights the effectiveness and relevance of traditional image processing methods in solving complex tasks.

This approach can be divided in several key features: the enhancement of edges using Gaussian and high-pass filters, the extrapolation of the Region Of Interest (ROI) from noise and other objects in the image, the utilization of the ROI for accurate separate the Rubik's cube. Then techniques such as `cv2.convexHull` and `cv2.HoughLines` are employed to define the cube's structure via new lines.

Subsequently the algorithm filters the intersections of these lines, by checking if the location of one this points is near the convex hull and if the angle is greater than a given threshold. If so, that particular intersection is used in the clustering during the generation of the cube's corners. It then uses geometric properties to identify the central corner (thus making possible to handle images taken from different angles and heights), essential for separating the three faces of the cube.

After the separation a normalization process is applied, to adapt the color detection to varying environmental lighting conditions, and camera characteristics. Finally the algorithm determines the color of each square on the cube's faces, using hue analysis, instead of the weaker method that relies only on BGR value readings.

Despite its strengths, the algorithm encounters limitations in scenarios where for example the cube is not isolated in the image, its edges are too close to the image ends or in cases of extreme lighting and perspective. However, the successes outweigh these challenges, particularly in controlled environments or professional photography settings.

Instead, the next phase of the project should focus on enhancing the algorithm's precision and versatility: using Machine Learning for optimal parameter adjustment, improving the algorithm to detect the Rubik's cube in more complex environments without isolation, and expanding its capability to analyze the other half of the cube for implementing a resolution feature.

In conclusion, the algorithm stands out for its precision in edge detection, perspective analysis, and color recognition. Its adaptability to different image qualities and robustness in handling background noise and minor clutter make it a good contribution to the field of computer vision, particularly in applications that require accurate color detection without the complexities of Neural Network-based approaches.

References

[1] Alexander Craggs. "How do I split up thresholds into squares in OpenCV2?" In: *Stack Overflow* (2020). Accessed: 28-01-2024. URL: https://stackoverflow.com/a/64567585.

[2] Bill Martin. *Perspective and the Cube*. Accessed: 28-01-2024. URL: https://guidetodrawing.com/drawing-the-basic-shapes-and-forms/perspective-and-the-cube/.

[3] Nursabillilah Mohd Ali. "Performance Comparison between RGB and HSV Color Segmentations for Road Signs Detection". In: *Applied Mechanics and Materials* 393 (Sept. 2013). DOI: 10.4028/www.scientific.net/AMM.393.550.

[4] Justin Ng. *Automated Rubik's Cube Recognition*. Accessed: 28-01-2024. URL: https://stacks.stanford.edu/file/druid:yj296hj2790/

Ng_Rubiks%20Cube_Reconstruction_from_Images.pdf.