Rubik's Cube Project

Sam Bekkis, Océane Charlery, Pierre Hubert-Brierre, Judith Millet
ATIV project report / ID3D master's degree / Group 09 and 10

I. Introduction

This project was implemented in Python using the OpenCV library as well as Tkinter libraries for the interface and Cython for the performance. The main purpose was to be able to detect the faces and the squares of a Rubik's Cube in 3D space from a webcam video.

II. Demo

To launch the demonstration from your computer, you can follow the following instructions :

1) Install the following dependencies (you may install them with pip, pip3 or homebrew depending on your environment) : tkinter, opencv, cython
2) Open a terminal window located at the root project of the project and run the following commands (you may run "python3" instead of "py" for the first command) :
   a) `py setup.py build_ext --inplace`
   b) `python3 detect2.py`
3) Place your Rubik's Cube in front of your computer's webcam and watch the face and square detection displayed in real time on the interface.

Since our program is based on edge detection, watch out for rectangular objects in the background such as ceiling lights, as these could disturb the detection. The Rubik's Cube should be relatively close to the webcam to get the best results.

III. Failed attempts & possible upgrades

failed attempts

We have initially tried to detect a cluster of color without using any edge detection (just extracting red pixel, yellow, ...) this work a bit if the lighting of the image is perfect, but as long as there is a face darker than it should be, it didn't work. We tried using different color spaces, but none were satisfying.

Upgrades

We could go a step further and try to match every detected squares with color. In order to do that we could convert our RGB colors into HSV colors and try to match every squares with one of the 6 colors of the original Rubik's Cube.

IV. Cube structures

A Rubik's Cube is made of 54 squares divided into 6 faces.

We implemented a structure to model the Rubik's Cube features :

- The number of faces
- The number of squares
- The input image path
- The output image path

V. Detection method

Our objective was to detect as many squares as possible to try to infer the cube structure and know the number of visible faces. To achieve that, we proceed image per image, performing a series of operations on each image to detect the squares.

We start by applying a Canny edge detection on our image (we are actually performing 3 edges detection, one on each color channel because it leads to better results). After that, we used an erosion-dilatation algorithm to erase artifacts created by edge detection. We then gather each pixel into clusters of connected pixels (delimited by the edges) and try to match every cluster with a parallelogram. (This is technically false because the projection of a square is NOT a parallelogram, but this is accurate enough to give some results).

Once we have extracted the clusters that look like a parallelogram, we gather them by similarity (those that have the same rotation and scale) and assume that they are on the same face. This gives us the number of faces, but also the structure of a face. Because the method is not perfect, some squares will not be detected. Knowing that a Rubik's Cube face is made of 9 squares, we can try to guess where are the missing squares by finding a homography that maps our clusters with a 3 by 3 grid. In this grid, it's easy to find missing squares and guess their location, we just have to apply the homography inverse to have the location of our missing squares in the image.

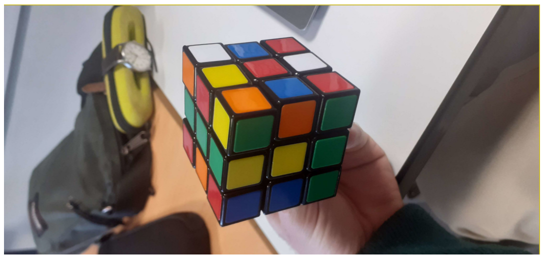


Figure 1. Reference image

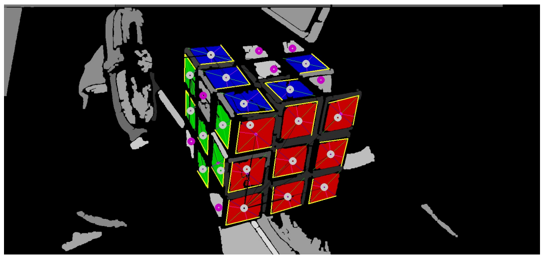


Figure 2. Image after treatment. Gray clusters are the clusters detected by edge detection, the lighter they are, the closer to a parallelogram they are. Red, green, and blue clusters are clustered very close to a parallelogram, they are supposed to be squares of the Rubik's cube (clusters of the same color are similar to each other). Finally, the white dots are the located squares and the purple dots are the guessed squares.

VI. Performance

During the development, we had to determine clusters of connected pixels as explained before. To do it, we used a function called "compute_clusters" that for each pixels of the picture and the edges image can detect a cluster of points.

At the beginning, this function did the job in approximately 12 seconds for each picture. That's a lot of time for a single function. This is why we worked on reducing its execution time.

Firstly, the algorithm. We removed and reorganized the code. That made us win 6 seconds. But 6 seconds of execution time is still too long. So we found a way, using library called Cython. That's a python file that can be compiled. To make it work, small changes have been done, and finally we finished with an execution time at approximately 1.5 seconds with our test image.

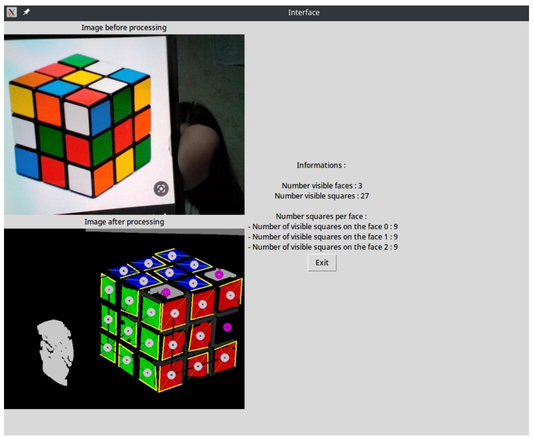
VII. Interface

Since Tkinter's main window's loop simulates a for loop, we could not use a typical while loop to analyse each frame of the webcam video stream for the features detection.

To bypass this constraint we implemented a main function that uses global variables and we make sure to only update the detected features after the TKinter's main loop window update. That way, we are able to avoid conflicts that would result in an unresponsive GUI.

We update the data displayed on the TKinter's window every two seconds depending on the framerate : if the video has a framerate of 24fps we analyse the frames of the video stream with a step of 48.

Since the features detection takes about 1 second after optimization to compute per frame, this refresh rate allows enough time to get the updated features data before displaying on the interface but remains fluid to watch.



```
Informations :

Number visible faces : 3
Number visible squares : 27

Number squares per face :
- Number of visible squares on the face 0 : 9
- Number of visible squares on the face 1 : 9
- Number of visible squares on the face 2 : 9
```

Figure 3. Display interface.

The initial webcam video is displayed normally, only the result update each 1 second.

VIII. Perspectives

With this method, we have the position of each small squares relative to the image being processed.

Thus, in order to generate a 3D Rubik's Cube, we could scan the entire cube to retrieve the position and color of all the squares of the Rubik's Cube, if possible. This can be done by following the orientation of the cube from the central small squares (which will never change) or by specifying on the interface which face we want.

If we are unable to capture all the squares, it is possible to deduce some cubes based on the overall configuration, or to leave them neutral while waiting to detect them later.

In this way, we will obtain the color of almost all squares for each face and thus, it would be possible to create a 3D model of our cube.

IX. Conclusion

By completing this project, we will obtain the detection of visible faces and small squares of a Rubik's Cube from a webcam video. Indeed, the detected squares are assigned to faces but moreover they can allow us to find other implicit squares that would not have been detected. The result allows to validate the method and to give a satisfying visual.