Rubik's Cube Solver

June 21, 2014

Yakir Dahan
yakir.dahan@post.idc.ac.il
Efi Arazi School of Computer Science
IDC Herzliya

Iosef Felberbaum
iosef.felberbaum@post.idc.ac.il
Efi Arazi School of Computer Science
IDC Herzliya

Google Play link: `https://play.google.com/store/apps/details?id=com.rubik.cubesolver`

Abstract

The Rubik's cube is a well-known puzzle invented in 1974 by Hungarian sculptor and professor of architecture *Ernő Rubik*. In the following study we're about to present an android application that solve one the Rubik's cube. Our goal is to create a simple application to help the user solve the cube, by scanning the cube in its current state and provide the user with a step-by-step guidance for solution of the cube. This application would have to handle two main issues - detection of the cube's structure and colors,and finding a possible solution process from its current state.

1 Introduction

The Rubik's cube is a famous puzzle around the world and is popular among both children and adults. This is a challenging game, and even though an algorithm for solving a cube's instance was already found in the 80's, it is still not an easy task for everyone. The purpose of this project is to create an application which can "understand" the user's current cube state and help him solve it by providing a step-by-step guidance. It is also important to make the application simple as possible - make it easy and intuitive to scan the cube's structure and current state (colors) and also to understand exactly what is the next step in the solution process.

2 Android Application and GUI

We used the android platform for developing the application. For the image processing we used OpenCV's java interface for android. No server is required for the image processing, nor for the process of finding a possible solution - the whole process is running on the device, which makes it independent and does not require network access.
We will now present the main activities of the GUI for our application.

2.1 User Manual

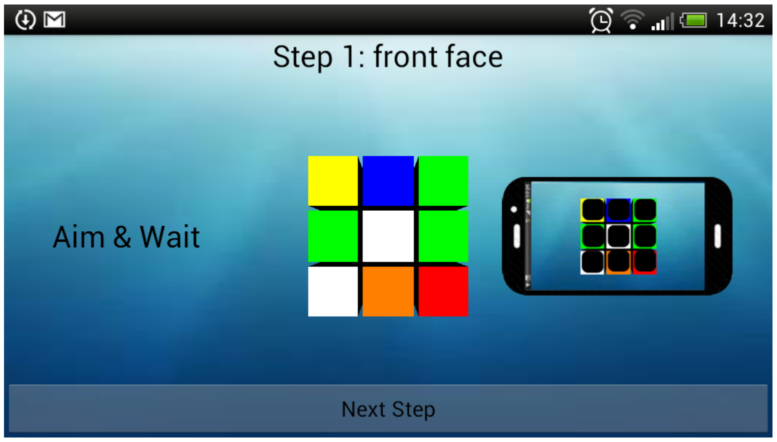


Figure 1: User Manual

One of our main concerns is to make the application simple and easy to use and understand. Therefore, we added a manual which describes the process of scanning the cube. The manual is important since we assume there's a specific order for scanning the cube, therefore for the application to run, the user must know and understand the correct order of cube sides scanning.

2.2 Cube Scanning

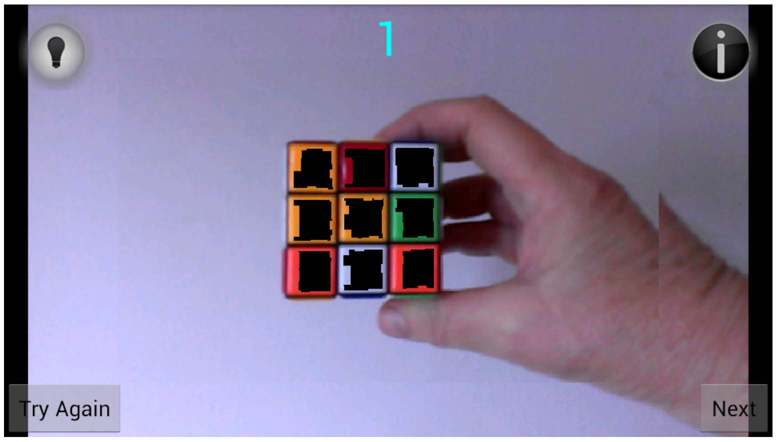


Figure 2: Cube's Side Scan

Due to bad lightning conditions or a noisy environment, the scan of the current side can detect contours which are not a valid square in this side. The user must understand what is a good scan, and when he should try to scan the current side again. Therefore after the application scans a side and finds nine candidates for squares - it first draws them on the screen and asks the user to approve them. This way the user can assure the squares were identified correctly. The whole process is described in the manual mentioned before.

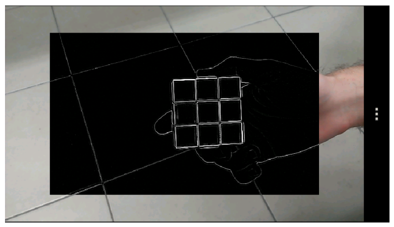
3 Implementation

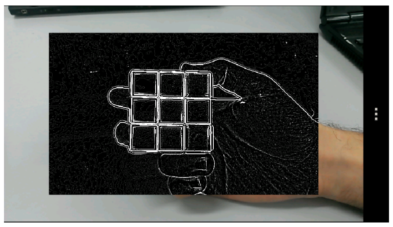
3.1 Cube Recognition Process:

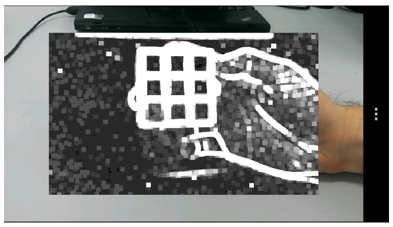
The first step in each scan of a cube's side is to recognize the locations of inner squares in the current side. The process for each image captured of a cube's side is composed of a series of image manipulations - *laplacian, scaling, dilate* and *finding contours*:

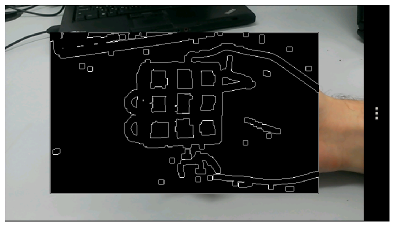
- *Laplacian* highlights regions of rapid intensity in the frame, therefore it is used to highlight the edges between the squares found in the current side of the cube
- *Scaling* (using *convertScaleAbs* in *OpenCV*) returns an 8-bit unsigned integer array filled with absolute values after scaling the image. This gives us a frame with sharper edges, which enables us to identify the cube's side edges more accurately
- *Dilate* takes the source image, received from scaling, and perform a dilation process - using a structuring element, it takes the maximum value over the neighborhood it creates. In our case, the best structuring element will be a square. This will return a frame with thicker edges, which will make it easier to identify contours as the squares in the current cube's side
- *finding contours* which finds all the contours in the dilated frame

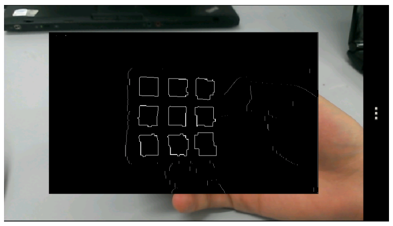
After contours were found, we filter out the squares in the image - only contours with a difference between their area and their bounding rectangle's area which is smaller than a predefined threshold are chosen (that means, only the contours with the area close enough their bounding rectangle's area). When **nine squares** are found in the image the user is presented with the result found in order to assure that all the nine squares in the current side scanned were perfectly scanned. After that, the user can move on to the next side of the cube.
In this step we only find the squares in each side, and do not attempt to find their colors. This will be done in a different process, after scanning the whole cube (will be explained later). Moreover, we require specific order of scanning the cube's sides in order to build the cube model correctly.

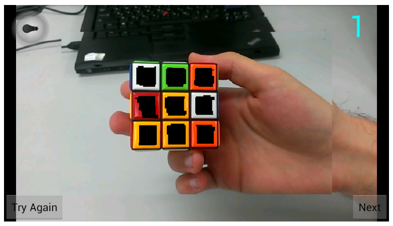

(a) Laplacian


(b) Scaling


(c) Dilate


(d) Find Contours


(e) Filter Squares



(f) Final View

Figure 3: Capturing Cube's Side Process

3.2 Colors Recognition Process:

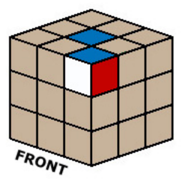
After scanning all the cube's sides we now know where to find each side's inner squares. In order to find each square color, we need to know the six colors available for the Rubik's cube: red, orange, yellow, blue, green and white. However, defining these color's RGB values in advance can be inaccurate when using running the process in an environment with bad lighting conditions - in this case, red and orange can be easily mixed up, and we get an illegal instance. Therefore, we used the fact that each side's central square has a different color - overall 6 different colors, which build the whole cube - in order to define the RGB values for the cube colors. Each square will now be compared to these six values in order to identify each square's color.
First, each central square in each side is taken to identify its RGB values. This is done by taking the center point of the square and creating a small rectangle around it. This area will determine the whole square color, by calculating the mean RGB value in this area. The mean value calculated for each central square is a valid color in the current instance.
Next, we move on to the other squares scanned before. The same process of calculating their mean RGB value is done, however this time we'll create different color options for each square, sorted by their *Euclidean-distance* between the square's mean RGB value and the known RGB values calculated before (and considered to be the possible colors for the cube). Then, we iterate over the options, taking the best option currently available (which means, the option with the smallest *euclidean-distance* from a known color), and erase other options which are now cannot be possible (for example - three squares which belongs to the same corner cannot be of the same color, therefore when deciding a color for a square which belongs to a corner, the options for the other squares in the corner with the chosen color are removed). We can summarize the process in the following algorithm:

1. For each facelet, we take the average color value of a sub-square in its middle as its initial color.
   (a) Set the initial color of the cube centers as the final 6 colors of the cube - which makes the process color-independent .
      i. Calculate the 3D Euclidean-distance of the cube facelets from each center (by colors) and put it all in a queue $Q < f_i, d_i, n_i, c_i >$which holds the distance d_i of facelet f_i form center n_i whose color is c_i - ordered ascending.
      ii. $MainLoop$(while $Q \neq \emptyset$):
         A. Pop the top element $< f, d, n, c >$from Q and set the final color of f to be c .
         B. Remove all elements in Q where $f_i = f$.
         C. Remove all elements in Q where f_i is a neighbor of f and $c_i = c$.
         D. Increment the counter of the color c .
         E. If the counter of the color c reached 9, Remove all elements in Q where $c_i = c$.

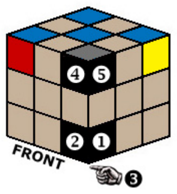
3.3 Finding a solution

In order to solve the recognized instance of the cube, we implemented a simple Rubik's cube solving strategy originally developed by Denny Dedmore. In this 7-steps-algorithm, we search for a certain state in the cube and then do some actions to take it to the next desired state. This way, the algorithm solves each layer of the cube, until it is complete.

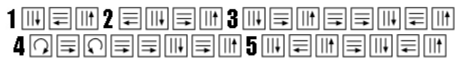
- Setup: pick a corner cubie and rotate the middle faces until the center of the upper face is the same as the upper side of the selected corner cubie



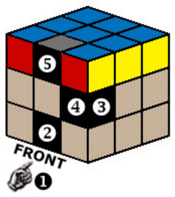
- **Step1: Place the corners of the upper face in their right position.**
  - In order to do that, we need to rotate the lower face until the corner cubie with our desired top color is in one of the marked locations (the numbered locations represent the possible locations of the blue facelet)

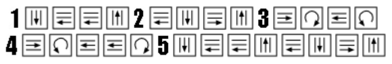


  - then we choose one of the following algorithms to move the cubie to the top and flip it around to line it up properly
  - Note that the target cubie may already be in its proper row position but not face the right way, in which case algorithm 4 or 5 should be used

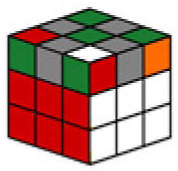


- **Step2: Place the edges and finish the top layer.**
  - In this step, we search for the proper edges (blue and red for the cube in the example) in three possible locations. This search requires checking the upper row (for algorithm 5), rotating the middle row for algorithms 4 and 3 and finally rotating the lower row for algorithms 1 and 2. As in the previous step, the numbered locations represent the possible locations of the blue facelet

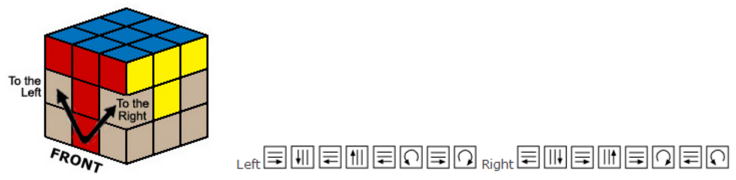




  - In the end of this step, the entire top layer of the cube is solved
- **Step3: Solve the middle layer.**

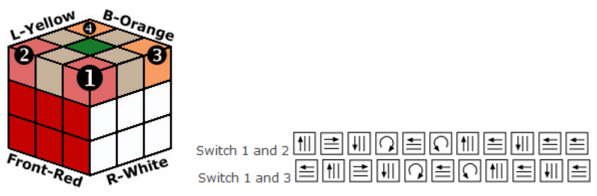


- This step, starts with rotating the middle layer until the center cubies in the middle layer match the rows above them
- Next, we rotate the middle layer until its middle cubie matches the center above it and form a T
- Then we shell check the bottom of the lower middle cubie to decide to which direction we should send this cubie (in the example it's Yellow to the right, otherwise to the left)



- In the end of this step, both the top and middle layers of the cube are solved.

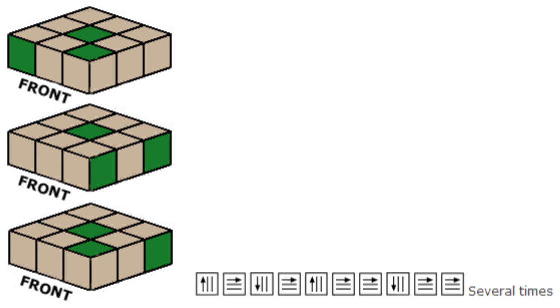
- **Step4: Turn the cube Over and Arrange the Last Layer Corners.**
  - First we turn the cube upside down so that the unsolved layer is on top.
  - Next we arrange the corners of this layer to their correct positions, although they may be flipped and not face the right way yet (we deal with that in the next step).
    To do that, we have 2 algorithms for swapping corners. We use them by deciding which corners we want to flip, placing them as corners 1, 2, or 3 (4 is out of the game in this step) and using the supplied swapping algorithms



  - In the end of this step, all upper layer corners are in place and may only need to be flipped

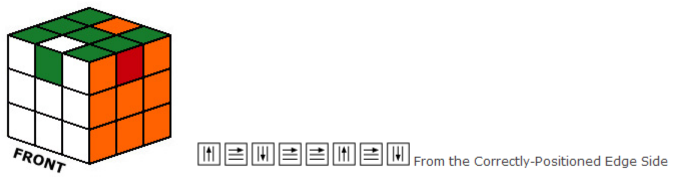
- **Step5: Completely Finish the Last Layer Corners.**

- In this step, we flip all the last layer corners into their final positions
- To do that, we need to find one of the three configurations shown below and perform the supplied algorithm
- We will probably have to perform the algorithm several times and will need to use at least 2 of the 3 different configurations to continue



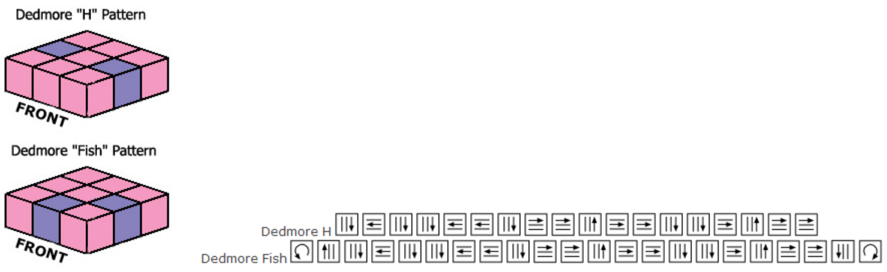
- If we can't find one of the configurations to begin with, we perform the algorithm once and then one of the configurations appear

- **Step6: Completely Finish 2 Edges and Prepare the Remaining Two.**
  - In the beginning of this step, there should be at least one edge in the upper layer in its final position, although not necessarily turned around correctly,
  - First, we turn the cube around so the side with a correctly positioned edge is the front
  - Then we perform the repositioning algorithm to move the rest of the edges into their proper places (up to 2 times)



  - If we can't find a correctly positioned edge to begin with, we perform the algorithm once and then proceed as usual

- **Step7: Solve the Rubik's Cube.**
  - At this point, there should be 2 edges completely solved and 2 edges unsolved. The 2 unsolved edges are properly positioned and simply need to be turned around
  - The 2 unsolved edges should form either a "Fish" Pattern or an "H" Pattern
  - Accordingly, we shall perform one of the following algorithms



- If all four edges are flipped after Step 6, perform the ”H” Pattern algorithm once and then proceed as usual

- **Finally, we added another step of our on, in which we remove redundant steps.**
  - Since the whole algorithm is based on searching for patterns, in some cases we may not find what we expected and then try to search somewhere else
  - We wouldn't want the users to do all these useless steps, so we take the solution steps and look for groups of steps that cancel each other. For instance, a rotation of a face to the left (as a part of the search process) and then rotating the same face to the right (as the first step in the appropriate algorithm). Same goes for 4 rotations of the same face in the same direction as a result of a failing search that happened
  - This way, we managed to reduce the number of step to less than half its size (~300 to ~130)

4 Results

Overall, the application works well and it is easy to use and understand. The process of colors recognition works good under fair lightning conditions, mainly due to the fact that we use the center of each side as a color in the cube (and do not assume constant RGB values in advance). Recognition also works when the background is not constant (can have different colors, patterns, etc.) - however it may require the user to rescan each side several times till all the squares are identified.
The process of recognizing the colors takes ~2 seconds, since it requires only simple calculations (handling a queue of options, calculating euclidean distances). Adding a queue of options and decide the colors using not only distances from known values has increased the number of successful recognitions. We also used elimination of options based on the cube's properties to reduce chance for false recognitions.
Finding a possible solution for the instance is also very fast, and takes 1-2 seconds. However, the solution found is probably not the optimal solution, and may have redundant steps, for example -

1. Turn the left column up
2. Turn the middle column up
3. Turn the right column up

simply represents a change in the cube's perspective, and has no effect at all in the progress of solving the cube.
The main bottleneck in the whole process is the recognition of the cube's squares. While being easy and intuitive, the user can find himself scanning the same side several times due to unsuccessful capturing of all the nine squares in the current cube's side. This can be reduced using assumptions on the shape of the cube - taking only nine squares with approximate constant distances between their centers (and the constant

will be the length of the square's edge), for example. This kind of heuristics will be considered in out next version of the application.

References

[1] Andrej Karpathy. Extracting sticker colors on rubik's cube, project report. http://www.cs.ubc.ca/ andrejk/525project/, May 2010.

[2] Wikipedia the free encyclopedia. Rubik's cube. http://en.wikipedia.org/wiki/Rubik's_Cube, June 2014.

[3] James Yates. Rubik's cube solution. http://www.chessandpoker.com/rubiks-cube-solution.html.