Title page

abstract

Table of Contents

Introduction to the project

What we want to do.

Why did we choose this idea?

Body:

Researched information

Main source (solvethecube.com)

Notation

Inspired from program

The proposed solution

3 steps to run the program

User input

Solve

Return solution to user

Will use the wiki to complete this section

The actual solution

Classes and how they interact

Graphical interface explained

Screen shots of the final form

Abstract

This technical report contains the method and thought process that has been involved in the creation of a Rubik’s Cube solver. The project consists in using the Qt C++ framework to write a graphic application that solves Rubik’s Cubes and the Scope Statement we wrote earlier this year contained the main functions this application should accomplish. We determined the final product should: allow the user to input a cube via GUI, solve the input cube, and output the solution in a *playable* fashion. We have managed to produce an application that can perform those three tasks; the program uses the Fridrich algorithm to output to the widget window, which contains a cube in 2D isometric, the solution to the user’s unsolved cube they input earlier. Potentially this program could be improved to use other algorithms, and using OpenGL to show the cube in 3D.

Introduction to the project

What do we want to create?

We were asked to come up with a project idea that would have us use the Qt framework and its graphical user interface mechanisms as thoroughly as possible. The first few ideas that came to mind were game ideas because they are very demanding in GUI but we were not very enthused by the fact that we would have to be working with C++, for C++ isn’t suitable for games. So we thought about it a little further and the idea which came to us as the most interesting, that uses both GUI and C++ effectively, and that would produce a *useful* product is solving a Rubik’s Cube using a computer. The potential in that project lies in the fact that it would have to interact with the user in such a way that the latter can give the current status of their unsolved Cube and the program will solve the cube and output the series of moves to the user in a comprehensive way. The difficulty of this project lies in the implementation of the algorithmic hidden behind the resolution of the cube and using C++ lightens the amount of resources required to solve the cube, for this language handles RAM very carefully.

*Why did we choose this idea?*

*It actually came from the fact that Quentin and Thomas have learned how to solve Rubik’s Cubes over many years and have become very good at it; so good, that Quentin’s fastest cube resolution lies around the 20 seconds mark. As for Karim, he decided it would be very interesting to create an application that will have a utility beyond a simple game without any. This project is more a challenge than it is simply creating a program that uses GUI.*

Researched information

The first step in starting this project was to do some research. We conveniently found a very diligent website called SolveTheCube that gathers many methods to solve Rubik’s Cubes in varying levels of difficulty. The method proposed by this website for “speedcubing” is called the Fridrich resolution, also known as CFOP, for it is the method used globally for solving cubes as fast as possible, but requires the *speedcuber* to learn and memorize 115 different algorithms each adapted for a very specific state that the cube is in. This means that the method is very case-based and the *speedcuber* has to verify constantly what is going on with the cube to know which algorithm to apply.

Notation

There exists an international notation for Rubik’s Cube solving also found on SolveTheCube. The basics are the following:

Moves are defined by the initials of the face it has to be executed in. For instance, the move U means that the Up face (relative to the way the cube is held) has to turn a quarter turn clockwise, and the move U' is a counter-clockwise turn of the Up face. Furthermore, the U2 move represents two clockwise quarter turn moves of the Up face.

For more details and a more in-depth explanation of the notation used for this project, the [notation](http://solvethecube.com/notation) page of SolveTheCube has all the information needed.

Inspired from program

As we didn’t know where to start with our project we looked over the internet if there were programs that already accomplished what we were trying to make. We found a few different programs written in different languages C#, Java, and C++ and the open source program Rubix Cube was definitely the more user-friendly program, therefore more inspiring GUI-wise. Since this program is open source we had a look at the code to see how the classes are implemented and what algorithm it uses to solve the cube. It has a function to solve the cube in the “god number” of moves, 20. Mathematicians and engineers have worked for years to finally discover that the maximum number of moves required to solve a cube as shuffled as it can be is 20 (more information at the following [link](http://www.cube20.org/)). The code was way too complicated so we decided to go from zero and outline our project as follows:

Scope Statement

|  |  |
| --- | --- |
| Must Have | Nice to Have |
| * An input handled on a graphical interface.   + The user inputs each square colour individually * An Algorithm that will take the unsolved cube and returns the moves it has used to solve it * The interface will be “playable,” showing each move one after the other   + With the list of movements using the standard notation used by mathematicians (U-R-L-D-B-F). | * Detect faces of the cube using uploaded pictures.   + Video treatment * The user can choose between different algorithms, or by default the program chooses the fastest accordingly. * Displaying the cube in 3D using Qt-OpenGL * Display the moves in motion for clearer understanding by the user. |

The proposed solution

The table on the previous page is the scope statement we decided was feasible within the four and a half month period we were given to create and develop the application. This section describes how we planned to accomplish each of the three functionalities.

The Input

|  |  |
| --- | --- |
| A Rubik's Cube consists of, in fact, 26 smaller 'cubies' as oposed to 54 colored stickers; this means that when a move is executed, there are 8 cubies that rotate around the central cubie. As illustrated in the image on the right.  As shown, the yellow, red, and blue axes (each colored by the central cubie of that face) do not move, only the up **U** face has turned counterclockwise **U'**. Subsequently, the orange and two yellow stickers on the blue, frontal face **F** in the first image end up in the same order on the red, right face **R**. |  |

Since the axes don't move that means that of the 26 cubies, the central ones don't move, which leaves us with 20 cubies (because there are 6 faces to a cube) that can move. Furthermore, these 20 cubies can be separated into two groups: corners and edges. Corners have three different colors and edges have two different colors, and each one of these cubies is unique. This means that we will have to handle the user's inputs in order for them to be possible; there can only be one blue-red edge like there can only be one yellow-green-orange cubie.

The program, Rubix Cube, has an input interface in a T-shape (Fig.X.X) where every time the user clicks on a square he may enter a color. We chose to use this method because it seemed like a really understandable way to input a cube and code-wise it would fit really well with the abstract representation of the cube.



The Output

The program will then take the state of the cube and the algorithm (look at the Algorithm section) will solve and output the set of moves. These moves will then be shown one after the other on the graphical interface. Now, as the input isn't very representative of a 3D cube, and as we have decided not to use three dimensional graphics, we had to find a compromise of both dimensions.

|  |  |  |
| --- | --- | --- |
| The first idea we came up with is using isometry. *“Isometric” refers to some form of parallel projection where the viewpoint is rotated slightly to reveal other facets of the game environment than are visible from a*[*top-down perspective*](http://en.wikipedia.org/wiki/Top-down_perspective)*or side view, thereby producing a three-dimensional effect.* As defined by wikipedia, but it is simpler to say that an isometric view is a projection of a three dimensional object on a two dimensional plane. | |  |
|  | |  |
| The second idea was to show the desired face of a cube that means there is no 3D or a resemblance to a 3D cube. The user would be able to rotate and show the face they would like to see. |  | |

Both these solutions have a problem though; holding a cube in your hand makes it very easy to look at any face of the cube and navigating through the cube to properly see each face is quite problematic when working in two dimensions-- there will always be three or more faces that you cannot see. Working with transparency could be a solution but the problem with that is that it can confuse the user more than it will help.

So, we are considering working with isometry. It makes more sense to work like that because the user will understand very well how to hold the cube in his hand with respect to the image the program is outputting.

The Algorithm – Fridrich

The cube resolution algorithm we are going to implement was invented by Jessica Fridrich because it’s the favorite algorithm for most “speed cubers” (people who spend atrocious amounts of time learning to resolve cubes as fast as possible) for its incredible efficiency and its moderate difficulty to learn. The main problem with this algorithm is that it requires knowing a lot of different sequences of moves that are difficult to master. Thankfully, computers can handle this “knowledge” and shouldn't be an issue for our program.

Moves

To handle the sequences of moves we decided the program would have methods that take as parameters a string withholding the list of moves. This means that we will have to code a decoder so that a string containing “F U R U' R' F'” actually performs the correct set of moves: F followed by U, followed by R, followed by etc. We chose to also implement each move individually, as there are only 18 possible moves on a cube {F, F', F2, U, U', U2, R, R', R2, L, L', L2, D, D', D2, B, B', B2} and considering FURLDB(2) are simply FURLDB launched twice, then there really only are 12 moves to code.

Representation

There is one last point to talk about in this section which is how the program will implement the cube. We have opted for an object system whereby the class Cube will have as child the class Cubie. The cube class will contain 26 cubies and their position, whereas the Cubie class will contain the one, two, or three colors assigned to the cubie and its orientation. Now, the orientation of the cubie will have to be handled carefully because when a move is executed, the cubie has to be facing the direction it should be facing as it would on a real cube.

Cube

To represent the cube in code form we have thought up two different methods. Both these methods are quite similar but don't handle the same.

*3x3x3 Matrix*

A 3x3x3 matrix that will contain all the cubies and the position of a cubie will be represented with a 3D vector e.g (1,1,1). Slight optimisation issue with this method-- “useless” cubies (centre cubies) are a possible position vector.

*3 3x3 Matrices*

Essentially the same model as the 3x3x3 Matrix-- the difference lies in the fact that we're working on each layer individually in this model as oposed to the previous one.

Cubie

Edge

Edge Cubies are slightly easier to handle because they can have only two possible orientations-- the first color on the right or on the left of the face it is in. As the position of the cubie is determined by the parent class, it is always known and therefore the orientation is relative to the position.

Corner

Corner Cubies are slightly more complex than Edge Cubies as they withhold three colours and therefore three possible orientations.

The actual solution

This section is written in chronological order of development. The core to the program, the only way for the program to function, is the virtual representation of the cube. A single cube has to be instantiated and used throughout the lifetime of the program. Naturally, we had to start by creating the Cube and Cubie classes.

Color

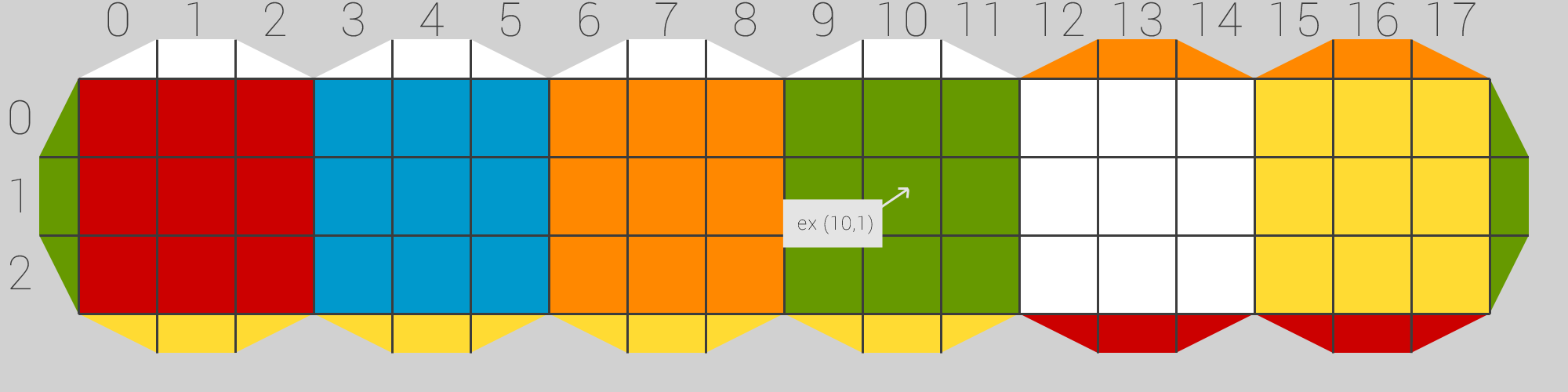
This class we had to create as a namespace. It contains an **enum** with an association color -> integer value.

|  |  |
| --- | --- |
| UNDEFINED = -1  RED = 0  BLUE = 1  ORANGE = 2 | GREEN = 3  WHITE = 4  YELLOW = 5 |

Cube

This is the more important class. As mentioned above, it creates a full **cube** object using smaller cubies. Before getting into more detail, there is a notable change between the implementation of the cube we had decided during the planning phase of the project and what is actually used now—originally, the cube was supposed to be a 3x3x3 matrix composed of cubies, but to work with such matrices in C++ made it quickly very complicated and difficult for us to keep up with the program and what the cube is doing. At this point we decided to change our virtual handling of the cube from a 3x3x3 matrix to an 18x3 matrix.

Visually this is what the cube “looks like” in code form:



Now, when a cube is created, we fill each square (0≤X<18 , 0≤Y<3) with the **color** that it holds. The Cube class also: handles whether or not the cube is possible (see input section), scrambles the cube object, records the moves applied to it during the solving process, implements every possible moves {U,D,B,F,R,L