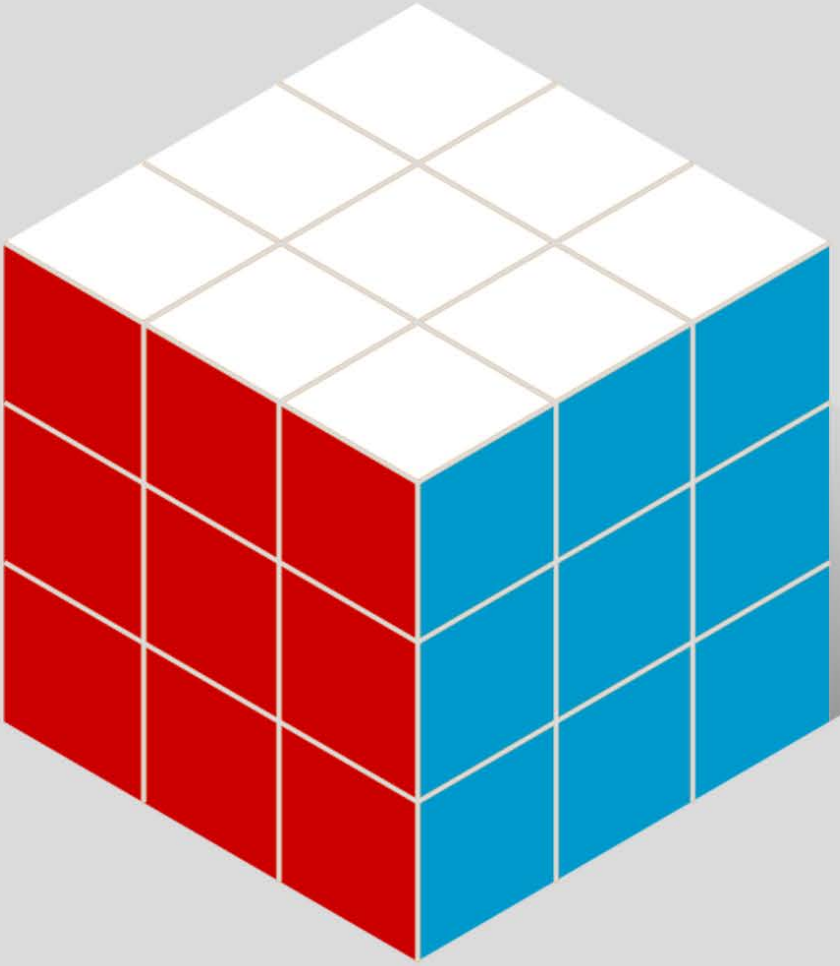
TURBOCUBE SOLVER



TECHNICAL REPORT

BY
Luy Karim
Jeanmonod Quentin
Roulin Thomas

haute école
neuchâtel berne jura
arc ingénierie
saint-imier le locle delémont

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 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). 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

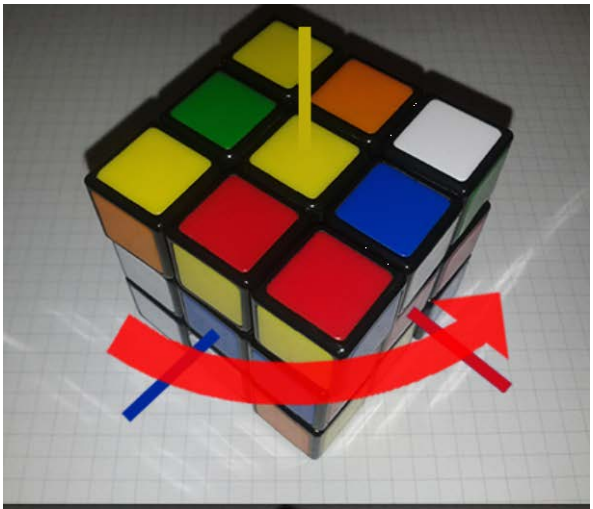
• An input handled on a graphical interface.
  ◦ The user inputs each square colour individually
• An Algorithm that will take the unsolved cube and returns the moves it has used to solve it
• The interface will be “playable,” showing each move one after the other
  ◦ With the list of movements using 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

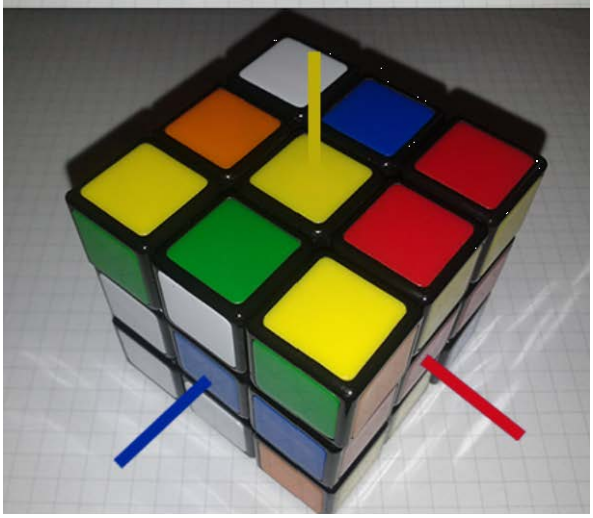
The table on the previous page is the scope statement we decided was feasible within the four and a half month period we were given to create and develop the application. This section describes how we planned to accomplish each of the three functionalities.

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.

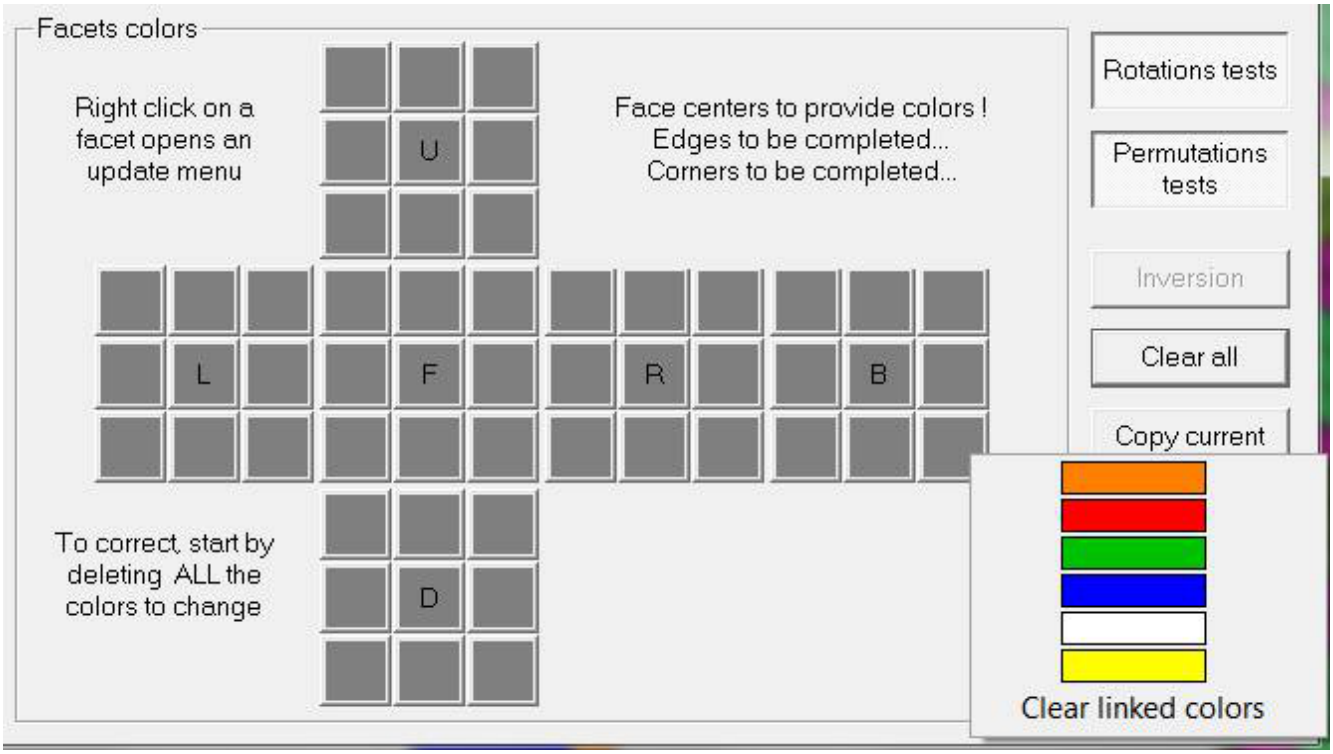


As shown, the yellow, red, and blue axes (each colored by the central cubie of that face) do not move, only the up **U** face has turned counterclockwise **U'**. Subsequently, the orange and two yellow stickers on the blue, frontal face **F** in the first image end up in the same order on the red, right 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.

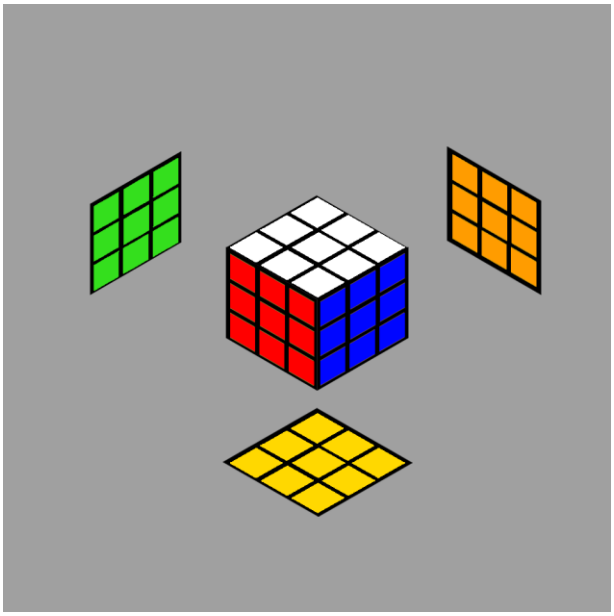
The program, Rubix Cube, has an input interface in a T-shape (Fig.X.X) where every time the user clicks on a square he may enter a color. We chose to use this method because it seemed like a really understandable way to input a cube and code-wise it would fit really well with the abstract representation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



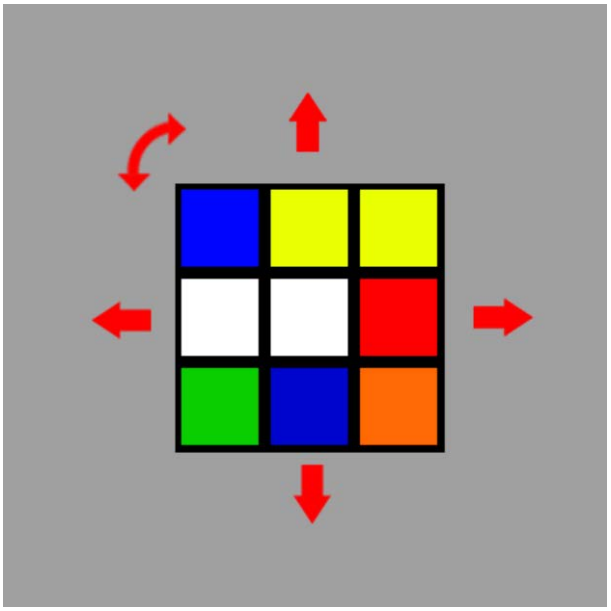
The Output

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 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 optimization issue with this method-- "useless" cubies (center 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 opposed 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 colors 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 class.

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

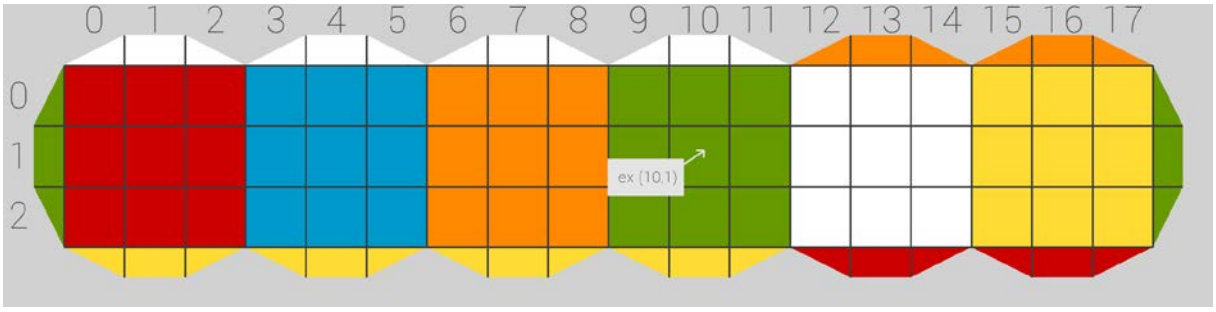
YELLOW = 5

We also implemented a toString(**color**) method which converts the given color into a QString, which allows for pop up messages to be more precise when interacting with the user (look at input section).

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++ unearthed a problem. We realized it was very difficult to figure out the orientation of the corner cubies efficiently; therefore changing the virtual cube was a necessary step to allow easier cube manipulation. So 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:



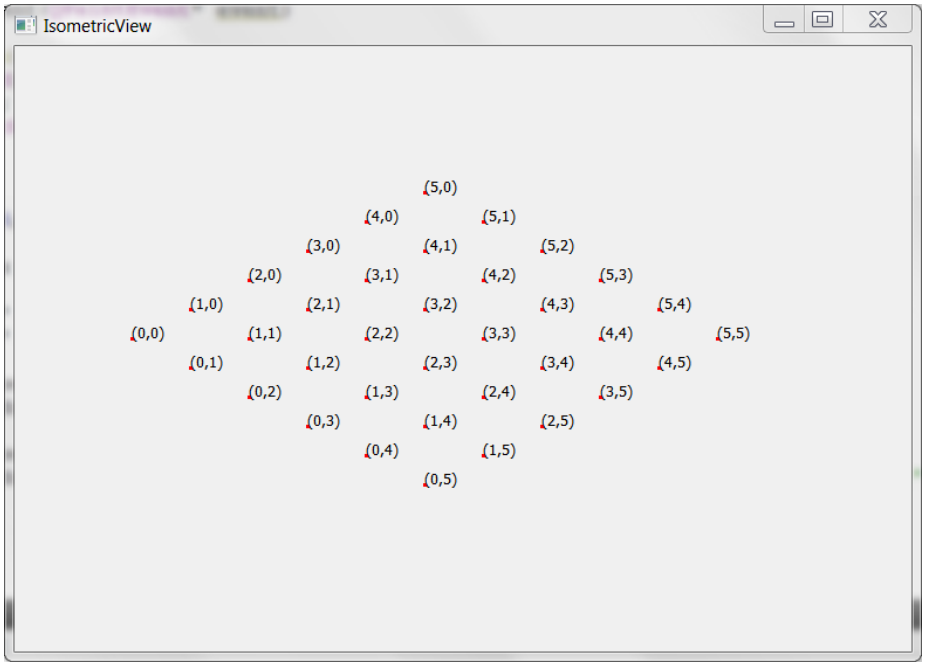
Now, when a cube is created, we fill each square ($0 \leq X < 18$, $0 \leq Y < 3$) with the **color** that it holds. The Cube class also: handles whether or not the cube is possible (see input section), scrambles the cube object, applies a sequence of moves or their reverse set of moves, implements every possible moves {U,D,B,F,R,L}, implements a method to turn a face relative to the adjacent colors.

The default orientation of the cube, both on the GUI and the virtual cube is: Up Face = Yellow, Front Face = Blue, and Right Face = Red because this is the default way of holding a cube when applying the Fridrich algorithm resolution.

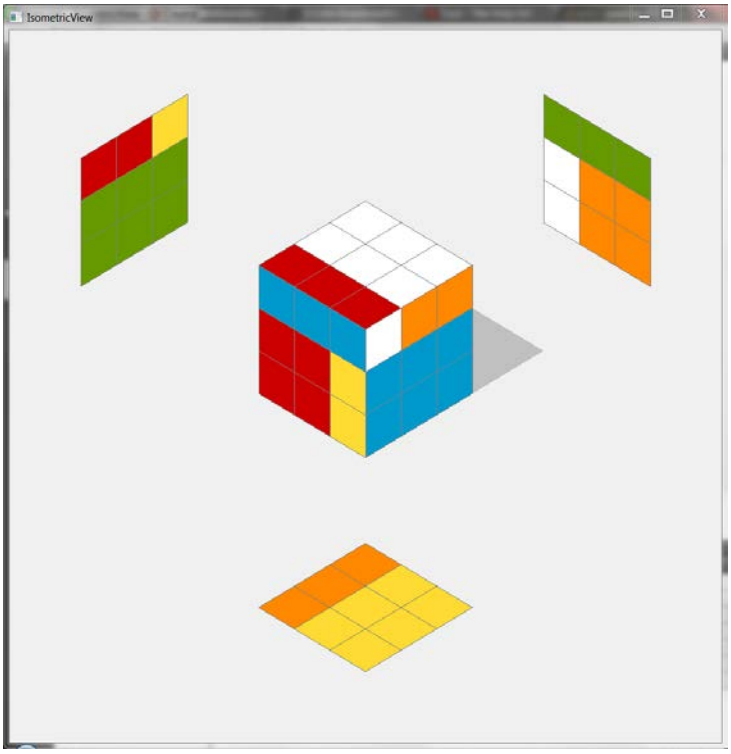
Alongside the internal representation of the cube was being coded, the isometric view of the cube was being written as well, which transitions nicely into the next part.

The Output

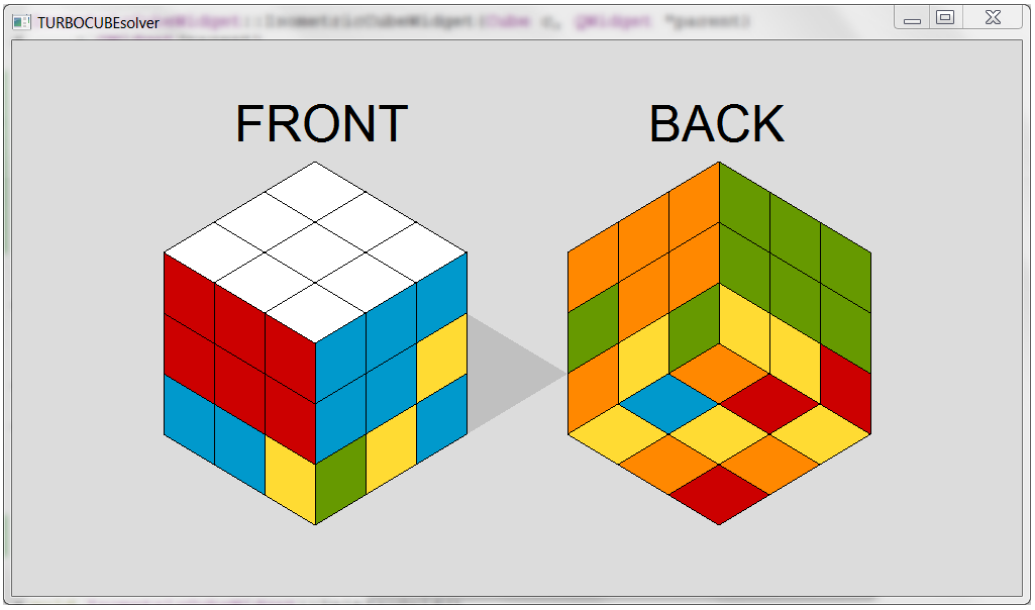
As mentioned previously in the research section of this report, the method we decided to use to output a visual representation of the cube to the user is by drawing an isometric view of the cube. To do this we used the QPainter class to draw the cube using the following grid:



Using this transposed plane, we can draw the cube in perfect isometry (this means there is no perspective effect; all the lines are parallel). The next step in outputting the cube is simple; using the coordinates, we draw the outline and we fill the squares with the colors we want therefore creating this image:



This resembles exactly what we had planned to use after our research. After having used this interface for a while, we realized that this output method is slightly confusing as the three “hidden” faces are not mirrored and they would have to be to imitate the user turning the cube in his hand. Instead of recreating mirrored images, we found another way to display the cube that makes it much easier to understand and takes a lot less space too.



This way of drawing the cube in 2D is much more efficacious because it imitates the user holding the cube on the left and turning it 180° to see what’s on the right. This is a much more natural and less confusing way to demonstrate the cube to the user.

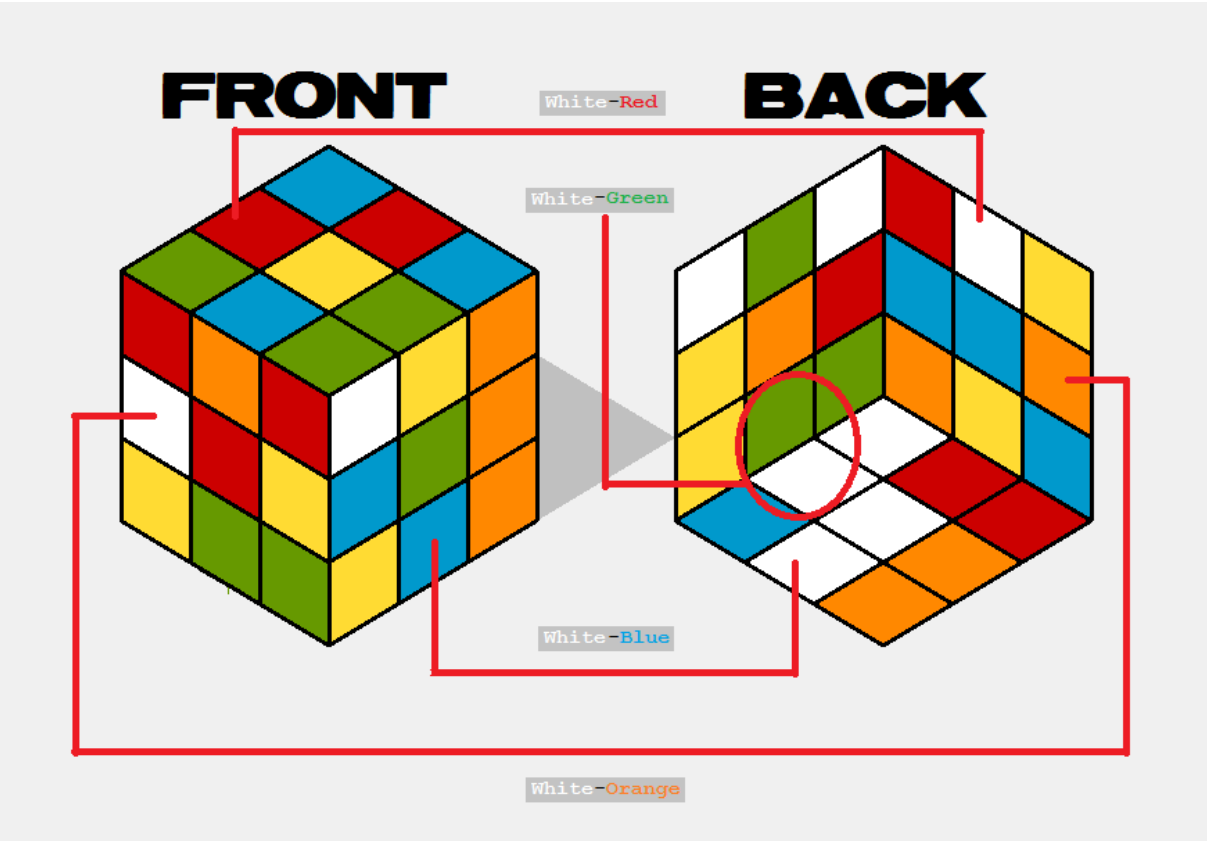
From here on out, this widget was used to output and test the Fridrich Algorithm alongside using the QDebug class to output on the Qt internal console and is now used as the main widget that is called at the launch of the program.

The Algorithm

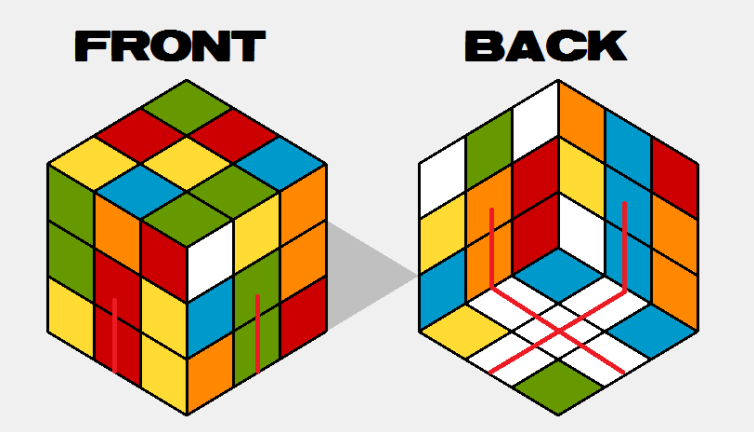
This section discusses the implementation of the Fridrich algorithm. It is separated in the 4 different sections that the resolution is performed in.

The Cross

The first step in resolving the Fridrich algorithm is creating the “bottom cross”. To achieve this, the program must find the white-red, white-blue, white-orange, and white-green cubies and move them to their corresponding, solved positions:

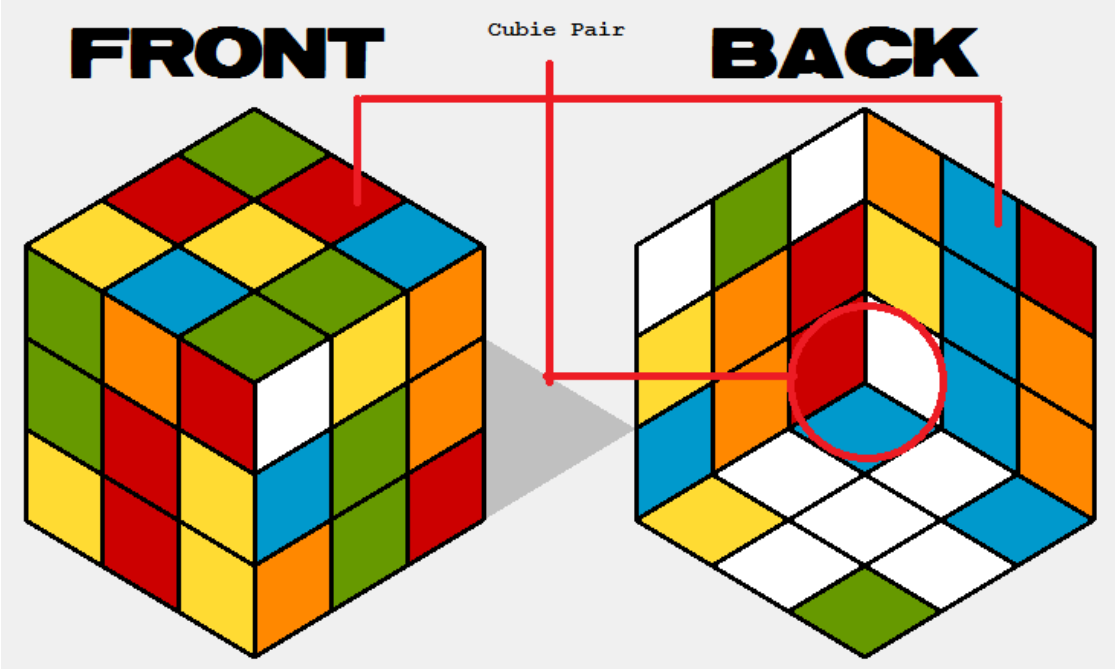


This is a scrambled cube and performing the cross creates the following:

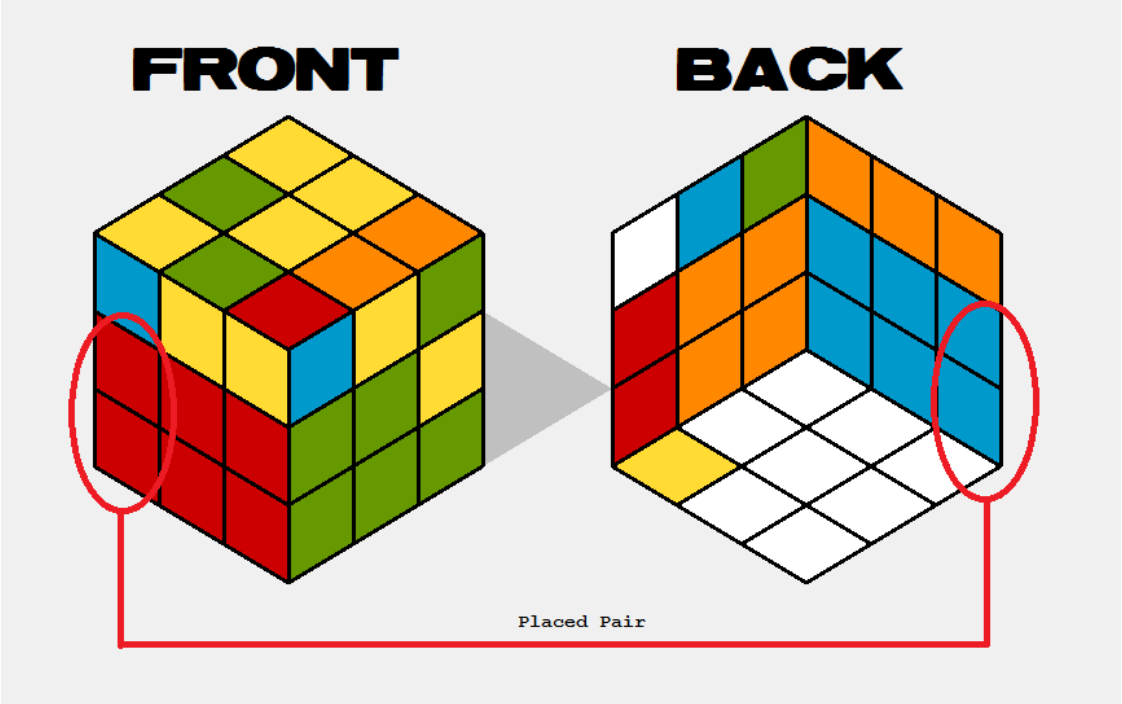


The First 2 Layers (F2L)

This part consists in finding the bottom layer corners {White-Red-Blue, White-Green-Red, White-Orange-Green, White-Blue-Orange} and their corresponding edge cubies {Red-Blue, Green-Red, Orange-Green, Blue-Orange}, and positioning them correctly in the least amount of possible moves to finalize the first two layers of the cube.

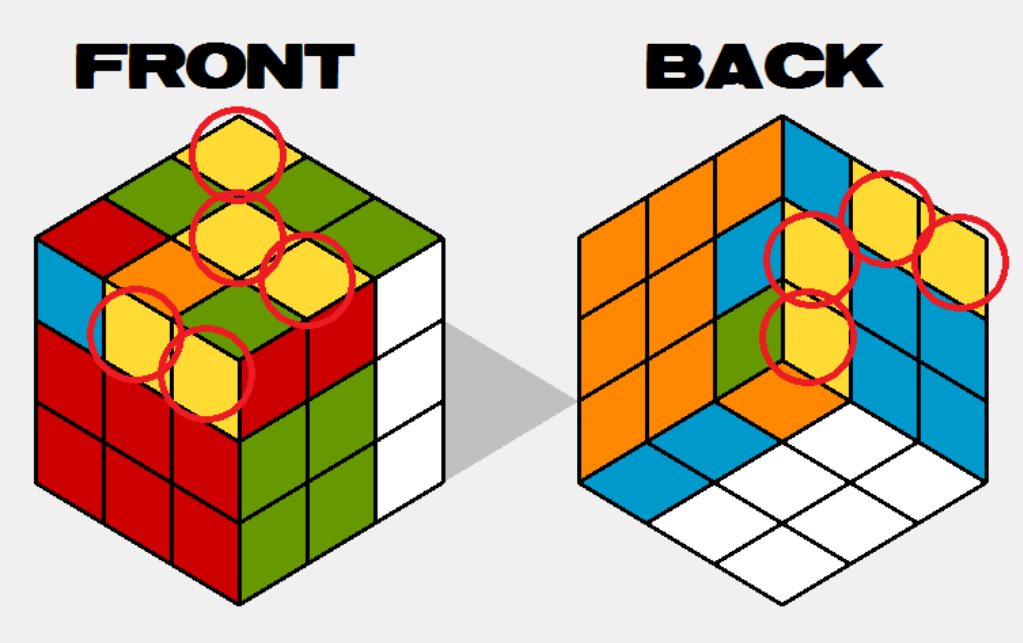


Once the F2L is complete the cube looks like the following image--the algorithm has been optimized to chain into the OLL algorithm, which explains why it doesn't look like the first two layers are placed:

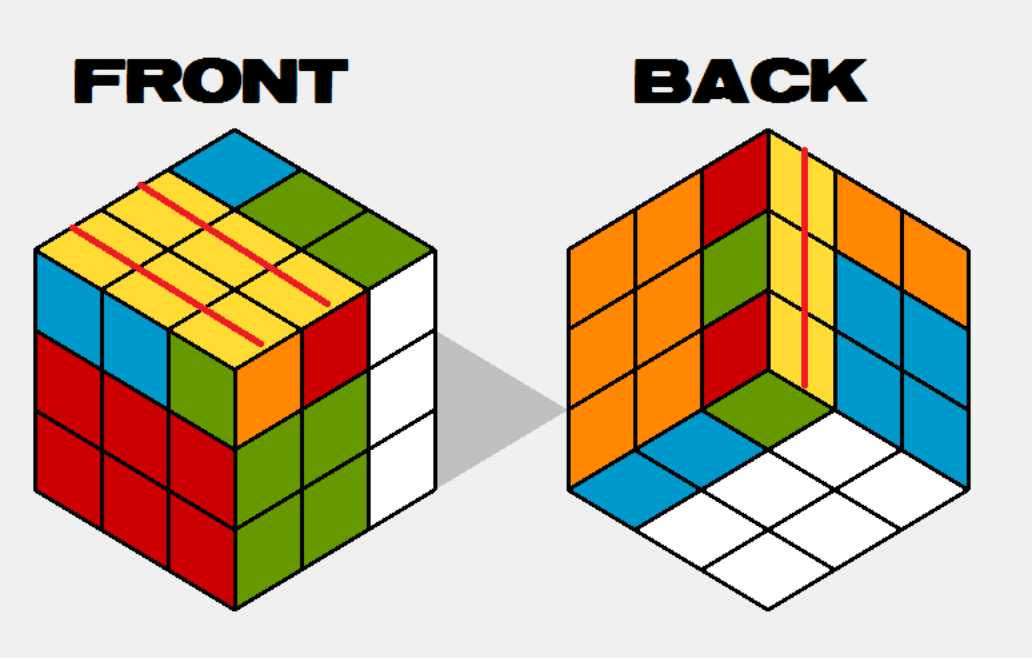


Orienting the Last Layer (OLL)

Named exactly after what this algorithm does, it orients the last layer upward. The last layer contains all the cubies that have a yellow face, since all the other are already placed, and orients the yellow face upwards.

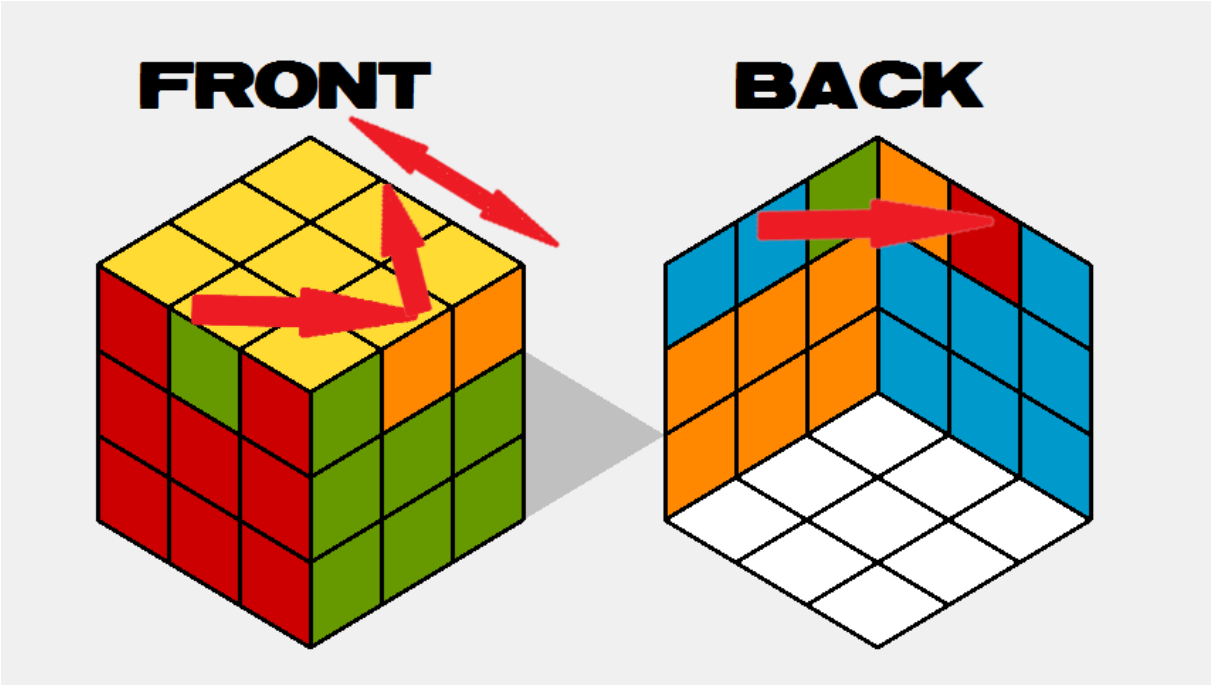


As mentioned on the previous page, the F2L anticipates the next, first move on the OLL and avoids making an unnecessary move (if F2L finishes with F' and OLL starts with F, they cancel each other out) which explains why the three vertical white stickers on the left are there instead of being in the bottom layer. The OLL algorithm then takes over and orients the yellow stickers up, and just like the F2L anticipates the OLL, the latter anticipates the PLL (the next, final algorithm).

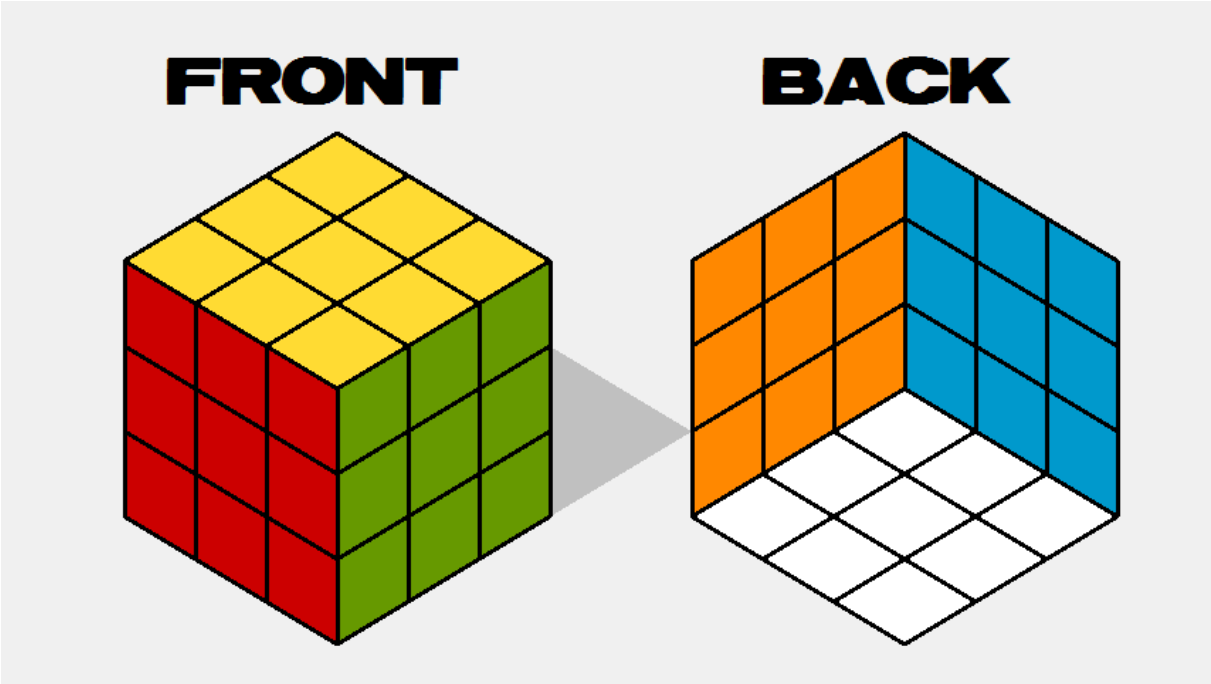


Positioning the Last Layer (PLL)

This is the final algorithm, which takes the top layer and moves the cubies around without changing their orientation until all the cubies are in their correct position. In this image we see where the cubies have to be moved for the cube to be finished:



For this algorithm, much like the F2L and OLL, there are specific move sets depending on the case (look at next section for more information) and the set of moves required to solve the last layer from the state of the cube from the second image on the previous page is {B R2 B' L' B R2 B' L' U}:



Voilà! The cube is solved!

The Algorithm's Evolution

This section discusses the evolution of the Fridrich algorithm with respect to the project's lifespan. Originally, the four main algorithms have been coded very simplistically and couldn't resolve the hundreds of different cases possible for each sub-algorithm.

The Cross

The first version of the cross would find each cross-relevant cubies and place them in order at their respective place to create the cross (order: white-red, white-blue, white-orange, white-green) using specific move sets to avoid disturbing the white face. This version would create the cross with a move count neighbouring the 15-20 moves. Then, it looked at all the cubie positions and determined the fastest move set. Now, it turns the bottom face relative to the cubies that are already in their correct spot which reduces the number of moves to less than 7 on average. This method was inspired from the website CubeLoop.

We implemented a test method that scrambles the cube and performs a cross, which is how we found that on average our program solves the cross in 7 moves.

F2L

Originally it would perform the set of moves in color order, now it checks which ones are the fastest and pre-resolved cubie pairs.

OLL

The "two-look" method (orientate the edges followed by the corners) was the first version of the OLL that we implemented. We reduced the number of moves to perform this algorithm by implementing the 57 different cases of the "one-look" method.

PLL

Same evolution as the OLL algorithm where there are 21 possible cases.

Global run-through

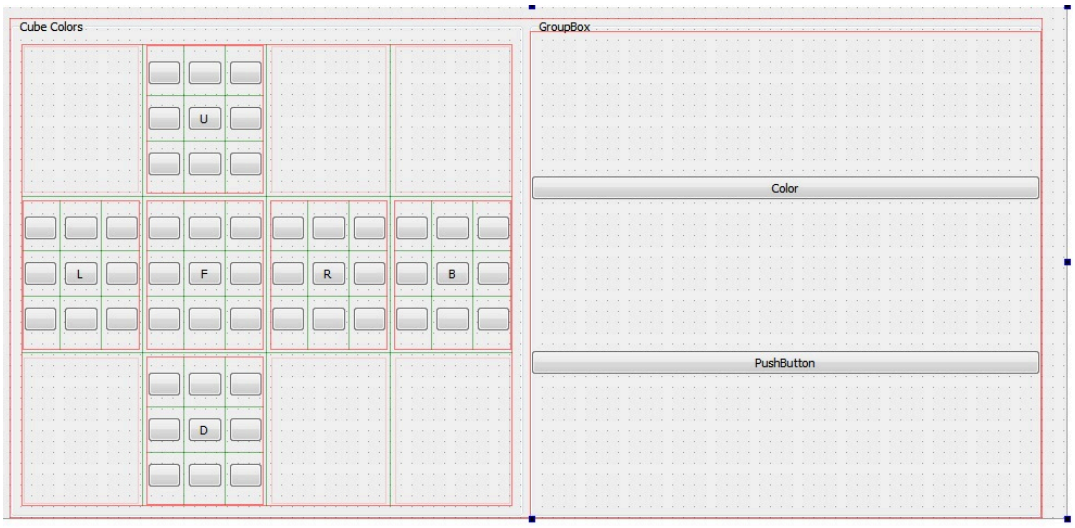
The complete run of a solve will try all the possible crosses and for each cross will try each F2L possible followed by the OLL and PLL; this generates all the possible Fridrich solutions and will select the one with the least amount of moves to output.

The evolution of the Fridrich resolution throughout our project resembles very much the way a human being learns to use this method. They would learn the easier methods and as time passes they would be confronted with multiple cases to complete the OLL and PLL which they would have to learn if they want to know how to solve any cube. Notably, code-wise, there are hundreds of conditions for each algorithm which makes the Fridrich class very long and painful to read, but it is the only way to implement it because it imitates the human solving the cube.

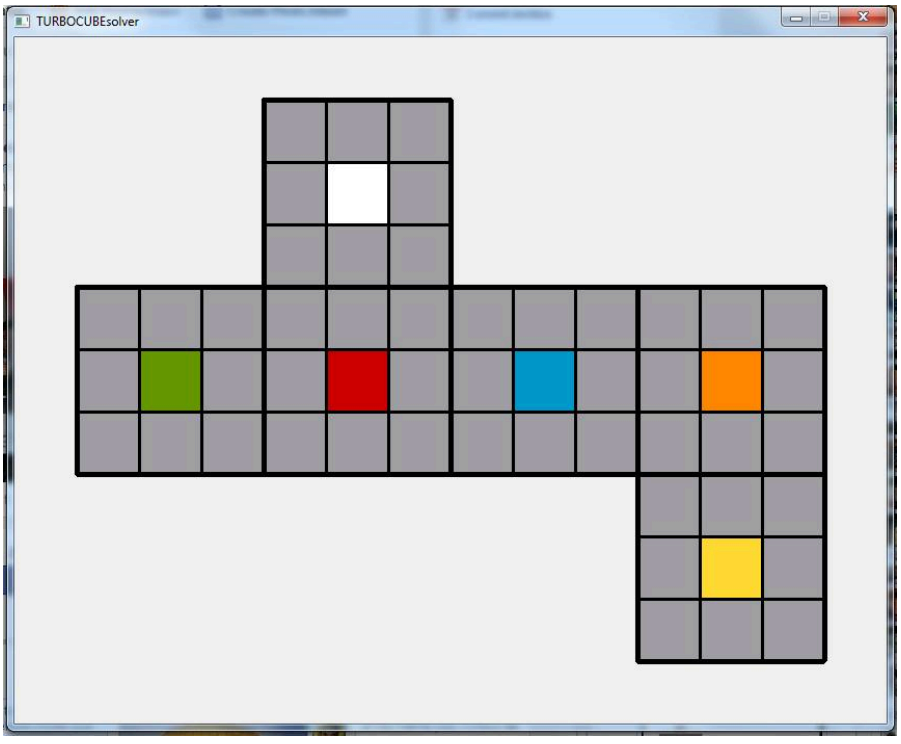
The Input

In the research section we determined that the best way to input the cube was to have another widget open and the user could fill each square on a T-shaped, flattened cube. After having discussed with the client, we determined that it was too complicated and confusing to fill the cube this way, so we worked out another way. The second version was to use the face-by-face output interface (see research section) as an input interface for the user to enter each face individually. However, this was before we had finished the isometric view out we have now and we realized we didn't need another widget to do this. We therefore decided to use only a single widget, which avoids eating up memory for an additional widget and all the possible memory leaks that method would induce.

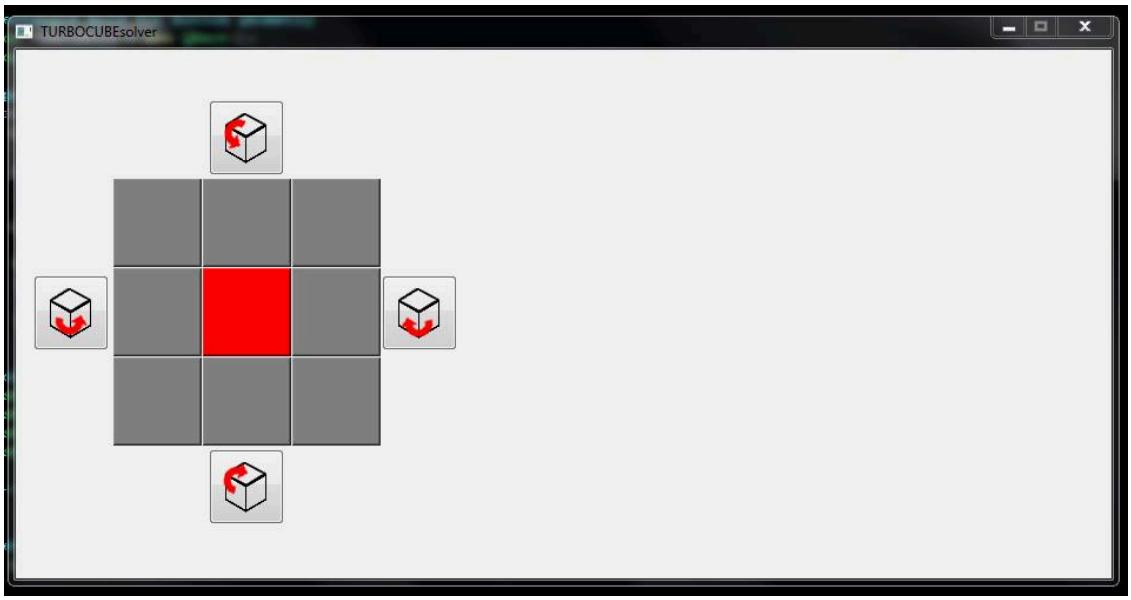
V1.1



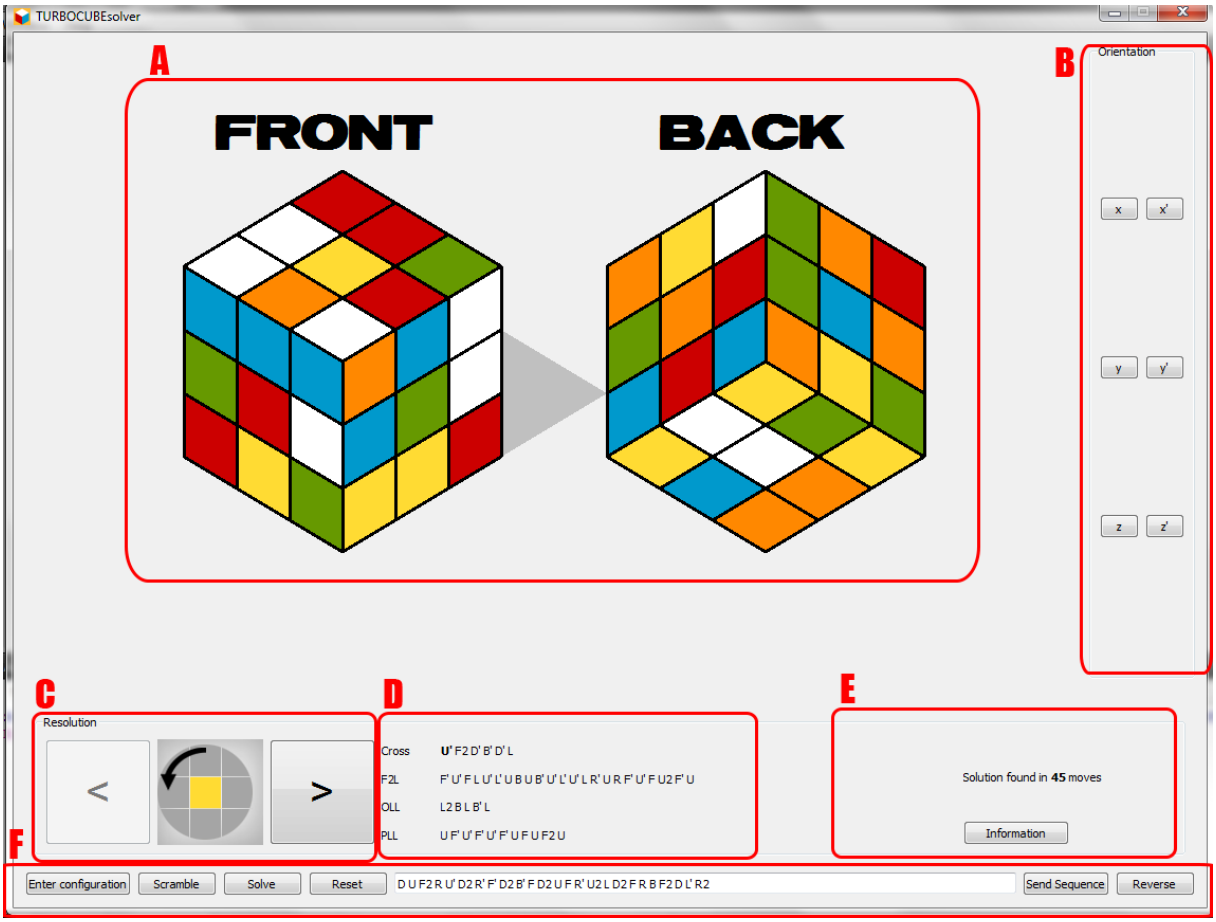
V1.2



V2



The Interface



Section	Description
A	QPainter region where the Cube is drawn in isometric view.
B	A QGroupBox which contains 6 buttons that when clicked changes the orientation of the cube respectively. X turns the entire cube clockwise around the X axis and X' does it counterclockwise.
C	C, D, and E are in the same QGroupBox called Resolution. All three of these sections are related to the demonstration of the cube's resolution. This section, C, shows the move to perform according to the face with the designated color and the two buttons on either side of the image (linked to the J and L key on the keyboard) performs the previous or next move.
D	This section shows the entire set of moves separated in Cross, F2L, OLL, and PLL.
E	The Information button pop's up a dialog that informs the user of which orientation the user should be holding the cube in their hand to solve the cube with the moves shown in D and C.
F	This section is basically the main menu. Enter Configuration—When this button is used, the user can click on the squares on the cube to change the color; right and left clicking scrolls through the possible colors. A label appears to inform the user that they may change the configuration of the cube and the button turns to "Confirm Configuration". This button can be activated with the accelerator key combination "Alt-C".

	Confirm Configuration—Once the user has finished entering the cube configuration, they may click this button to confirm and it will launch the validateCube() method of the Cube class which will check if the cube is possible, if not an error message shows up to let the user know. Scramble—This button scrambles the cube with a set of moves sent to the QLineEdit. Accelerator: “Alt-A”. Solve—This button launches the Fridrich algorithm using the current state of the cube. Accelerator: “Alt-S”. Reset—This button resets the cube to a solved state. Accelerator: “Alt-R”. QLineEdit—The user can enter a set of moves {R,L,U,D,B,F}, their primes, or their doubles and the “Return” and “Enter” keys of the keyboard are linked to the “Send Sequence” button to perform the moves to the cube. Other letters are discarded. Send Sequence—Performs the moves in the QLineEdit. Reverse—Reverses the sequence in the QLineEdit to nullify it.

Conclusion

Overall, this project has been brought to what we had planned and the Scope Statement was accomplished. Indeed, the input mechanism, though it was changed and thought through over and over again, works using the main window; the user can configure the cube as they have it in their hand and the program checks whether or not the cube they entered is possible. We then implemented the Fridrich algorithm, which was the part that required the largest amount of coding due to the number of cases possible, to solve a scrambled or a user's cube. We then display the cube on the main output widget where the solution can be performed and the cube can be handled. For future advancements on the project; the cube could be displayed using OpenGl in 3D, which would ultimately simplify the visual representation of the cube for the user; include image treatment to input a cube using video or a set of images; code other algorithms that could be selected before solving the cube, which would allow more freedom by the user; and add additional parameters such as saving cube states and solutions to a file or port the program to Android.

Bibliography

- "Cross." *Cubeloop*. WordPress, 24 Nov. 2011. Web. 24 Jan. 2015. <http://http://cubeloop.com/?page_id=749>

- Mark. "Advanced Method." *SolveTheCube*. SolvetheCube, 2012. Web. 20 Jan. 2015. <http://solvethecube.com/advanced>

- Rokicki, Tomas. "God's Number Is 20." *God's Number Is 20*. Ed. Herbert Kociemba, Morley Davidson, and John Dethridge. Seven Towns, Ltd, n.d. Web. 20 Jan. 2015. http://www.cube20.org/

- *Rubix Cube*. Computer software. Vers. 5.6.00. N.p., 2012. Web. 20 Jan. 2015.