Khuyen Do

3X3X3 RUBIK’S CUBE SIMULATOR AND GROUP THEORY

Bachelor’s Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Department of Engineering, Information Technology

May, 2016

ABSTRACT

Unit Kokkola-Pietessari	**Date** May 2016	**Author/s** Khuyen, Do
Degree programme Bachelor of Engineering, Information Technology		
Name of thesis 3X3X3 RUBIK'S CUBE SIMULATOR AND GROUP THEORY		
Instructor Dr. Grzegorz Szewczyk		**Pages** [54 + 4]
Supervisor Dr. Grzegorz Szewczyk		

The aim of the thesis was to write a software program to simulate the traditional 3x3x3 Rubik's Cube and also introduce and explain the algorithm for solving the cube by means of group theory. The virtual cube was created using OpenGL/SDL which are C++ graphic libraries for rendering and defining 2D, 3D images.

Rubik's Cube is not only well-known as a best-selling toy all over the world but it can also be used as a tool to employ a rich underlying mathematical structure. And in this thesis work, we show that Rubik's Cube can be explained using group theory which is a brand of modern abstract algebra.

The thesis first introduces the structure and notations of the Cube and the mathematical background including permutation and group theory. The next part provides the architecture and the essential features of the software. After that, the strategy for solving the cube and the explanation of the application of the virtual cube for applying the algorithms will be described. The final part shows some of the lacking features that need to be improved in the future.

Keywords:
Rubik's Cube, permutation, group theory, algorithm, OpenGL, simulator

CONTENTS

GRAPHS

TABLES

FOREWORD

This is my Bachelor thesis of Engineering, Information Technology at Centria University of Applied Sciences. Thesis's aims are to:

- write a software program to simulate the traditional 3x3x3 Rubik's Cube.
- provide and explain the application of the virtual cube to develop the strategy for solving the cube.

The virtual cube was created using OpenGL/SDL which are graphic libraries for rendering and defining 2D, 3D images.

Rubik's Cube is not only well-known as a best-selling toy all over the world but it can also be used as a tool to employ a rich underlying mathematical structure. And in this thesis work, we show that Rubik's Cube can be explained using group theory which is a brand of modern abstract algebra.

The thesis first introduces the structure and notations of the Cube and the mathematical background including permutation and group theory. The next part provides the architecture and the essential features of the software. After that, the strategy for solving the cube and the explanation of the application of the virtual cube for applying the algorithms will be described. The final part shows some of the lacking features that need to be improved in the future.

ABBREVIATIONS

WAC	World Cube Association
3D	three dimensional
2D	two dimensional
IDE	Integrated Development Environment
OpenGL	Open Graphics Library
API	Application Programming Interface
SDL	Simple Direct-Media Layer
LCM	Least common multiple
GLU	OpenGL utility library
UML	Unified modeling language
F2L	Friedrich First Two Layers

1. INTRODUCTION

Rubik’s Cube is probably one of the most illustrious puzzles up to date. And the reason made it so prominent is still not certain. Since its invention in 1974 by Hungarian sculptor and professor of architecture Erno Rubik, it has captured the imagination of millions of all age and is widely considered to be the world’s best-selling toy. There are competitions held every year for cube lover to compete the records and organized by WCA (World Cube Association). The current world record for a single time on the 3×3×3 Rubik’s Cube in competition is 5.25 seconds, set by Collin Burns in April 2015. There is a variation of cubes which are sold commercially from 2x2x2 to 7x7x7. This paper will discuss about the traditional 3x3x3 Rubik’s Cube, one that is simple enough to study and understand but complex enough to exploit and investigate from many points of view.

The project’s aim is to write a software program to simulate the traditional 3x3x3 Rubik’s Cube. The virtual cube is used for simulating the cube and keeping track of the moves. Instead of using the physical cube to design and test any algorithms. It is better to do it with a virtual cube. Besides the basic operations which can be applied on the normal Rubik’s Cube, the program also implemented the features which can help to scramble and solve the cube with built-in algorithms. The virtual cube was created using OpenGL/SDL which are C++ graphic libraries for rendering and defining 2D, 3D images.

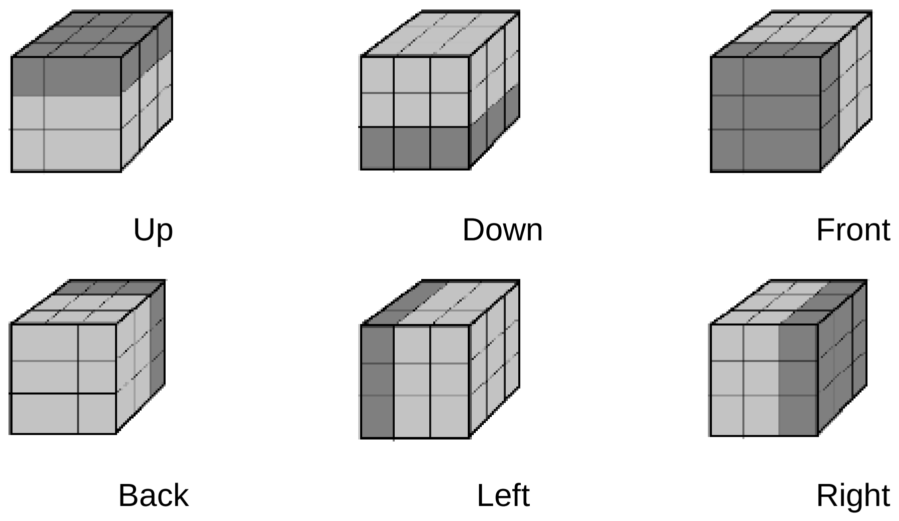
The version control I used during the programming stage in this project was GitHub. Version control can be useful at helping keep track of progress throughout a project. By keeping a record of previous commits and branches previous versions of a project can be restored if required. Below is the link to my project repository: https://github.com/imama22/Rubik-s-Cube-Simulator/tree/master/RubiksCubeSimulator

1 RUBIK'S CUBE

This chapter provides us with the basic understanding of the structure of Rubik's Cube, the notations of the Cube's rotations and some calculations on the bounds of the Cube. In this paper, we use the word "the Cube" or sometimes "Rubik's Cube" in this paper to refer to the entire cube.

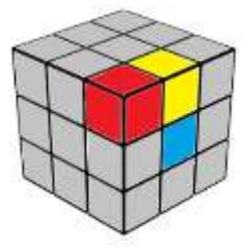
1.1 Structure

Six faces of the Cube will be differentiated by 6 different solid colors: white, yellow, blue, green, red and orange. The model we use to give the demonstrations in this thesis is: white is opposite yellow, blue is opposite green and red is opposite orange; and this is currently the most popular sold model also (Dempsey 1988.). The face is currently facing you when you hold the Cube is the front face, then the face opposite the front is the back face, the face above or on the top of the front is the up face, the face opposite the up is the down face, the face directly to the left of the front is left face; and the face directly to the right of the front is the right face. The demonstration of faces can be seen in Graph 1.



GRAPH 1. Six faces of a 3x3x3 Rubik's Cube

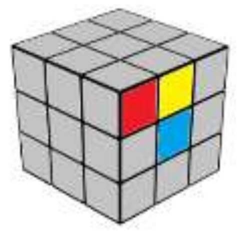
The Cube consists of smaller cubes which are called **cubies**. In the actual physical cube, just 26 cubies are visible, the 27th cubie in the center does not exist. There are 3 different types of cubies: center cubies, edge cubies and corner cubies. These cubies can be seen in graph 2. In total, there are 6 center cubies, 12 edge cubies and 8 corner cubies. After any sequence of moves or rotations, center cubies will be replaced by other center positions, edge cubies will be replaced by other edge cubies, and corner cubies will be replaced by other corner cubies.



blue: center cubie
yellow: edge cubie
red: corner cubie

GRAPH 2. Cuibes of a Rubik's Cube

On each face, there are 9 squares which are called **facets**. They are usually covered by colored stickers. And there are total 6 x 9 = 54 facets on a cube. There are also 3 types of facets: center facets, edge facets and corner facets based on their positions. These facets can be seen in Graph 3.



blue: center facet
yellow: edge facet
red: corner facet

GRAPH 3. Facets of a Rubik's Cube

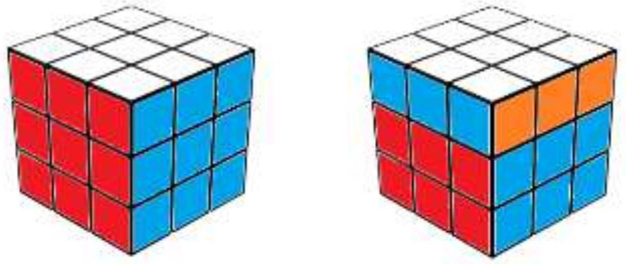
1.2 Notations

Notation plays an important role because it is meaningful, concise and really helpful to give the better demonstration for the algorithms. There are variation of ways to denote faces, cubies and other things relating to Rubik’s Cube. In this paper, some of the notations are adopted from “Singmaster notation” which were developed from David Singmaster, an English mathematician, to denote a sequence of moves (Joyner 2008.). The upper-case letters are used to denote the faces of the Cube. They are: F (Front), B (Back), U (Up), D (Down), L (Left), R (Right).

Cubies will be denoted using string of upper-case letters. The edge cubies will be denoted by XY (= YX), where X and Y are the faces on which the cubie is located. The corner cubies will be denoted by XYZ, where X is the face on which the cubie is located, Y and Z are the faces the cubie borders to. For example, UF is the edge cubie whose one face is on the up face, and the other is on the front face; ULB is the corner cubie whose 3 faces are on up face, left face and back face.

Facets will be denoted using string of lower-case letters. The edge facets will be denoted by xy, where x is the face on which the facet is located and y is the face the facet borders to. The corner facets will be denoted by xyz, where x is the face on which the facet is located, y and z are the faces the facet borders to.

There are same notations for the faces to denote the rotations (or the moves). Therefore, U, D, F, B, L and R are used to denote the rotations of that face a quarter-turn clockwise. For example, U means a clockwise quarter-turn on up face (graph 4), F means a clockwise quarter-turn on front face and so on. The same letters but with lower-case will be used to denote the quarter-turn counter-clockwise rotation. Therefore, u means the quarter rotation of the up face counter-clockwise, d means the quarter rotation of the down face counter-clockwise and so on.



GRAPH 4. Up-face rotation turns the Cube from state 1 to state 2.

When a single move is repeated several times, instead of writing each separately, the move sequence can be written with the power number. For example, UUU will be rewritten U^3 (or simply U3) and UUUUU will be rewritten U^5 (or simply U5). Notice that rotating the up face 4 times will return the Cube the state before applying the moves; on the other hand, U^4 equals doing nothing to the Cube. After this if applying one more U move, it simply rotates the up face a quarter turn clockwise and just like starting a new cycle, then we have $U^5 = U$, $U^6 = U^2$, etc.

Later, with the help of mathematical tool, it will be proven that because the move U has the order of 4, then after every 4 single moves, the effect will be the same, like $U = U^5 = U^9 = \dots$ Also rotating the up face 90^0 clockwise three times would end up the same result as rotating the up face 90^0 counter-clockwise; hence, $U^3 = u$. In conclusion, for any single face, say U for example, we will have 4 different types of rotations: e, U, U^2, and u. To sum up, there are 3 primitive moves on a face because e means doing nothing, hence there are 3 x 6 = 18 primitive moves in total. Any other moves on the Cube can be written as a combination of primitive moves.

For the notation of the rotations of the entire cube. X will denote the rotation of the entire cube as if doing an R turn; and x will be the inverse of X. Y will denote the rotation of the entire cube as if doing a U turn; and y will be the inverse of Y. Z will denote the rotation of the entire cube as if doing a F turn; and z will be the inverse of Z. These moves are not primitive moves and are often not counted as a single move in most situations.

A sequence of moves is a combination of moves followed by one each other and can be written like a string of letters like this “RDDuF2fd”. The algorithm described in this paper will be presented as a set of sequences of moves.

1.3 Bounds on Rubik’s Cube

Let’s call a permutation of all the facets on a Rubik’s Cube is a configuration. Let’s find out how many possible configurations there can be in total. There are 8 corner cubies, so there are 8! (= 40320) ways to arrange them. Similarly, there 12 edge cubies, so there are 12! (= 479001600) ways to arrange them. Moreover, each corner cubie can be arranged in 3 orientations, giving 3^8 possibilities for each permutation; each edge cubie can be arranged in 2 orientations, giving 2^{12} possibilities for each permutation. This simply gives
$(8! \cdot 3^8 \cdot 12! \cdot 2^{12}) = 519{,}024{,}039{,}293{,}878{,}272{,}000 \approx 5.19 \times 10^{20}$
possible configurations.

Because a configuration of the Cube is not only regarding the positions of all cubies, but also regarding the orientations of all cubies on the Cube. Hence not every configuration of the Cube is solvable. If you disassemble the cube and assemble the cubies back, there is probability that the Cube will be unsolvable (Joyner 2008).

There are 8 corner cubies, but the orientations of 7 corner cubies will decide the orientation of the last corner cubie. Similarly, the orientations of 11 edge cubies will decide the orientation of the last edge cubie. The more detailed explanation about how to come up with these numbers will be presented in the “Unsolvable configurations” part where we have enough theoretically mathematical background sufficient to explain the phenomenon. This narrows down to

$$\frac{8! \cdot 38 \cdot 12! \cdot 212}{3.2.2} \approx 4.3252 \text{ x } 10^{19}$$

possible configurations of the Cube.

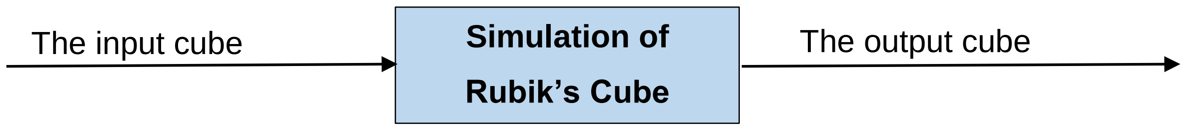
3 IMPLEMENTATION AND DESIGN

This chapter gives us the general overview of the implementation and the design of the program. It starts by explaining the main functions of the program, how the software is designed and some of the features of the program in detail. Then the architecture of the software will be explained by showing some UML diagrams, and some implementation code snippets of the main functions and finally the software development environment of the program.

3.1 Overview

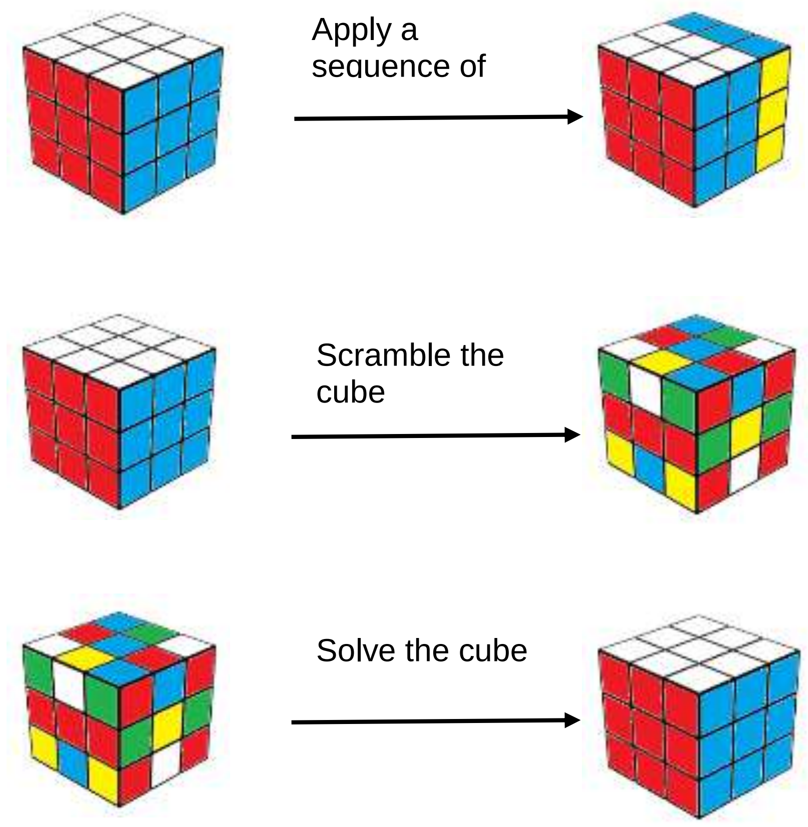
The essential purpose of the virtual cube is for the users to simulate the cube as if they are doing on the real physical cube. Specifically, the users can apply most of the basic moves on the cube such as the twists on a single face or the rotations on the entire cube. Moreover, the virtual cube also has some more built-in special functions which are scrambling the cube and solving the cube.

The process can be viewed as an input/output simulation model as below:



GRAPH 5. Simulation model.

In this case, the cube with a given state can be the input for the process and the output of that process will be the cube with a different state. Therefore, three main features of the program can be considered as three processes in the simulation model and can be summarized visually in Graph 6. Graph 6 shows that the program can provide us the utilities to simulate the real cube such as applying any sequence of moves, scrambling the cube and even solving the cube.

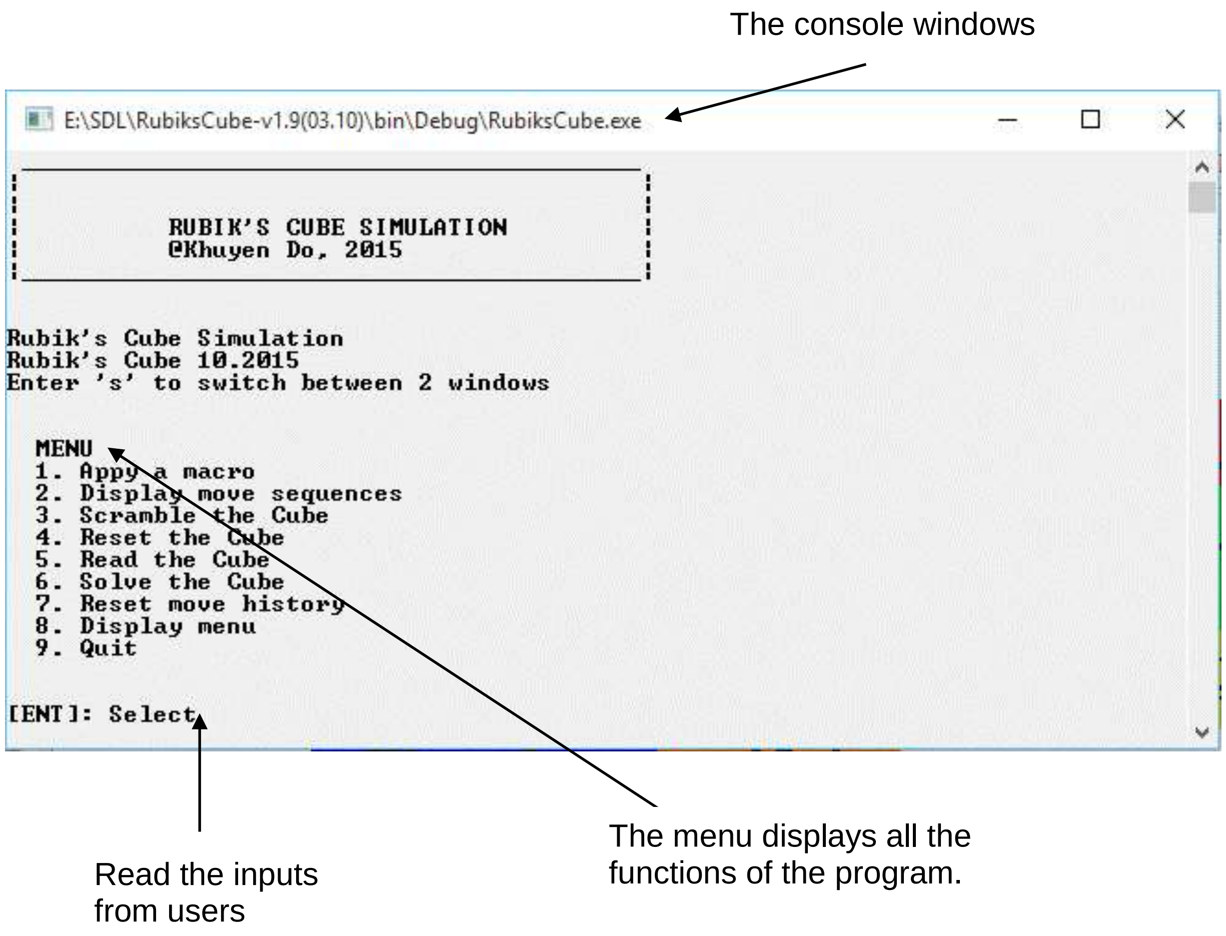


GRAPH 6. The visual display of 3 main processes of the program.

3.2 GUI

The GUI of the virtual cube consists 2 windows. One is the normal console window to interact with users and deal with input/output functions as well; the other is the SDL window which displays the cube in 3D representation.

3.2.1 Console windows

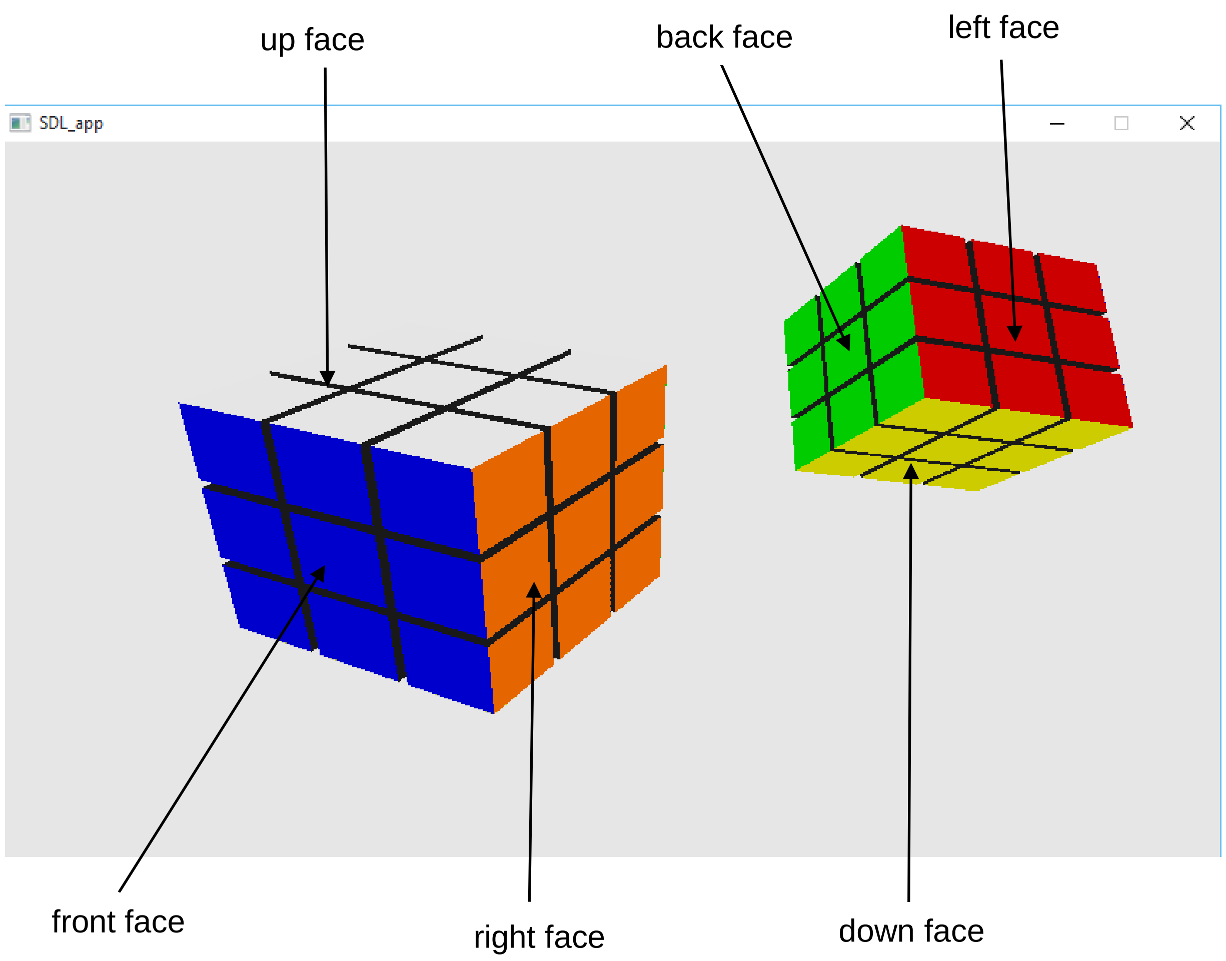

The console windows
Read the inputs from users
The menu displays all the functions of the program.

```
E:\SDL\RubiksCube-v1.9(03.10)\bin\Debug\RubiksCube.exe

           RUBIK'S CUBE SIMULATION
           @Khuyen Do, 2015

Rubik's Cube Simulation
Rubik's Cube 10.2015
Enter 's' to switch between 2 windows

  MENU
  1. Appy a macro
  2. Display move sequences
  3. Scramble the Cube
  4. Reset the Cube
  5. Read the Cube
  6. Solve the Cube
  7. Reset move history
  8. Display menu
  9. Quit

[ENT]: Select
```

GRAPH 7. The console windows.

3.2.2 SDL windows

In the SDL windows, the cube is displayed as 2 images of the same cube; one shows the front, up and right faces, the other shows the down, back and left faces.

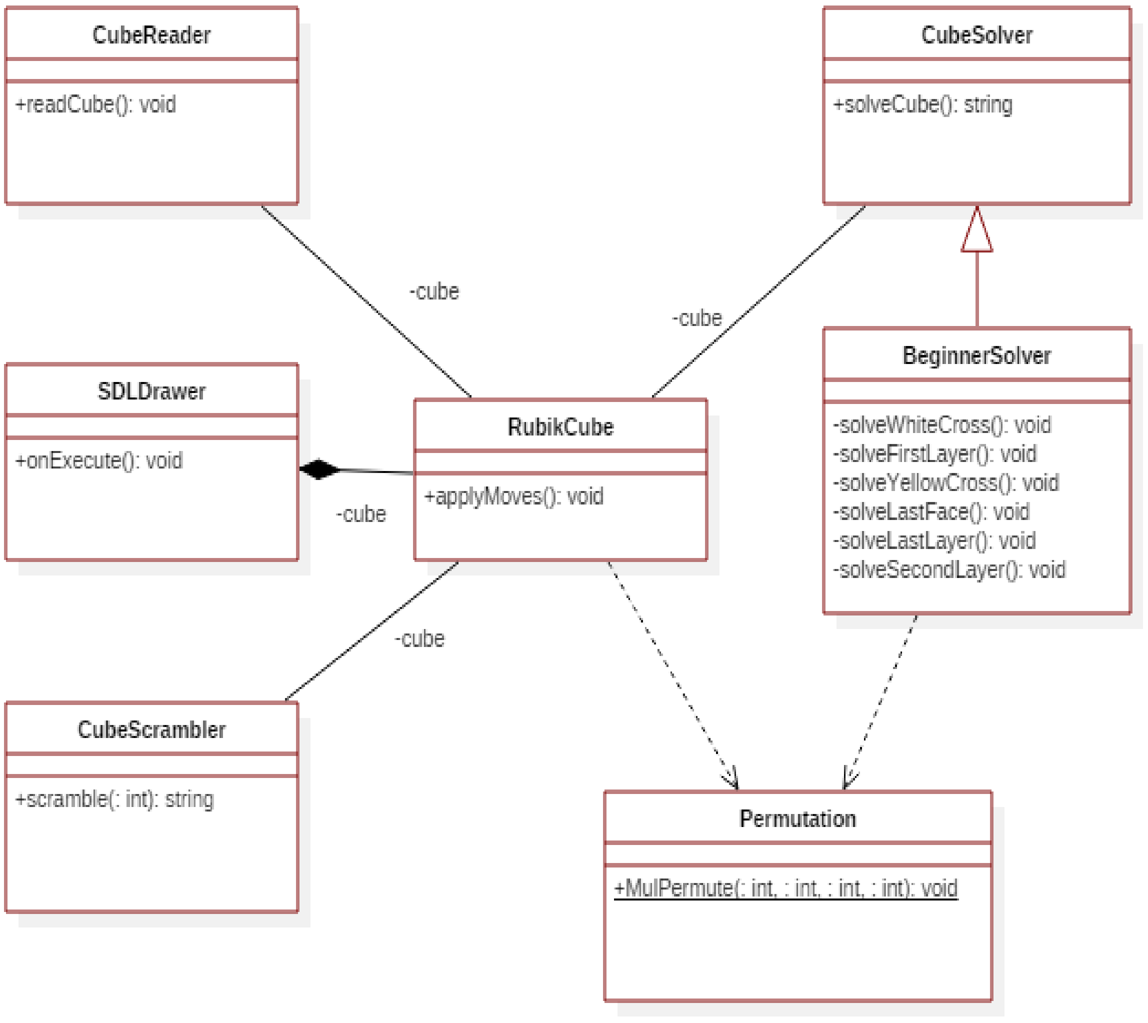


GRAPH 8. SDL windows.

3.3 Architecture

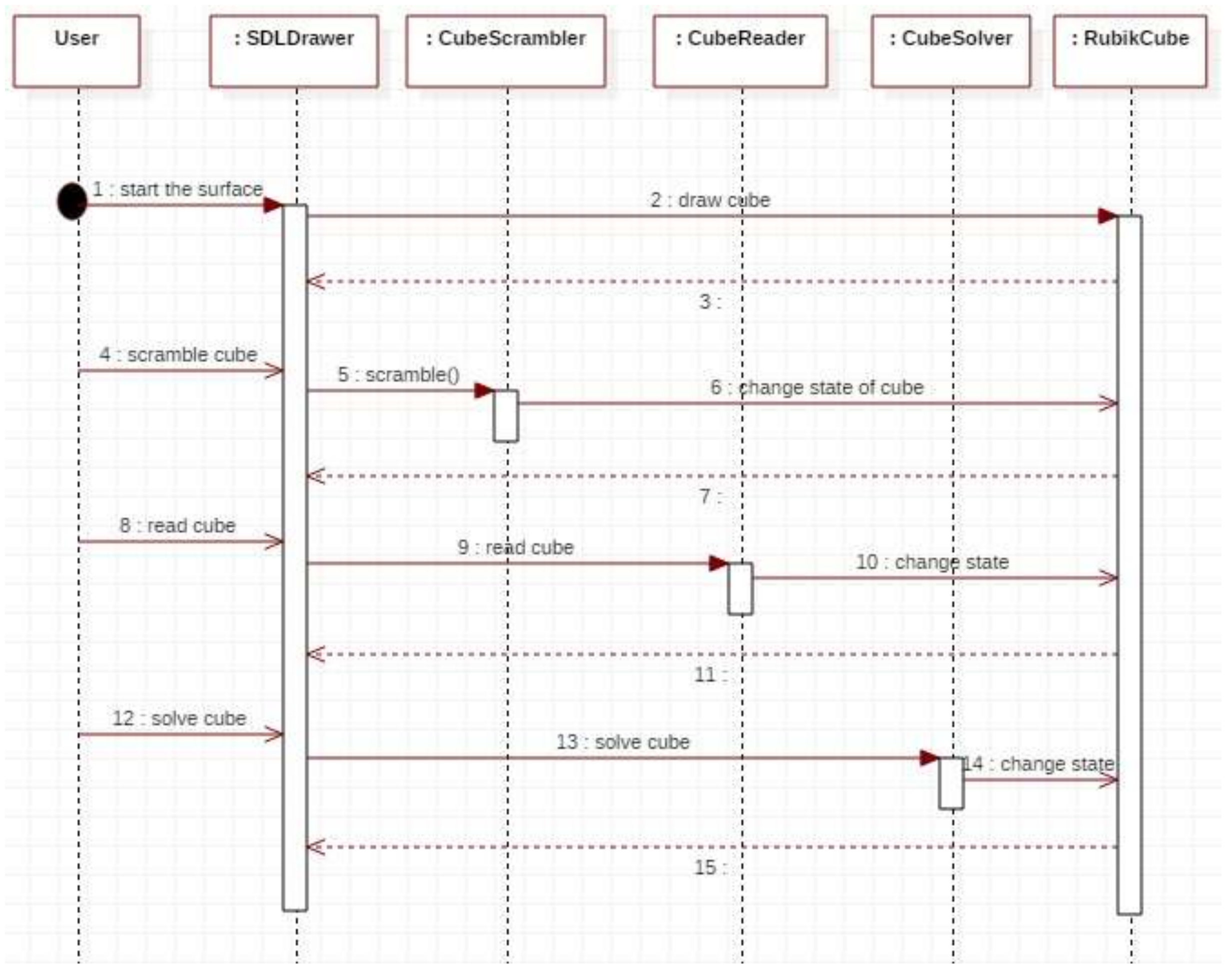
3.3.1 Diagrams

Below is the UML class diagrams which describes the relation between the classes of the program.



GRAPH 9. Class diagram.

Below is the UML sequence diagram of the virtual cube system.



GRAPH 10. UML Sequence Diagram.

The UML sequence diagram shows the sequential order of the interactions in the virtual cube system. Once the SDL surface starts, the SDLDrawer wait for the orders to come and pass to the specific class to process that order and send the reply back after the process is done.

3.3.2 Rubik’s Cube class

```
class RubikCube
{
public:
    RubikCube();
    virtual ~RubikCube();
    void applyMove(char, int);
    void applyMoves(string);
    void drawCube(char);
    void resetCube();

private:
    void rotateUpFaceClockwise();
    void rotateDownFaceClockwise();
    void rotateFrontFaceClockwise();
    void rotateBackFaceClockwise();
    void rotateLeftFaceClockwise();
    void rotateRightFaceClockwise();
    void rotateUpFaceCounterClockwise();
    void rotateDownFaceCounterClockwise();
    void rotateFrontFaceCounterClockwise();
    void rotateBackFaceCounterClockwise();
    void rotateLeftFaceCounterClockwise();
    void rotateRightFaceCounterClockwise();
    void rotateCubeX();
    void rotateCubeXCounterClockwise();
    void rotateCubeY();
    void rotateCubeYCounterClockwise();
    void rotateCubeZ();
    void rotateCubeZCounterClockwise();
    void rotateMiddle();
    void rotateUpMiddle();

    int* face_color_buffer_data;
    int* facets_buffer_data;

    static int const face_cubie_buffer_data[6][9];
    static int const cubie_buffer_data[27][6];
};
```

GRAPH 11. RubikCube class.

The shaded blue part shows the most two important data fields in RubikCube class which are:

face_color_buffer_data and ***facets_buffer_data***.

The face_color_buffer_data is a 2d array which contains the color of all 54 facets of the cube and the facets_buffer_data is a 2d array which contains the position of all 54 facets of the cube. After any move, the positions of 54 facets on the cube will be changed. Hence, face_color_buffer_data and facets_buffer_data will also change and they maintain the current state of the cube.

The RubikCube class also implements all the functions for doing the basic moves of the cube. There are in total 20 private helper methods for doing the rotations which are shown in the shaded gray part of the RubikCube class. Below is the code for implementing the 'applyMove' method for the RubikCube where all the basic operations can be done on the cube.

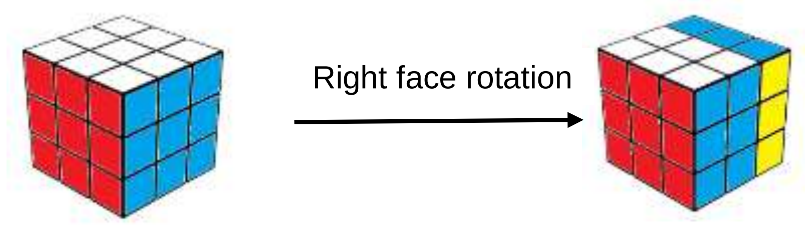
```
void RubikCube::applyMove(char _move, int time)
{
    rotating = true;
    for (int i = 0; i < time; ++i){
    switch (_move)
    {
        case 'U': rotateUpFaceClockwise(); break;
        case 'u': rotateUpFaceCounterClockwise(); break;
        case 'D': rotateDownFaceClockwise(); break;
        case 'd': rotateDownFaceCounterClockwise(); break;
        case 'F': rotateFrontFaceClockwise(); break;
        case 'f': rotateFrontFaceCounterClockwise(); break;
        case 'B': rotateBackFaceClockwise(); break;
        case 'b': rotateBackFaceCounterClockwise(); break;
        case 'L': rotateLeftFaceClockwise(); break;
        case 'l': rotateLeftFaceCounterClockwise(); break;
        case 'R': rotateRightFaceClockwise(); break;
        case 'r': rotateRightFaceCounterClockwise(); break;
        case 'X': rotateCubeX(); break;
        case 'Y': rotateCubeY(); break;
        case 'Z': rotateCubeZ(); break;
        case 'x': rotateCubeXCounterClockwise(); break;
        case 'y': rotateCubeYCounterClockwise(); break;
        case 'z': rotateCubeZCounterClockwise(); break;
    }
    …
…
}
```

GRAPH 12. The method applyMove in RubikCube class.

3.3.3 Implementation of Rubik's Cube moves

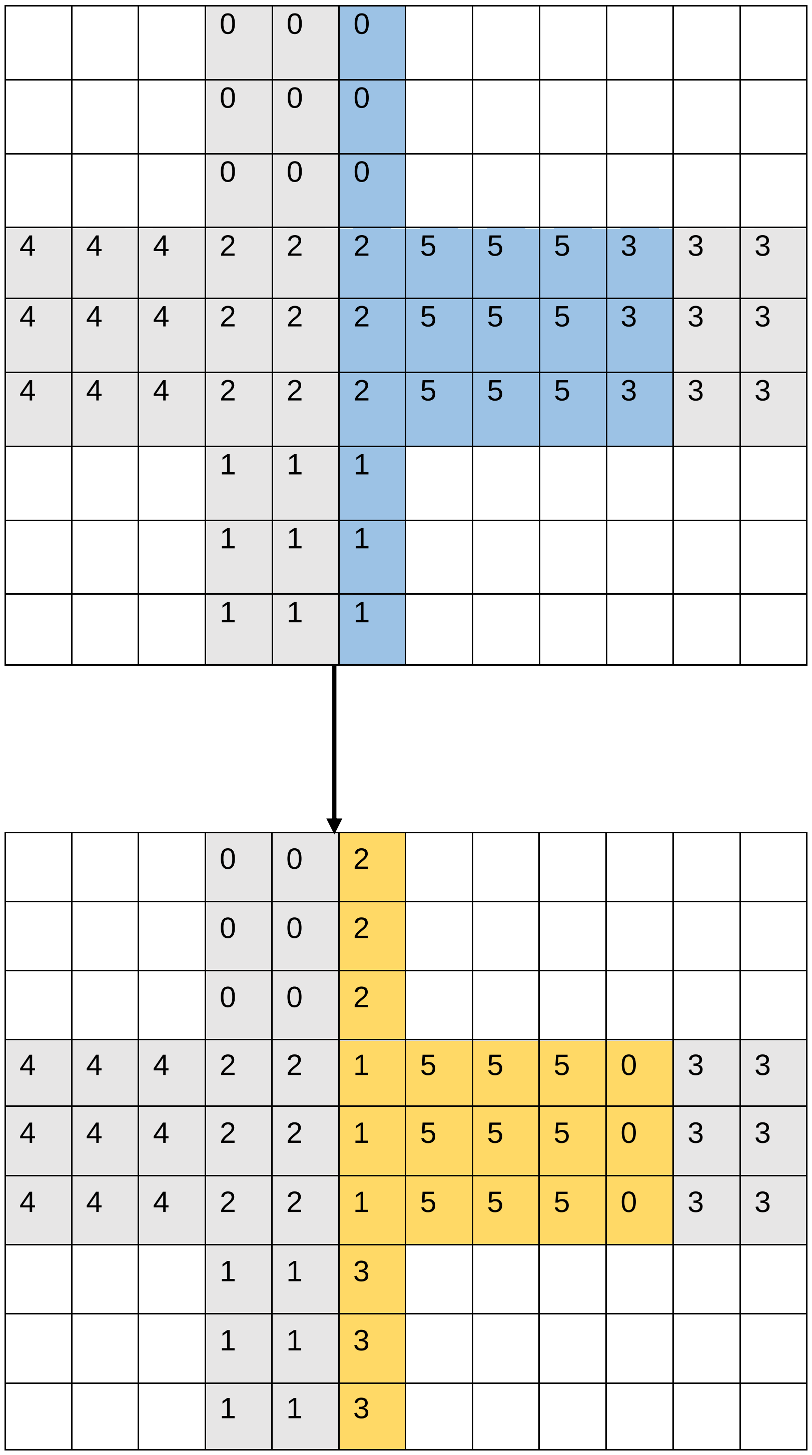
The main operation that can be applied on a cube is doing one of the basic rotations. Scrambling the cube or solving the cube can just be done through a long sequence of many rotations or moves. The main key or the most significant point in a rotation is the permutation or the rearrangement of the cubies and facets of the cube.

Let's look at an example in which the right-face rotation will transform the cube in the solved state to another state.



GRAPH 13. The right-face rotation.

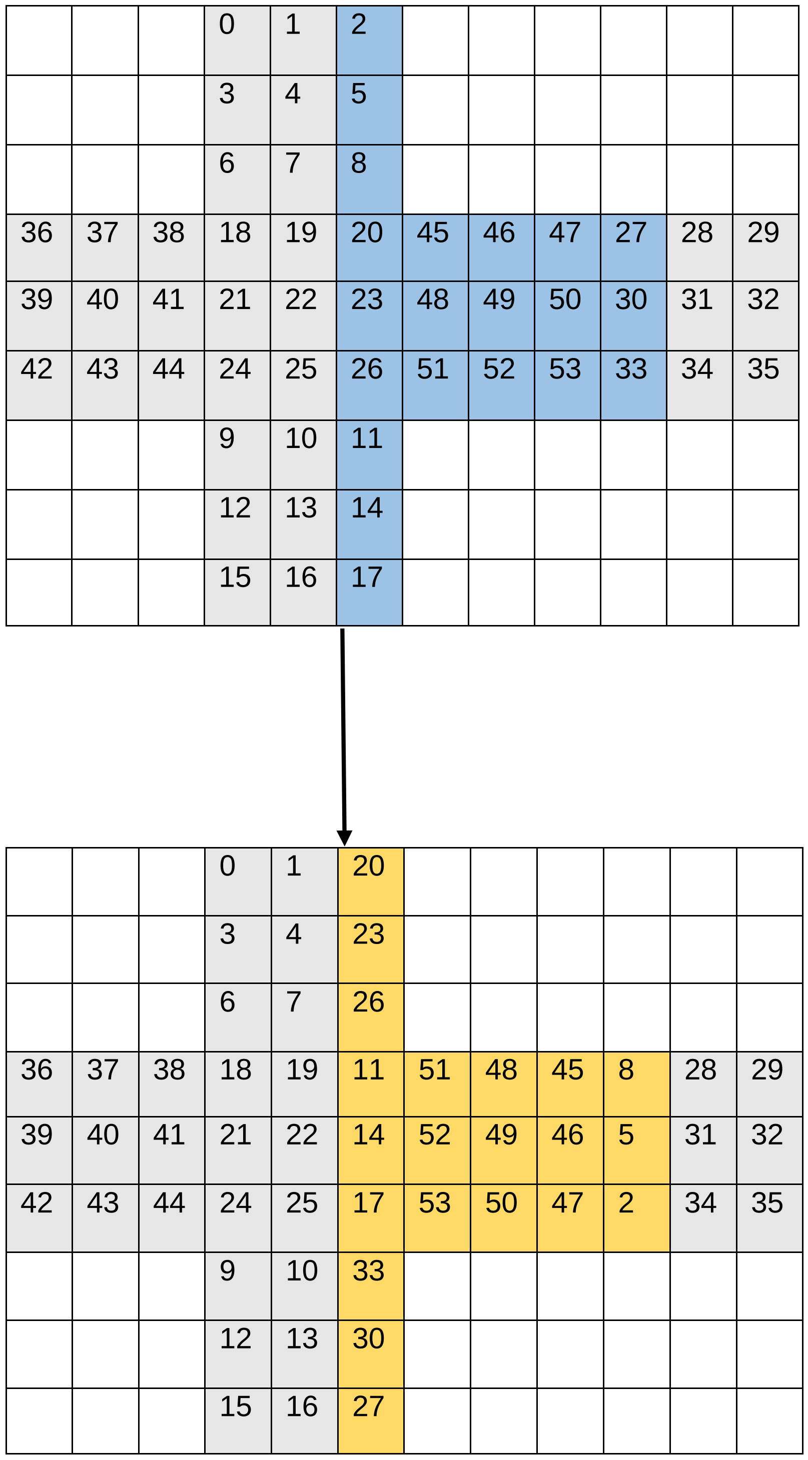
Reminded that he face_color_buffer_data is a 2d array which contains the color of all 54 facets of the cube and the facets_buffer_data is a 2d array which contains the position of all 54 facets of the cube. The graph 14 and graph 15 below show the changes of the face_color_buffer_data and facets_buffer_data:



			0	0	0						
			0	0	0						
			0	0	0						
4	4	4	2	2	2	5	5	5	3	3	3
4	4	4	2	2	2	5	5	5	3	3	3
4	4	4	2	2	2	5	5	5	3	3	3
			1	1	1						
			1	1	1						
			1	1	1						

			0	0	2						
			0	0	2						
			0	0	2						
4	4	4	2	2	1	5	5	5	0	3	3
4	4	4	2	2	1	5	5	5	0	3	3
4	4	4	2	2	1	5	5	5	0	3	3
			1	1	3						
			1	1	3						
			1	1	3						

GRAPH 14. The changes in face_color_buffer_data.



			0	1	2						
			3	4	5						
			6	7	8						
36	37	38	18	19	20	45	46	47	27	28	29
39	40	41	21	22	23	48	49	50	30	31	32
42	43	44	24	25	26	51	52	53	33	34	35
			9	10	11						
			12	13	14						
			15	16	17						

			0	1	20						
			3	4	23						
			6	7	26						
36	37	38	18	19	11	51	48	45	8	28	29
39	40	41	21	22	14	52	49	46	5	31	32
42	43	44	24	25	17	53	50	47	2	34	35
			9	10	33						
			12	13	30						
			15	16	27						

GRAPH 15. The changes in facets_buffer_data.

The permutation of the positions of 54 facets of the cube can be viewed as the following mapping:

45→ 47, 47 → 53, 53 → 51, 51 → 45.

46 → 50, 50 → 52, 52 → 48, 48 → 46.

17 → 26, 26 → 8, 8 → 27, 27 → 17.

14 → 23, 23 → 5, 5 → 30, 30 → 14.

11 → 20, 20 → 2, 2 → 33, 33 → 11.

GRAPH 16. The mapping of facets' changes

To implement the changes in face_color_buffer_data and facets_buffer_data described in the graph 14 and 15, we need the help of the permute function in the Permutation class which was implemented as an algorithm class in which most of the functions are public static.

```
void Permutation::Permute(int* _old, int* _new, int n)
{
    int temp = _old[_new[n - 1]];
    for (int i = n - 1; i > 0; --i)
        _old[_new[i]] = _old[_new[i - 1]];
    _old[_new[0]] = temp;
}
```

GRAPH 17. Main Permute function.

Then, each basic move can be implemented using the permutation method shown as below, and the implementations for other moves are done in a similar way.

```
void RubikCube::rotateRightFaceClockwise()
{
    int arr[5][4] = {{45, 47, 53, 51},
                    {46, 50, 52, 48},
                    {17, 26, 8, 27},
                    {14, 23, 5, 30},
                    {11, 20, 2, 33},};
    Permutation::MulPermute((int*)facets_buffer_data, (int*)arr, 4, 5);
    Permutation::MulPermute((int*)face_color_buffer_data, (int*)arr, 4, 5);
}
```

GRAPH 18. rotateRightFaceClockwise method.

3.4 Development platform

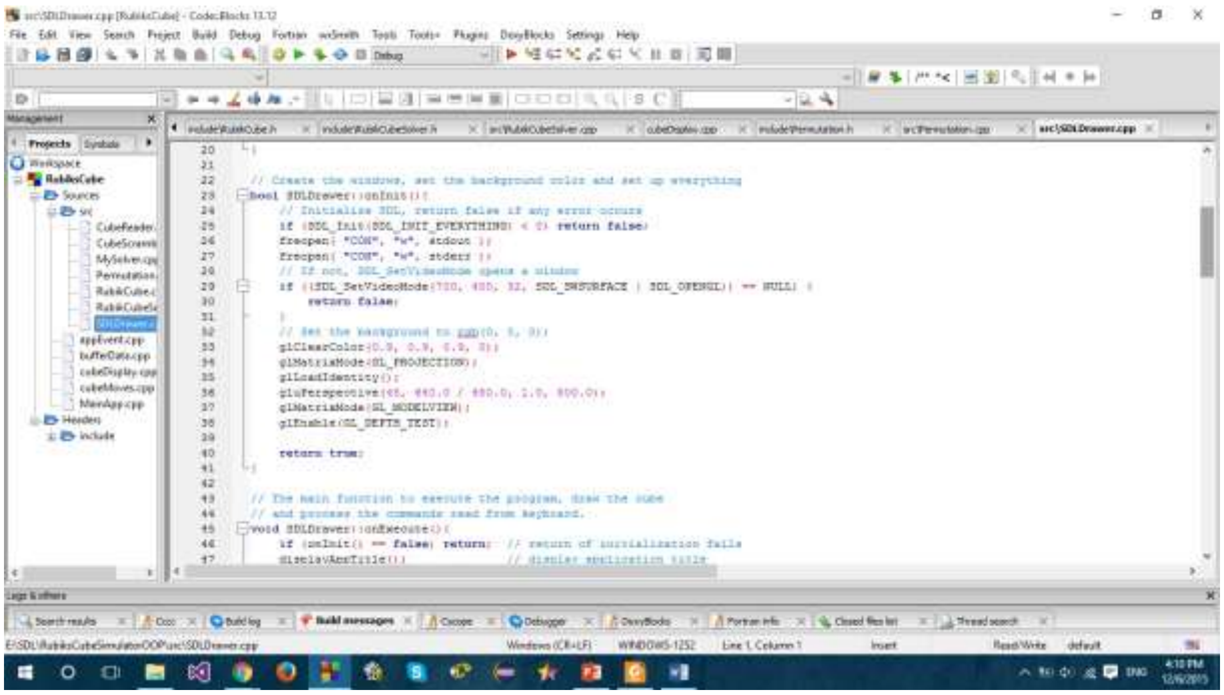
SDL can run on various platforms such as Windows, Mac OS X, Linux, iOS, Android. This project was done under Windows platform. Specifically, it uses Win32 APIs for display, taking advantage of Direct3D for hardware acceleration. It also uses DirectSound and XAudio2 for sound (SDL Wiki 2015.).

3.5 Software requirements

3.5.1 Code Blocks as an IDE

Code::Blocks is a cross-platform IDE that supports compiling and running multiple programming languages. Code::Blocks is a free C, C++ and Fortran IDE built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable. (Codeblocks 2015.) You can check the following link to download and install the software if you want: http://www.codeblocks.org/downloads. Code::Blocks allows to create and test the programs easily. It is also fast and easy to install.

There are several other choices for IDEs such as QtCreator, Eclipse, Visual Studio. They have both advantages and disadvantages. However, I chose Code::Blocks for this project just because I used to work with it before in a school course; hence, that is the one I feel comfortable the most to work with.



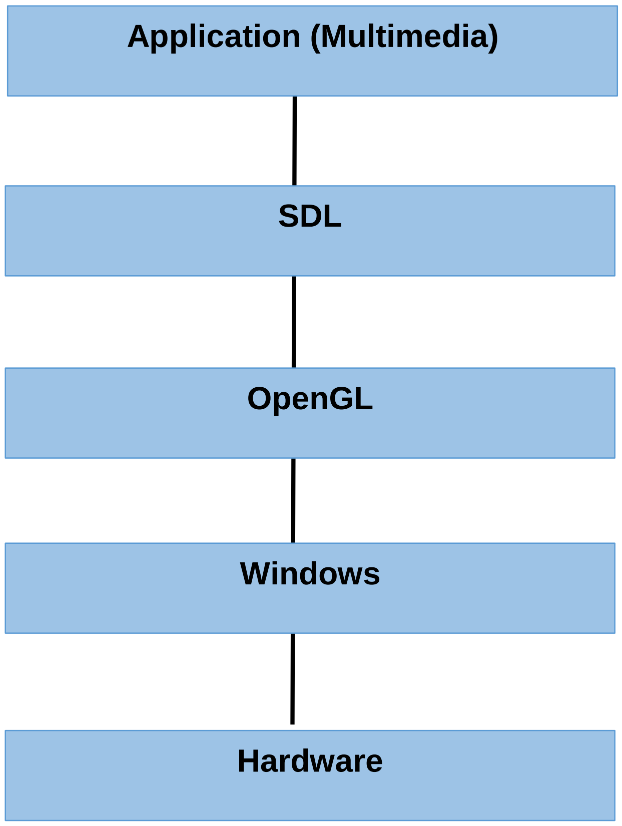
```
}

// Create the window, set the background color and set up everything
bool SDLDrawer::onInit(){
    // Initialize SDL, return false if any error occurs
    if (SDL_Init(SDL_INIT_EVERYTHING) < 0) return false;
    freopen( "CON", "w", stdout );
    freopen( "CON", "w", stderr );
    // If not, SDL_SetVideoMode opens a window
    if ((SDL_SetVideoMode(720, 480, 32, SDL_SWSURFACE | SDL_OPENGL)) == NULL) {
        return false;
    }
    // Set the background to rgb(0, 0, 0);
    glClearColor(0.9, 0.9, 0.9, 0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45, 640.0 / 480.0, 1.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_DEPTH_TEST);

    return true;
}

// The main function to execute the program, draw the cube
// and process the commands read from keyboard.
void SDLDrawer::onExecute(){
    if (onInit() == false) return;  // return if initialization fails
    displayAppTitle();              // display application title
```

GRAPH 19. Code::Blocks IDE.

3.5.2 OpenGL with SDL

The cube was drawn using OpenGL with SDL 1.0 and the programming language of the application is C++. OpenGL is an API which consists of a set of functions for defining 2D and 3D graphic images. It comprises several libraries with varying levels of abstraction like GL, GLU. But OpenGL only deals with rendering graphics, it does not provide functions for animations, timing, file I/O, and so forth (OpenGL Wiki 2014). That is why I use OpenGL along with SDL to deal with all those drawbacks.

SDL is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D (SDL Wiki 2015.). Check the following link to download and install the library: https://www.libsdl.org/download-2.0.php



GRAPH 20. Abstraction layers of SDL under Windows platform.

3.6 SDL features used in this program

Below are some of the SDL features used in my project.

3.6.1 Video

The video feature in SDL help to render 3D graphics, and can be used in combination with the OpenGL API for 3D graphics. It is also useful for accelerated 2D render API. It can support easy rotation, scaling and alpha blending, all accelerated using modern 3D APIs. (SDL Wiki 2015.)

```
if (SDL_Init(SDL_INIT_EVERYTHING) < 0) return false;
if ((SDL_SetVideoMode(700, 400, 32, SDL_SWSURFACE |
SDL_OPENGL)) == NULL) {
        return false;
        }
glClearColor(0.9, 0.9, 0.9, 0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45, 640.0 / 480.0, 1.0, 500.0);
glMatrixMode(GL_MODELVIEW);
glEnable(GL_DEPTH_TEST);
…
…
```

GRAPH 21. SDL video features.

3.6.2 Timers

The Timers feature in SDL helps to get the number of milliseconds elapsed or wait a specified number of milliseconds. It also can create timers that run alongside your code in a separate thread and use high resolution counter for profiling. (SDL Wiki 2015.)

In this project, the SDL_Delay() function was used to create the animation for the cube when being rotated. This function is used to wait for a specified number of milliseconds before returning (SDL Wiki 2015.).

3.6.2 Input events

The input events feature in SDL helps to read inputs from mouse, keyboard, joystick and other game controllers. Each event can be enabled or disabled with SDL_EventState(). Events are passed through a user-specified filter function before being posted to the internal event queue. (SDL Wiki 2015.)

The SDL_PollEvent() in line 8 of graph 22 is used for polling for currently pending events. It will return 1 if there is a pending event or 0 if there are none available. Then the SDL_GetKeyState() in line 17 of graph 22 gets a snapshot of the current keyboard state. The current state is return as a pointer to an array, the size of this array is stored in numkeys. The array is indexed by the SDLK_* symbols. A value of 1 means the key is pressed and a value of 0 means it is not. The pointer returned is a pointer to an internal SDL array and should not be freed by the caller. (SDL Wiki 2015.)

```
 1  void SDLDrawer::onExecute(){
 2          if (onInit() == false) return;  // return of initialization fails
 3          displayAppTitle();              // display application title
 4          displayAppMenu();               // display application menu
 5          SDL_Event evnt;                 // create SDL_event variable
 6          while (appState == ON)
 7          {
 8                  while (SDL_PollEvent(&evnt)){
 9                          onEvent(&evnt);}
10                  readKeyboardInput();
11          }
12          onCleanup();
13  }
14  …
15  void SDLDrawer::readKeyboardInput()
16  {
```

```
            Uint8* keystate = SDL_GetKeyState(NULL);
            if (keystate[SDLK_s])
    {
        while (true)
        {
        cout << "\n\n\[ENT]: Select ";
        string command;
        getline(cin, command);
        …
    }
…
}
```

GRAPH 22. SDL Input Events features.

4 ALGORITHM

This chapter introduces the mathematical model and explains how to apply those concepts on Rubik’s Cube. After that, the strategy for solving the cube and the explanation of the application of the virtual cube for applying the algorithms will be described.

4.1 Permutation

A **permutation** of a set of distinct symbols is an arrangement of them in a line in some order (Goodaire, Parmenter 2002). For example: abc, acb, bac, bca, cab and cba are permutations of the symbols a, b and c; 12345, 12354 and 23451 are permutations of the symbols 1, 2, 3, 4, and 5.

The more formal way to define is: A **permutation** of a finite set T is a bijection from T to itself (Goodaire, Parmenter 2002). A bijection function is both one-to-one (injection) and onto (surjective). With new definition, if we have a set T = {1, 2, 3, 4, 5} and a bijection f : P → P defined as following: f(1) = 2, f(2) = 3, f(3) = 4, f(4) = 5 and f(5) = 1. So such bijection is also a permutation.

We already know that the cube consists of 26 visible cubies and 54 facets. The cubie permutation and facet permutation will tell us how the cube changes in a rotation. If every cubie is seen as an object, the cubie permutation is similar to the arrangement of the cubies in slots of the Cube. Hence, every move will be presented by a cubie permutation which tells us where the cubies go. For example: the up-face rotation will produce the following permutation of cubies: f(UR) = UF; f(UF) = UL; f(UL) = UB; f(UB) = UR; f(URB) = URF; f(URF) = ULF; f(ULF) = UBL; f(UBL) = URB.

The above notation is pretty long and tedious. A more convenient way to denote a permutation is using a 2xn matrix. Then the bijection f of set T above can be rewritten like this:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ f(1) & f(2) & f(3) & f(4) & f(5) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 5 & 4 \\ 2 & 3 & 4 & 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 2 & 4 & 5 \\ 2 & 4 & 3 & 5 & 1 \end{pmatrix}$$

All matrices above may be different in the order of numbers on the first row but they define the same permutation.

With this elegant notation, now we can rewrite the permutation of cubies of up-face rotation in the example above like this:

$$\mathrm{U} = \begin{pmatrix} \mathrm{UR} & \mathrm{UF} & \mathrm{UL} & \mathrm{UB} & \mathrm{URB} & \mathrm{URF} & \mathrm{ULF} & \mathrm{UBL} \\ \mathrm{UF} & \mathrm{UL} & \mathrm{UB} & \mathrm{UR} & \mathrm{URF} & \mathrm{ULF} & \mathrm{UBL} & \mathrm{URB} \end{pmatrix}$$

But there is still a more compact and elegant way to denote permutations. It is called ***canonical cycle notation***. In this notation, a permutation can be expressed as a product of cycles. For example: (1 2 3 4 5) means that 1 goes to 2, 2 goes to 3, 3 goes to 4, 4 goes to 5, and 5 goes to 1; (1 2 3)(4 5) means that 1 goes to 2, 2 goes 3, 3 goes to 1, 4 goes to 5, and 5 goes to 4.

Similar to the matrix notation, the cycle notation is also not unique. All the permutations below are basically the same:

(1 2 3 4 5) (2 3 4 5 1) (3 4 5 1 2) (4 5 1 2 3) (5 1 2 3 4)

In this notation, if there is any element that stays in place, we can simply put it in a separate parenthesis or even omit it. Consider the following permutation where the bijection is defined: f(1) = 1, f(2) = 3, f(3) = 4, f(4) = 5, f(5) = 2 where 1 stays in place. Then the cycle notation is (1)(2 3 4 5) or just simply (2 3 4 5).

The steps in writing down the products of permutations in canonical cycle notations are as follows:

Find the smallest item in the list and begin a cycle with it. In the example above: 1 is the smallest item then start a cycle with 1.

Complete this first cycle by following the movement of the objects by the permutation until it closes the cycle. In this example, 1 goes to 2 in the first cycle, then 2 goes to 5 in the cycle 3, then 1 goes to 5 in the resulting permutation. Thus, write down 1 and 5. Next, following 5 shows that it goes back to 1. So 1 and 5 form 2-cycle.

Continue another cycle with the next smallest number as in the step 2 until we use up all the numbers.

The resulting combination is:

PQ = (1 2 3) (4 5) (2 5) (1 3 4) = (1 5) (2 4) (3)

4.2 Group Theory

In mathematics, group is an algebraic structure which is defined by some axioms and theorems. A set of moves (permutations) of Rubik's Cube comprises a group. And this group is quite large and complex to investigate. Let's have a closer look at how the properties as group elements can be used as a tool to help us manipulate the Cube.

4.2.1 Definitions and properties

A group G is a non-empty set of objects and a binary operator • for which the following properties are satisfied:

- (Closure) If a and b $\in$ G, then a • b is also $\in$ G.
- (Associativity) a • (b • c) = (a • b) • c, for all a, b, c $\in$ G.
- (Identity) There is an element e $\in$ G such that a • e = e • a = a, for all a $\in$ G.
- (Inverse) If a $\in$ G, then there is an element a^{-1} $\in$ G such that a • a^{-1} = a^{-1} • a = e.

Some examples of group are: The set of all non-empty set of integers Z (= {… -3, -2, -1, 0, 1, 2, 3 …}) forms a group under the operation of addition. But the set of natural numbers N (= {0, 1, 2, 3 …}) under addition does not form a group because there are no inverse for any element. The set of all non-zero rational numbers also forms a group under the operation of multiplication.

The **order of a group** G, denoted by |G|, is the cardinality of G that is the number of elements in G. From now, for the sake of convenience, sometimes we can adopt the multiplicative notation for groups, where the operation a • b of 2 elements a and b of a group G is denoted by ab.

So how Rubik's Cube moves can form a group? At this point, we know that rotations of the faces of Rubik's Cube produce permutations of the cubies and the facets on the Cube. If we define a binary operator, •, like this: if r_1 and r_2 are 2 permutations (or elements) of R, then $r_1 \bullet r_2$ means applying the permutation (or move) r_1 followed by the permutation (or move) r_2, then the set of permutations on the Cube will satisfy all the above properties to form a group under the binary operator • we just defined. We denote this group to be R.

- (Closure) Applying any sequence of moves (permutation) will leave us also a move (permutation) of the Cube which is in R.
- (Associative) No matter how we group the permutations in a combination, as long as we manipulate it in the right order, the result will end up the same.
- (Identity) The identity element of R is not changing the Cube at all.
- (Inverse) If r is a move in R, then we can reverse all the steps we do in r to transform the Cube back to the state before r. If we call that reverse move is r^{-1} then $r \bullet r^{-1} = e$ (1) equals we do nothing with the Cube.

Because the elements of R are permutations, so we R can also be called **permutation group**. Even though Rubik's Cube group is finite, from the previous chapter, we know this group consists of approximately 43 quintillion elements. Because this group is fairly large and complex, let's look in detail at a much smaller permutation group, say A, is a group of all permutations of the set {1 2 3}. There are 6 elements in this group. They are:

(1), (1 2), (1 3), (2 3), (1 2 3), (1 3 2)

Below is the multiplication table for the permutation group of 3 objects described above.

	(1)	(1 2)	(1 3)	(2 3)	(1 2 3)	(1 3 2)
(1)	**(1)**	(1 2)	(1 3)	(2 3)	(1 2 3)	(1 3 2)
(1 2)	(1 2)	**(1)**	(1 3 2)	(1 2 3)	(2 3)	(1 3)
(1 3)	(1 3)	(1 2 3)	**(1)**	(1 3 2)	(1 2)	(2 3)
(2 3)	(2 3)	(1 3 2)	(1 2 3)	**(1)**	(1 3)	(1 2)
(1 2 3)	(1 2 3)	(1 3)	(2 3)	(1 2)	(1 3 2)	**(1)**
(1 3 2)	(1 3 2)	(2 3)	(1 2)	(1 3)	**(1)**	(1 2 3)

GRAPH 23. Multiplication table for permutation group with 3.

Observing from the multiplication table, we notice that in each column, there is one and only one identity element, and it means that every element in the group has the inverses and they are all unique.

One property was not stated in the definition of group is Commutative Law. So, Rubik's Cube group R is commutative? The integer group under addition has this property since ab = ba with for all integers a and b. Is this the case for permutation group? Let consider an example of 2 permutations P_1 = (1 2) and P_2 = (2 3). Then the products P_1P_2 = (1 2) (2 3) = (1 3 2) and P_2P_1 = (2 3) (1 2) = (2 3 1) do not have the same result. Thus, the order of the permutations in the products do matter. If we apply the front twist followed by the right twist to the Cube, the result is not the same as we apply the right twist followed by the front twist to the Cube. If a group is commutative, we call that **Abelian group**.

4.2.2 Special classes of group

Permutation groups

A permutation group is a finite group G whose elements are permutations of a given set and whose group operations composition of permutations in G. Permutation groups have orders dividing n!. Rubik's Cube group is a typical example in this class which we have already studied and examined some properties of it.

Symmetric groups

Symmetric group is the group of all permutations we can make of n objects. More precisely, the symmetric group on a finite set X is the group whose elements are all bijective functions from X to X and whose group operation is that of function composition. The symmetric group of **degree** n is the symmetric group on the set X = {1, 2... n}. Since there are n! possible permutation operations that can be performed on *n* symbols, the **order** of the symmetric group S_n is *n*!.(Mulholland 2015.)

Cyclic groups

A group *G* is called cyclic if there exists an element *g* in *G* such that $G = \langle g \rangle = \{ g^n \mid n$ is an integer $\}$. Since any group generated by an element in a group is a subgroup of that group, showing that the only subgroup of a group *G* that contains *g* is *G* itself suffices to show that *G* is cyclic. For example, the set of integers, with the operation of addition, forms a group. It is an infinite cyclic group, because all integers can be written as a finite sum or difference of copies of the number 1. In this group, 1 and −1 are the only generators. (Mulholland 2015.).

4.3 Cycle structure of cube moves

From the previous part, we know how to combine or multiply permutations. Let's define P = (2 3 4 5), we can easily calculate that:

P^2 = PP = (2 3 4 5)(2 3 4 5) = (2 4)(3 5);

P^3 = PPP = (2 3 4 5)(2 3 4 5)(2 3 4 5) = (2 5 4 3);

P^4 = PPPP = (2 3 4 5)(2 3 4 5)(2 3 4 5)(2 3 4 5) = (2)(3)(4)(5)=e.

If we continue with P^5, we will have $P^5 = P^4P = eP = P$. The permutation P contains just a single 4-cycle. This cycle has the length of 4. After repeating this cycle 4 times, the result will be the same as changing nothing. So this cycle is said to have order of 4. More general, if $P = (x_1\ \ x_2 \ldots x_n)$ has **order** n, then $P^{kn} = e$ (or 1) with k = 1, 2, 3,… Consider the above example, if we apply the operation 4 times, 8 times, or 12 times, the result will be the identity permutation.

If P consists of multiple cycles of varying length, then the **order of permutation** P is the least common multiple (LCM) of the lengths of the cycles, since that number will return all the cycles to their beginning states. Let's consider the permutation that looks like this: P = (1 2) (3 4 5) contains both a 2-cycle a 3-cycle. These 2 cycles are disjoint and do not have any elements in common. So if we apply P two times and ignore the 3-cycle, the 2-cycle will disappear. Similarly, if we apply P three times and ignore the 2-cycle, the 3-cycle will disappear. Specifically, $P^2 = (3\ \ 5\ \ 4)$, $P^3 = (1\ \ 2)$. Moreover, P has order of 6, 6 is both multiple of 2 and 3. So if we apply P six times, both cycles will disappear and we have $P^6 = e$.

This time let's consider the real example on Rubik's Cube. Consider the following sequence of moves "RUru": in order to write this sequence of moves in cycle notation, first let's examine the cycle notations of cubies for each single move.

R = (FR UR BR DR) (DRF URF URB DBR)
r = (BR UR FR DR) (URB URF DRF DBR)
U = (UR UF UL UB) (URB UFR ULF UBL)
u = (UL UF UR UB) (ULF UFR URB UBL)

Combining all these moves, we have:

RUru = (FR UR BR DR) (DRF URF URB DBR) (UR UF UL UB) (URB UFR ULF UBL) (BR UR FR DR) (URB URF DRF DBR) (UL UF UR UB) (ULF UFR URB UBL)
= (FR UR UB) (DRF UFR) (UBL URB)

The resulting permutation consists of 2 2-cycles and 1 3-cycle. The order of the permutation is LCM (3, 2) = 6. If we applies *(*RUru*)* 2 times, then all the 2-cycles disappear, just 3-cycle stays in the permutation. It means that if we do the operation *(*RUru*)* ^{2}, the operation just really exchanges 3 cubies, leaving all those 4 cubies fixed. Similarly, if we do the operation *(*RUru*)* ^{3}, then 3-cycle will disappear, the other 4 cubies still stay on the cycles.

4.4 Strategy

If you did some searching, you would see that there was really a legion of methods for solving the Cube from beginner method which just requires to learn around 10 macros to a really advanced method which requires you to learn by heart more than 100 macros. The more macros you have in your hand, the more confidence you can feel and probably the more efficient method that you can build up.

Solving the Cube is all about putting the 12 edge cubies and 8 corner cubies in the right positions and orientations. So whatever method you try to reach for, they are probably different in recipes because of the type of macros you use and the order of cubies you want to fix, but they essentially serve for the same purpose. So based on this thought, the solution could be built, for example, one could fix the edge cubies first and corner cubies later or vice versa. When fixing the edge cubies, the edge macros will be used, after the edge cubies are done, the corner macros will be used. That is why the more macros you know, the more efficient the solution you could think of.

Next, a number of essential macros will be listed along with the descriptions. With this list of macros, you are free to build your own way for solving the Cube by yourself. After that, the layer method will be introduced and the explanation for the macros used in the method will help to clarify how to the cubies will be fixed. This is one of the most simple but easy-to-approach method and this method just uses not-too-long macros which are easy to remember and follow. And finally, the edge-corner method will be introduced. Because there are no systematically step-by-step guides for this method, it will be described briefly how to apply the macros for it.

4.5 Design macros

Macro is the sequence of moves in which after doing the macro, some properties of the Cube are achieved. Some examples are: the macro to cycle three corner cubies (URB UFR, ULF) clockwise on the top face leaving everything else of the Cube unchanged, the macro to flip two corner cubies (ULF, URF) on the top face in place leaving everything else of the Cube unchanged.

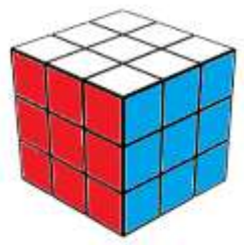
4.5.1 Conjugates

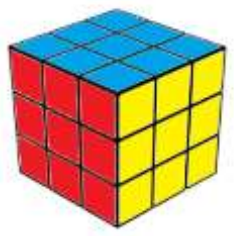
Suppose you know the formula for cycling 3 edge cubies on the top face but now you need to cycle 3 edge cubies on the other faces and do not want to waste time for learning formulas. Let M be the macro for cycling 3 edge cubies on the top face, and P be the macro that transforms the face you want to apply M into the top face. If now you apply P first to move the face you want to apply M to the top face, after that apply the macro M to cycle 3 edge cubies on the top face as you always know, finally apply P^{-1} to restore the initial state of the cubies that are not affected by M. Therefore, $\mathbf{PMP^{-1}}$ is called the **conjugate** of A (David 2006.). Sometimes, conjugations can be referred to be the changes of coordinates in mathematics.

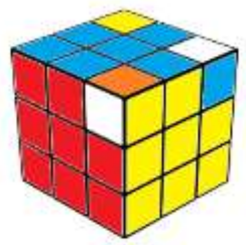
People use conjugations a lot when solving the Cube, sometimes even subconsciously. They know the macro that fixes some cubies or facets on a face but not for other faces;

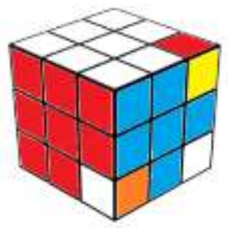
so just need to turn to the face for which you know the formula to apply and after applying the macro M to that face, finally turn it back to the state before applying M.

In the part when the strategy for solving the Cube is mentioned in this paper, whenever I say you need to apply macro, say M = “rULuRUlu”, on the right face, but you know that this macro is cycling 3 edge cubies on the top face, then you should need to turn the right face to the top face first, after that apply M, and turn the cube back to its initial state. The graph 24 will give you the clear demonstration about the example that I just stated.


Step 0: Initial state


Step 1: Make front face(blue) as top face


Step 2: Apply “rULuRUlu”


Step 3: Change back to the front

GRAPH 24. The conjugate example.

4.5.2 Commutators

In general group, if the group operation is commutative then the following property is satisfied: If a, b are the elements of the group then ab = ba. But we all know that this is not the case for Rubik's Cube group because that property is not satisfied in every case. Let's consider a simple example with 2 group elements in R, say F and R, applying the front face rotation followed by the right face rotation will not end up the same result as applying the right face rotation first followed by the front face rotation. But not all pairs of permutations do not commute, for example the U and D rotations are commutative, doing U before D, or D before U does not affect the result.

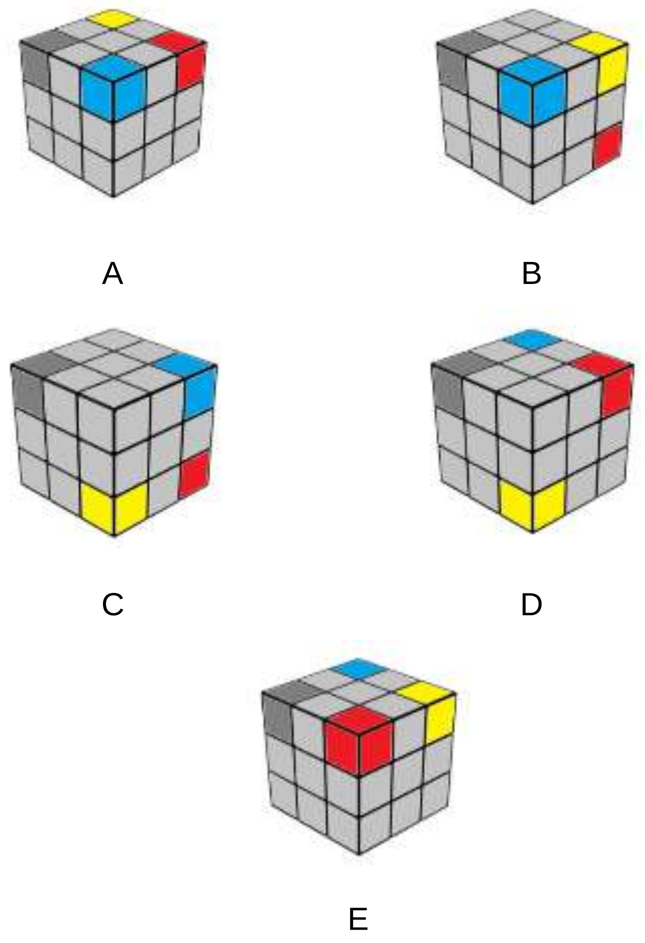
To formalize this, If P and M are two elements of a group (are two permutations, for example), then the **commutator** of P and M, sometimes written [P, M] is defined to be **$PMP^{-1}M^{-1}$** (David 2006.). It is easy to notice that if the group is commutative, then the commutator is simply the identity element of the group because $PMP^{-1}M^{-1} = P(MP^{-1})M^{-1} = P(P^{-1}M)M^{-1} = (PP^{-1})MM^{-1} = ee = e$. And with Rubik's Cube group, the identity element is doing nothing with the Cube. But when the group is not commutative and the commutator is not the identity element of the group, but it can become the measure of the degree of commutativity by the number of cubies or facets being changed. The purpose is to find the commutator which changes just a few of cubies or facets and maintains some properties.

Let's have a closer look at a detailed example to fully understand the idea behind this. The purpose is to find the macro to cycle 3 corner cubies on the top face, say UBR (the yellow cubie), UFR (the red cubie) and ULF (the blue cubie) in graph 25.A, clockwise leaving everything else unchanged. So in the end, the yellow cubie should go to the red cubie, the red cubie should go to the blue cubie and the blue cubie should go to the yellow cubie. Assume that you know the counter-clockwise right-face rotation will make the yellow and red cubies move like in the graph 25.B. Then the macro "ULu" will move the blue cubie to the yellow cubie and move the yellow cubie to the cubie DFL like in the graph 25.C. Now we need to restore the Cube by apply the inverse of "r" followed by the inverse of "ULu". The macro "R" will change back the UFR (now the blue cubie) to URB position and the new DRF cubie (now the red cubie) to the UFR position. The macro "Ulu" will cycle the yellow cubie back to UFR position, and move the red cubie the ULF position as we want.

If the plain explanation is not completely clear, probably more mathematical explanation will make sense. Consider the permutations for each sequence of moves below:

P = r = (BR UR FR DR) (URB UFR DRF DBR)
P^{-1} = R = (FR UR BR DR) (DRF UFR URB DBR)
M = ULu = (BL UF FL DL) (ULF UFR DFL   DLB)
M^{-1} = Ulu = (FL UF BL DL) (DFL UFR ULF DLB)
$PMP^{-1}M^{-1}$ = r(ULu)R(Ulu) = (URB UFR ULF)

The resulting permutation looks spectacular and goes as we want. The permutations “r” and “ULu” move a lot of cubies but their commutator just really cycles 3 corner cubies on the top face. Because the permutations of edge cubies and permutations of corner cubies are disjoint so if we just consider the product of all the permutations of edge cubies, it is trivial to see that “R” will cancel all the changes in “r”, and “Ulu” will cancel all the changes in “ULu” because they are inverses to each other. Then the product is just the permutations of corner cubies.



GRAPH 25. The commutator example.

Commutators are really helpful in developing macros. Some of the most essential macros will be introduced in the next part are also built from commutative property. Of course, you can try to work out your own set of macros. Some useful building blocks for commutators are provided below; and they can help you to work out the puzzle by yourself. (David 2006.)

FUDLLUUDDRU : flips exactly one edge cubie on the top face

rDRFDf : twists one cubie on a face

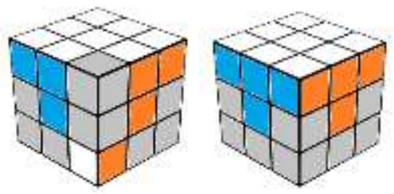
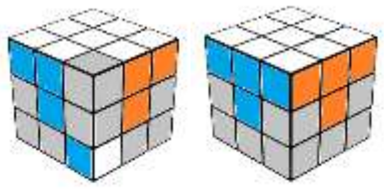
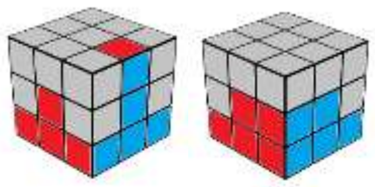
FF : swaps a par of edges in a slice

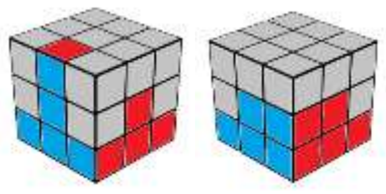
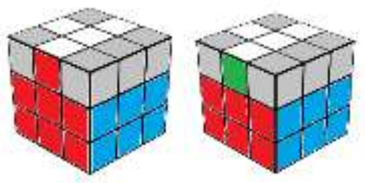
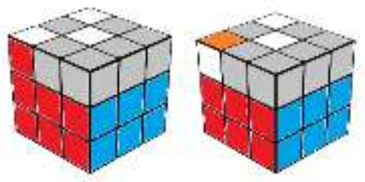
rDR : cycles three corners

4.5.3 Essential macros

Simple macros

These macros are usually short and easy to remember and often change a lot cubies.

Name	***Figure***	***Effect and explanation***
M1	 FDf	Bring a corner cubie (FRD) from the bottom to the top face directly above it and rotate it clockwise on the way up without changing any cubies on the top face.
M2	 rdR	Bring a corner cubie (FRD) from the bottom to the top face directly above it and rotate it counter-clockwise on the way up without changing any cubies on the top face.
M3	 ulULUFuf	Move an edge cubie (UF) from the top face to the middle face cubie (FL) without altering the down face at all.

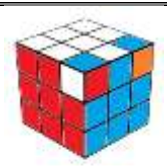
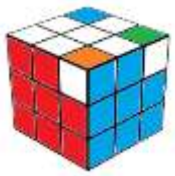
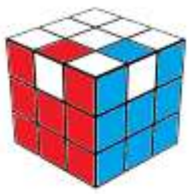
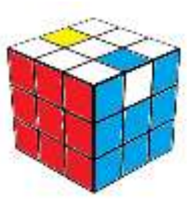
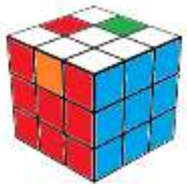
M4	 URurufUF	Move an edge cubie (UF) from the top face to the middle face cubie (FR) without altering the down face at all.
M5	 RUrURU2r	Cycle 3 edge cubies UL, UB and UR counter-clockwise without touching 2 bottom layers.
M6	 FubUfuBU2	Swap two corner cubies (ULB, UBR) at the back of the up face and do not change any cubies on 2 bottom layers.

GRAPH 26. List of simple macros

More advanced macros

These macros are usually longer in formula and often touch 2-3 cubies.

Name	***Figure***	***Effect and explanation***
t2c		Twist two corner cubies (ULF, URF) in place leaving everything unchanged. This is the commutator with P =

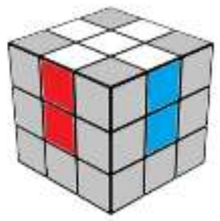
	 LdlfdFUfDFLDlu	LdlfdF, M = U. Then $MPM^{-1}P^{-1}$ gives the cubie permutation: (URB UFR ULF)
c3c	 rULuRUlu	Cycle three corner cubies (URB UFR, ULF) clockwise leaving everything unchanged. This is the commutator with P = r, M = ULu. Then $MPM^{-1}P^{-1}$ gives the cubie permutation: (URB UFR ULF)
f2ae	 FRBLUlUbrfluLu	Flip two adjacent edge cubies (UL, UF) in place leaving everything unchanged. This is the commutator with P = FRB, M = UlU. Then $MPM^{-1}P^{-1}$ gives the facet permutation: (lu ul) (fu uf).
f2oe	 LfUlFbUrFuRfBu	Flip two opposite edge cubies (UF, UB) in place leaving everything unchanged. The facet permutation is (bu ub) (fu uf).
c3e	 LrFRlU2LrFRl	Cycle three edge cubies (UL, UB, UR) clockwise and change orientations of 2 among 3 cubies. The cubie permutation is (UB UR UL). The facet permutation is (ru lu ub) (bu ur ul).

GRAPH 27. List of more advanced macros.

4.6 Beginner method

STEP 1: SOLVE THE FIRST LAYER

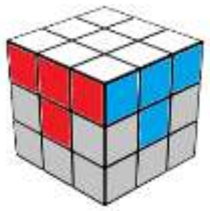
At first, choose a face and put all the edge cubies in place. In other words, solve the cross on a face. Let's say, we choose the white face to demonstrate the method. This first step is often done intuitively and this was proven that the cross can be done in 6 moves or less than 7 moves most of the time. The cross must be completed with the correct orientations of 4 white edges on the top face and looks like the graph 28. There are no particular formulas for this first step because this is pretty easy even if you are the beginner to the Cube.



GRAPH 28. The white cross is completed.

After that, we need to complete the first layer by putting 4 corner cubies in the right positions and orientations. First, try to improvise a bit to put the corner cubie on the down face right under the position on the top face we want to put it in then apply the macro **M1** or **M2**.

At this step, the macros are not so long and complex because we do not really have to care about many things on other layers and even you can solve it with just intuition and through a lot of practice. If this is done properly, you should have something like in the graph 29. But later, when we complete the first layer and the second layer, we need macro which changes some cubies and facets on the last layer but do not touch anything on the solved layers. That is when you have to face more complicated algorithms.

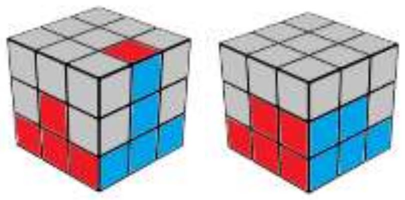


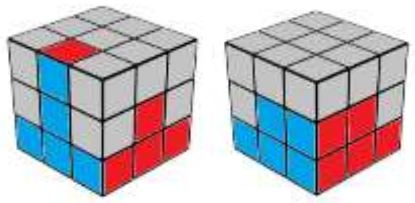
GRAPH 29. The top layer is completed.

STEP 2: SOLVE THE MIDDLE LAYER

Once the white face is completed, turn the cube over so that the white face is now the down face. The next step is put 4 middle layer edge cubies in the right positions and orientations. There are 2 algorithms used in this step. They are **M3** in the graph 30 and **M4** in the graph 31. They are used to replace either the edge cubie FL or FR by the edge cubie UF.

Let's take a closer look at how these macros actually work. I will choose one to give a detailed explanation and that for the other macro is totally the same. Let's consider the macro M3 = "ulULUFuf". It will move the UF to FL and does not touch any cubies on the 2 bottom layers. It is actually the product of commutators. One is "ulUL" and the other is "ULuf". When we perform "ulUL", the resulting permutation is (FL UB UL) (UBL URB) (DFL ULF). If we stop here, it really damages a lot cubies on the 2 bottom layers. But we do not really care about the cycle (UBL URB) because those are the corner cubies on the top face. What we care is how to cancel the cycle (DFL ULF) and how make the UF appear. The resulting permutation of commutator "UFuf" is (UR UF FL) (URB UFR) (DFL ULF). This really cancels the cycle (DFL ULF) and makes UF go to FL and all the other changes are about the edge cubies on the top face. So we succeeded in moving UF to FL along with protecting the two bottom layers.

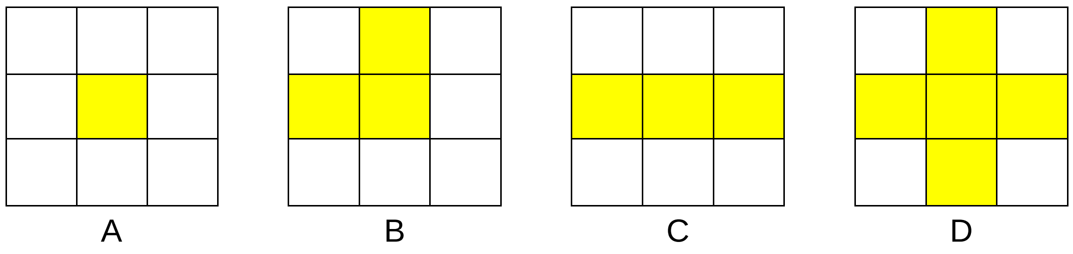

GRAPH 30. Demonstration for the effect of the macro = “ulULUFuf”.


GRAPH 31. Demonstration for the effect of the macro = “URurufUF”.

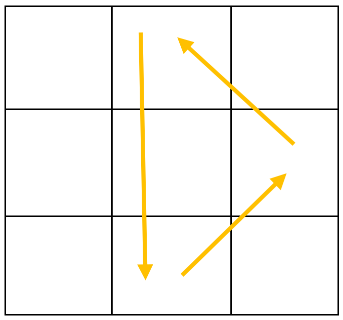
Repeat the algorithms until you put all the 4 middle layer edge cubies in place and with proper orientations. The conjugation tells us that we need to apply some Y or y move on the Cube before applying the appropriate macro. After this is done properly, you will have the Cube with 2 solved layers.

STEP 3: SOLVE THE LAST LAYER

The very first thing to do in this step is to make the yellow cross on the last face (or top face now). The algorithm used here is “FRUruf”. Repeating this algorithm several times will complete the yellow cross. So why does it work? The top face will fall into 4 of the cases like in the graph 32. If it is the case D, then you can proceed to the next step. If it is the case A, then apply algorithm once, we will have the top face look like one of the 3 later cases.



GRAPH 32. Cases to solve yellow cross.

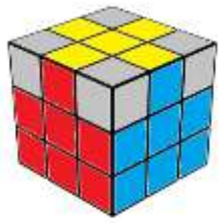


GRAPH 33. The cubie permutation of the macro “FRUruf”.

The cubie permutation of the macro “FRUruf” is (UB UF UR) (ULF UFR) (UBL URB). At this point, we just need to care about edge cubies, now let’s ignore the 2 corner cubies 2-cycles for now. The 3-cycle tells us the cubies UB, UF and UR will cycle like in the graph 36. The facet permutation of the macro is (rub blu bru lub urb ulb) (ruf ful fur luf urf ulf) (fu ur ub) (ru bu uf). The changes of edge cubies in the permutations will make the top face change from case A to B, from B to C, then from C to D. That is why based on where you already are, you will have to apply the macro several times to achieve the yellow cross like graph 32D.

The next concern is how to put the edge cubies on the top face in their proper positions and the algorithm used for now is “RUrURU2r”. Similarly, in order to understand how the macro work, let’s dive into the cubie and facet permutations of the macro.
The cubie permutation = (UL UR UB) (UBL UFR) (ULF URB).
The facet permutation = (ruf blu urf lub fur ulb) (ul ur ub) (ful bru luf rub ulf urb) (ru bu lu).

Before applying this algorithm, you need to do some up face twist to put the cubie UF in its position. The macro will cycle three edge cubies UL, UR and UB without changing their orientations. Then applying this several times plus some observation to rotate the Cube when needed to help you to complete this step and you should have the Cube look like the graph 34.



GRAPH 34. The Cube after completing 2 bottom layers.

The final step is to put the last 4 corner cubies in right place with proper orientations. I suggest 2 algorithms which can help you to complete this step. They are **c3c** = “rULuRUlu” and **t2c** = “LdlfdFUfDFLDlu“. The macro “rULuRUlu” will cycle three corner cubies (URB UFR, ULF) clockwise leaving everything unchanged. This is the commutator with P = r, M = ULu. And the macro “LdlfdFUfDFLDlu” will flip two corner cubies (ULF, URF) in place leaving everything unchanged. This is the commutator with P = LdlfdF, M = U. You also need a quick observation to complete this step fast. If this is done properly, you will have the solved cube.

4.7 God’s algorithm

The God’s number tells us that we can solve any instance of Rubik’s Cube in 20 moves or fewer with haft-turn metric system. But it does not tell us specifically how to solve a given cube. We just know that a solution exists for all possible legal cube configurations, and that solution is guaranteed to be achievable in 20 moves or fewer. In fact, the algorithm which produces a solution having the fewest possible number of moves is called **God’s algorithm**. It also refers to the optimal method of solution for any other permutation puzzles. From the Wikipedia article “Optimal solutions for Rubik’s Cube”, you can see the latest progress concerning God’s algorithm people have made to date. There are few algorithms which produce the optimal solutions so far; some examples are: Thistlethwaite's algorithm by Douglas Hofstadfer in 1981 with 52 moves for upper bound, Kociemba’s algorithm by Herbert Kociemba in 1992 with 29 moves for upper bound, and may other algorithm improvements.

4.8 Unsolvable cubes

The permutation theory tells us that there are 5.19 x 10^{20} possible configurations of the Cube. But it was already stated that there are approximately 12 times less than the original calculation which is approximately 4 quintillion possible configurations of the Cube. This part tells us how exactly that number is calculated.

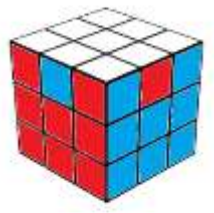
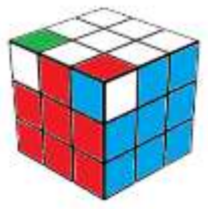
Even permutation

First, let's examine the cubie permutation of a single face turn, say F, we have:
F = (FL UF FR DF) (ULF UFR DRF DFL)
= (FL UF) (FL FR) (FL DR) (ULF UFR) (ULF DRF) (ULF DFL).

The permutation of the front face turn F gives us the total of 6 2-cycles meaning that there is an even parity of cubies exchanged after its starting position. And this applies to any face turn, since all face turns, no matter which face they are applied to, are essentially equivalent.

After single face turn, the cube has an even parity of cubies exchanged. Hence, any combination of moves will also end up an even parity of cubies exchanged. This means that there is no move which exchanges a single pair of cubies. That explains why the cubes in the graph 35 are in unsolvable or invalid states. The cubes in the graph 35.A and 35.B have odd parity of cubies exchanged that contradicts to what we just proved. That is why they are not valid configurations of the Cube.

Moreover, since only half of the permutations of the Cube has even parity and half of the permutations of the Cube has odd parity, the total number of reachable cube configurations can be recalculated like this:
(5.19 x 10^{20}) / 2.

A B

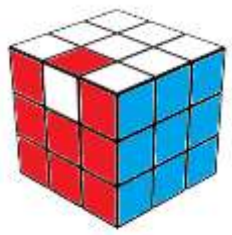
GRAPH 35. Some unsolvable configurations.

Flips of edges

This time, let's examine the facet permutation of a single face turn, say F, we have
F = (ruf drf ldf ulf) (rf df lf uf) (rfd dlf luf urf) (fur frd fld ful) (fr fd fl fu)
Ignoring the permutation of corner facets, we have:
F = (rf df lf uf) (fr fd fl fu)

This can also be written in even number of parities. After single face turn, the cube has an even parity of edge facets exchanged after its starting position. Hence, any combination of moves will also end up an even parity of edge facets exchanged. The permutation of the edge facets for the cube in graph 36 is (lu ul) which has an odd parity and leads to an invalid cube configuration.

Since only half of the cube permutations will satisfy the property we just proved, the total number of reachable cube configurations can be recalculated like this: (5.19 x 10^{20}) / (2 x 2).

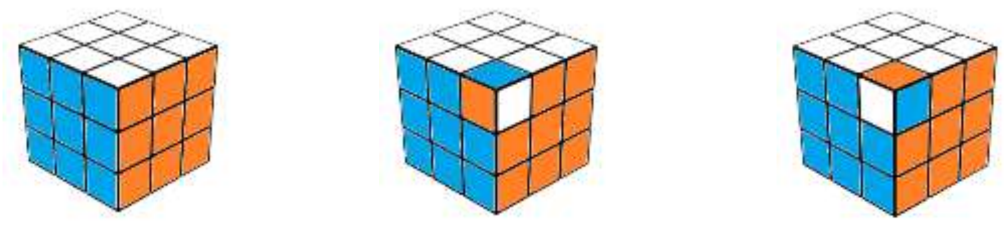


GRAPH 36. Unsolvable flip of edge cubie.

Twists of corners

This time, we have to approach the problem in different way. A corner cubie can be twisted in three different ways like the graph 37. We denote the original state as 0, a twist of 120^0 clockwise as 1, and a twist of 120^0 counter-clockwise as -1.

Notice that every corner cubie has a facet belonging to either the top face or the down face. And one more assumption is that if a corner cubie has its top or bottom facet is facing up or down, then that corner cubie is in the right orientation. Otherwise, it will fall into either "1" or "-1" cases.



GRAPH 37. Notation of orientation of corner cubies cases.

Let's number eight corner cubies of the entire cube.

UFL = 1	URF = 2	UBR = 3	ULB = 4
DLF = 5	DFR = 6	DBR = 7	DLB = 8

Let's prove the fact that a legal move of the cube always produce the sum of the orientation of all corner cubies of 0. We will calculate the value of the sum of the orientation of all corner cubies for each single face turn.

The up-face move: U = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0.

The down-face move: D = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0.

The front-face move: F = 1 + 1 + 0 + 0 + (-1) + (-1) + 0 + 0 = 0.

The back-face move: B = 0 + 0 + (-1) + (-1) + 0 + 0 + 1 + 1 = 0.

The left-face move: L = (-1) + 0 + 0 + (-1) + 1 + 0 + 0 + 1 = 0.

The right-face move: R = 0 + 1 + (-1) + 0 + 0 + (-1) + 1 = 0.

After any single face turn, the sum of the orientation of all corner cubies will end up 0. Hence, any combination of moves will also result in the sum of orientation of all corner cubies of the Cube as 0. The cubes in the graph 9 with orientation of “1” and “-1” are in invalid or unsolvable configurations because the sum of the orientation of all corner cubies is not 0. Since only one-third of the cube permutations will satisfy the property we just discussed, the total number of reachable cube configurations can be recalculated like this: $(5.19 \times 10^{20}) / (2 \times 2 \times 3)$.

Summing up, the half of the permutations are valid, half of the edge cubie permutations are valid, and one-third of the corner cubie permutations are valid. Thus, if we disassemble the Cube and reassemble the cubies back, we will have the chance of one-twelfth that the cube is solvable. This also gives us the resulting calculation for all the possible cube configurations:

$(5.19 \times 10^{20}) / (2 \times 2 \times 3) = 4.3252 \times 10^{19}$.

5 VALIDATION AND TESTING

5.2 Software

5.3.1 Function testing

The correctness of all the functionalities of the program will be tested. The functions of the program will be test by input/output model. The input will be fed into the function to see if we can get the desired output.

5.3.2 GUI testing

Besides the function testing, we also need to test the GUI or graphic user interface. This is also one of the most important phases of software testing. And in software engineering, this is the process of testing the product’s GUI to ensure it meets its specifications and requirements.

5.3 Algorithm

The test for the correctness and efficiency of the algorithm is implemented in the ‘solver’ feature.

The algorithm is based on the beginner method introduced in chapter 4. We shall test thousands of cube scrambles and measure the average number of moves to solve the Cube using this method.

5.2.1 Test design

We used the ‘scramble’ feature in the program to generate 10,000 scrambles. In 10,000 scrambles, we have measured the following: the total number of moves for solving the Cube, the number of moves in each step, the minimum number of moves in each step, the maximum number of moves in each step.

The method that was implemented in the program can be considered to consist of 6 smaller steps as follows: solving white cross, solving the first layer, solving the second layer, solving the yellow cross, solving the yellow face, solving the third layer.

5.2.2 Results

The solver successfully solved 10000 scrambles with the following results.

TABLE 1. The test result on 10000 scrambles.

	Average number of moves	**Min number of moves**	**Max number of moves**
Total	91.924	68	108
White cross	13.427	6	20
First layer	14.324	9	19
Second layer	18.621	11	25
Yellow cross	9.112	3	18
Yellow face	8.967	5	11
Last layer	19.538	12	24

In the table 1, the average number of moves in total and in each step can be seen as well. And the minimum number of moves and the maximum number of moves for each step can also be observed. As being stated in the previous chapter, the total number of moves for solving the Cube using the beginner method introduced in the last chapter falls in the range 80 ~ 100 moves.

6 CONCLUSION

For further improvement, the console program may be entirely changed to GUI program with buttons and textbox to get easier for the users to employ and implement the features of the program. One of the solutions can be the GUI libraries for OpenGL. For further improvement, I will try to improve the efficiency of the solver by implementing several more-advanced algorithms such as Friedrich First Two Layers (F2L) method, or one of the God's algorithms, or mixing between those methods to give the better one.

The program should also implement the features to detect the unsolvable cubes. As we discussed in the previous chapters, not all configurations on Rubik's Cube are solvable. And the current version of the software did not implement the functionality to detect this; hence this needs to be improved in the future.

Thesis's aims were to write a software program to simulate the traditional 3x3x3 Rubik's Cube and provide and explain the application of the virtual cube to develop the strategy for solving the cube. The chapter 2 was committed to helping you get familiar with the structure and notations of the Cube. Chapter 3 provided the architecture and the essential features of the software. After that, the strategy for solving the cube and the explanation of the application of the virtual cube for applying the algorithms were described in chapter 4. Chapter 4 also showed how group theory can be a great tool to develop a solution for solving the cube. As intended, specifying the methods required a high level of abstraction as well as a little hard work. Finally, chapter 5 discussed about the validation and testing phase of the software; and also showed some of the lacking features that need to be improved in the future.

REFERENCES

Christophe Goudey. 2001. Rubik’s Cube History. Available: http://cubeland.free.fr/infos/infos.htm. Accessed 20 October 2015.

Fourie, Daniel. "3x3x3 World Record 5.253 seconds". Youtube. Accessed 25 April 2015.

Rokicki, Tom. 2008. Twenty-Two Moves Suffice.

Michael W. Dempsey. 1988. Growing up with science: The illustrated encyclopedia of invention. London: Marshall Cavendish.

David Joyner. 2008. Adventures in Group Theory: Rubik’s Cube, Merlin’s Machine, and Other Mathematical Toys.

Tom Davis. 2006. Group Theory via Rubik’s Cube.

Edgar G. Goodaire, Michael M. Parmenter. 2002. Discrete mathematics with Graph Theory, 2nd edition.

Reuters. 2008. eGames, Mindscape Put International Twist On Rubik's Cube PC Game. Accessed 20 Octocber 2015.

Basic Cube Notation. Available: https://www.speedsolving.com/wiki/index.php/3x3x3_notation. Accessed 20 October 2015.

Mathematics of Rubik’s Cube. Available: http://ruwix.com/the-rubiks-cube/mathematics-of-the-rubiks-cube-permutation-group/. Accessed 20 October 2015.

Alan Chang. 2001. Available: www.learn2cube.com. Accessed 20 October 2015.

Introduction to SDL. Available: https://wiki.libsdl.org/Introduction. Accessed 20 October 2015.

Kenneth Sloan. 2008. A Gentle Guide to Rubik's Cube.

Rik van Grol. 2010. The Quest For God’s Number. Math Horizons.

Jamine Mulholland. 2015. Permutation Puzzles: A Mathematical Perspective.

Josef Jelinek. 2000. Corners-First Solution Method for Rubik's Cube - for Beginners. Available: http://rubikscube.info/beginner.php. Accessed 23 October 2015.

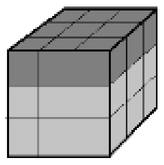
Christopher P. Benton. Doc Benton’s Fantastic Guide to Group Theory, Rubik’s Cube, Permutations, Symmetry, and all that is!

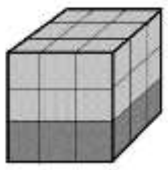
APPENDIX 1

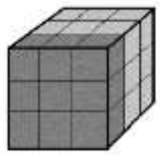
NOTATIONS

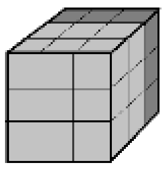
FACES

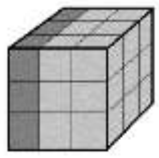
U:	Up	D:	Down
F:	Front	B:	Back
L:	Left	R:	Right

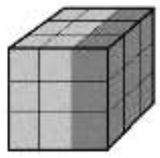

U


D


F


B


L


R

MOVES

U:	The rotation of up face a quarter-turn clockwise.
D:	The rotation of down face a quarter-turn clockwise.
F:	The rotation of front face a quarter-turn clockwise.
B:	The rotation of back face a quarter-turn clockwise.
L:	The rotation of left face a quarter-turn clockwise.
R:	The rotation of right face a quarter-turn clockwise.
u:	The rotation of up face a quarter-turn counter-clockwise.
d:	The rotation of down face a quarter-turn counter-clockwise.
f:	The rotation of front face a quarter-turn counter-clockwise.

b:	The rotation of back face a quarter-turn counter-clockwise.
l:	The rotation of left face a quarter-turn counter-clockwise.
r:	The rotation of right face a quarter-turn counter-clockwise.
U2:	The rotation of up face 180^0 clockwise.
D2:	The rotation of down face 180^0 clockwise.
F2:	The rotation of front face 180^0 clockwise.
B2:	The rotation of back face 180^0 clockwise.
L2:	The rotation of left face 180^0 clockwise.
R2:	The rotation of right face 180^0 clockwise.
X:	The rotation of the entire cube as if doing an R turn.
Y:	The rotation of the entire cube as if doing a U turn.
Z:	The rotation of the whole cube as if doing an F turn.
x:	The rotation of the whole cube as if doing an r turn.
y:	The rotation of the whole cube as if doing a u turn.
z:	The rotation of the whole cube as if doing an f turn.