Computer Graphics Project (UCS505)

Rubik's Cube Simulation

Submitted by:

Akshat Garg (102116084)
Smriti (102116094)
Ikjot Singh (102116071)

SubGroup: 3CS11

Submitted To:

Dr. Anupam Garg

I. PROJECT DESCRIPTION

The Rubik's Cube Simulation using OpenGL and C++ is an interactive computer graphics project aimed at simulating the iconic Rubik's Cube puzzle in a 3D environment. The project allows users to manipulate the cube's layers, rotate faces, and solve the puzzle.

Key Features:

1. ***3D Visualization***: Utilizing OpenGL, the project provides a visually appealing 3D representation of the Rubik's Cube, allowing users to view and interact with the puzzle from different angles.

2. ***User Interaction***: Users can manipulate the Rubik's Cube using mouse input to rotate individual layers or faces in different directions, simulating the actions performed when solving the puzzle.

3. ***Customization Options:*** The project offers customization options such as changing the cube's size, colors, and appearance to suit the user's preferences.

4. ***Solver Algorithm***: Implementing a solving algorithm allows users to automatically solve the Rubik's Cube, providing hints or solutions when requested.

5. ***User Interface***: The project features an intuitive user interface with buttons for starting, learning about the Rubik's Cube, seeking help, and exiting the simulation.

6. ***Animation and Transition Effects***: Smooth animation effects are incorporated to enhance the user experience, providing seamless transitions between rotations and actions.

7. ***Documentation and Learning Resources:*** Detailed documentation and learning resources are provided to help users understand the mechanics of the Rubik's Cube, learn solving strategies, and explore the project's codebase.

Project Goals:

- Develop a fully functional Rubik's Cube simulation using modern computer graphics techniques and OpenGL.
- Provide an engaging and interactive user experience, allowing users to learn, practice, and enjoy solving the Rubik's Cube virtually.
- Implement efficient algorithms for cube manipulation, rotation, and solving, ensuring smooth performance and accurate simulation results.

Target Audience:

- Rubik's Cube enthusiasts, puzzle solvers, and hobbyists interested in virtual simulations and computer graphics projects.
- Students and educators seeking to learn or teach concepts of 3D graphics programming, OpenGL, and game development through a fun and challenging project.

II. COMPUTER GRAPHICS CONCEPTS USED

The code you provided appears to be a C++ program that utilizes the OpenGL library for computer graphics. Here's a breakdown of the key concepts used in the code:

1. ***OpenGL (Open Graphics Library):*** It's a cross-language, cross-platform API for rendering 2D and 3D vector graphics. In this code, OpenGL is used for rendering graphics on the screen.

2. ***Graphics Primitives*** : The code defines various geometric primitives like points, lines, and polygons (e.g., rectangles for buttons) using OpenGL functions like **glBegin()** and **glEnd().**

3. ***Transformation*** *:* The program applies various transformations like translation **(glTranslatef())** and scaling **(glScalef())** to position and size the graphics objects appropriately on the screen.

4. ***Color Handling*** **:** Colors are specified using RGB values. OpenGL functions like **glColor3fv()** are used to set the color of graphics primitives.

5. **Event Handling :** The program appears to handle mouse click events (isMousePressed) and animate certain graphics elements based on these events.

6. **Button Creation and Interaction**: Buttons are created as objects with properties like position, size, background color, foreground color, and text. These buttons can be clicked, triggering certain actions or animations.

7. **State Management** : The code includes a State class representing the state of a cube, possibly for a Rubik's Cube simulation. Various methods in the State class perform rotations (e.g., clockwise, anticlockwise) on the cube's faces.

8. **Windows and Viewports**: The code performs shifts in overall positioning of the screen to view different parts of the window by moving the viewport according to input.

Overall, the code demonstrates how to create a basic interactive graphics application using OpenGL in C++, including rendering primitives, handling user input, and animating objects.

III. USED DEFINED FUNCTION

In the code provided, several user-defined functions are used to encapsulate specific functionalities and make the code more modular. Here are some of the user-defined functions:

1. DrawCube():
This function is used to draw a cube using OpenGL commands. It likely contains code to define the vertices, edges, and faces of the cube, along with color and transformation settings.

2. initRendering():
This function initializes various rendering settings such as background color, enabling depth testing, and setting the projection matrix.

3. handleResize():
This function is called whenever the window is resized and adjusts the viewport and projection matrix accordingly to maintain correct aspect ratio.

4. handleMouseclick():
This function is likely called when a mouse click event occurs. It may contain logic to detect which button was clicked and trigger corresponding actions.

5. drawButton():
This function is used to draw a graphical button on the screen. It may include drawing a rectangle for the button shape, adding text labels, and handling button colors.

6. isMouseOverButton():
This function checks if the mouse cursor is over a specific button on the screen. It likely involves comparing the mouse coordinates with the button's position and dimensions.

7. animate(void actionEvent()):
This function animates the rectangular object by gradually moving it horizontally and vertically based on predefined elevation values (xelev and yelev). If the object is not currently animating (isAnimating is false), it gradually moves the object back to its original position. The actionEvent() parameter specifies a callback function to be executed once the animation is complete.

8. collision(float px, float py):
This function checks whether a given point (specified by coordinates px and py) is within the boundaries of the rectangular object. It returns true if the point is inside the object's boundaries and false otherwise.

9. front_anticlock(), front_clock(), back_anticlock(), back_clock(), left_anticlock(), left_clock(), right_anticlock(), right_clock(), up_anticlock(), up_clock(), down_anticlock(), down_clock()**:**
These functions perform clockwise and anti-clockwise rotations on the front, back, left, right, up, and down faces of the Rubik's Cube. They take into account the layer of the cube being rotated, and they update the faces array accordingly. Each function includes a temporary array t to store values temporarily during the rotation process. Additionally, when layer is 0 (indicating the outermost layer), these functions perform additional operations to rotate the corresponding center pieces of each face.

***10. toggleHollow()*:**

This function toggles between two states for a button, likely switching between a solid and hollow appearance. If hollow is currently true, indicating that the button is hollow, it sets hollow to false and creates a new button with solid appearance labeled "HOLLOW". If hollow is false, indicating that the button is solid, it sets hollow to true and creates a new button with hollow appearance labeled "SOLID".

11. printText():

This function print s text on the screen at the specified position (x, y). It allows customization of the text size (size), font (font), foreground color (fg), and stroke width (stroke). Inside the function, OpenGL transformations are used to translate and scale the text before drawing each character using glutStrokeCharacter.

12. enableTransition():

This function sets the isTransition flag to true, presumably to enable some form of transition effect or animation. The purpose of this transition and its implementation details are likely handled elsewhere in the codebase.

13. printCube():

This function prints the current state of the Rubik's Cube. It can print the cube in two different formats based on the value of the type parameter. If type is 0, it prints the cube face by face, with each face represented by a 3x3 grid of sticker colors. If type is 1, it prints a more compact representation of the cube, with each face represented by a 3x3 grid of sticker colors printed side by side.

14. isSolved():

This function checks if the Rubik's Cube is solved by iterating through each face of the cube and comparing the color of each sticker with the color of the stickers on the initial face. If any sticker on any face does not match the initial color, it returns false indicating that the cube is not solved. If all stickers match, it returns true indicating that the cube is solved.

***15. getRotation()*:**

Matrix function calculates and returns the rotation matrix corresponding to a given quaternion rotation q. It extracts the rotation matrix from the normalized quaternion, fills a 4x4 matrix with the rotation values and identity elements, and then returns a pointer to this matrix.

16. changeState():

This function updates the Rubik's Cube's state after rotations by clearing the rotation queue, executing the appropriate rotation based on the rotation type, and resetting the rotation type to zero once completed.

17. BFS Algorithm:

This code implements a Breadth First Search (BFS) algorithm to solve a Rubik's Cube.

The BFS function initializes necessary data structures such as sets, queues, and maps to keep track of visited states, the queue of states to explore, and the parent-child relationships between states, respectively.

It starts with the initial state of the Rubik's Cube and explores all possible moves from that state, adding them to the queue if they haven't been visited before. It checks if each move leads to a solved state and stops when a solution is found.

Moves are performed in clockwise and anticlockwise directions for each face of the Rubik's Cube (front, back, left, right, up, down), exploring all possible combinations until a solution is found.

Once a solution state is found, the function backtracks through the parent-child relationships to reconstruct the sequence of moves needed to solve the cube. This sequence is stored in the moves vector.

The solve function clears the moveList (vector of moves) and then calls the BFS function to find a solution and store it in moveList

These user-defined functions help organize the code into manageable and reusable components, making it easier to understand, maintain, and extend the functionality of the program.

IV. CODE

```
#include <Eigen/Eigen>
#include <math.h>
#include <time.h>
#include <ctype.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <map>
#include <set>
#include <deque>
#include <queue>
#include <stack>
#include <bitset>
#include <string>
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
#include <GL/freeglut.h>

using namespace std;
using namespace Eigen;
#define sp system("pause")
#define FOR(i,a,b) for(int i=a;i<=b;++i)
#define FORD(i,a,b) for(int i=a;i>=b;--i)
#define REP(i,n) for(int i=0;i<n;++i)
#define ll long long

#define CUBE_SIZE 2
int layer = 0;
int width = 1200, height = 750;

class point {
public:
    double x, y, z;
    point() {
    }
    point(double px, double py, double pz) {
        x = px;
        y = py;
        z = pz;
    }
```

```
};

class color {
public:
    float r, g, b;
    color() {
        r = 0;
        g = 0;
        b = 0;
    }
    color(float ir, float ig, float ib) {
        r = ir;
        g = ig;
        b = ib;
    }
    float* getArray() {
        float c[3];
        c[0] = r;
        c[1] = g;
        c[2] = b;
        return c;
    }
};

class button {
    static float xelev;
    static float yelev;
    static float xshadowMul;
    static float yshadowMul;
    static float clickdelay;
public:
    bool is3D, isAnimating;
    float x, y;
    float xoff, yoff;
    float w, h;
    color bg, fg;
    color yshadow, xshadow;
    string text;

    button() {
        is3D = true;
        isAnimating = false;
        x = 0;
        y = 0;
        xoff = yoff = 0;
```

```
        w = 1;
        h = 0.35;
        bg = color(0.9, 0.9, 0.9);
        xshadow = color(bg.r * xshadowMul, bg.g * xshadowMul, bg.b * xshadowMul);
        yshadow = color(bg.r * yshadowMul, bg.g * yshadowMul, bg.b * yshadowMul);
        fg = color(0, 0, 0);
        text = "Button";
    }
    button(float px, float py, float iw, float ih) {
        x = px;
        y = py;
        w = iw;
        h = ih;
        is3D = true;
        isAnimating = false;
        xoff = yoff = 0;
        bg = color(0.9, 0.9, 0.9);
        xshadow = color(bg.r * xshadowMul, bg.g * xshadowMul, bg.b * xshadowMul);
        yshadow = color(bg.r * yshadowMul, bg.g * yshadowMul, bg.b * yshadowMul);
        fg = color(0, 0, 0);
        text = "Button";
    }
    button(float px, float py, float iw, float ih, color background, color foreground,
string disptext) {
        x = px;
        y = py;
        w = iw;
        h = ih;
        is3D = true;
        isAnimating = false;
        bg = background;
        xshadow = color(bg.r * xshadowMul, bg.g * xshadowMul, bg.b * xshadowMul);
        yshadow = color(bg.r * yshadowMul, bg.g * yshadowMul, bg.b * yshadowMul);
        fg = foreground;
        text = disptext;
        xoff = yoff = 0;
    }
    void draw() {

        glColor3fv(bg.getArray());
        glBegin(GL_POLYGON);
        glVertex3f(x + xoff, y + yoff, 1);
        glVertex3f(x + w + xoff, y + yoff, 1);
        glVertex3f(x + w + xoff, y - h + yoff, 1);
        glVertex3f(x + xoff, y - h + yoff, 1);
```

```
        glEnd();

        if (is3D) {
            // Top Side
            glColor3fv(yshadow.getArray());
            glBegin(GL_POLYGON);
            glVertex3f(x + xoff, y + yoff, 1);
            glVertex3f(x + w + xoff, y + yoff, 1);
            glVertex3f(x + w + xelev, y + yelev, 0);
            glVertex3f(x + xelev, y + yelev, 0);
            glEnd();
            // Right Side
            glColor3fv(xshadow.getArray());
            glBegin(GL_POLYGON);
            glVertex3f(x + w + xoff, y + yoff, 1);
            glVertex3f(x + w + xelev, y + yelev, 0);
            glVertex3f(x + w + xelev, y - h + yelev, 0);
            glVertex3f(x + w + xoff, y - h + yoff, 1);
            glEnd();
        }
        glPushMatrix();
        glTranslatef(x + xoff + (w - text.size() * 0.135) / 2, y + yoff - (h + 0.12) /
2, 1.1);
        glScalef(1 / 800.0, 1 / 800.0, 0);
        glLineWidth(2);
        glColor3fv(fg.getArray());

        REP(i, text.size())
            glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, text[i]);
        glPopMatrix();

    }
    bool collision(float px, float py) {
        if (!isAnimating && px > x + xoff && px<x + w + xoff && py>y - h + yoff && py <
y + yoff)
            return true;
        return false;
    }
    void animate(void actionEvent()) {

        if (isAnimating) {
            xoff += xelev / clickdelay;
            yoff += yelev / clickdelay;

            if (xoff + x >= x + xelev) {
```

```
                isAnimating = false;
                actionEvent();
            }
        }
        else {
            xoff = max(0, xoff - xelev / clickdelay);
            yoff = max(0, yoff - yelev / clickdelay);
        }
    }

};

float button::xelev = 0.07;
float button::yelev = 0.05;
float button::xshadowMul = 0.7;
float button::yshadowMul = 0.5;
float button::clickdelay = max(10, 1000 / (CUBE_SIZE * sqrt(CUBE_SIZE)));

class State {
public:
    int faces[6][CUBE_SIZE][CUBE_SIZE];
    State() {
        REP(k, 6) {
            REP(i, CUBE_SIZE)
                REP(j, CUBE_SIZE)
                faces[k][i][j] = k;
        }
    }
    bool isSolved() {
        REP(k, 6) {
            int color = faces[k][0][0];
            REP(j, CUBE_SIZE) {
                REP(i, CUBE_SIZE) {
                    if (faces[k][j][i] != color)
                        return false;
                }
            }
        }
        return true;
    }

    void printCube(int type = 0) {
        if (type == 0) {
            printf("~~~~~~~~~~~~~~  C U B E  ~~~~~~~~~~~~~~\n");
            REP(i, CUBE_SIZE) {
```

```
            REP(j, CUBE_SIZE)
                printf("  ");
            REP(j, CUBE_SIZE)
                printf("%2d", faces[1][i][j]);
            REP(j, CUBE_SIZE)
                printf("  ");
            printf("\n");
        }
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                printf("  ");
            REP(j, CUBE_SIZE)
                printf("%2d", faces[4][i][j]);
            REP(j, CUBE_SIZE)
                printf("  ");
            printf("\n");
        }
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                printf("%2d", faces[2][i][j]);
            REP(j, CUBE_SIZE)
                printf("%2d", faces[0][i][j]);
            REP(j, CUBE_SIZE)
                printf("%2d", faces[3][i][j]);
            printf("\n");
        }
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                printf("  ");
            REP(j, CUBE_SIZE)
                printf("%2d", faces[5][i][j]);
            REP(j, CUBE_SIZE)
                printf("  ");
            printf("\n");
        };
        printf("\n");
    }
    else if (type == 1) {
        REP(i, 3) {
            REP(k, 6) {
                REP(j, 3)
                    printf("%2d", faces[k][i][j]);
                printf(" ");
            }
            printf("\n");
```

```
            }
        }
    }
    void front_anticlock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i];
            faces[4][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = faces[3][CUBE_SIZE - 1
- i][layer];
            faces[3][CUBE_SIZE - 1 - i][layer] = faces[5][layer][i];
            faces[5][layer][i] = faces[2][i][CUBE_SIZE - 1 - layer];
            faces[2][i][CUBE_SIZE - 1 - layer] = t[i];
        }
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[0][l][CUBE_SIZE - 1 - i];
                    faces[0][l][CUBE_SIZE - 1 - i] = faces[0][CUBE_SIZE - 1 -
i][CUBE_SIZE - 1 - l];
                    faces[0][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[0][CUBE_SIZE
- 1 - l][i];
                    faces[0][CUBE_SIZE - 1 - l][i] = faces[0][i][l];
                    faces[0][i][l] = t[i];
                }
            }
        }
    }
    void front_clock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][CUBE_SIZE - 1 - layer][i];
            faces[4][CUBE_SIZE - 1 - layer][i] = faces[2][CUBE_SIZE - 1 - i][CUBE_SIZE
- 1 - layer];
            faces[2][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] =
faces[5][layer][CUBE_SIZE - 1 - i];
            faces[5][layer][CUBE_SIZE - 1 - i] = faces[3][i][layer];
            faces[3][i][layer] = t[i];
        }
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[0][l][i];
                    faces[0][l][i] = faces[0][CUBE_SIZE - 1 - i][l];
                    faces[0][CUBE_SIZE - 1 - i][l] = faces[0][CUBE_SIZE - 1 -
l][CUBE_SIZE - 1 - i];
```

```
                    faces[0][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] =
faces[0][i][CUBE_SIZE - 1 - l];
                    faces[0][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }

    }
    void back_anticlock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][layer][i];
            faces[4][layer][i] = faces[2][CUBE_SIZE - 1 - i][layer];
            faces[2][CUBE_SIZE - 1 - i][layer] = faces[5][CUBE_SIZE - 1 -
layer][CUBE_SIZE - 1 - i];
            faces[5][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = faces[3][i][CUBE_SIZE
- 1 - layer];
            faces[3][i][CUBE_SIZE - 1 - layer] = t[i];
        }

        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[1][l][CUBE_SIZE - 1 - i];
                    faces[1][l][CUBE_SIZE - 1 - i] = faces[1][CUBE_SIZE - 1 -
i][CUBE_SIZE - 1 - l];
                    faces[1][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[1][CUBE_SIZE
- 1 - l][i];
                    faces[1][CUBE_SIZE - 1 - l][i] = faces[1][i][l];
                    faces[1][i][l] = t[i];
                }
            }
        }
    }
    void back_clock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][layer][CUBE_SIZE - 1 - i];
            faces[4][layer][CUBE_SIZE - 1 - i] = faces[3][CUBE_SIZE - 1 - i][CUBE_SIZE
- 1 - layer];
            faces[3][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] = faces[5][CUBE_SIZE - 1
- layer][i];
            faces[5][CUBE_SIZE - 1 - layer][i] = faces[2][i][layer];
            faces[2][i][layer] = t[i];
        }
```

```
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[1][l][i];
                    faces[1][l][i] = faces[1][CUBE_SIZE - 1 - i][l];
                    faces[1][CUBE_SIZE - 1 - i][l] = faces[1][CUBE_SIZE - 1 -
l][CUBE_SIZE - 1 - i];
                    faces[1][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] =
faces[1][i][CUBE_SIZE - 1 - l];
                    faces[1][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }
    }
    void left_anticlock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][CUBE_SIZE - 1 - i][layer];
            faces[4][CUBE_SIZE - 1 - i][layer] = faces[0][CUBE_SIZE - 1 - i][layer];
            faces[0][CUBE_SIZE - 1 - i][layer] = faces[5][CUBE_SIZE - 1 - i][layer];
            faces[5][CUBE_SIZE - 1 - i][layer] = faces[1][i][CUBE_SIZE - 1 - layer];
            faces[1][i][CUBE_SIZE - 1 - layer] = t[i];
        }
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[2][l][CUBE_SIZE - 1 - i];
                    faces[2][l][CUBE_SIZE - 1 - i] = faces[2][CUBE_SIZE - 1 -
i][CUBE_SIZE - 1 - l];
                    faces[2][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[2][CUBE_SIZE
- 1 - l][i];
                    faces[2][CUBE_SIZE - 1 - l][i] = faces[2][i][l];
                    faces[2][i][l] = t[i];
                }
            }
        }
    }
    void left_clock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][i][layer];
            faces[4][i][layer] = faces[1][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer];
            faces[1][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] = faces[5][i][layer];
            faces[5][i][layer] = faces[0][i][layer];
```

```
            faces[0][i][layer] = t[i];
        }
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[2][l][i];
                    faces[2][l][i] = faces[2][CUBE_SIZE - 1 - i][l];
                    faces[2][CUBE_SIZE - 1 - i][l] = faces[2][CUBE_SIZE - 1 -
l][CUBE_SIZE - 1 - i];
                    faces[2][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] =
faces[2][i][CUBE_SIZE - 1 - l];
                    faces[2][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }
    }

    void right_anticlock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][i][CUBE_SIZE - 1 - layer];
            faces[4][i][CUBE_SIZE - 1 - layer] = faces[1][CUBE_SIZE - 1 - i][layer];
            faces[1][CUBE_SIZE - 1 - i][layer] = faces[5][i][CUBE_SIZE - 1 - layer];
            faces[5][i][CUBE_SIZE - 1 - layer] = faces[0][i][CUBE_SIZE - 1 - layer];
            faces[0][i][CUBE_SIZE - 1 - layer] = t[i];
        }
        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[3][l][CUBE_SIZE - 1 - i];
                    faces[3][l][CUBE_SIZE - 1 - i] = faces[3][CUBE_SIZE - 1 -
i][CUBE_SIZE - 1 - l];
                    faces[3][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[3][CUBE_SIZE
- 1 - l][i];
                    faces[3][CUBE_SIZE - 1 - l][i] = faces[3][i][l];
                    faces[3][i][l] = t[i];
                }
            }
        }
    }

    void right_clock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[4][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer];
```

```
            faces[4][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] = faces[0][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer];
            faces[0][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] = faces[5][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer];
            faces[5][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - layer] = faces[1][i][layer];
            faces[1][i][layer] = t[i];
        }

        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[3][l][i];
                    faces[3][l][i] = faces[3][CUBE_SIZE - 1 - i][l];
                    faces[3][CUBE_SIZE - 1 - i][l] = faces[3][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i];
                    faces[3][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] = faces[3][i][CUBE_SIZE - 1 - l];
                    faces[3][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }

    }

    void up_anticlock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[1][layer][i];
            faces[1][layer][i] = faces[3][layer][i];
            faces[3][layer][i] = faces[0][layer][i];
            faces[0][layer][i] = faces[2][layer][i];
            faces[2][layer][i] = t[i];
        }

        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[4][l][CUBE_SIZE - 1 - i];
                    faces[4][l][CUBE_SIZE - 1 - i] = faces[4][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l];
                    faces[4][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[4][CUBE_SIZE - 1 - l][i];
                    faces[4][CUBE_SIZE - 1 - l][i] = faces[4][i][l];
                    faces[4][i][l] = t[i];
                }
```

```
            }
        }
    }

    void up_clock() {
        int t[CUBE_SIZE];

        REP(i, CUBE_SIZE) {
            t[i] = faces[1][layer][CUBE_SIZE - 1 - i];
            faces[1][layer][CUBE_SIZE - 1 - i] = faces[2][layer][CUBE_SIZE - 1 - i];
            faces[2][layer][CUBE_SIZE - 1 - i] = faces[0][layer][CUBE_SIZE - 1 - i];
            faces[0][layer][CUBE_SIZE - 1 - i] = faces[3][layer][CUBE_SIZE - 1 - i];
            faces[3][layer][CUBE_SIZE - 1 - i] = t[i];
        }

        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[4][l][i];
                    faces[4][l][i] = faces[4][CUBE_SIZE - 1 - i][l];
                    faces[4][CUBE_SIZE - 1 - i][l] = faces[4][CUBE_SIZE - 1 -
l][CUBE_SIZE - 1 - i];
                    faces[4][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] =
faces[4][i][CUBE_SIZE - 1 - l];
                    faces[4][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }
    }

    void down_anticlock() {
        int t[CUBE_SIZE];

        REP(i, CUBE_SIZE) {
            t[i] = faces[0][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i];
            faces[0][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = faces[3][CUBE_SIZE - 1
- layer][CUBE_SIZE - 1 - i];
            faces[3][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = faces[1][CUBE_SIZE - 1
- layer][CUBE_SIZE - 1 - i];
            faces[1][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = faces[2][CUBE_SIZE - 1
- layer][CUBE_SIZE - 1 - i];
            faces[2][CUBE_SIZE - 1 - layer][CUBE_SIZE - 1 - i] = t[i];
        }

        if (layer == 0) {
```

```
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[5][l][CUBE_SIZE - 1 - i];
                    faces[5][l][CUBE_SIZE - 1 - i] = faces[5][CUBE_SIZE - 1 -
i][CUBE_SIZE - 1 - l];
                    faces[5][CUBE_SIZE - 1 - i][CUBE_SIZE - 1 - l] = faces[5][CUBE_SIZE
- 1 - l][i];
                    faces[5][CUBE_SIZE - 1 - l][i] = faces[5][i][l];
                    faces[5][i][l] = t[i];
                }
            }
        }

    }

    void down_clock() {
        int t[CUBE_SIZE];
        REP(i, CUBE_SIZE) {
            t[i] = faces[0][CUBE_SIZE - 1 - layer][i];
            faces[0][CUBE_SIZE - 1 - layer][i] = faces[2][CUBE_SIZE - 1 - layer][i];
            faces[2][CUBE_SIZE - 1 - layer][i] = faces[1][CUBE_SIZE - 1 - layer][i];
            faces[1][CUBE_SIZE - 1 - layer][i] = faces[3][CUBE_SIZE - 1 - layer][i];
            faces[3][CUBE_SIZE - 1 - layer][i] = t[i];
        }

        if (layer == 0) {
            REP(l, CUBE_SIZE / 2) {
                FOR(i, l, CUBE_SIZE - 2 - l) {
                    t[i] = faces[5][l][i];
                    faces[5][l][i] = faces[5][CUBE_SIZE - 1 - i][l];
                    faces[5][CUBE_SIZE - 1 - i][l] = faces[5][CUBE_SIZE - 1 -
l][CUBE_SIZE - 1 - i];
                    faces[5][CUBE_SIZE - 1 - l][CUBE_SIZE - 1 - i] =
faces[5][i][CUBE_SIZE - 1 - l];
                    faces[5][i][CUBE_SIZE - 1 - l] = t[i];
                }
            }
        }
    }

    bool operator<(const State& rhs) const {
        REP(k, 6) {
            REP(j, CUBE_SIZE) {
                REP(i, CUBE_SIZE) {
                    if (faces[k][j][i] != rhs.faces[k][j][i])
```

```
                        return faces[k][j][i] < rhs.faces[k][j][i];
                }
            }
        }
        return false;
    }

    bool operator>(const State& rhs) const {
        REP(k, 6) {
            REP(j, 3) {
                REP(i, 3) {
                    if (faces[k][j][i] != rhs.faces[k][j][i])
                        return faces[k][j][i] > rhs.faces[k][j][i];
                }
            }
        }
        return false;
    }

    bool operator==(const State& rhs) const {
        REP(k, 6) {
            REP(j, 3) {
                REP(i, 3) {
                    if (faces[k][j][i] != rhs.faces[k][j][i])
                        return false;
                }
            }
        }
        return true;
    }
};

bool hollow = false;
bool change = false;
bool isTransition = false;
bool isMousePressed = false;

int px = -1, py = -1;
float xscreen = 0, fromXScreen = 0, toXScreen = 0;
float yscreen = 0, fromYScreen = 0, toYScreen = 0;
float transition_percent = 0;
float cameraX = -0, cameraY = 0, cameraZ = 0;
float viewportX = 8, viewportY = 5;

double transitionSpeed = min(1, pow(10, floor((CUBE_SIZE - 3) / 10)) / 1000);
```

```
int rotationType = 0;
const double PI = 3.1415926535;
double rorationSpeed = min(90.0, max(0.15, (double)CUBE_SIZE * sqrt(CUBE_SIZE) / 50));

double totalRotation = 0;
Vector3d rotationAxis;

button bstart = button(-0.5, 0.8, 0.8, 0.3, color(0, 0.8, 0), color(0, 0, 0), "START");
button babout = button(-0.5, 0.3, 0.8, 0.3, color(1, 0.8, 0), color(0, 0, 0), "LEARN");
button bhelp = button(-0.5, -0.3, 0.8, 0.3, color(0, 0.6, 1), color(0, 0, 0), "HELP");
button bexit = button(-0.5, -0.8, 0.8, 0.3, color(1, 0.1, 0.1), color(0, 0, 0),
"EXIT");

button bback10 = button(4.5, -2, 0.8, 0.3, color(1, 0.1, 0.1), color(0, 0, 0), "BACK");
button bhollow = button(4.5, 2, 0.8, 0.3, color(0.2, 0.2, 0.2), color(1, 1, 1),
"HOLLOW");
button bsolve = button(4.5, 1.5, 0.8, 0.3, color(1, 0, 0.6), color(0, 0, 0), "SOLVE");

button bback01 = button(-3.5, 3, 0.8, 0.3, color(1, 0.1, 0.1), color(0, 0, 0), "BACK");

button bback0_1 = button(-3.5, -7, 0.8, 0.3, color(1, 0.1, 0.1), color(0, 0, 0),
"BACK");

State cube;
Quaterniond camera = Quaterniond{ AngleAxisd{1, Vector3d{0,0,0}} };

double* matrix = new double[16];
vector<char> moveList;
vector<point> rotationQueue;
color colorList[7] = { color(0.3,0.8,0), color(0,0.5,1), color(1,0.8,0),
color(0.9,0.9,0.9),  color(1,0.4,0), color(0.9,0,0), color(0.2,0.2,0.2) };

Quaterniond cubesRotation[CUBE_SIZE][CUBE_SIZE][CUBE_SIZE];

void printText(float x, float y, string text, float size, void* font =
GLUT_STROKE_ROMAN, color fg = color(1, 1, 1), float stroke = 2) {

    glPushMatrix();
    glTranslatef(x, y, 0);
    glScalef(size / 800.0, size / 800.0, 0);
    glLineWidth(stroke);
    glColor3fv(fg.getArray());

    REP(i, text.size())
```

```
        glutStrokeCharacter(font, text[i]);
    glPopMatrix();

}

void toggleHollow() {

    if (hollow) {
        hollow = false;
        bhollow = button(4.5, 2, 0.8, 0.3, color(0.2, 0.2, 0.2), color(1, 1, 1),
"HOLLOW");
    }
    else {
        hollow = true;
        bhollow = button(4.5, 2, 0.8, 0.3, color(0.9, 0.9, 0.9), color(0, 0, 0),
"SOLID");
    }
}


void enableTransition() {
    isTransition = true;
}

void nothing() {
}

void exitProgram() {

    printf("Exiting");
    Sleep(1000);
    exit(0);
}

void BFS(vector<char>& moves) {

    State cur;
    set<State> visited;
    queue<State> q;
    map<State, pair<char, State>> parent;


    visited.insert(::cube);
    q.push(::cube);
```

```
    while (!q.empty()) {

        State s = q.front();
        State copy = s;
        q.pop();

        if (s.isSolved()) {
            cur = s;
            break;
        }

        // Clockwise turns
        s.front_clock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('q', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.front_anticlock();

        s.back_clock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('w', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.back_anticlock();

        s.left_clock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('a', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
```

```
    }
    s.left_anticlock();

    s.right_clock();
    if (visited.find(s) == visited.end()) {
        visited.insert(s);
        parent[s] = make_pair('s', copy);
        q.push(s);
        if (s.isSolved()) {
            cur = s;
            break;
        }
    }
    s.right_anticlock();

    s.up_clock();
    if (visited.find(s) == visited.end()) {
        visited.insert(s);
        parent[s] = make_pair('z', copy);
        q.push(s);
        if (s.isSolved()) {
            cur = s;
            break;
        }
    }
    s.up_anticlock();

    s.down_clock();
    if (visited.find(s) == visited.end()) {
        visited.insert(s);
        parent[s] = make_pair('x', copy);
        q.push(s);
        if (s.isSolved()) {
            cur = s;
            break;
        }
    }
    s.down_anticlock();

    // Anti-Clockwise turns
    s.front_anticlock();
    if (visited.find(s) == visited.end()) {
        visited.insert(s);
        parent[s] = make_pair('Q', copy);
        q.push(s);
```

```
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.front_clock();

        s.back_anticlock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('W', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.back_clock();

        s.left_anticlock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('A', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.left_clock();

        s.right_anticlock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('S', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.right_clock();

        s.up_anticlock();
        if (visited.find(s) == visited.end()) {
```

```
            visited.insert(s);
            parent[s] = make_pair('Z', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.up_clock();

        s.down_anticlock();
        if (visited.find(s) == visited.end()) {
            visited.insert(s);
            parent[s] = make_pair('X', copy);
            q.push(s);
            if (s.isSolved()) {
                cur = s;
                break;
            }
        }
        s.down_clock();

    }

    while (parent.count(cur)) {
        pair<char, State> s = parent[cur];
        moves.push_back(s.first);;
        cur = s.second;
    }

    reverse(moves.begin(), moves.end());

}

void solve() {
    moveList.clear();
    BFS(moveList);
}

void animateButtons() {

    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == 0 || floor(yscreen) == 0)) {
        bstart.animate(enableTransition);
        babout.animate(enableTransition);
```

```
        bhelp.animate(enableTransition);
        bexit.animate(exitProgram);
    }
    if ((ceil(xscreen) == 1 || floor(xscreen) == 1) &&
        (ceil(yscreen) == 0 || floor(yscreen) == 0)) {
        bback10.animate(enableTransition);
        bhollow.animate(toggleHollow);
        if (CUBE_SIZE == 2 || CUBE_SIZE == 3)
            bsolve.animate(solve);
    }
    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == 1 || floor(yscreen) == 1)) {
        bback01.animate(enableTransition);
    }
    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == -1 || floor(yscreen) == -1)) {
        bback0_1.animate(enableTransition);
    }
}

void changeState() {

    REP(a, rotationQueue.size()) {
        point mci = rotationQueue[a];
        int i = mci.z, j = mci.y, k = mci.x;
        cubesRotation[i][j][k] = Quaterniond{ AngleAxisd{ 1, Vector3d{ 0,0,0 } } };
    }

    rotationQueue.clear();
    totalRotation = 0;

    if (rotationType == 1)
        ::cube.front_anticlock();
    else if (rotationType == 2)
        ::cube.back_anticlock();
    else if (rotationType == 3)
        ::cube.left_anticlock();
    else if (rotationType == 4)
        ::cube.right_anticlock();
    else if (rotationType == 5)
        ::cube.up_anticlock();
    else if (rotationType == 6)
        ::cube.down_anticlock();

    else if (rotationType == 7)
```

```
        ::cube.front_clock();
    else if (rotationType == 8)
        ::cube.back_clock();
    else if (rotationType == 9)
        ::cube.left_clock();
    else if (rotationType == 10)
        ::cube.right_clock();
    else if (rotationType == 11)
        ::cube.up_clock();
    else if (rotationType == 12)
        ::cube.down_clock();

    //cube.printCube();
    rotationType = 0;

}

inline double degtorad(double deg) {
    return PI * deg / 180;
}

void printMatrix(double m[]) {

    REP(i, 4) {
        REP(j, 4)
            printf("% .6lf ", m[i * 4 + j]);
        printf("\n");
    }
    printf("\n");

}

double* getRotationMatrix(Quaterniond& q) {

    Matrix3d rotMat = q.normalized().toRotationMatrix();

    matrix[0] = rotMat(0, 0);   matrix[4] = rotMat(0, 1);   matrix[8] = rotMat(0, 2);
matrix[12] = 0;
    matrix[1] = rotMat(1, 0);   matrix[5] = rotMat(1, 1);   matrix[9] = rotMat(1, 2);
matrix[13] = 0;
    matrix[2] = rotMat(2, 0);   matrix[6] = rotMat(2, 1);   matrix[10] = rotMat(2, 2);
matrix[14] = 0;
    matrix[3] = 0;              matrix[7] = 0;              matrix[11] = 0;
matrix[15] = 1;
```

```
    return matrix;

}

void keyboard(unsigned char key, int x, int y) {
    if (xscreen == 1 && yscreen == 0) {
        if (rotationType == 0) {
            if (key >= '1' && key <= '9') {
                layer = min((CUBE_SIZE - 1) / 2, key - '1');
            }
            else if (key == '+') {
                layer = min((CUBE_SIZE - 1) / 2, layer + 1);
            }
            else if (key == '-') {
                layer = max(0, layer - 1);
            }
            if (key == 'j' || key == 'J') {
                int cacheLayer = layer;
                FOR(i, 1, 1) {
                    layer = rand() % (CUBE_SIZE / 2);
                    rotationType = 1 + rand() % 12;
                    changeState();
                }
                layer = cacheLayer;
            }

            if (key == 'Q') {
                rotationAxis = { 0, 0, 1 };
                REP(i, CUBE_SIZE) {
                    REP(j, CUBE_SIZE)
                        rotationQueue.push_back(point(j, i, layer));
                }
                if (rorationSpeed < 0)
                    rorationSpeed = -rorationSpeed;
                rotationType = 1;
            }
            else if (key == 'q') {
                rotationAxis = { 0, 0, 1 };
                REP(i, CUBE_SIZE) {
                    REP(j, CUBE_SIZE)
                        rotationQueue.push_back(point(j, i, layer));
                }
                if (rorationSpeed > 0)
                    rorationSpeed = -rorationSpeed;
                rotationType = 7;
```

```
        }
        else if (key == 'W') {
            rotationAxis = { 0, 0, 1 };
            REP(i, CUBE_SIZE) {
                REP(j, CUBE_SIZE)
                    rotationQueue.push_back(point(j, i, CUBE_SIZE - 1 - layer));
            }
            if (rorationSpeed > 0)
                rorationSpeed = -rorationSpeed;
            rotationType = 2;
        }
        else if (key == 'w') {
            rotationAxis = { 0, 0, 1 };
            REP(i, CUBE_SIZE) {
                REP(j, CUBE_SIZE)
                    rotationQueue.push_back(point(j, i, CUBE_SIZE - 1 - layer));
            }
            if (rorationSpeed < 0)
                rorationSpeed = -rorationSpeed;
            rotationType = 8;
        }
        else if (key == 'A') {
            rotationAxis = { 1, 0, 0 };
            REP(i, CUBE_SIZE) {
                REP(j, CUBE_SIZE)
                    rotationQueue.push_back(point(layer, i, j));
            }
            if (rorationSpeed > 0)
                rorationSpeed = -rorationSpeed;
            rotationType = 3;
        }
        else if (key == 'a') {
            rotationAxis = { 1, 0, 0 };
            REP(i, CUBE_SIZE) {
                REP(j, CUBE_SIZE)
                    rotationQueue.push_back(point(layer, i, j));
            }
            if (rorationSpeed < 0)
                rorationSpeed = -rorationSpeed;
            rotationType = 9;
        }
        else if (key == 'S') {
            rotationAxis = { 1, 0, 0 };
            REP(i, CUBE_SIZE) {
                REP(j, CUBE_SIZE)
```

```
                rotationQueue.push_back(point(CUBE_SIZE - 1 - layer, i, j));
        }
        if (rorationSpeed < 0)
            rorationSpeed = -rorationSpeed;
        rotationType = 4;
    }
    else if (key == 's') {
        rotationAxis = { 1, 0, 0 };
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                rotationQueue.push_back(point(CUBE_SIZE - 1 - layer, i, j));
        }
        if (rorationSpeed > 0)
            rorationSpeed = -rorationSpeed;
        rotationType = 10;
    }
    else if (key == 'Z') {
        rotationAxis = { 0, 1, 0 };
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                rotationQueue.push_back(point(i, layer, j));
        }
        if (rorationSpeed < 0)
            rorationSpeed = -rorationSpeed;
        rotationType = 5;
    }
    else if (key == 'z') {
        rotationAxis = { 0, 1, 0 };
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                rotationQueue.push_back(point(i, layer, j));
        }
        if (rorationSpeed > 0)
            rorationSpeed = -rorationSpeed;
        rotationType = 11;
    }
    else if (key == 'X') {
        rotationAxis = { 0, 1, 0 };
        REP(i, CUBE_SIZE) {
            REP(j, CUBE_SIZE)
                rotationQueue.push_back(point(i, CUBE_SIZE - 1 - layer, j));
        }
        if (rorationSpeed > 0)
            rorationSpeed = -rorationSpeed;
        rotationType = 6;
```

```
            }
            else if (key == 'x') {
                rotationAxis = { 0, 1, 0 };
                REP(i, CUBE_SIZE) {
                    REP(j, CUBE_SIZE)
                        rotationQueue.push_back(point(i, CUBE_SIZE - 1 - layer, j));
                }
                if (rorationSpeed < 0)
                    rorationSpeed = -rorationSpeed;
                rotationType = 12;
            }
        }
    }
}

void mouse(int button, int state, int x, int y) {

    if (isMousePressed == false) {
        float glx = (x - (float)width / 2) * viewportX / width + cameraX;
        float gly = ((float)height / 2 - y) * viewportY / height + cameraY;
        printf("Mouse Click at %f, %f\n", glx, gly);
        if (xscreen == 0 && yscreen == 0) {
            if (bstart.collision(glx, gly)) {
                bstart.isAnimating = true;
                xscreen = 0.5;
                fromXScreen = 0;
                toXScreen = 1;
                yscreen = 0;
                fromYScreen = 0;
                toYScreen = 0;
            }
            else if (babout.collision(glx, gly)) {
                babout.isAnimating = true;
                xscreen = 0;
                fromXScreen = 0;
                toXScreen = 0;
                yscreen = 0.5;
                fromYScreen = 0;
                toYScreen = 1;
            }
            else if (bhelp.collision(glx, gly)) {
                bhelp.isAnimating = true;
                xscreen = 0;
                fromXScreen = 0;
                toXScreen = 0;
```

```
            yscreen = -0.5;
            fromYScreen = 0;
            toYScreen = -1;
        }
        else if (bexit.collision(glx, gly)) {
            bexit.isAnimating = true;
        }
    }
    if (xscreen == 1 && yscreen == 0) {
        if (bback10.collision(glx, gly)) {
            bback10.isAnimating = true;
            xscreen = 0.5;
            fromXScreen = 1;
            toXScreen = 0;
            yscreen = 0;
            fromYScreen = 0;
            toYScreen = 0;
        }
        if (bhollow.collision(glx, gly)) {
            bhollow.isAnimating = true;
        }
        if (bsolve.collision(glx, gly) && (CUBE_SIZE == 2 || CUBE_SIZE == 3)) {
            bsolve.isAnimating = true;
        }
    }
    if (xscreen == 0 && yscreen == 1) {
        if (bback01.collision(glx, gly)) {
            bback01.isAnimating = true;
            xscreen = 0;
            fromXScreen = 0;
            toXScreen = 0;
            yscreen = 0.5;
            fromYScreen = 1;
            toYScreen = 0;
        }
    }
    if (xscreen == 0 && yscreen == -1) {
        if (bback0_1.collision(glx, gly)) {
            bback0_1.isAnimating = true;
            xscreen = 0;
            fromXScreen = 0;
            toXScreen = 0;
            yscreen = -0.5;
            fromYScreen = -1;
            toYScreen = 0;
```

```
            }
        }
    }
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        px = x;
        py = y;
        isMousePressed = true;
    }
    if (button == GLUT_LEFT_BUTTON && state == GLUT_UP) {
        py = px = -1;
        isMousePressed = false;
    }
}

void mouseWheel(int button, int dir, int x, int y) {
    if (xscreen == 1 && yscreen == 0) {
        double factor = 0.05;
        Quaterniond qz = Quaterniond{ AngleAxisd{ dir * factor, Vector3d{ 0, 0, 1 }}};
        camera = qz * camera;
    }
}

void motion(int x, int y) {
    if (xscreen == 1 && yscreen == 0) {
        if (px != -1 && py != -1) {
            double factor = 0.005;
            Quaterniond qx = Quaterniond{ AngleAxisd{ (y - py) * factor, Vector3d{ 1,
0, 0 } } };
            Quaterniond qy = Quaterniond{ AngleAxisd{ (x - px) * factor, Vector3d{ 0,
1, 0 } } };
            camera = qx * qy * camera;
        }
    }
    px = x;
    py = y;

}

void reshape(int w, int h) {

    double widthScale = (double)w / width, heightScale = (double)h / height;

    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    glOrtho(-viewportX / 2 * widthScale, viewportX / 2 * widthScale, -viewportY / 2 *
heightScale, viewportY / 2 * heightScale, -5, 5);
    glMatrixMode(GL_MODELVIEW);
}

void drawFace(float* a, float* b, float* c, float* d, int face) {

    glColor3fv(colorList[face].getArray());
    glBegin(GL_QUADS);
    glVertex3fv(a);
    glVertex3fv(b);
    glVertex3fv(c);
    glVertex3fv(d);
    glEnd();
}

void drawCube(float* a, float* b, float* c, float* d,
    float* e, float* f, float* g, float* h,
    int x, int y, int z) {
    int front = 6, back = 6, left = 6, right = 6, up = 6, down = 6;
    if (z == 0)
        front = ::cube.faces[0][y][x];
    if (z == CUBE_SIZE - 1)
        back = ::cube.faces[1][y][CUBE_SIZE - 1 - x];
    if (y == 0)
        up = ::cube.faces[4][CUBE_SIZE - 1 - z][x];
    if (y == CUBE_SIZE - 1)
        down = ::cube.faces[5][z][x];
    if (x == 0)
        left = ::cube.faces[2][y][CUBE_SIZE - 1 - z];
    if (x == CUBE_SIZE - 1)
        right = ::cube.faces[3][y][z];
    if (!hollow || front != 6)
        drawFace(a, b, c, d, front);    // Front
    if (!hollow || back != 6)
        drawFace(f, e, h, g, back);     // Back
    if (!hollow || left != 6)
        drawFace(e, a, d, h, left);     // Left
    if (!hollow || right != 6)
        drawFace(b, f, g, c, right);    // Right
    if (!hollow || up != 6)
        drawFace(e, f, b, a, up);       // Up
    if (!hollow || down != 6)
        drawFace(d, c, g, h, down);     // Down
}
```

```
void buildRubiksCube() {
    double big_szie = 2.4;
    double small_size = big_szie / CUBE_SIZE;
    double intercube_spacing = small_size * 0.05;
    if (CUBE_SIZE > 20)
        intercube_spacing = 0.01;
    double start = big_szie / 2 + intercube_spacing * (CUBE_SIZE - 1) / 2;
    for (double z = start; z > -start; z -= small_size + intercube_spacing) {
        for (double y = start; y > -start; y -= small_size + intercube_spacing) {
            for (double x = -start; x < start; x += small_size + intercube_spacing) {

                int cx = (int)round((x + big_szie/2)/(small_size+ intercube_spacing));
                int cy = (int)round((-y + big_szie/2)/(small_size+ intercube_spacing));
                int cz = (int)round((-z + big_szie/2)/(small_size+ intercube_spacing));
                glPushMatrix();
                glTranslatef(8, 0, 0);
                glMultMatrixd(getRotationMatrix(cubesRotation[cz][cy][cx]));
                glTranslatef(-8, 0, 0);
                float v[8][3] = {
                    { x + 8, y, z },
                    { x + small_size + 8, y, z },
                    { x + small_size + 8, y - small_size, z },
                    { x + 8, y - small_size, z },
                    { x + 8, y, z - small_size },
                    { x + small_size + 8, y, z - small_size },
                    { x + small_size + 8, y - small_size, z - small_size },
                    { x + 8, y - small_size, z - small_size }
                };
                drawCube(v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7], cx, cy, cz);
                glPopMatrix();
            }
        }
    }
}

void updateRotation() {
    REP(a, rotationQueue.size()) {
        point mci = rotationQueue[a];
        int i = mci.z, j = mci.y, k = mci.x;
        Quaterniond qr = Quaterniond{ AngleAxisd{ degtorad(rorationSpeed),
rotationAxis} }; // degree to radian
        cubesRotation[i][j][k] = qr * cubesRotation[i][j][k];
    }

    totalRotation += rorationSpeed;
```

```
    if (totalRotation >= 90 || totalRotation <= -90) {
        changeState();
    }
}

void doTransition(float fromX, float toX, float fromY, float toY) {
    if (isTransition) {
        if ((ceil(xscreen) == 1 || floor(xscreen) == 1) &&
            (ceil(yscreen) == 0 || floor(yscreen) == 0)) {
            cameraX += transitionSpeed * (toX - fromX) * (8);
            cameraY += transitionSpeed * (toY - fromY) * (5);
            transition_percent += transitionSpeed;
            if (transition_percent >= 1) {
                isTransition = false;
                fromXScreen = xscreen = toXScreen;
                fromYScreen = yscreen = toYScreen;
                transition_percent = 0;
            }
        }
        else {
            cameraX += 0.001 * (toX - fromX) * (8);
            cameraY += 0.001 * (toY - fromY) * (5);
            transition_percent += 0.001;

            if (transition_percent >= 1) {
                isTransition = false;
                fromXScreen = xscreen = toXScreen;
                fromYScreen = yscreen = toYScreen;
                transition_percent = 0;
            }
        }
    }
    glTranslatef(-cameraX, -cameraY, -cameraZ);
}

void displayTitle() {

    printText(-3.5, 1.5, "R", 5, GLUT_STROKE_MONO_ROMAN, colorList[5], 5);
    printText(-2.9, 1.5, "U", 5, GLUT_STROKE_MONO_ROMAN, colorList[4], 5);
    printText(-2.3, 1.5, "B", 5, GLUT_STROKE_MONO_ROMAN, colorList[2], 5);
    printText(-1.7, 1.5, "I", 5, GLUT_STROKE_MONO_ROMAN, colorList[0], 5);
    printText(-1.1, 1.5, "K", 5, GLUT_STROKE_MONO_ROMAN, colorList[1], 5);
    printText(-0.8, 1.5, "'", 5, GLUT_STROKE_MONO_ROMAN, colorList[1], 5);
    printText(-0.5, 1.5, "S", 5, GLUT_STROKE_MONO_ROMAN, colorList[3], 5);
    printText(0.5, 1.5, "C", 5, GLUT_STROKE_MONO_ROMAN, colorList[5], 5);
```

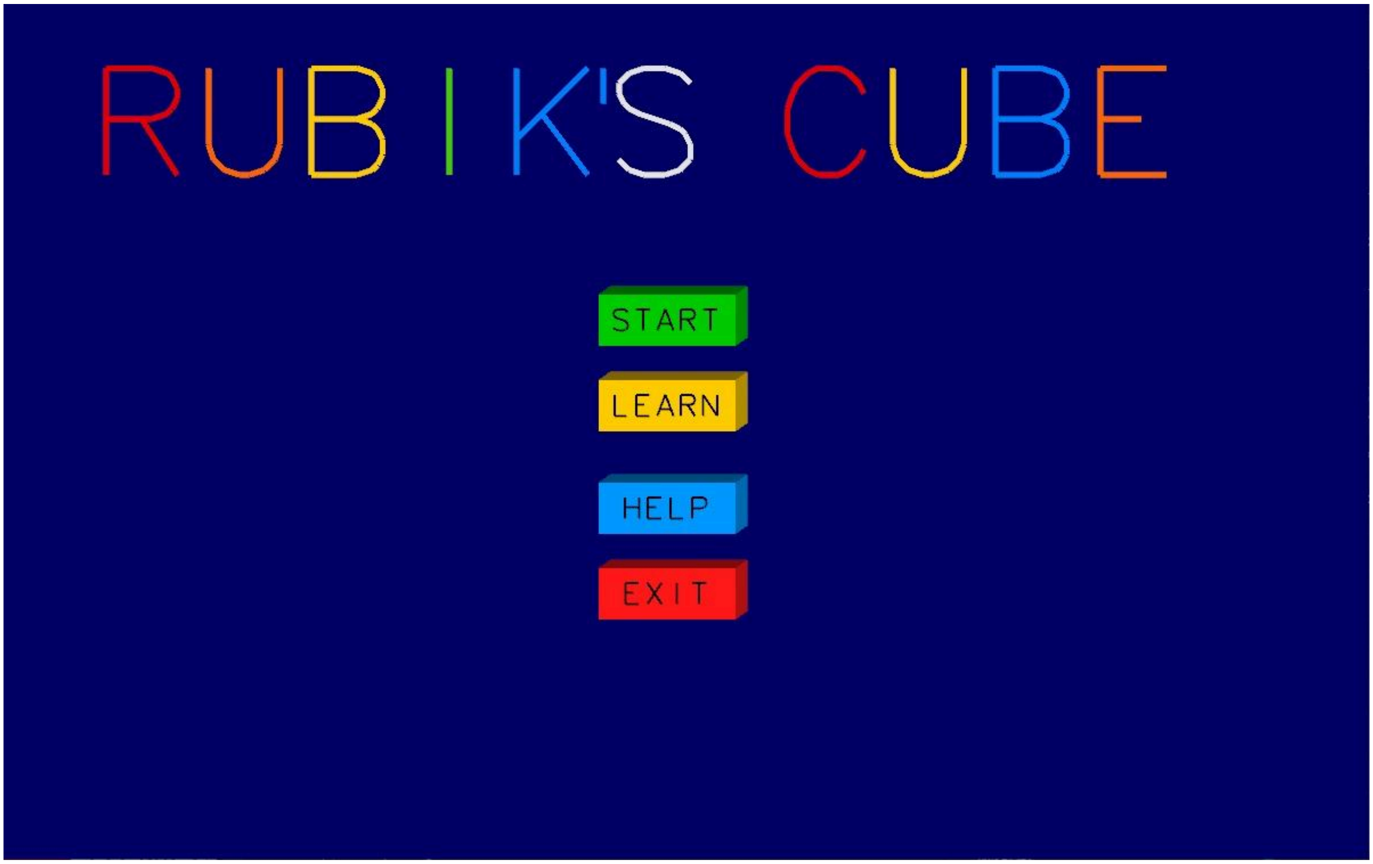
```
    printText(1.1, 1.5, "U", 5, GLUT_STROKE_MONO_ROMAN, colorList[2], 5);
    printText(1.7, 1.5, "B", 5, GLUT_STROKE_MONO_ROMAN, colorList[1], 5);
    printText(2.3, 1.5, "E", 5, GLUT_STROKE_MONO_ROMAN, colorList[4], 5);
}
void displayHelp() {
    printText(-3.8, -3.2, "INSTRUCTIONS",3,GLUT_STROKE_ROMAN, color(0, 0.6, 1), 3);
    printText(-3.5,-3.55, "1. KEYS :",1.5,GLUT_STROKE_ROMAN,color(0.8, 0.8, 0.8), 2.5);
    printText(-3.2, -3.8, "Q : Front clockwise", 1, GLUT_STROKE_ROMAN,colorList[0], 1);
    printText(-3.2, -4.0, "W : Back clockwise", 1, GLUT_STROKE_ROMAN, colorList[1], 1);
    printText(-3.2, -4.2, "A : Left clockwise", 1, GLUT_STROKE_ROMAN, colorList[2], 1);
    printText(-3.2, -4.4, "S : Right clockwise", 1, GLUT_STROKE_ROMAN,colorList[3], 1);
    printText(-3.2, -4.6, "Z : Top clockwise", 1, GLUT_STROKE_ROMAN, colorList[4], 1);
    printText(-3.2, -4.8, "X : Bottom clockwise", 1,GLUT_STROKE_ROMAN,colorList[5], 1);
    printText(-3.2, -5.0, "J : Random Rotations", 1,GLUT_STROKE_ROMAN,colorList[3], 1);
    printText(-3.2, -5.2, "1-9 : Choose the corresponding layer", 1, GLUT_STROKE_ROMAN,
colorList[3], 1);
    printText(-3.2, -5.4, "-/+ : Increment/Decrement the layer", 1, GLUT_STROKE_ROMAN,
colorList[3], 1);
    printText(-3.2, -5.6, "SHIFT : Anti-clockwise Turns", 1, GLUT_STROKE_ROMAN,
colorList[3], 1);
    printText(-3.5, -5.95,"2. MOUSE",1.5,GLUT_STROKE_ROMAN,color(0.8, 0.8, 0.8), 2.5);
    printText(-3.2, -6.2, "Drag Vertically : Rotate along X-Axis", 1,
GLUT_STROKE_ROMAN, colorList[5], 1);
    printText(-3.2, -6.4, "Drag Horizontally : Rotate along Y-Axis", 1,
GLUT_STROKE_ROMAN, colorList[0], 1);
    printText(-3.2, -6.6, "Scroll Up/Down : Rotate along Z-Axis", 1, GLUT_STROKE_ROMAN,
colorList[1], 1);
}
void displayAbout() {
    printText(-3.0, 6.8, "LEARN HOW TO SOLVE", 3, GLUT_STROKE_ROMAN, color(1, 0.8, 0),
3);
    printText(-3.5, 6.5, "                 This is our college project. It utilizes ",
1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 6.3, "        to provide Rubik's Cube enthusiasts with a
sophisticated tool for interacting ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 6.1, "     with their favorite puzzle. The program enables smooth
manipulation and rotation ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 5.9, "       of the cube, accommodating large sizes up to 50x50x50,
depending on available ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 5.7, "  memory and computational power. With its user-friendly
interface, 3D buttons, and ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 5.5, "           seamless transitions, it offers an intuitive
experience. ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 5.3, "          Have fun exploring and experimenting with this
project!", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
```

```
    printText(-3.5, 5.1, "      To Learn how to solve the RUBIK'S CUBE watch the following Link:", 1, GLUT_STROKE_ROMAN, colorList[3], 1);
    printText(-3.5, 4.9, "              https://www.youtube.com/watch?v=7Ron6MN45LY ", 1, GLUT_STROKE_ROMAN, colorList[3], 1);

    printText(-3.8, 4.0, "DEVELOPED BY", 1.2, GLUT_STROKE_MONO_ROMAN, colorList[2], 2);
    printText(-3.8, 3.7, "   AKSHAT,SMRITI,IKJOT", 1.2, GLUT_STROKE_MONO_ROMAN, colorList[4], 2);
}
void displaySolution() {

    REP(i, moveList.size()) {
        string s = "";
        s += moveList[i];
        printText(11, 2 - i * 0.3, s, 1.5, GLUT_STROKE_ROMAN, colorList[3], 2);
    }

}
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    animateButtons();
    doTransition(fromXScreen, toXScreen, fromYScreen, toYScreen);
    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == 0 || floor(yscreen) == 0)) {
        bstart.draw();
        babout.draw();
        bhelp.draw();
        bexit.draw();
        displayTitle();
    }
    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == 1 || floor(yscreen) == 1)) {
        bback01.draw();
        displayAbout();
    }
    if ((ceil(xscreen) == 0 || floor(xscreen) == 0) &&
        (ceil(yscreen) == -1 || floor(yscreen) == -1)) {
        bback0_1.draw();
        displayHelp();
    }
    if ((ceil(xscreen) == 1 || floor(xscreen) == 1) &&
        (ceil(yscreen) == 0 || floor(yscreen) == 0)) {
        //bhollow.draw();
        bback10.draw();
```

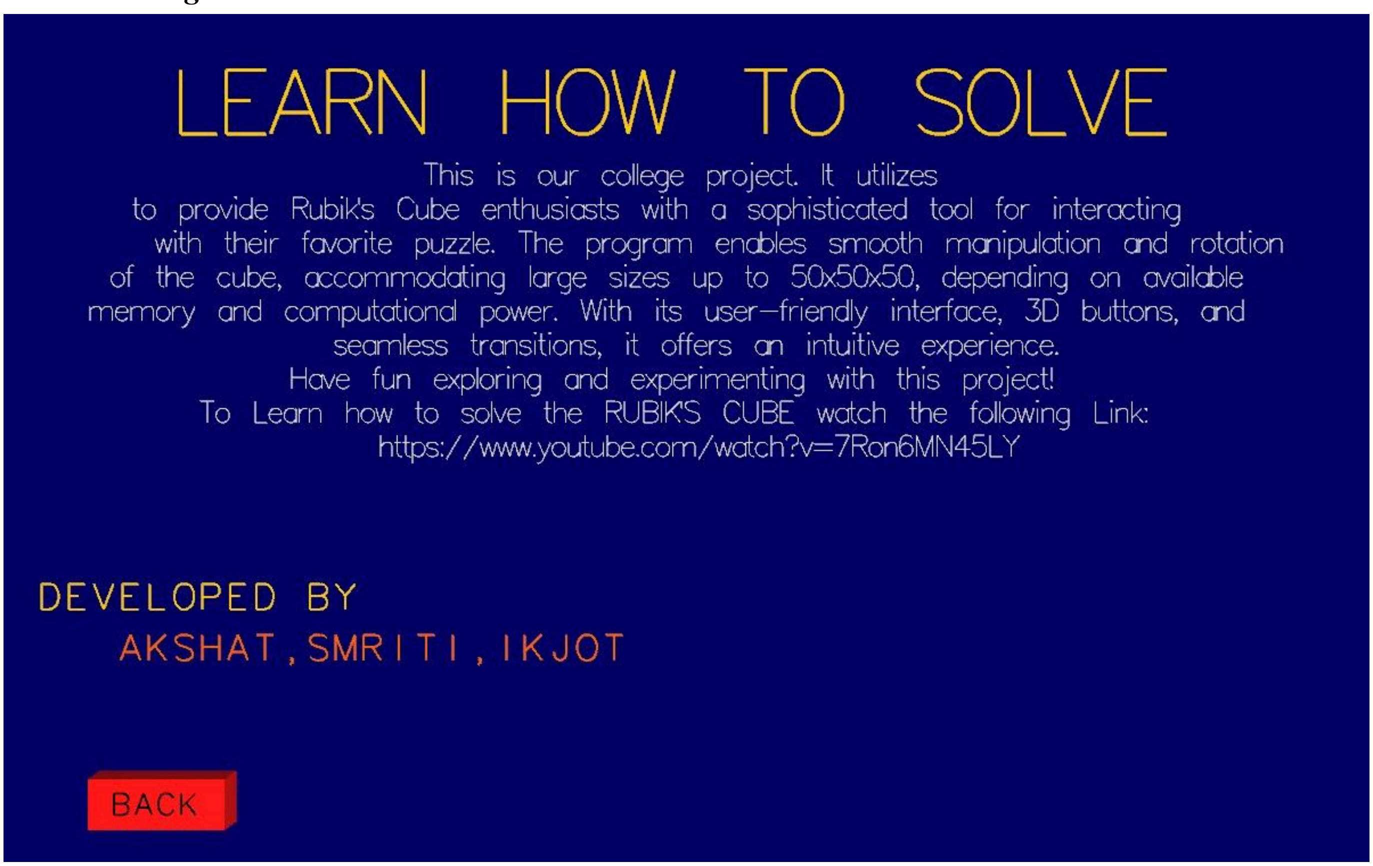
```
        if (CUBE_SIZE == 2 || CUBE_SIZE == 3)
            bsolve.draw();
        displaySolution();
        glTranslatef(8, 0, 0);
        glMultMatrixd(getRotationMatrix(camera));
        glTranslatef(-8, 0, 0);
        buildRubiksCube();

        if (rotationType) {
            updateRotation();
        }
    }
    glutSwapBuffers();
    glutPostRedisplay();
}
void init() {
    if (CUBE_SIZE > 50)
        hollow = true;
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutKeyboardFunc(keyboard);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
    glutMouseWheelFunc(mouseWheel);
    REP(i, CUBE_SIZE) {
        REP(j, CUBE_SIZE) {
            REP(k, CUBE_SIZE) {
                cubesRotation[i][j][k] = Quaterniond{ AngleAxisd{1,Vector3d{0,0,0}} };
            }
        }
    }
}
int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(width, height);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Rubik's Cube Solver ");
    glClearColor(0.0f, 0.0f, 0.4f, 1.0f);
    init();
    glEnable(GL_DEPTH_TEST);
    glLoadIdentity();
    glutMainLoop();
    return 0;
}
```

V. OUTPUT SCREENSHOTS

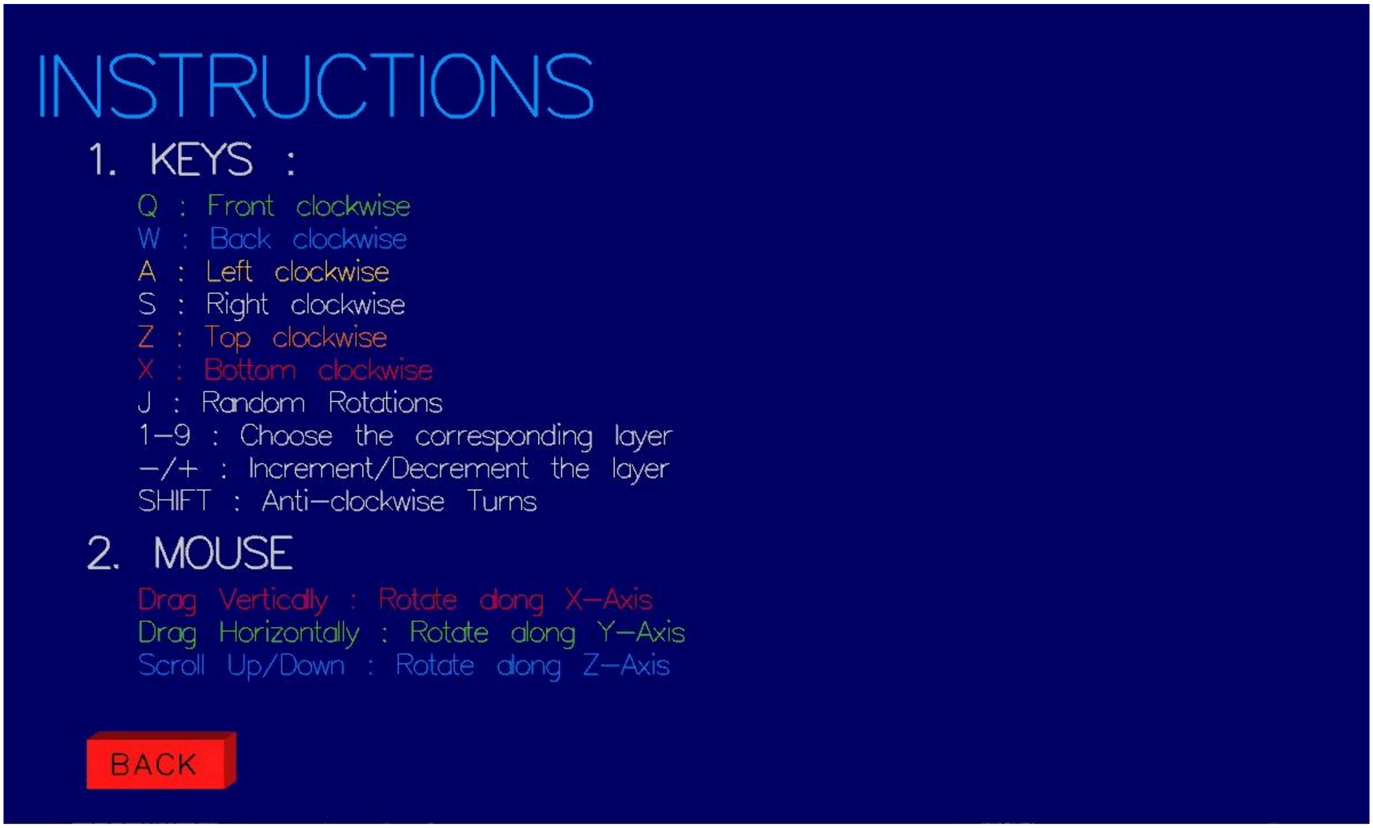
1. Home Page:



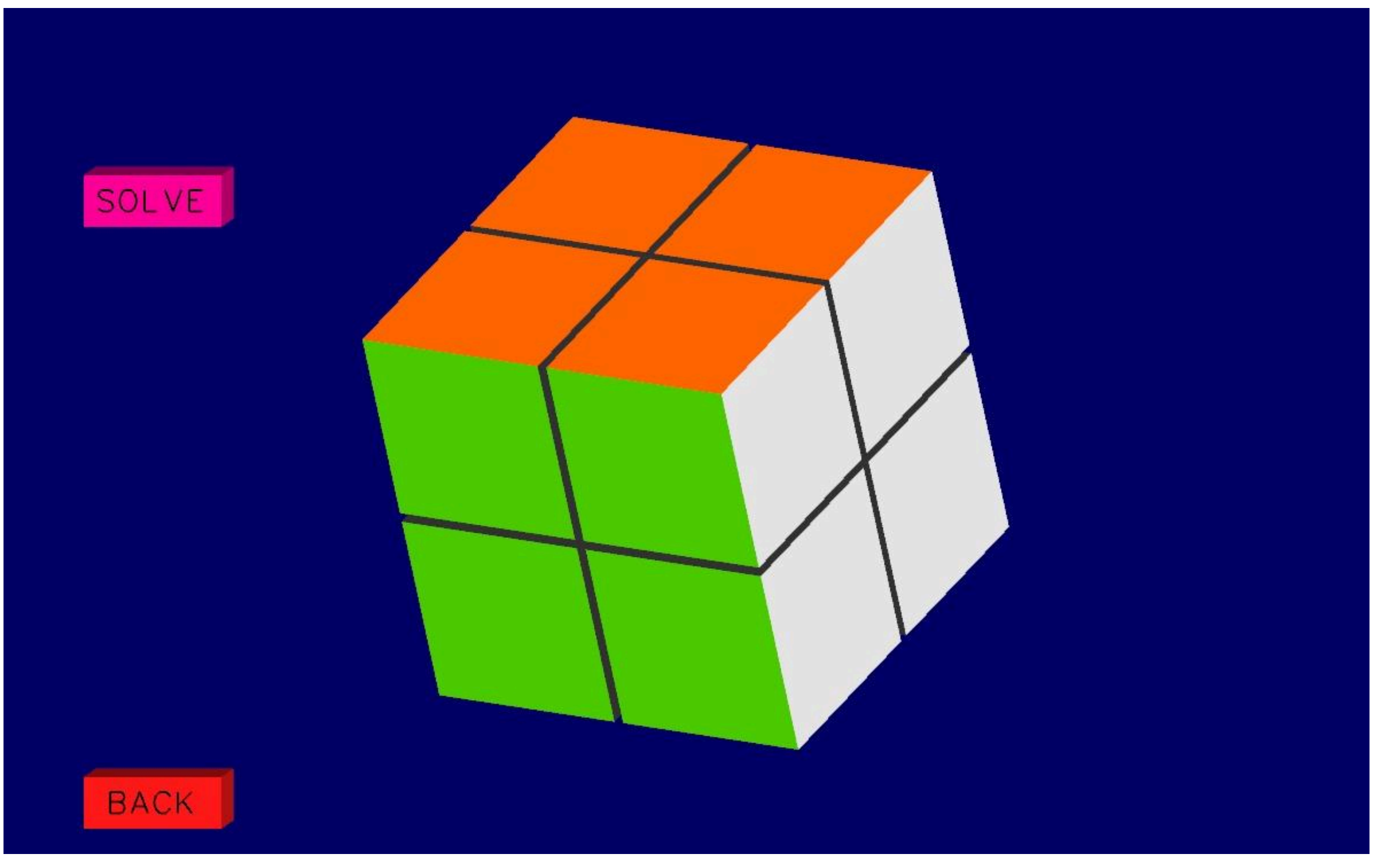
2. Learn Page:



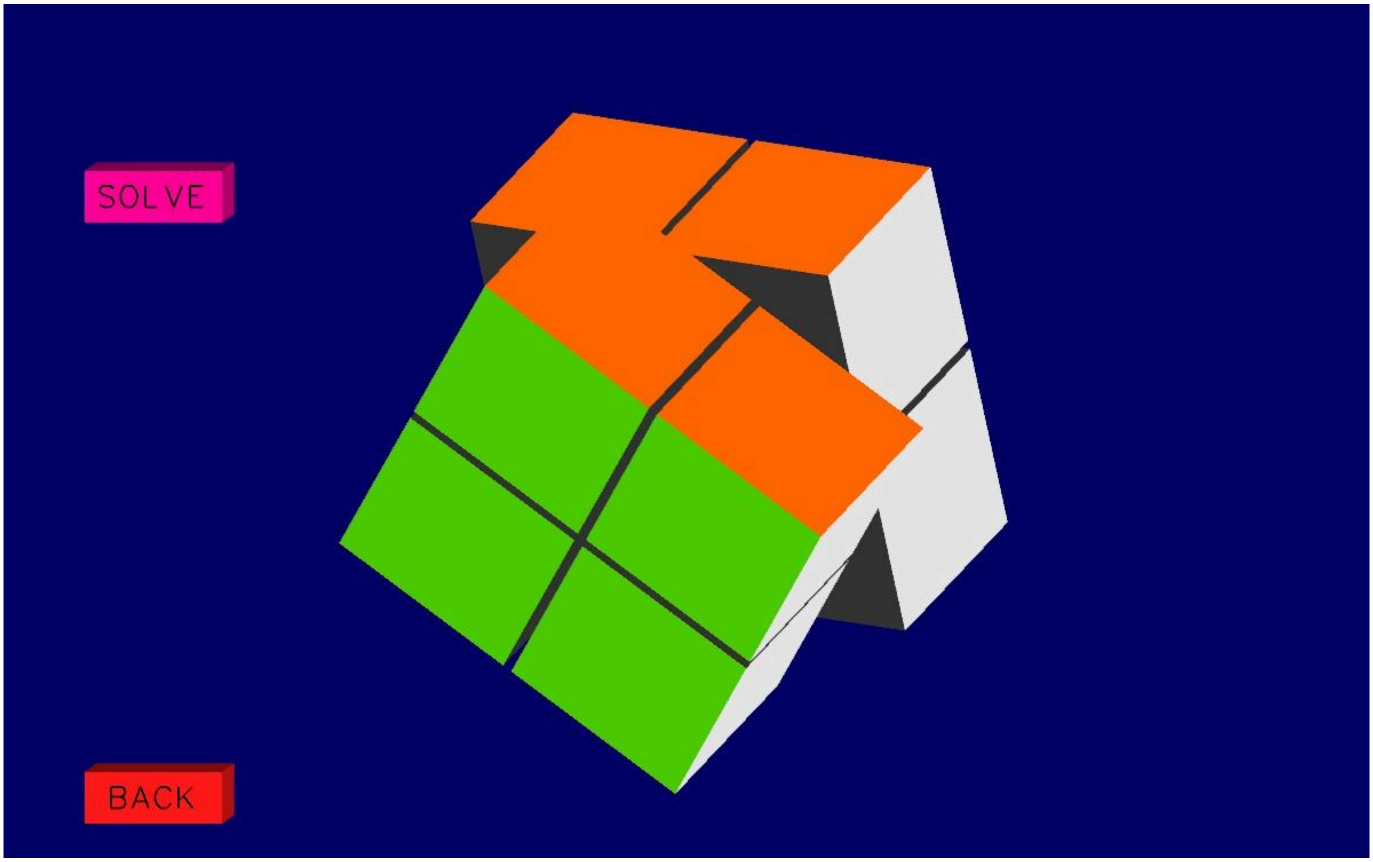
3. Instructions Page:



4. Cube Page:



5. Interacting with the Cube:



6. Solving the cube and Displaying the Solution: