# 

# Computer Graphics (UCS505)

# Project on

# Rubik's Cube Visualization

# in OpenGL

**Submitted By**

Anisha Watts 102117093

Kirat Kaur Rai 102117096

Soham Biswas 102117109

# 3CS4

**B.E. Third Year – COPC**

**Submitted To:**

**Diksha Arora**



**Computer Science and Engineering Department Thapar Institute of Engineering and Technology Patiala – 147001**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Page No.** |
| 1. | Introduction to Project | 1 |
| 2. | Computer Graphics concepts used | 2 |
| 3. | User Defined Functions | 5-6 |
| 4. | Code | 7-23 |
| 5. | Output/ Screen shots | 24-26 |

**INTRODUCTION**

The presented software artifact represents a meticulously crafted framework designed to visualize the iconic Rubik's Cube within a three-dimensional space using the OpenGL graphics library in the C++ programming language. This implementation serves as a testament to the convergence of computational graphics and classic puzzle-solving, offering enthusiasts and developers alike a comprehensive platform to explore, manipulate, and study the intricate dynamics of the Rubik's Cube.

At its core, this software encapsulates the quintessential elements of Rubik's Cube manipulation, providing users with a rich array of functionalities to interact with and observe the cube's complex configurations. Leveraging OpenGL's capabilities, the framework meticulously renders each individual cubelet, meticulously accounting for their positions, orientations, and interrelations, thus ensuring a faithful representation of the Rubik's Cube in a virtual environment.

**Key Features:**

1. **Realistic Rendering**: Through a judicious combination of matrix transformations and shading techniques, the framework achieves a visually immersive depiction of the Rubik's Cube, meticulously replicating its iconic appearance and aesthetic appeal.
2. **Interactive Manipulation**: Users are empowered to dynamically manipulate the Rubik's Cube through intuitive keyboard and mouse controls, facilitating rotation, zooming, and face selection functionalities to explore its myriad permutations.
3. **Parametric Customization**: With adjustable parameters for cube size, rotation increments, and camera perspectives, the framework offers flexibility in tailoring the visualization experience to suit diverse preferences and analytical requirements.
4. **Efficient Algorithms**: Behind the scenes, the software employs optimized algorithms to manage cube rotations, ensuring smooth and responsive interactions even amidst complex configurations and rapid user inputs.

**COMPUTER GRAPHICS CONCEPTS USED**

In our project on Rubik’s Cube Visualization in OpenGL, we have employed a variety of computer graphics concepts to achieve a realistic and immersive rendering of the Rubik’s Cube in a three-dimensional space. Below, we provide an overview of the key computer graphics concepts utilized in our project:

1. **3D Modeling and Transformation**:
   * We utilized 3D odelling techniques to represent the Rubik’s Cube as a collection of individual cubelets within a three-dimensional Cartesian coordinate system.
   * Matrix transformations, including translation, rotation, and scaling, were employed to manipulate the position, orientation, and size of each cubelet within the virtual space.
   * **Detailed overview on Scaling, Rotation and Translation:**
2. **Scaling**:
   * Scaling was instrumental in accurately depicting the size and proportions of individual cubelets comprising the Rubik’s Cube.
   * By applying scaling transformations to each cubelet, we ensured uniformity in size while maintaining the overall structure and symmetry of the cube.
   * Additionally, scaling allowed for the adjustment of cubelet dimensions to accommodate variations in cube size and resolution, enhancing the flexibility and customizability of the visualization.
3. **Rotation**:
   * Rotation formed the core mechanism for manipulating the Rubik’s Cube, simulating the twisting and turning of its individual layers.
   * Through rotational transformations, we enabled users to dynamically rotate specific layers of the Rubik’s Cube around designated axes, thus altering its configuration and revealing new patterns.
   * Implementing rotation involved calculating the appropriate transformation matrices and applying them to the cubelets’ coordinates, ensuring seamless and responsive manipulation of the cube’s geometry.
4. **Translation**:
   * Translation facilitated the precise positioning of each cubelet within the virtual space, allowing for the creation of a cohesive and visually coherent Rubik’s Cube.
   * By applying translation transformations, we determined the spatial coordinates of each cubelet relative to a common reference point, such as the center of the cube.
   * Translation also enabled the arrangement of cubelets into distinct layers and columns, aligning them along predefined axes to form the intricate structure of the Rubik’s Cube.
5. **Integration and Interactivity**:
   * The seamless integration of scaling, rotation, and translation transformations enabled users to interact with the Rubik’s Cube intuitively and dynamically.
   * Interactive controls allowed users to manipulate the cube’s rotation, scaling, and translation parameters in real-time, providing a responsive and engaging user experience.
   * Through these interactive features, users could explore the Rubik’s Cube from different perspectives, zoom in/out to examine intricate details, and rotate individual layers to solve the puzzle and uncover hidden patterns.
6. **Shading and Lighting**:
   * To enhance the visual realism of the Rubik’s Cube, we implemented shading techniques to simulate the interaction of light with the cubelets’ surfaces.
   * We employed lighting models, such as Phong shading, to calculate the intensity of light reflection and achieve smooth transitions between light and shadow regions on the cube’s surface.
   * Ambient, diffuse, and specular lighting components were integrated to simulate the complex interplay of light sources and materials.
7. **Projection and Viewing**:
   * We employed perspective projection to transform the three-dimensional coordinates of the Rubik’s Cube into a two-dimensional image plane, replicating the effect of human vision.
   * Viewing parameters, including the position and orientation of the virtual camera, were manipulated to control the viewer’s perspective and facilitate dynamic exploration of the Rubik’s Cube.
8. **Rasterization and Rendering**:
   * We utilized rasterization techniques to convert geometric primitives, such as points, lines, and polygons representing the cubelets, into pixel-based images on the screen.
   * OpenGL’s rendering pipeline was leveraged to process and render the cubelets efficiently, taking into account their position, orientation, and surface properties.
9. **Interaction and User Interface**:
   * Interactive controls were implemented to enable users to manipulate the Rubik’s Cube dynamically, including rotating individual layers, zooming in/out, and selecting specific cube faces.
   * User interface elements, such as keyboard and mouse input handlers, were integrated to facilitate user interaction and enhance the immersive experience of interacting with the Rubik’s Cube.

By incorporating these computer graphics concepts into our project, we aimed to create a visually compelling and interactive visualization of the Rubik’s Cube, offering users an engaging platform to explore and interact with this iconic puzzle in a virtual environment.

**USER DEFINED FUNCTION**

In our Rubik's Cube Visualization project, several user-defined functions were created to facilitate various aspects of the visualization, interaction, and manipulation of the Rubik's Cube. Here is a list of the key user-defined functions:

1. **load\_visualization\_parameters()**:
   * This function initializes the visualization parameters, such as the projection matrix and camera position, to set up the viewing perspective for the Rubik's Cube.
2. **apply\_rotation(GLfloat angle)**:
   * This function applies a rotation transformation to the Rubik's Cube based on the specified angle, allowing users to rotate individual layers of the cube around designated axes.
3. **reset\_selected\_face()**:
   * This function resets the parameters for selecting a specific face of the Rubik's Cube, ensuring that subsequent rotation operations target the entire cube or individual layers as desired.
4. **set\_camera()**:
   * This function defines the position and orientation of the virtual camera within the three-dimensional space, establishing the viewer's perspective for observing the Rubik's Cube.
5. **draw\_cube(int x, int y, int z)**:
   * This function draws an individual cubelet of the Rubik's Cube at the specified position (x, y, z) within the three-dimensional grid, applying appropriate transformations for scaling, rotation, and translation.
6. **init\_func()**:
   * This function initializes the rendering parameters, including the cube size, lighting model, material properties, and OpenGL rendering settings, to prepare the environment for visualizing the Rubik's Cube.
7. **draw\_func()**:
   * This function serves as the main rendering loop for the Rubik's Cube visualization, iterating through the grid of cube positions and invoking the draw\_cube() function to render each cubelet.
8. **reshape\_func(GLsizei w, GLsizei h)**:
   * This function handles window resizing events, adjusting the viewport and aspect ratio to ensure that the Rubik's Cube maintains its proportions and remains visible within the window.
9. **keyboard\_func(unsigned char key, int x, int y)**:
   * This function captures keyboard input from the user, allowing them to interact with the Rubik's Cube by rotating layers, zooming in/out, and selecting specific faces for manipulation.
10. **mouse\_func(int button, int state, int x, int y)**:
    * This function captures mouse input events, enabling users to zoom in/out or perform other interactive actions on the Rubik's Cube using mouse clicks.

**CODE**

#include <GL/gl.h>

#include <GL/glut.h>

#include <vector>

using namespace std;

struct cube\_rotate{

GLfloat angle, x, y, z;

};

GLfloat angle, fAspect, cube\_size;

GLint rot\_x, rot\_y, crement, x\_0, x\_k, y\_0, y\_k, z\_0, z\_k, gap, gap\_crement;

//cube\_rotate cube\_rotations[3][3][3];

vector<cube\_rotate> cube\_rotations[3][3][3];

void load\_visualization\_parameters(void);

void apply\_rotation(GLfloat angle){

vector<cube\_rotate> face[3][3];

int index;

cube\_rotate rotation;

// copy face to be rotated

// apply rotation to face

for(int i = 0; i < 3; ++i)

for(int j = 0; j < 3; ++j) {

index = 2 - j%3;

if(x\_0 == x\_k){

rotation = {angle, 1.0, 0.0, 0.0};

face[index][i] = cube\_rotations[x\_k][i][j];}

if(y\_0 == y\_k){

rotation = {angle, 0.0, 1.0, 0.0};

face[index][i] = cube\_rotations[j][y\_k][i];

}

if(z\_0 == z\_k){

rotation = {-1 \* angle, 0.0, 0.0, 1.0};

face[index][i] = cube\_rotations[j][i][z\_k];

}

face[index][i].push\_back(rotation);

}

// copy back rotated face

for(int i = 0; i < 3; ++i)

for(int j = 0; j < 3; ++j) {

if(x\_0 == x\_k)

cube\_rotations[x\_k][i][j] = face[i][j];

if(y\_0 == y\_k)

cube\_rotations[j][y\_k][i] = face[i][j];

if(z\_0 == z\_k)

cube\_rotations[j][i][z\_k] = face[i][j];

}

}

// reset face selection parameters

void reset\_selected\_face(){

x\_0 = 0;

x\_k = 2;

y\_0 = 0;

y\_k = 2;

z\_0 = 0;

z\_k = 2;

}

void set\_camera()

{

gluLookAt(0,80,200, 0,0,0, 0,1,0);

}

// draw a cube

void draw\_cube(int x, int y, int z)

{

vector<cube\_rotate> lrot = cube\_rotations[x][y][z];

glPushMatrix();

// translate to final position

glTranslatef((x - 1) \* cube\_size + x \* gap, (y - 1) \* cube\_size + y \* gap, (z - 1) \* cube\_size + z \* gap);

// rotate cube to correct position

for(int i = lrot.size() - 1; i >= 0; --i)

glRotatef(lrot[i].angle, lrot[i].x, lrot[i].y, lrot[i].z);

glColor3f(1.0f, 0.0f, 0.0f);

glBegin(GL\_QUADS); // front

glNormal3f(0.0, 0.0, 1.0); // face normal

glVertex3f(cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(-cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(-cube\_size/2, -cube\_size/2, cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, cube\_size/2);

glEnd();

glColor3f(1.0f, 0.5f, 0.0f);

glBegin(GL\_QUADS); // back

glNormal3f(0.0, 0.0, -1.0); // face normal

glVertex3f(cube\_size/2, cube\_size/2, -cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(-cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(-cube\_size/2, cube\_size/2, -cube\_size/2);

glEnd();

glColor3f(0.0f, 0.0f, 1.0f);

glBegin(GL\_QUADS); // left

glNormal3f(-1.0, 0.0, 0.0); // face normal

glVertex3f(-cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(-cube\_size/2, cube\_size/2, -cube\_size/2);

glVertex3f(-cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(-cube\_size/2, -cube\_size/2, cube\_size/2);

glEnd();

glColor3f(0.0f, 1.0f, 0.0f);

glBegin(GL\_QUADS); // right

glNormal3f(1.0, 0.0, 0.0); // face normal

glVertex3f(cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(cube\_size/2, cube\_size/2, -cube\_size/2);

glEnd();

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS); // top

glNormal3f(0.0, 1.0, 0.0); // face normal

glVertex3f(-cube\_size/2, cube\_size/2, -cube\_size/2);

glVertex3f(-cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(cube\_size/2, cube\_size/2, cube\_size/2);

glVertex3f(cube\_size/2, cube\_size/2, -cube\_size/2);

glEnd();

glColor3f(1.0f, 1.0f, 0.0f);

glBegin(GL\_QUADS); // bottom

glNormal3f(0.0, -1.0, 0.0); // face normal

glVertex3f(-cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, -cube\_size/2);

glVertex3f(cube\_size/2, -cube\_size/2, cube\_size/2);

glVertex3f(-cube\_size/2, -cube\_size/2, cube\_size/2);

glEnd();

glPopMatrix();

} // draw cube function

// draw function

void draw\_func(void)

{

int x = -cube\_size, y = -cube\_size, z = -cube\_size;

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// reset transformations

glLoadIdentity();

// set camera position

set\_camera();

// apply visualization transformations

glRotatef(rot\_x, 1.0, 0.0, 0.0); // rotate in y axis

glRotatef(rot\_y, 0.0, 1.0, 0.0); // rotate in x axis

for(int i = 0; i < 3; ++i) // step through x axis

for(int j = 0; j < 3; ++j) // step through y axis

for(int k = 0; k < 3; ++k) { // step through z axis

// draw a single cube

draw\_cube(i, j, k);

}

// flush opengl commands

glutSwapBuffers();

}

// init rendering parameters

void init\_func (void)

{

// init parameters

cube\_size = 30.0; // cuboid size

rot\_x = 0.0; // view rotation x

rot\_y = 0.0; // view rotation y

crement = 5; // rotation (in/de)crement

gap = 5;

gap\_crement = 3;

// initialize cuboid rotations

// init lighting

GLfloat ambient\_lighte[4]={0.2,0.2,0.2,1.0};

GLfloat diffuse\_light[4]={0.7,0.7,0.7,1.0}; // color

GLfloat specular\_light[4]={1.0, 1.0, 1.0, 1.0}; // brightness

GLfloat light\_position[4]={0.0, 50.0, 50.0, 1.0};

// material brightness capacity

GLfloat specularity[4]={1.0,1.0,1.0,1.0};

GLint material\_specularity = 60;

// black background

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

// Gouraud colorization model

glShadeModel(GL\_SMOOTH);

// material reflectability

glMaterialfv(GL\_FRONT,GL\_SPECULAR, specularity);

// brightness concentration

glMateriali(GL\_FRONT,GL\_SHININESS,material\_specularity);

// activate ambient light

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, ambient\_lighte);

// define light parameters

glLightfv(GL\_LIGHT0, GL\_AMBIENT, ambient\_lighte);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, diffuse\_light );

glLightfv(GL\_LIGHT0, GL\_SPECULAR, specular\_light );

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position );

// enable changing material color

glEnable(GL\_COLOR\_MATERIAL);

// enable lighting

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);

// enable depth buffering

glEnable(GL\_DEPTH\_TEST);

angle=45;

} // init

// specify what's shown in the window

void load\_visualization\_parameters(void)

{

// specify projection coordinate system

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

// specify projection perspective

gluPerspective(angle,fAspect,0.4,500);

// init model coordinate system

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

// specify observer and target positions

set\_camera();

} // load visualization parameters

// window reshape callback

void reshape\_func(GLsizei w, GLsizei h)

{

// prevents division by zero

if ( h == 0 ) h = 1;

// viewport size

glViewport(0, 0, w, h);

// aspect ratio

fAspect = (GLfloat)w/(GLfloat)h;

load\_visualization\_parameters();

} // reshape function

// keyboard function callback

void keyboard\_func(unsigned char key, int x, int y)

{

switch(key) {

case '+':

gap += gap\_crement;

break;

case '-':

gap -= gap\_crement;

break;

// view rotation

// INcrement or DEcrement

case 'L': // right

case 'l':

rot\_y = (rot\_y - crement) % 360;

break;

case 'J': // left

case 'j':

rot\_y = (rot\_y + crement) % 360;

break;

case 'I': // down

case 'i':

rot\_x = (rot\_x + crement) % 360;

break;

case 'K': // up

case 'k':

rot\_x = (rot\_x - crement) % 360;

break;

// end of view rotation

// cube movements

// select cube face

// x-axis faces

case 'Q':

case 'q':

reset\_selected\_face();

x\_0 = 0;

x\_k = 0;

break;

case 'W':

case 'w':

reset\_selected\_face();

x\_0 = 1;

x\_k = 1;

break;

case 'E':

case 'e':

reset\_selected\_face();

x\_0 = 2;

x\_k = 2;

break;

// y-axis faces

case 'A':

case 'a':

reset\_selected\_face();

y\_0 = 0;

y\_k = 0;

break;

case 'S':

case 's':

reset\_selected\_face();

y\_0 = 1;

y\_k = 1;

break;

case 'D':

case 'd':

reset\_selected\_face();

y\_0 = 2;

y\_k = 2;

break;

// z-axis faces

case 'C':

case 'c':

reset\_selected\_face();

z\_0 = 0;

z\_k = 0;

break;

case 'X':

case 'x':

reset\_selected\_face();

z\_0 = 1;

z\_k = 1;

break;

case 'Z':

case 'z':

reset\_selected\_face();

z\_0 = 2;

z\_k = 2;

break;

// move selected face

case 'U': // counter-clockwise

case 'u':

apply\_rotation(-90);

break;

case 'O': // clockwise

case 'o':

apply\_rotation(90);

break;

// end of cube movements

default:

break;

}

glutPostRedisplay();

}

// mouse function callback

void mouse\_func(int button, int state, int x, int y)

{

if (button == GLUT\_LEFT\_BUTTON)

if (state == GLUT\_DOWN) { // Zoom-in

if (angle >= 10) angle -= 5;

}

if (button == GLUT\_RIGHT\_BUTTON)

if (state == GLUT\_DOWN) { // Zoom-out

if (angle <= 130) angle += 5;

}

load\_visualization\_parameters();

glutPostRedisplay();

} // mouse function

int main(int argc, char \*\*argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(400,350);

glutCreateWindow("Rubik’s Cube");

glutDisplayFunc(draw\_func);

glutReshapeFunc(reshape\_func);

glutMouseFunc(mouse\_func);

glutKeyboardFunc(keyboard\_func);

init\_func();

glutMainLoop();

}

**SCREENSHOTS**

