Visvesvaraya Technological University
Belagavi-590 018, Karnataka

A Mini Project Report on

"RUBIKS CUBE"

Mini Project Report submitted in partial fulfillment of the requirement for the Computer Graphics Laboratory with Mini Project [18CSL67]

Bachelor of Engineering
in
Computer Science and Engineering

Submitted by,
Y V Sai Parimala[1JT19CS107]
Sahana H S [1JT19CS116]

Department of Computer Science and Engineering
Accredit by NBA, New Delhi
Jyothy Institute of Technology
Tataguni, Bengaluru-560082

Jyothy Institute of Technology

Tataguni, Bengaluru-560082

Department of Computer Science and Engineering

CERTIFICATE

Certified that the mini project work entitled **“Rubiks Cube”** carried out by **Y V Sai Parimala[1JT19CS107]** and **Sahana H S [1JT19CS116],** bonafide students of Jyothy Institute of Technology, in partial fulfilment for the award of **Bachelor of Engineering** in **Computer Science and Engineering** department of the **Visvesvaraya Technological University, Belagavi** during academic the year **2022-2023**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Degree.

Mr. Vallabh Mahale	**Dr. Prabhanjan S**
Guide, Asst. Professor	Professor & HOD
Dept. of CSE	Dept. of CSE

External Viva Examiner

Signature with Date:

1.
2.

ACKNOWLEDGEMENT

Firstly, we are very grateful to this esteemed institution **“Jyothy Institute of Technology**” for providing us an opportunity to complete our project.

We express our sincere thanks to our **Principal Dr. Gopalakrishna K** for providing us with adequate facilities to undertake this project.

We would like to thank **Dr. Prabhanjan S, Professor and Head of Computer Science** and Engineering Department for providing his valuable support.

We would like to thank our guide **Mr. Vallabh Mahale, Assistant Professor** for his keen interest and guidance in preparing this work.

Finally, we would thank all our friends who have helped us directly or indirectly in this project.

Y V Sai Parimala[1JT19CS107]
Sahana H S[1JT19CS116]

ABSTRACT

This project is about the simulation of Rubik Cube. I am implementing it using different OpenGL primitives, OpenGL libraries and combining them together in a required manner.

It highlights the key feature of a data structure and its high-quality efficiency that is obtained on its usage in the application program. This project consists of Rubik Cube which is constructed by using graphics primitives available in OpenGL library. It illustrates the role of different call back functions that provides easier way to accomplish our project in an effective manner. Here the Rubik Cube is rotating. We use the mouse interaction to increase the speed of rotation.

The project has been implemented by efficiently using the data structures to obtain the optimized results and also various functions and features that are made available by the OpenGL software package have been utilized effectively.

Table of Contents

CHAPTER 1
INTRODUCTION

1. INTRODUCTION

1.1 Introduction to CG

Graphics is defined as any sketch or a drawing or a special network that pictorially represents some meaningful information. Computer Graphics is used where a set of image needs to be manipulated or the creation of the image in the form of pixels and is drawn on the computer. Computer Graphics can be used in digital photography, film, entertainment, electronic gadgets and all other core technologies which are required. It is a vast subject and area in the field of computer science. Computer Graphics can be used in UI design, rendering, geometric object, animation and many more. In most area, computer graphics is an abbreviation of CG. There are several tools used for implementation of Computer Graphics.

Computer Graphics refers to several things:

- The manipulation and the representation of the image or the data in a graphical manner.
- Various technologies required for the creation and manipulation.
- Digital synthesis and its manipulation.

Applications

- **Computer Graphics are used for aided design for engineering and architectural system-** These are used in electrical automobile, electro-mechanical, mechanical, electronic devices. For example: gears and bolts.
- **Computer Art** – MS Paint, Adobe Photoshop.
- **Presentation Graphics** – It is used to summarize financial statistical scientific or economic data. For example- Bar chart, Line chart.
- **Entertainment-** It is used in motion picture, music video, television gaming.
- **Education and training-** It is used to understand operations of complex system. It is also used for specialized system such for framing for captains, pilots and so on.
- **Visualization-** To study trends and patterns. For example- Analyzing satellite photo of earth.

1.2 Introduction to OpenGL

Interface

OpenGL is an application program interface (API) offering various functions to implement primitives, models and images. This offers functions to create and manipulate render lighting, coloring, viewing the models. OpenGL offers different coordinate system and frames. OpenGL offers translation, rotation and scaling of objects. Functions in the main GL library have names that begin with gl and are stored in a library usually referred to as GL. The second is the OpenGL Utility Library (GLU). The library uses only GL functions but contains code for creating common objects and simplifying viewing. All functions in GLU can be created from the core GL library but application programmers prefer not to write the code repeatedly. The GLU library is available in all OpenGL implementations; functions in the GLU library begin with the letter glu. Rather than using a different library for each system we use a readily available library called OpenGL Utility Toolkit (GLUT), which provides the minimum functionality that should be expected in any modern windowing system.

Overview

- OpenGL (Open Graphics Library) is the interface between a graphics program and graphics hardware. It is streamlined. In other words, it provides low-level functionality. For example, all objects are built from points, lines and convex polygons. Higher level objects like cubes are implemented as six four-sided polygons.
- OpenGL supports features like 3-dimensions, lighting, anti-aliasing, shadows, textures, depth effects, etc.
- It is system-independent. It does not assume anything about hardware or operating system and is only concerned with efficiently rendering mathematically described scenes. As a result, it does not provide any windowing capabilities.
- It is a state machine. At any moment during the execution of a program there is a current model transformation.
- It is a rendering pipeline. The rendering pipeline consists of the following steps:

* Defines objects mathematically.
* Arranges objects in space relative to a viewpoint.
* Calculates the colorof the objects.

CHAPTER 2
OpenGL Architecture

2. OpenGL Architecture

1) Pipeline Architectures

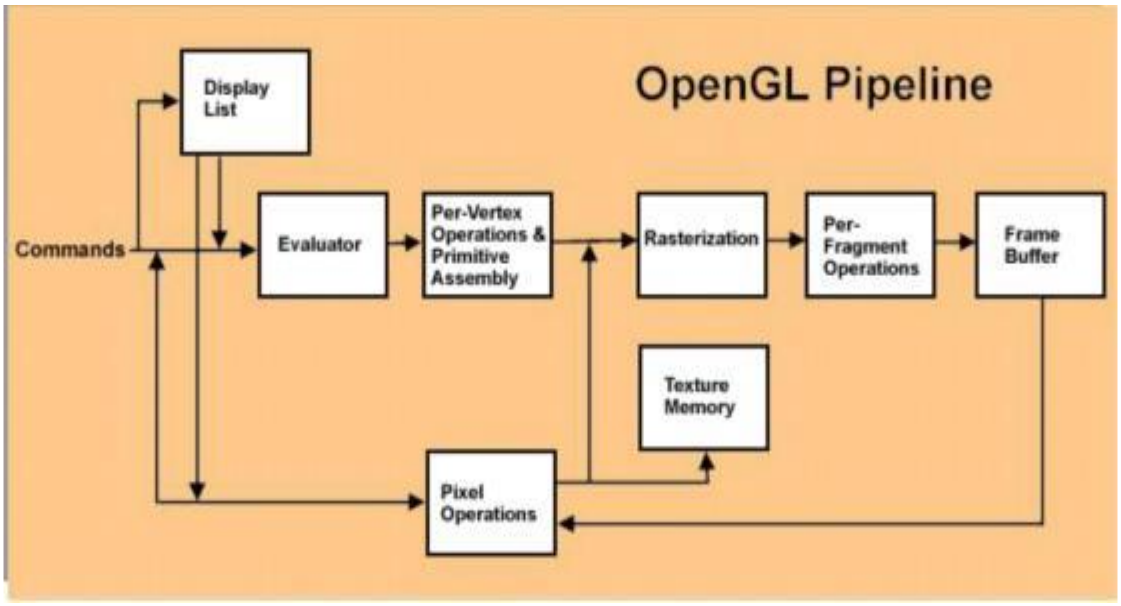


FIG:2.1 OPENGL PIPELINE ARCHITECTURE

• **Display Lists:** All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (1 alternative to retaining data in a display list is processing the data immediately - also known as immediate mode.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

• **Evaluators:** All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colours, and spatial coordinate values from the control points.

• **Per-Vertex Operations:** For vertex data, next is the "per-vertex operations" stage, which converts. The vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a colour value.

• **Primitive Assembly:** Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then view port and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. The results or this stage are complete geometric primitives, which are the transformed and clipped vertices with related colour, depth, and sometimes texture coordinate values and guidelines for the rasterization step.

• **Pixel Operations:** While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory. There are special pixel copy operations to copy data in the frame buffer to other parts of the frame buffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the frame buffer.

• **Texture Assembly:** An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them. Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

• **Rasterization:** Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the frame buffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support ant aliasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

2) OpenGL Engine and Drivers

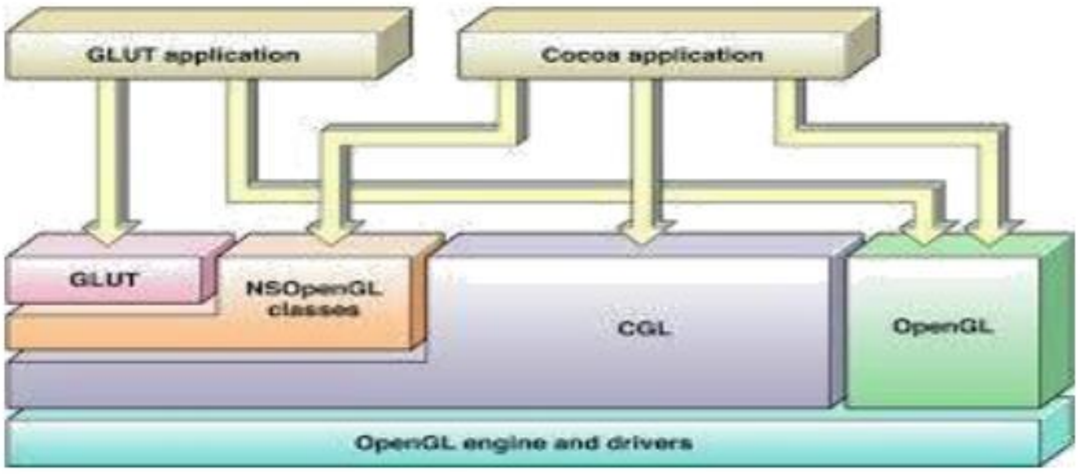


FIG:2.2 OPENGL ENGINE AND DRIVERS

CPU-GPU Cooperation

The architecture of OpenGL is based on a client-server model. An application program written to use the OpenGL API is the "client" and runs on the CPU. The implementation of the OpenGL graphics engine (including the GLSL shader programs we write) is the "server" and runs on the GPU. Geometry and many other types of attributes are stored in buffers called Vertex Buffer Objects (or VBOs). These buffers are allocated on the GPU and filled by your CPU program. We will get our first glimpse into this process (including how these buffers are allocated, used, and deleted) in the first sample program we will study.
Modeling, rendering, and interaction is very much a cooperative process between the CPU client program and the GPU server programs written in GLSL. An important part of the design process is to decide how best to divide the work and how best to package and communicate required information from the CPU to the GPU. There is no standard "best way" to do this that is applicable to all programs, but we will study a few very common approaches.

Window Manager Interfaces

OpenGL is a pure output-oriented modeling and rendering API. It has no facilities for creating and managing windows, obtaining runtime events, or any other such window system dependent operation. OpenGL implicitly assumes a window-system interface that fills these needs and invokes user-written event handlers as appropriate.
It is beyond the scope of these notes to list, let alone compare and contrast, all window manager interfaces that can be used with OpenGL. Two very common window system interfaces are:

GLFW: Runs on Linux, Macintosh, and Windows.
GLUT: (actually *freeglut*): A window interface that used to be the most common one used when teaching OpenGL. It, too, runs on Linux, Macintosh, and Windows.

3) <u>Application Development-API’s</u>

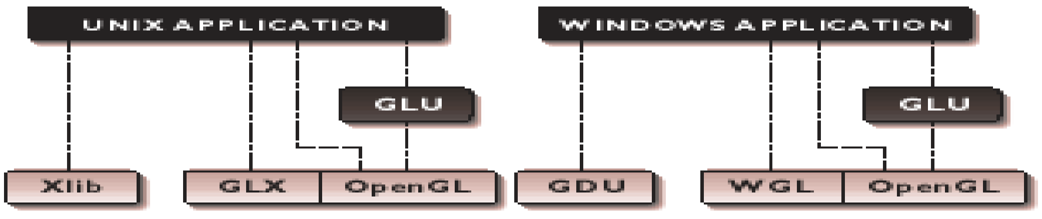


FIG:2.3 APPLICATIONS DEVELOPMENT(API’S)

This diagram demonstrates the relationship between OpenGL GLU and windowing APIs.

Leading software developers use OpenGL, with its robust rendering libraries, as the 2D/3D graphics foundation for higher-level APIs. Developers leverage the capabilities of OpenGL to deliver highly differentiated, yet widely supported vertical market solutions. For example, Open Inventor provides a cross-platform user interface and flexible scene graph that makes it easy to create OpenGL applications. IRIS Performer < leverages OpenGL functionality and delivers additional features tailored for the demanding high frame rate markets such as visual simulation and virtual sets OpenGL Optimizer is a toolkit for real-time interaction, modification, and rendering of complex surface-based models such as those found in CAD/CAM and special effects creation. OpenGL Volumizer is a high-level immediate mode volume rendering API for the energy, medical and sciences markets. OpenGL Shader provides a common interface to support realistic visual effects, bump mapping, multiple textures, environment maps, volume shading and an unlimited array of new effects using hardware acceleration on standard OpenGL graphics cards.

CHAPTER 3 IMPLEMENTATION

Code section:

```
#include <string.h>
#include<glut.h>
#include<stdio.h>
void *font = GLUT_BITMAP_TIMES_ROMAN_24;
char defaultMessage[] = "Rotation Speed:";
char *message = defaultMessage;
void
output(int x, int y, char *string)
{
 int len, i;
 glRasterPos2f(x, y);
 len = (int) strlen(string);
 for (i = 0; i < len; i++) {
 glutBitmapCharacter(font, string[i]);
  }
}
static float speed=0.0;
static int top[3][3]={{0,0,0},{0,0,0},{0,0,0}},
right[3][3]={{1,1,1},{1,1,1},{1,1,1}},
front[3][3]={{2,2,2},{2,2,2},{2,2,2}},
back[3][3]={{3,3,3},{3,3,3},{3,3,3}},
bottom[3][3]={{4,4,4},{4,4,4},{4,4,4}},
left[3][3]={{5,5,5},{5,5,5},{5,5,5}},
temp[3][3];
int solve[300];
int count=0;
int solve1=0;
static int rotation=0;
int rotationcomplete=0;
static GLfloat theta=0.0;
static GLint axis=0;
static GLfloat p=0.0,q=0.0,r=0.0;
static GLint inverse=0;
static GLfloat angle=0.0;
int beginx=0,beginy=0;
int moving=0;


static int speedmetercolor[15]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
static int speedmetercount=-1;
GLfloat vertices[][3]={{-1.0,-1.0,-1.0},
 {1.0,-1.0,-1.0},
 {1.0,1.0,-1.0},
 {-1.0,1.0,-1.0}, //center
 {-1.0,-1.0,1.0},
 {1.0,-1.0,1.0},
 {1.0,1.0,1.0},
 {-1.0,1.0,1.0},

 {-1.0,-3.0,-1.0},
 {1.0,-3.0,-1.0},
 {1.0,-1.0,-1.0},
 {-1.0,-1.0,-1.0}, //bottom center
 {-1.0,-3.0,1.0},
 {1.0,-3.0,1.0},
 {1.0,-1.0,1.0},
 {-1.0,-1.0,1.0},

 {-3.0,-1.0,-1.0},
 {-1.0,-1.0,-1.0},
 {-1.0,1.0,-1.0},
 {-3.0,1.0,-1.0}, //left center
 {-3.0,-1.0,1.0},
 {-1.0,-1.0,1.0},
 {-1.0,1.0,1.0},
 {-3.0,1.0,1.0},

 {1.0,-1.0,-1.0},
 {3.0,-1.0,-1.0},
 {3.0,1.0,-1.0},
 {1.0,1.0,-1.0}, // right center
 {1.0,-1.0,1.0},
 {3.0,-1.0,1.0},
 {3.0,1.0,1.0},
 {1.0,1.0,1.0},
 GLfloat color[][3]={{1.0,1.0,1.0}, //white
 {1.0,0.5,0.0}, //orange
 {0.0,0.0,1.0}, //blue
 {0.0,1.0,0.0}, //green
 {1.0,1.0,0.0}, //yellow
 {1.0,0.0,0.0}, //red
{0.5,0.5,0.5}, //grey used to represent faces of cube without colour
{.6,.5,.6}//speed meter colour
};
void polygon(int a,int b,int c,int d,int e)
{
glColor3f(0,0,0);
glLineWidth(3.0);
glBegin(GL_LINE_LOOP);
glVertex3fv(vertices[b]);
glVertex3fv(vertices[c]);
glVertex3fv(vertices[d]);
glVertex3fv(vertices[e]);
glEnd();
glColor3fv(color[a]);
glBegin(GL_POLYGON);
glVertex3fv(vertices[b]);
glVertex3fv(vertices[c]);
glVertex3fv(vertices[d]);
glVertex3fv(vertices[e]);
glEnd();
}
void colorcube1()
{
polygon(6,0,3,2,1);
polygon(6,2,3,7,6);
polygon(6,0,4,7,3); // center piece
 polygon(6,1,2,6,5);
polygon(6,4,5,6,7);
polygon(6,0,1,5,4);
}
void colorcube2()
```

```
{
polygon(6,8,11,10,9);
polygon(6,10,11,15,14);
polygon(6,8,12,15,11); // bottom center
 polygon(6,9,10,14,13);
polygon(6,12,13,14,15);
polygon(bottom[1][1],8,9,13,12);
}
void colorcube3()
{
polygon(6,16,19,18,17);
polygon(6,18,19,23,22);
polygon(left[1][1],16,20,23,19); // left center
 polygon(6,17,18,22,21);
polygon(6,20,21,22,23);
polygon(6,16,17,21,20);
}
void colorcube4()
{
polygon(6,24,27,26,25);
polygon(6,26,27,31,30);
polygon(6,24,28,31,27); // right center
 polygon(right[1][1],25,26,30,29);
polygon(6,28,29,30,31);
polygon(6,24,25,29,28);
}
void colorcube5()
{
polygon(6,32,35,34,33);
polygon(top[1][1],34,35,39,38);
polygon(6,32,36,39,35); // top center
 polygon(6,33,34,38,37);
polygon(6,36,37,38,39);
polygon(6,32,33,37,36);
}
void colorcube6()
{
polygon(6,40,43,42,41);
polygon(6,42,43,47,46);
polygon(6,40,44,47,43); // front center
```

```
 polygon(6,41,42,46,45);
polygon(front[1][1],44,45,46,47);
polygon(6,40,41,45,44);
}
void colorcube7()
{
polygon(back[1][1],48,51,50,49);
polygon(6,50,51,55,54);
polygon(6,48,52,55,51); //back center
 polygon(6,49,50,54,53);
polygon(6,52,53,54,55);
polygon(6,48,49,53,52);
}
void colorcube8()
{
polygon(6,56,59,58,57);
polygon(top[1][0],58,59,63,62);
polygon(left[0][1],56,60,63,59); // top left center
 polygon(6,57,58,62,61);
polygon(6,60,61,62,63);
polygon(6,56,57,61,60);
}
void colorcube9()
{
polygon(6,64,67,66,65);
polygon(top[1][2],66,67,71,70);
polygon(6,64,68,71,67); // top right center
 polygon(right[0][1],65,66,70,69);
polygon(6,68,69,70,71);
polygon(6,64,65,69,68);
}
void colorcube10()
{
polygon(6,72,75,74,73);
polygon(top[2][1],74,75,79,78);
polygon(6,72,76,79,75); // top front center
 polygon(6,73,74,78,77);
polygon(front[0][1],76,77,78,79);
polygon(6,72,73,77,76);
}
```

```
void colorcube11()
{
polygon(back[0][1],80,83,82,81);
polygon(top[0][1],82,83,87,86);
polygon(6,80,84,87,83); // top back center
 polygon(6,81,82,86,85);
polygon(6,84,85,86,87);
polygon(6,80,81,85,84);
}
void colorcube12()
{
polygon(6,80+8,83+8,82+8,81+8);
polygon(6,82+8,83+8,87+8,86+8);
polygon(left[2][1],80+8,84+8,87+8,83+8); // bottom left center
 polygon(6,81+8,82+8,86+8,85+8);
polygon(6,84+8,85+8,86+8,87+8);
polygon(bottom[1][0],80+8,81+8,85+8,84+8);
}
void colorcube13()
{
polygon(6,80+16,83+16,82+16,81+16);
polygon(6,82+16,83+16,87+16,86+16);
polygon(6,80+16,84+16,87+16,83+16); // bottom right center
 polygon(right[2][1],81+16,82+16,86+16,85+16);
polygon(6,84+16,85+16,86+16,87+16);
polygon(bottom[1][2],80+16,81+16,85+16,84+16);
}
void colorcube14()
{
polygon(6,80+24,83+24,82+24,81+24);
polygon(6,82+24,83+24,87+24,86+24);
polygon(6,80+24,84+24,87+24,83+24); // bottom front center
 polygon(6,81+24,82+24,86+24,85+24);
polygon(front[2][1],84+24,85+24,86+24,87+24);
polygon(bottom[0][1],80+24,81+24,85+24,84+24);
}
void colorcube15()
{
polygon(back[2][1],112,115,114,113);
polygon(6,114,115,119,118);
polygon(6,112,116,119,115); // bottom back center
 polygon(6,113,114,118,117);
polygon(6,116,117,118,119);
polygon(bottom[2][1],112,113,117,116);
}
void colorcube16()
{
polygon(back[0][2],120,123,122,121);
polygon(top[0][0],122,123,127,126);
polygon(left[0][0],120,124,127,123); // top left back
 polygon(6,121,122,126,125);
polygon(6,124,125,126,127);
polygon(6,120,121,125,124);
}
void colorcube17()
{
polygon(6,128,131,130,129);
polygon(top[2][0],130,131,135,134);
polygon(left[0][2],128,132,135,131); // top left front
 polygon(6,129,130,134,133);
polygon(front[0][0],132,133,134,135);
polygon(6,128,129,133,132);
}
void colorcube18()
{
polygon(back[0][0],136,139,138,137);
polygon(top[0][2],138,139,143,142);
polygon(6,136,140,143,139); // top right back
 polygon(right[0][2],137,138,142,141);
polygon(6,140,141,142,143);
polygon(6,136,137,141,140);
}
void colorcube19()
{
polygon(6,144,147,146,145);
polygon(top[2][2],146,147,151,150);
polygon(6,144,148,151,147); // top right front
 polygon(right[0][0],145,146,150,149);
polygon(front[0][2],148,149,150,151);
polygon(6,144,145,149,148);
```

```
}
void colorcube20()
{
polygon(back[1][2],152,155,154,153);
polygon(6,154,155,159,158);
polygon(left[1][0],152,156,159,155); //center left back
 polygon(6,153,154,158,157);
polygon(6,156,157,158,159);
polygon(6,152,153,157,156);
}
void colorcube21()
{
polygon(6,160,163,162,161);
polygon(6,162,163,167,166);
polygon(left[1][2],160,164,167,163); // center left front
 polygon(6,161,162,166,165);
polygon(front[1][0],164,165,166,167);
polygon(6,160,161,165,164);
}
void colorcube22()
{
polygon(back[1][0],168,171,170,169);
polygon(6,170,171,175,174);
polygon(6,168,172,175,171); // center right back
 polygon(right[1][2],169,170,174,173);
polygon(6,172,173,174,175);
polygon(6,168,169,173,172);
}
void colorcube23()
{
polygon(6,176,179,178,177);
polygon(6,178,179,183,182);
polygon(6,176,180,183,179); //center right front
 polygon(right[1][0],177,178,182,181);
polygon(front[1][2],180,181,182,183);
polygon(6,176,177,181,180);
}
void colorcube24()
{
polygon(back[2][2],184,187,186,185);
polygon(6,186,187,191,190);
polygon(left[2][0],184,188,191,187); // bottom left back
 polygon(6,185,186,190,189);
polygon(6,188,189,190,191);
polygon(bottom[2][0],184,185,189,188);
}
void colorcube25()
{
polygon(6,192,195,194,193);
polygon(6,194,195,199,198);
polygon(left[2][2],192,196,199,195); // bottom left front
 polygon(6,193,194,198,197);
polygon(front[2][0],196,197,198,199);
polygon(bottom[0][0],192,193,197,196);
}
void speedmeter()
{
 glColor3fv(color[7]);
glBegin(GL_POLYGON);
glVertex3f(0.0,7.2,0.0);
glVertex3f(1.0,7.0,0.0);
glVertex3f(1.0,7.5,0.0);
glEnd();
glPushMatrix();
 glTranslatef(1.0,0.0,0.0);
polygon(speedmetercolor[0],216,217,218,219);
glPopMatrix();
 glPushMatrix();
 glTranslatef(1.5,0.0,0.0);
polygon(speedmetercolor[1],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(2.0,0.0,0.0);
polygon(speedmetercolor[2],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(2.5,0.0,0.0);
polygon(speedmetercolor[3],216,217,218,219);
glPopMatrix();
glPushMatrix();
```

```
 glTranslatef(3.0,0.0,0.0);
polygon(speedmetercolor[4],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(3.5,0.0,0.0);
polygon(speedmetercolor[5],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(4.0,0.0,0.0);
polygon(speedmetercolor[6],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(4.5,0.0,0.0);
polygon(speedmetercolor[7],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(5.0,0.0,0.0);
polygon(speedmetercolor[8],216,217,218,219);
glPopMatrix();
glPushMatrix();
 glTranslatef(5.5,0.0,0.0);
polygon(speedmetercolor[9],216,217,218,219);
glPopMatrix();
glPushMatrix();
 }
void display()
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
speedmeter();
glColor3fv(color[0]);
output(0,8,message);
glPushMatrix();
 glRotatef(25.0+p,1.0,0.0,0.0);
 glRotatef(-30.0+q,0.0,1.0,0.0);
 glRotatef(0.0+r,0.0,0.0,1.0);
if(rotation==0)
{
colorcube1();
colorcube2();
```

```
colorcube3();
colorcube4();
colorcube5();
colorcube6();
colorcube7();
colorcube8();
colorcube9();
colorcube10();
colorcube11();
colorcube12();
colorcube13();
colorcube14();
colorcube15();
colorcube16();
colorcube17();
colorcube18();
colorcube19();
colorcube20();
colorcube21();
colorcube22();
colorcube23();
colorcube24();
colorcube25();
colorcube26();
colorcube27();
}
if(rotation==1)
{
colorcube1();
colorcube2();
colorcube3();
colorcube4();
colorcube6();
colorcube7();
colorcube12();
colorcube13();
colorcube14();
colorcube15();
colorcube20();
colorcube21();
```

```
colorcube22();
colorcube23();
colorcube24();
colorcube25();
colorcube26();
colorcube27();
if(inverse==0)
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Top");
glPopMatrix();
glRotatef(-theta,0.0,1.0,0.0);
}
else
{glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"TopInverted");
glPopMatrix();
glRotatef(theta,0.0,1.0,0.0);
}
colorcube5();
colorcube8();
colorcube9();
colorcube10();
colorcube11();
colorcube16();
colorcube17();
colorcube18();
colorcube19();
}
if(rotation==2)
{
colorcube1();
colorcube2();
colorcube3();
colorcube5();
colorcube6();
colorcube7();
colorcube8();
```

```
colorcube10();
colorcube11();
colorcube12();
colorcube14();
colorcube15();
colorcube16();
colorcube17();
colorcube20();
colorcube21();
colorcube24();
colorcube25();
if(inverse==0)
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Right");
glPopMatrix();
glRotatef(-theta,1.0,0.0,0.0);
}
else
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"RightInverted");
glPopMatrix();
glRotatef(theta,1.0,0.0,0.0);
}
}
if(rotation==3)
{
colorcube1();
colorcube2();
colorcube3();
colorcube4();
colorcube5();
colorcube7();
colorcube8();
colorcube9();
colorcube11();
colorcube12();
```

```
colorcube13();
colorcube15();
colorcube16();
colorcube18();
colorcube20();
colorcube22();
colorcube24();
colorcube26();
if(inverse==0)
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Front");
glPopMatrix();
glRotatef(-theta,0.0,0.0,1.0);
}
else
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"FrontInverted");
glPopMatrix();
glRotatef(theta,0.0,0.0,1.0);
}
colorcube6();
colorcube10();
colorcube14();
colorcube17();
colorcube19();
colorcube21();
colorcube23();
colorcube25();
colorcube27();
}
if(rotation==4)
{
colorcube1();
colorcube2();
colorcube4();
colorcube5();
```

```
colorcube6();
colorcube7();
colorcube9();
colorcube10();
colorcube11();
colorcube13();
colorcube14();
colorcube15();
colorcube18();
colorcube19();
colorcube22();
colorcube23();
colorcube26();
colorcube27();
if(inverse==0)
{glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Left");
glPopMatrix();
glRotatef(theta,1.0,0.0,0.0);
}
else
{glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"LeftInverted");
 glPopMatrix();
glRotatef(-theta,1.0,0.0,0.0);
}
colorcube3();
colorcube8();
colorcube12();
colorcube16();
colorcube17();
colorcube20();
colorcube21();
colorcube24();
colorcube25();
}
if(rotation==5)
{
```

```
colorcube1();
colorcube2();
colorcube3();
colorcube4();
colorcube5();
colorcube6();
colorcube8();
colorcube9();
colorcube10();
colorcube12();
colorcube13();
colorcube14();
colorcube17();
colorcube19();
colorcube21();
colorcube23();
colorcube25();
colorcube27();
if(inverse==0)
{glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Back");
glPopMatrix();
glRotatef(theta,0.0,0.0,1.0);
}
else
{
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"BackInverted");
glPopMatrix();
glRotatef(-theta,0.0,0.0,1.0);
}
}
if(rotation==6)
{
colorcube1();
colorcube3();
colorcube4();
colorcube5();
```

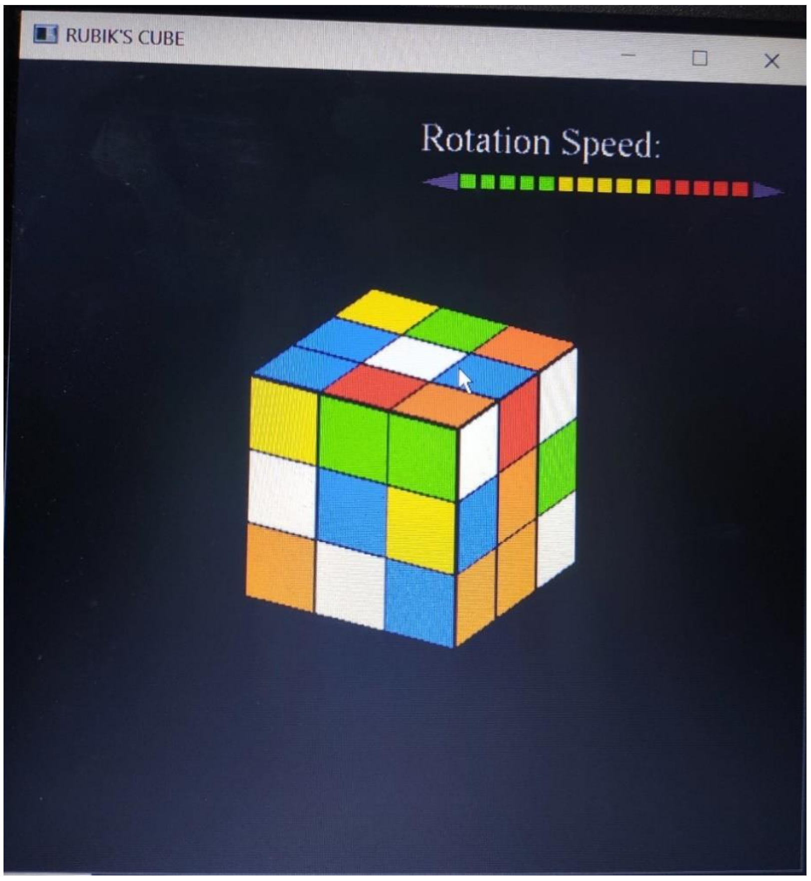
```
colorcube6();
colorcube7();
colorcube8();
colorcube9();
colorcube10();
colorcube11();
colorcube16();
colorcube17();
colorcube18();
colorcube19();
colorcube20();
colorcube21();
colorcube22();
colorcube23();

int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize (500, 500);
glutCreateWindow ("RUBIK'S CUBE");
glutReshapeFunc (myreshape);
glutIdleFunc(spincube);
glutMouseFunc(mouse);
 glutMotionFunc(motion);
glutCreateMenu(mymenu);
glutAddMenuEntry("Top :a",1);
glutAddMenuEntry("Top Inverted :q",2);
glutAddMenuEntry("Right :s",3);
glutAddMenuEntry("Right Inverted :w",4);
glutAddMenuEntry("Front :d",5);
glutAddMenuEntry("Front Inverted :e",6);
glutAddMenuEntry("Left :f",7);
glutAddMenuEntry("Left Inverted :r",8);
glutAddMenuEntry("Back :g",9);
glutAddMenuEntry("Back Inverted :t",10);
glutAddMenuEntry("Bottom :h",11);
glutAddMenuEntry("Bottom Inverted :y",12);
glutAddMenuEntry("Exit",13);
glutDisplayFunc (display);
glEnable(GL_DEPTH_TEST);
glutMainLoop();
//return 0;
}
```

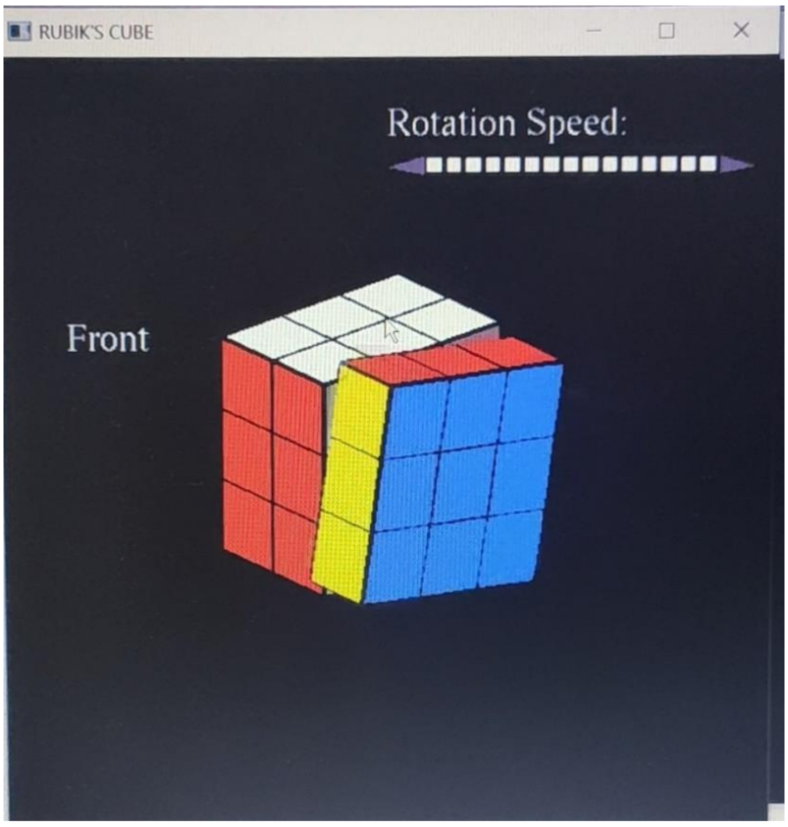
Commonly used Functions of OpenGL includes:

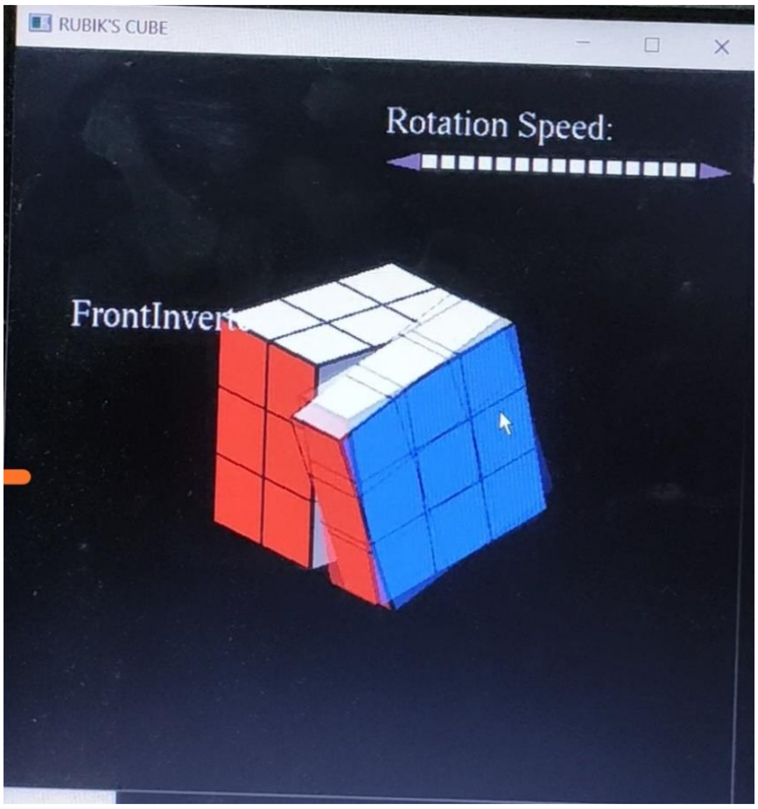
glBegin, glEnd	The glBegin and glEnd functions delimit the vertices of a primitive or a group of like primitives.
glClear	The glClear function clears buffers to preset values.
glColor	These functions set the current color
glFlush	The glFlush function forces execution of OpenGL functions in finite time.
glLoadIdentity	The glLoadIdentity function replaces the current matrix with the identity matrix.
glMatrixMode	The glMatrix Mode function specifies which matrix is the current matrix.
glVertex	These functions specify a vertex.
gluOrtho2D	The gluOrtho2D function defines a 2-D Orthographic projection matrix .

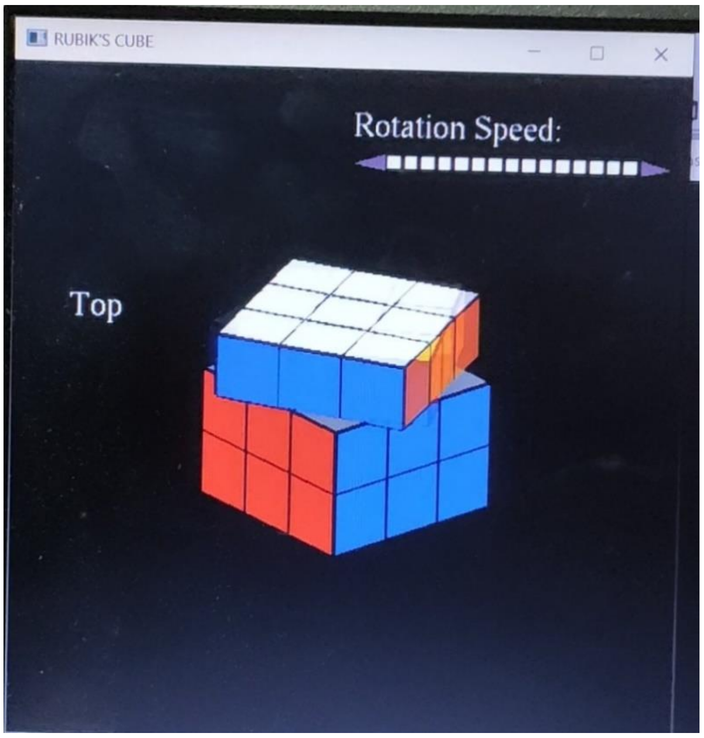
CHAPTER 4
RESULTS AND SNAPSHOTS

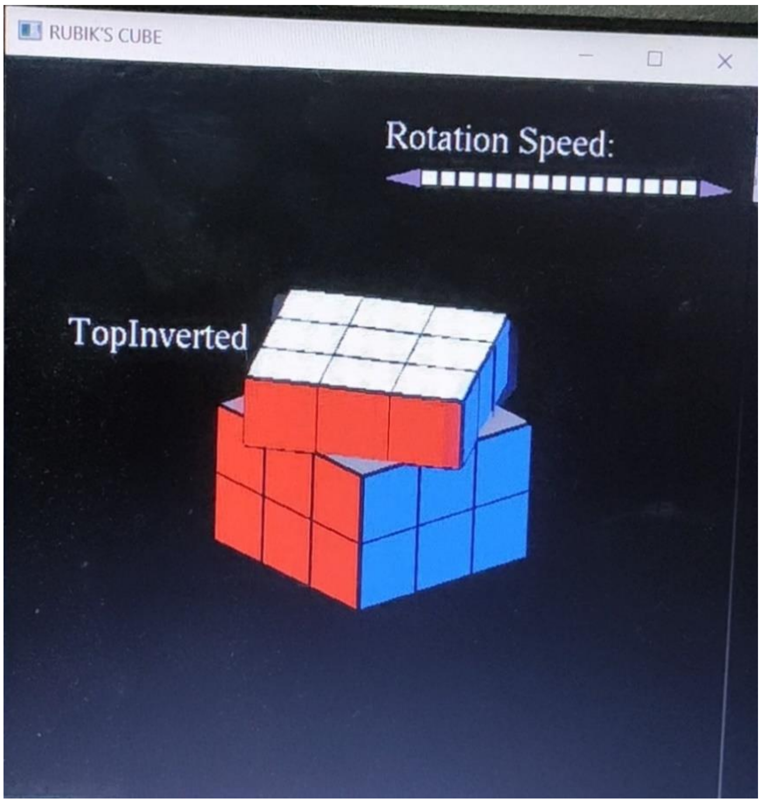
Snapshots:

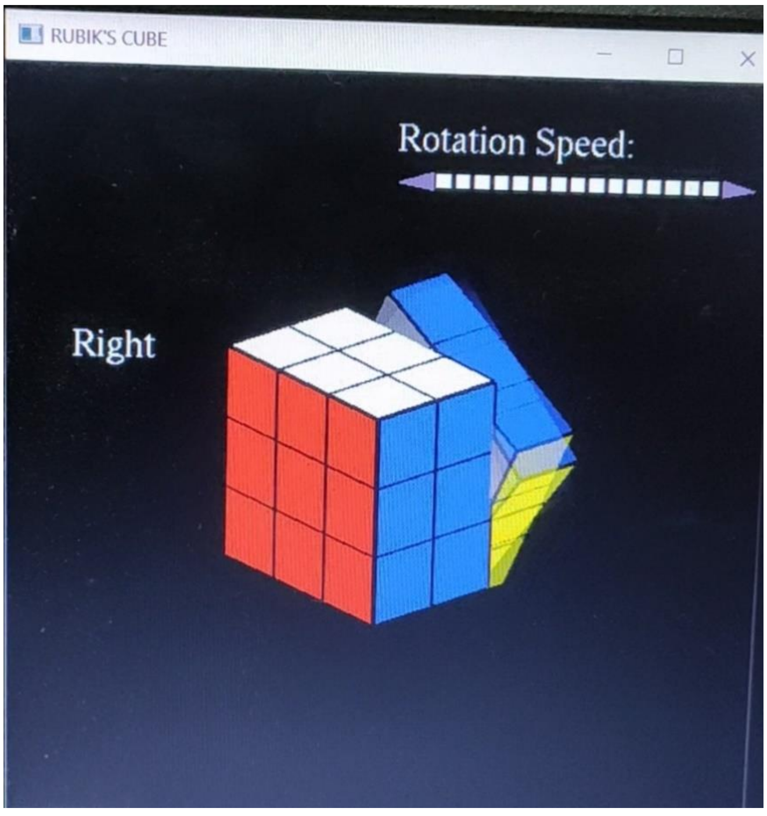

RUBIK'S CUBE
Rotation Speed:

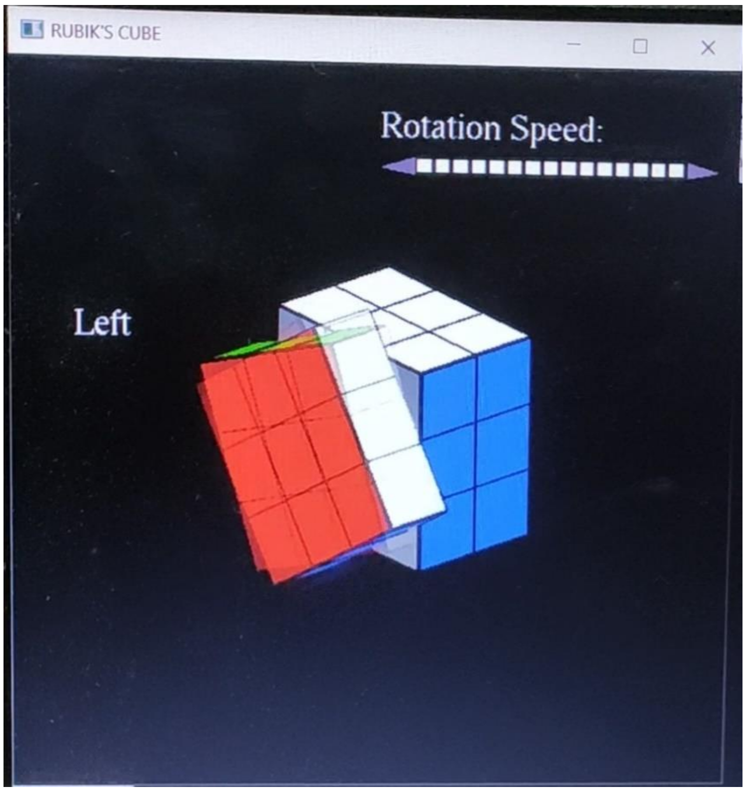

RUBIK'S CUBE
Rotation Speed:
Front

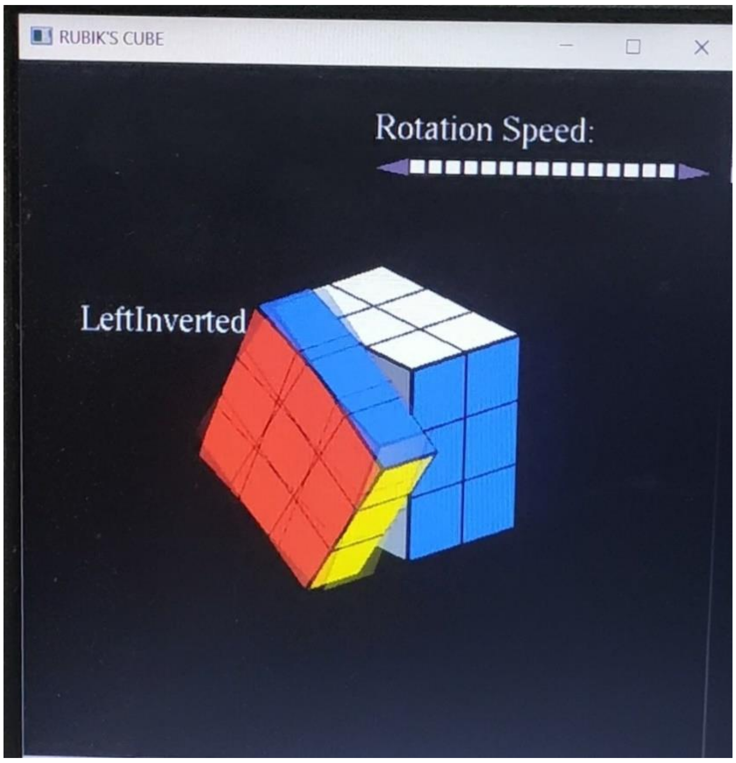

RUBIK'S CUBE
Rotation Speed:
FrontInvert

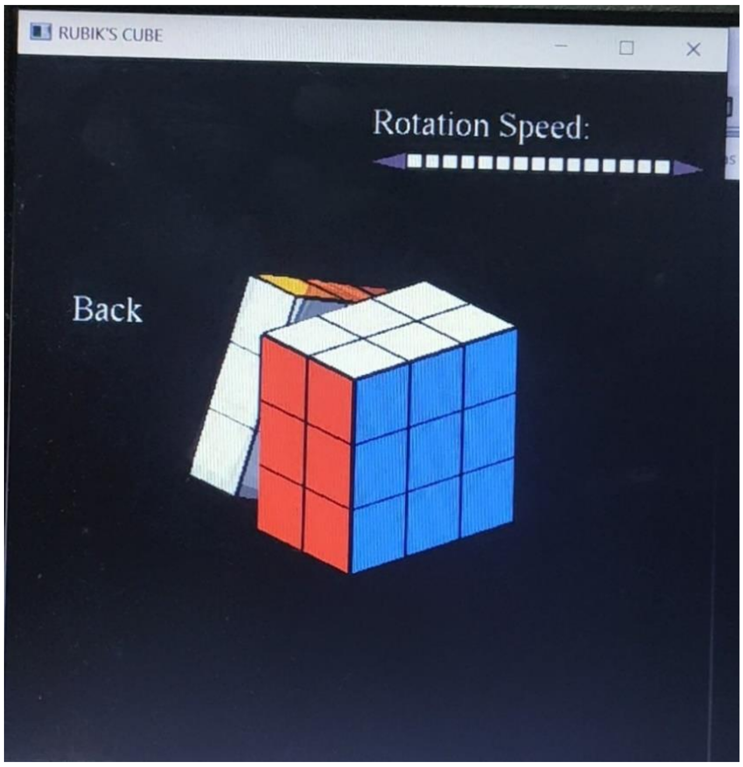

RUBIK'S CUBE
Rotation Speed:
Top

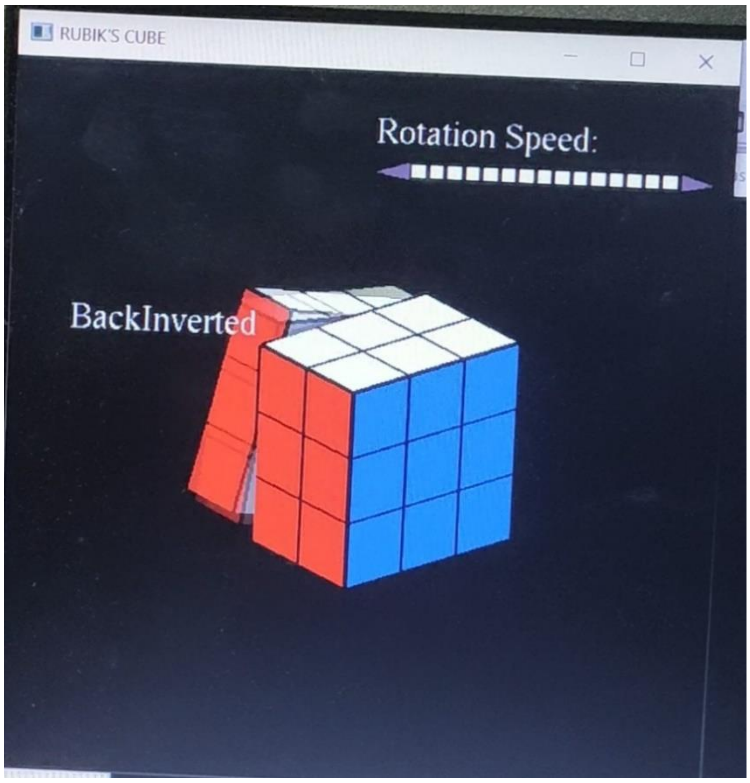

RUBIK'S CUBE
Rotation Speed:
TopInverted

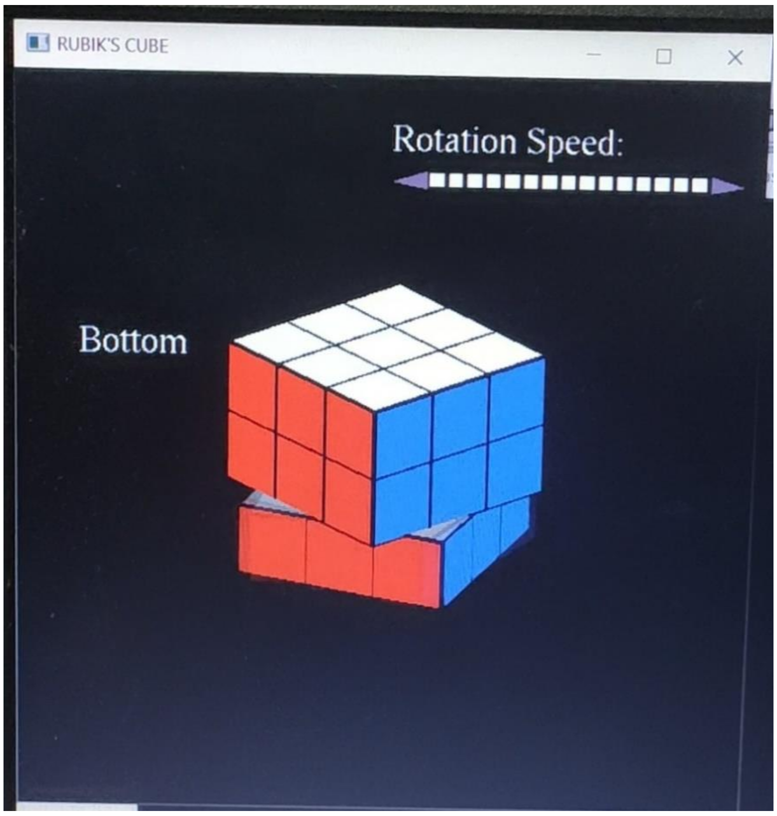

RUBIK'S CUBE
Rotation Speed:
Right

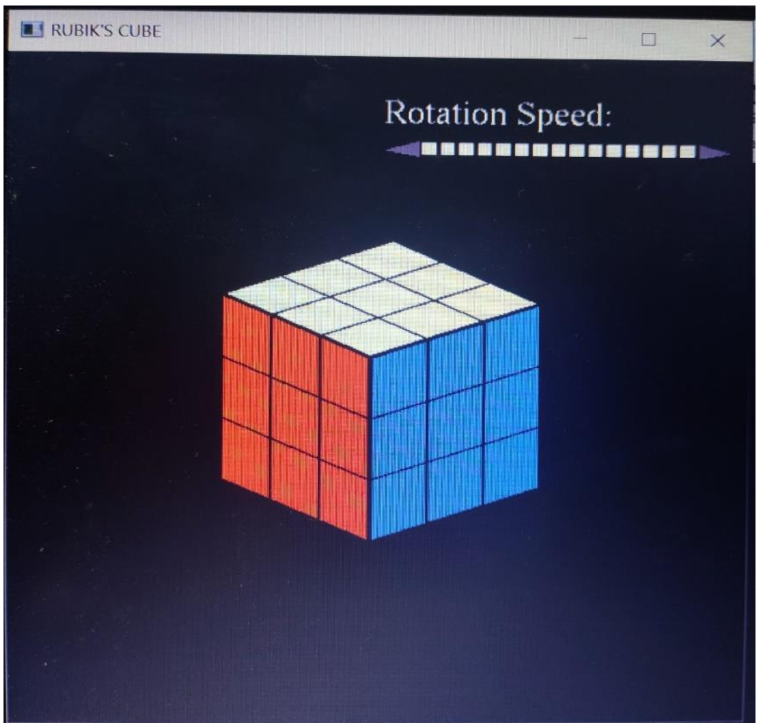

RUBIK'S CUBE
Rotation Speed:
Left


RUBIK'S CUBE
Rotation Speed:
LeftInverted


RUBIK'S CUBE
Rotation Speed:
Back


RUBIK'S CUBE
Rotation Speed:
BackInverted


RUBIK'S CUBE
Rotation Speed:
Bottom


RUBIK'S CUBE
Rotation Speed:

CHAPTER 5
CONCLUSION

CONCLUSION:

We successfully implemented the Rubicks cube game with different frames and speedmeter.

REFERENCES

[1] The OpenGL Utility Toolkit (GLUT) documentation
https://www.opengl.org/resources/libraries/glut/spec3/spec3.html/