Chapter 1

INTRODUCTION

1.1 What is OpenGl?

OpenGL is a software interface to graphics hardware. This interface consists of about150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of *geometric primitives* - points, lines, and polygons.

1.2 What is GLUT?

GLUT is a complete API written by Mark Kilgard which lets you create windows and handle the messages. It exists for several platforms, that means that a program which uses GLUT can be compiled on many platforms without (or at least with very few) changes in the code.

1.3 How does OpenGL work?

OpenGL bases on the state variables. There are many values, for example the color, that remain after being specified. That means, you can specify a color once and draw several polygons, lines or whatever with this color then. There are no classes like in DirectX. However, it is logically structured. Before we come to the commands themselves, here is another thing:

To be hardware independent, OpenGL provides its own data types. They all begin with "GL". For example, GLfloat, GLint and so on. There are also many symbolic constants, they all begin with "GL_", like GL_POINTS, GL_POLYGON. Finally the commands have

the prefix "gl" like glVertex3f(). There is a utility library called GLU, here the prefixes are "GLU_" and "glu". GLUT commands begin with "glut", it is the same for every library. You want to know which libraries coexist with the ones called before? There are libraries for every system, Windows has the wgl*-Functions, Unix systems glx* and so on.

A very important thing is to know that there are two important matrices, which affect the transformation from the 3d-world to the 2d-screen: The projection matrix and the modelview matrix. The projection matrix contains information, such as a vertex – let's say a "point" in space – that is mapped to the screen. This contains, whether the projection shall be isometric or from a perspective, how wide the field of view is and so on. Into the other matrix you put information, how the objects are moved, where the viewer is and so on.

Don't like matrices? Don't be afraid, you probably won't really see them, at least at the beginning. There are commands that do all the maths for you.

Some basic commands are explained later in this tutorial.

1.4 How can I use GLUT?

GLUT provides some routines for the initialization and creating the window (or fullscreen mode, if you want to). Those functions are called first in a GLUT application:

In your first line you always write glutInit(&argc, argv). After this, you must tell GLUT which display mode you want – single or double buffering, color index mode or RGB and so on. This is done by calling glutInitDisplayMode(). The symbolic constants are connected by a logical OR, so you could use glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE). In later tutorials we will use some more constants here.

After the initialization you call glCreateWindow() with the window name as parameter.Then you can (and should) pass some methods for certain events. The most important ones are "reshape" and "display". In reshape you need to (re)define the field of view and specify a new area in the window, where OpenGL is allowed to draw to.Display should clear the so-called color buffer – let's say this is the sheet of paper – and draw our objects.You pass the methods by glut*Func(), for example glutDisplayFunc(). At the end of the main function you call glutMainLoop(). This function doesn't return, but calls the several functions passed by glut*Func.

1.5 Opengl Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in Figure 1-2, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

If you are new to three-dimensional graphics, the upcoming description may seem like drinking water out of a fire hose. You can skim this now, but come back to Figure 1-2 as you go through each chapter in this book.

The following diagram shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

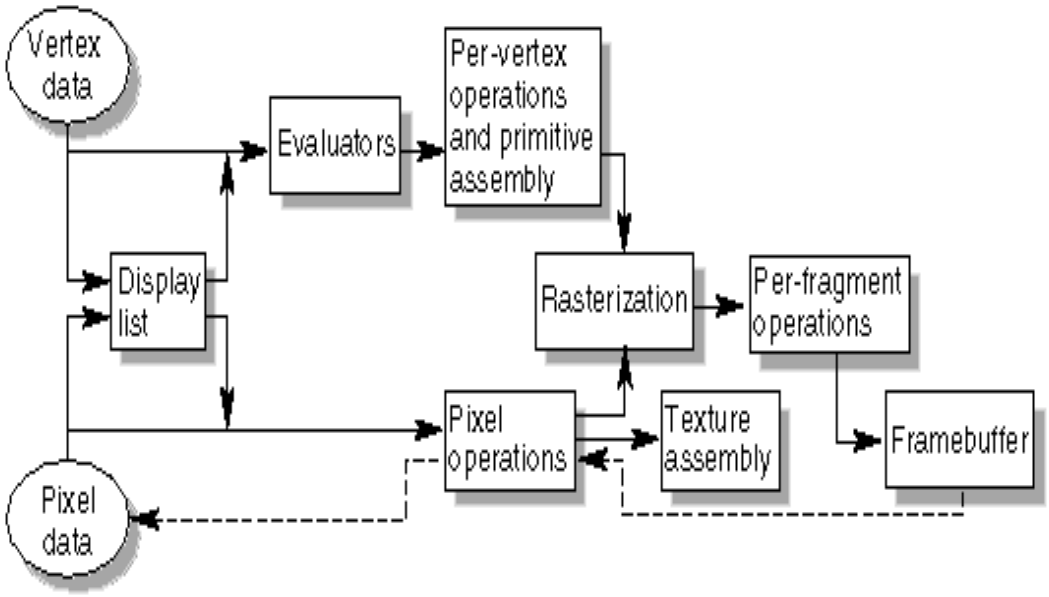

Figure 1.1**:** Order of Operations

Chapter 2

SYSTEM REQUIREMENTS

The minimum hardware and software requirements for this Rubik's-Cube Game are specified below.

2.1 Hardware Requirements

Minimum hardware specification

Processor : Intel/AMD processor

Main memory : 512 MB RAM

Hard Disk : 40 MB

2.2 Software Requirements

Minimum software specification

Operating system : Windows

Software Used : Microsoft visual 8.0 using C++ (or) CodeBlocks 17.12

Library Used : OPENGL Library

Chapter 3

Analysis and Design

3.1 Detailed Design

#include<GL/glut.h> #include<stdio.h> #include<math.h>

#include<windows.h>

- GL/glut.h: This header file provides an interface with application programs that are installed on the system.
- Stdio.h: Input & Output operations can also be performed in C using C Standard input,output library.
- Math.h: contains constant definitions and external subordinate declarations for the math subroutine library.
- Windows h: handles the output window creation.

3.2 Application Design

Application design for graphics is the TC++. This system will support only the 2D graphics. The 2D graphics package being designed should be easy to use and understand.

It should provide various options such as free hand drawing, line drawing, polygon drawing, filled polygons, flood fill, translation, rotation, scaling, clipping etc.

Even though these properties were supported, it was difficult to render 2D graphics. It wasn't very difficult to get a 3 Dimensional object.Even the effects like lighting, shading cannot be provided. So we go for code blocks

OpenGL provides a powerful but primitive set of rendering commands, and all higher level drawing must be done in terms of these commands. There are several libraries that allow you to simplify your programming tasks, including the following:

OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections and rendering surfaces.

3.3 Flowchart

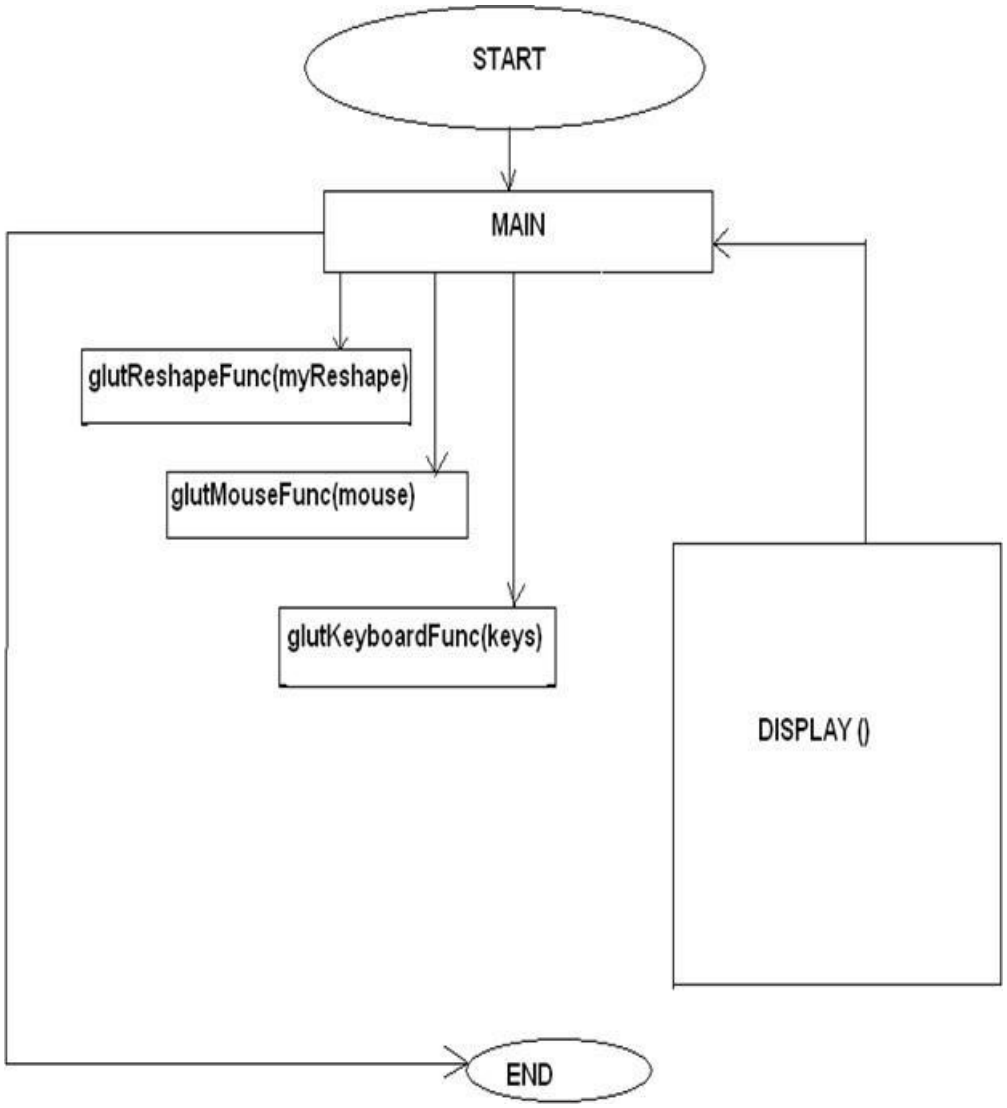

Figure 3.1 : Flow Chart Rubik's-Cube

The above flow diagram illustrates the working of the game,where each step of the game is described in a number of steps starting from the main function .

Chapter 4

IMPLEMENTATION

4.1 Header Files

The <stdlib.h> Header File

include <stdlib.h>

The ISO C standard introduced this header file as a place to declare certain standard library functions. These include the Memory management functions (malloc, free, et. al.) communication with the environment (abort, exit) and others. Not yet all the standard functions of this header file are supported. If a declaration is present in the supplied header file, then uCR supports it and will continue to support it. If a function is not there, it will be added in time.

The <stdio.h> Header File

include <stdio.h>

Load the file SIMPLEIO.C for our first look at a file with standard I/O. Standard I/O refers to the most usual places where data is either read from, the keyboard, or written to, the video monitor. Since they are used so much, they are used as the default I/O devices and do not need to be named in the Input/Output instructions. This will make more sense when we actually start to use them so let's look at the file in front of you.

The first thing you will notice is the second line of the file, the #include the "stdio.h" line. This is very much like the #define we have already studied, except that instead of a simple substitution, an entire file is read in at this point. The system will find the file named "stdio.h" and read its entire contents in, replacing this statement. Obviously then, the file named "stdio.h" must contain valid C source statements that can be compiled as part of a program. This particular file is composed of several standard #defines to define some of the standard I/O operations. The file is called a header file and you will find several different header files on the source disks that came with your C compiler. Each of the header files has a specific purpose and any or all of them can be included in any program.

4.2 FUNCTIONS

4.2.1 glColor3f Function

Set the current color.

SYNTAX:

void glColor3f(GLfloat red, GLfloat green, GLfloat blue);

PARAMETERS:

red

The new red value for the current color.

green

The new green value for the current color.

4.2.5 glBegin, glEnd Function

The glBegin and glEnd functions delimit the vertices of a primitive or a group of like primitives.

SYNTAX:

void glBegin, glEnd(GLenum mode);

PARAMETERS:

mode

The primitive or primitives that will be created from vertices presented between glBegin and the subsequent glEnd. The following are accepted symbolic constants and their meanings:

GL_LINES Treats each pair of vertices as an independent line segment.

Vertices 2n - 1 and 2n define line n. N/2 lines are drawn.

GL_LINE_STRIP Draws a connected group of line segments from the first vertex to the last. Vertices n and n+1 define line n. N - 1 lines are drawn.

GL_LINE_LOOP Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n + 1 define line n. The last line, however, is defined by vertices N and N lines are drawn.

GL_TRIANGLES Treats each triplet of vertices as an independent triangle. Vertices 3n - 2, 3n - 1, and 3n define triangle n. N/3 triangles are drawn.

GL_QUADS Treats each group of four vertices as an independent quadrilateral. Vertices 4n - 3, 4n - 2, 4n - 1, and 4n defined quadrilateral n. N/4 quadrilaterals are drawn.

4.2.2 glNormal3f Function

Sets the current normal vector.

SYNTAX:

void glNormal3b(GLfloat nx, GLfloat ny, GLfloat nz);

PARAMETERS:

nx

Specifies the x-coordinate for the new current normal vector.

ny

Specifies the y-coordinate for the new current normal vector.

nz

Specifies the z-coordinate for the new current normal vector.

glNormal3f(1.0,0.0,0.0);

4.2.3 glVertex3f Function

Specifies a vertex.

SYNTAX:

void glVertex3f(GLfloat x, GLfloat y, GLfloat z);

PARAMETERS:

x

Specifies the x-coordinate of a vertex.

y

Specifies the y-coordinate of a vertex.

z

Specifies the z-coordinate of a vertex.

glVertex3f(-3.0,3.0,4.0);

4.2.4 glTranslate Function

The glTranslate and glTranslatef functions multiply the current matrix by a translation matrix.

SYNTAX:

void glTranslate(x, y, z);

PARAMETERS:

x, y, z

The x, y, and z coordinates of a translation vector.

glTranslatef(0.0,0.0,-1.0);

4.3 Functions Used to Display

4.3.1 glClear Function

The glClear function clears buffers to preset values.

SYNTAX:

glClear(GLbitfield mask);

PARAMETERS:

Bitwise OR operators of masks that indicate the buffers to be cleared. The four masks are as follows.

Value	Meaning
GL_COLOR_BUFFER_BIT	The buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT	The depth buffer.

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

4.3.2. glMatrixMode Function

The glMatrixMode function specifies which matrix is the current matrix.

SYNTAX:

void glMatrixMode(GLenum mode);

PARAMETERS:

mode

The matrix stack that is the target for subsequent matrix operations. The mode parameter can assume one of three values:

Value	Meaning
GL_MODELVIEW	Applies subsequent matrix operations to the modelview matrix stack.

glMatrixMode(GL_MODELVIEW);

4.3.3 glLoadIdentity Function

The glLoadIdentity function replaces the current matrix with the identity matrix.

SYNTAX:

void glLoadIdentity(void);

PARAMETERS:

This function has no parameters.

glLoadIdentity();

glRotatef Function

glPushMatrix, glPopMatrix Function

glScalef Function

glTranslate Function

glClear Function

glPushAttrib, glPopAttrib

Pushes or pops the attribute stack.

Name	Meaning
glPushAttrib	Pushes the attribute stack.
glPopAttrib	Pops the attribute stack.

glPushAttrib(0xffffffff);

glPopAttrib();

glEnable, glDisable Function

4.3.4 glColor4f Function

Sets the current color.

SYNTAX:

void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

PARAMETERS:

red

The new red value for the current color.

green

The new green value for the current color.

blue

The new blue value for the current color.

alpha

The new alpha value for the current color.

glColor4f(0.0,0.0,0.0,0.05);

glBegin, glEnd Function

glVertex3f Function

4.3.5 glutSwapBuffers

glutSwapBuffers swaps the buffers of the current window if double buffered.

SYNTAX:

void glutSwapBuffers(void);

glutSwapBuffers();

4.4.1 glViewport Function

The glViewport function sets the viewport.

SYNTAX:

void glViewport(x, y,width, height);

PARAMETERS:

x, y

The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).

width, height

The width and height, respectively, of the viewport. When an OpenGL context is first attached to a window, width and height are set to the display.

glViewport(0,0,w,h);

glMatrixMode Function

glLoadIdentity Function

4.4.2.gluPerspective Function

set up a perspective projection matrix

SYNTAX:

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear,GLdoublezFar);

PARAMETERS:

fovy Specifies the field of view angle, in degrees, in the y direction.

aspect Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).

zNear Specifies the distance from the viewer to the near clipping plane (always positive).

zFar Specifies the distance from the viewer to the far clipping plane (always positive).

gluPerspective (50.0,(float)w/(float)h,1.0,20.0);

msecs Number of milliseconds to pass before calling the callback.

Func The timer callback function.

value

Integer value to pass to the timer callback.

glutTimerFunc(TIMER,timer,0);

glutDisplayFunc Function

glutDisplayFunc sets the display callback for the current window.

SYNTAX:

void glutDisplayFunc(void (*func)(void));

PARAMETERS:

func

The new display callback function.

glutDisplayFunc(display);

glutReshapeFunc Function

glutReshapeFunc sets the reshape callback for the current window.

SYNTAX:

void glutReshapeFunc(void (*func)(int width, int height));

PARAMETERS:

func

The new reshape callback function.

glutReshapeFunc(reshape);

4.5 Main Function

4.5.1. glutInit Function

glutInit is used to initialize the GLUT library.

SYNTAX:

glutInit(int *argcp, char **argv);

PARAMETERS:

argcp

A pointer to the program's unmodified argc variable from main. Upon return, the value pointed to by argcp will be updated, because glutInit extracts any command line options intended for the GLUT library.

Argv

The program's unmodified argv variable from main. Like argcp, the data for argv will be updated because glutInit extracts any command line options understood by the GLUT library.

glutInit(&argc,argv);

4.5.2. glutInitDisplayMode Function

glutInitDisplayMode sets the initial display mode.

SYNTAX:

void glutInitDisplayMode(unsigned int mode);

PARAMETERS:

mode

Display mode, normally the bitwise OR-ing of GLUT display mode bit masks. See values below:

GLUT_RGB: An alias for GLUT_RGBA.

GLUT_DOUBLE: Bit mask to select a double buffered window. This overrides GLUT_SINGLE if it is also specified.

GLUT_DEPTH: Bit mask to select a window with a depth buffer.

glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH|GLUT_DOUBLE);

4.5.3. glutInitWindowPosition, glutInitWindowSize Functions

glutInitWindowPosition and glutInitWindowSize set the initial window position and size respectively.

SYNTAX:

void glutInitWindowSize(int width, int height);

void glutInitWindowPosition(int x, int y);

PARAMETERS:

width

Width in pixels.

height

Height in pixels.

x

Window X location in pixels.

y

Window Y location in pixels.

glutInitWindowSize(300,300);

4.5.4. glutCreateWindow Function

glutCreateWindow creates a top-level window.

SYNTAX:

int glutCreateWindow(char *name);

PARAMETERS:

name

ASCII character string for use as window name.

glutCreateWindow("two pass mirror");

4.5.5. glutMainLoop Function

glutMainLoop enters the GLUT event processsing loop.

4.6 Sample code

```
#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},
{-1.0,1.0,-1.0},
{1.0,1.0,-1.0},
```

```
{1.0,3.0,-1.0},
{-1.0,3.0,-1.0}, // top center
{-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},
{1.0,1.0,1.0},
{-1.0,1.0,1.0}, //front center
{-1.0,-1.0,3.0},
{1.0,-1.0,3.0},
{1.0,1.0,3.0},
{-1.0,1.0,3.0},
{-1.0,-1.0,-3.0},
{1.0,-1.0,-3.0},
{1.0,1.0,-3.0},
{-1.0,1.0,-3.0}, //back 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},
{-3.0,1.0,-1.0},
{-1.0,1.0,-1.0},
{-1.0,3.0,-1.0},
{-3.0,3.0,-1.0}, // top left center
{-3.0,1.0,1.0},
{-1.0,1.0,1.0},
{-1.0,3.0,1.0},
{-3.0,3.0,1.0},
 {1.0,1.0,-1.0},
{3.0,1.0,-1.0},
{3.0,3.0,-1.0},
{1.0,3.0,-1.0}, // top right center
```

```
{1.0,1.0,1.0},
{3.0,1.0,1.0},
{3.0,3.0,1.0},
{1.0,3.0,1.0},
{-1.0,1.0,1.0},
{1.0,1.0,1.0},
{1.0,3.0,1.0},
{-1.0,3.0,1.0}, // top front center
{-1.0,1.0,3.0},
{1.0,1.0,3.0},
{1.0,3.0,3.0},
{-1.0,3.0,3.0},
 {-1.0,1.0,-3.0},
{1.0,1.0,-3.0},
{1.0,3.0,-3.0},
{-1.0,3.0,-3.0}, // top back center
{-1.0,1.0,-1.0},
{1.0,1.0,-1.0},
{1.0,3.0,-1.0},
{-1.0,3.0,-1.0},
 {-3.0,-3.0,-1.0},
{-1.0,-3.0,-1.0},
{-1.0,-1.0,-1.0},
{-3.0,-1.0,-1.0}, //bottom left center
{-3.0,-3.0,1.0},
{-1.0,-3.0,1.0},
{-1.0,-1.0,1.0}
{3.0,1.0,-1.0},
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
glPushMatrix();
glColor3fv(color[0]);
output(-11,6,"Right");
glPopMatrix();
glRotatef(-theta,1.0,0.0,0.0);
}
else
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);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutKeyboardFunc(keyboard);
glutDisplayFunc (display);
glEnable(GL_DEPTH_TEST);
glutMainLoop();
//return 0;
}
```

Chapter 5

RESULTS

5.1 Rubik's-Cube Game When it starts

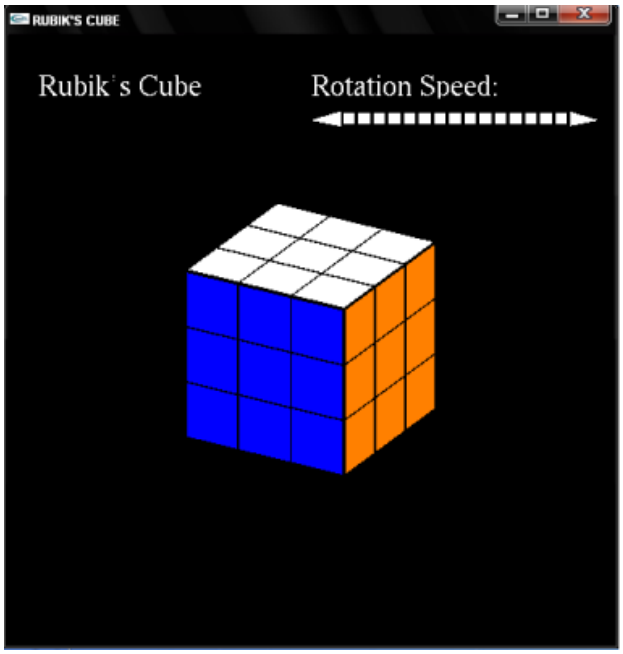

Figure 5.1 It displays the Screenshot of game when it starts

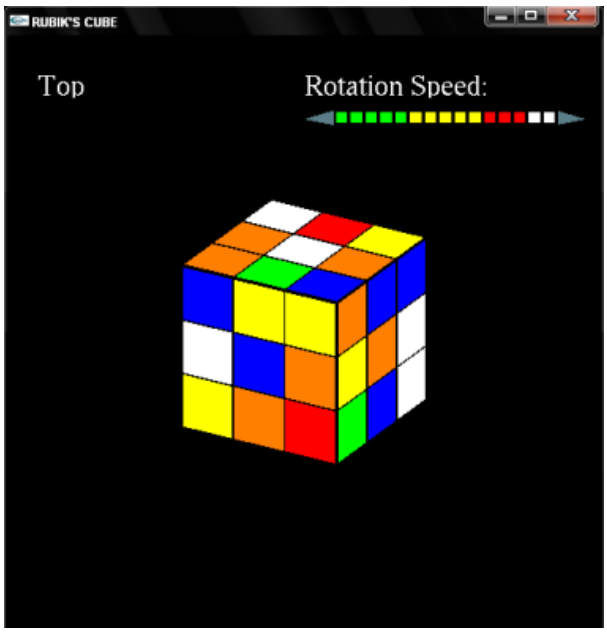

Figure 5.2 It displays the Screenshot of when you mix it

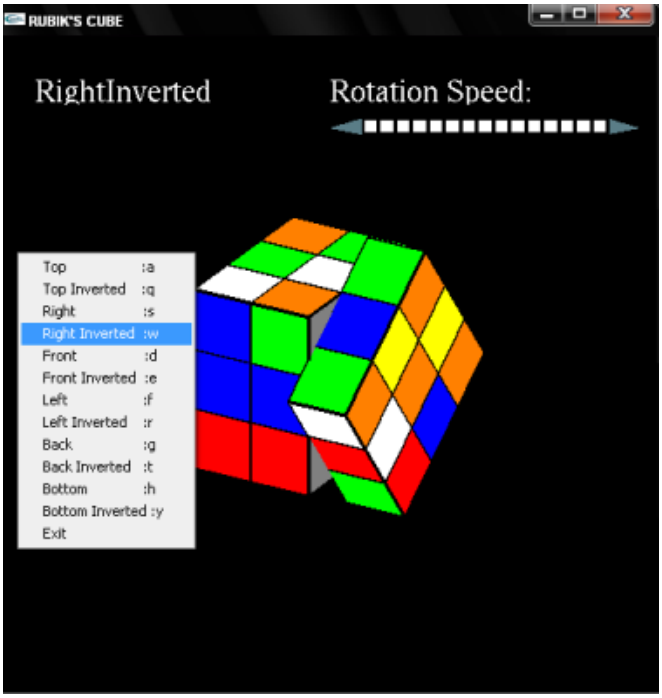

Figure 5.3 It displays the Screenshot of Keys

Chapter 6

CONCLUSION

SUMMARY:

The project '**Rubik-Cube Game' implemented** on Windows platform using C++ was completed successfully. It involves various translating, scaling and rotating effects effectively bringing out a simple animation. The utilities and functions provided by the OpenGL have been used in an optimal way to achieve a completely working project.

The work done during the process of completion of this project has helped me understand the various functionalities available in a better and a more practical environment and I realize that the scope of OpenGL platform has a premier game developing launch pad. Hence it has indeed been useful in developing many games. OpenGL in its own right is good for low cost and simple game development.

SCOPE OF IMPROVEMENT:

Though the package that is designed here does not include a complex OpenGL package, we intend to improve this by including extra features like lighting effects, changing the background color and adding alarms. We do this with an aim to attract more people to use our package.

REFERENCES

BOOK REFERENCES

[1] - Donald Hearn & Pauline Baker: Computer Graphics with OpenGL Version, 3rd / 4th Edition, Pearson Education, 2011.

[2] - Edward Angel: Interactive Computer Graphics- a Top Down approach with OpenGL, 5th edition. Pearson Education, 2008.

WEBSITE REFERENCES

[1] - https://www.khronos.org/opengl/wiki/Getting_Started

[2] - https://www.opengl.org/sdk/docs/tutorials/OGLSamples/

[3] - https://www.geeksforgeeks.org/getting-started-with-opengl/

[4] - https://www.khronos.org/opengl/wiki//Code_Resources

[5] - http://www.lighthouse3d.com/tutorials/glut-tutorial/

[6] - http://code-blocks.blogspot.in/2014/12/bresenhams-circle-drawing-algorithm.html

[7] - https://github.com/sprintr/opengl-examples/blob/master/OpenGL-Menu.cpp