ACS COLLEGE OF ENGINEERING

#207, Kambipura, Mysore road, Banglore-74

Department of Computer Science & Engineering

LAB MANUAL

Academic Year 2020-21 (Even Sem)

Computer Graphics Laboratory with Mini Project

(For VI Semester BE)

SUBJECT CODE: 18CSL68

Prepared by,

Mrs. Neetha Das

Asst. Prof, Dept of CSE

ACSCE, Bangalore

Sponsored by

MOOGAMBIGAI CHARITABLE & EDUCATION TRUST

BANGALORE-560074

Sl.no	List of Programs
	Design, develop, and implement the following programs using OpenGL API
1	Implement Brenham's line drawing algorithm for all types of slope. **Refer:Text-1: Chapter 3.5 Refer:Text-2: Chapter 8**
2	Create and rotate a triangle about the origin and a fixed point. **Refer:Text-1: Chapter 5-4 3**
3	Draw a color cube and spin it using OpenGL transformation matrices. **Refer:Text-2: Modelling a Coloured Cube**
4	Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing. **Refer:Text-2: Topic: Positioning of Camera**
5	Clip a lines using Cohen-Sutherland algorithm **Refer:Text-1: Chapter 6.7 & Refer:Text-2: Chapter 8**
6	To draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the surfaces of the solid object used in the scene. **Refer:Text-2: Topic: Lighting and Shading**
7	Design, develop and implement recursively subdivide a tetrahedron to form 3D sierpinski gasket. The number of recursive steps is to be specified by the user. **Refer: Text-2: Topic: sierpinski gasket.**
8	Develop a menu driven program to animate a flag using Bezier Curve algorithm **Refer: Text-1: Chapter 8-10**
9	Develop a menu driven program to fill the polygon using scan line algorithm

Project:

PART –B (MINI-PROJECT) :

Student should develop mini project on the topics mentioned below or similar applications using Open GL API. Consider all types of attributes like color, thickness, styles, font, background, speed etc., while doing mini project.

(**During the practical exam: the students should demonstrate and answer Viva-Voce**)

Sample Topics: Simulation of concepts of OS, Data structures, algorithms etc.

Course outcomes: The students should be able to:

Apply the concepts of computer graphics
Implement computer graphics applications using OpenGL
Animate real world problems using OpenGL

Conduction of Practical Examination:

1. All laboratory experiments from part A are to be included for practical examination.
2. Mini project has to be evaluated for 30 Marks as per 6(b).
3. Report should be prepared in a standard format prescribed for project work.
4. Students are allowed to pick one experiment from the lot.
5. Strictly follow the instructions as printed on the cover page of answer script.
6. Marks distribution:
   a) Part A: Procedure + Conduction + Viva:10 + 35 +5 =50 Marks
   b) Part B: Demonstration + Report + Viva voce = 15+10+05 = 30 Marks
7. Change of experiment is allowed only once and marks allotted to the procedure part to be made zero.

Reference books:

1. Donald Hearn & Pauline Baker: Computer Graphics-OpenGL Version,3rd Edition, Pearson Education,2011
2. Edward Angel: Interactive computer graphics- A Top Down approach with OpenGL, 5th edition. Pearson Education, 2011
3. M M Raikar, Computer Graphics using OpenGL, Fillip Learning / Elsevier, Bangalore / New Delhi (2013)

Introduction to OpenGL

OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in tw[o or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

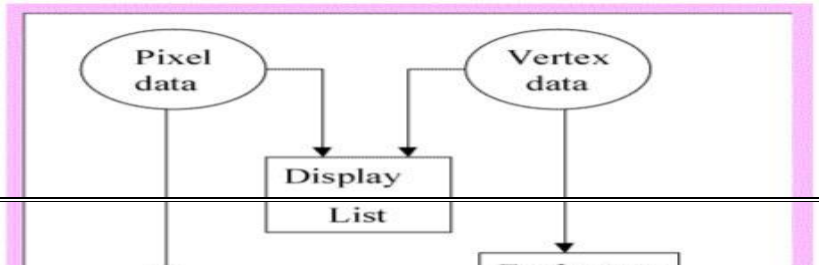
Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the OpenGL Utility Toolkit (GLUT) has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may not be satisfactory for full-featured OpenGL applications, but it is a useful starting point for learning OpenGL.

GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. The interface is designed to be simple yet still meet the needs of useful OpenGL programs. Removing window system operations from OpenGL is a sound decision because it allows the OpenGL graphics system to be retargeted to various systems including powerful but expensive graphics workstations as well as mass-production graphics systems like video games, set-top boxes for interactive television, and PCs.

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT routines also take relatively few parameters.

1.1 Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. Although this is not a strict rule of how OpenGL is implemented, it provides a reliable guide for predicting what OpenGL will do. Geometric data (vertices, line, and polygons) follow a 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 step (rasterization) before the final pixel data is written to the frame buffer.



Display Lists: All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (The alternative to retaining data in a display list is processing the data immediately-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. Evaluators provide a method for deriving the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, colors, and spatial coordinate values from the control points.
Per-Vertex and Primitive Assembly: For vertex data, the next step converts the vertices into primitives. Some types of vertex data are transformed by 4x4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then view port and depth operations are applied. The results at this point are geometric primitives, which are transformed with related color and depth 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 form one of a variety of formats into the proper number of components. Next the data is scaled, biased, processed by a pixel map, and sent to the rasterization step.
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 width, point size, shading model, and coverage calculations to support antialiasing are taken it to 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. The processed fragment is then drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

1.2 Libraries

OpenGL provides a powerful but primitive set of rendering command, 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.
OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

1.3 Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

```
#include<GL/gl.h>
#include <GL/glu.h>
```

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

```
#include <GL/glut.h>
```

Note that glut.h guarantees that gl.h and glu.h are properly included for you so including these three files is redundant. To make your GLUT programs portable, include glut.h and do not include gl.h or glu.h explicitly.

1.4 Setting up Compilers

Windows Using MS Visual C++

Installing GLUT

Most of the following files (ie. OpenGL and GLU) will already be present if you have installed MS Visual C++ v5.0 or later. The following GLUT files will need to be copied into the specified directories.

1. To install:
   - right-click each link
   - choose **Save Link As...**
   - accept the default name (just click **Save**)
   - libraries (place in the **lib**\ subdirectory of Visual C++)
     - opengl32.lib
     - glu32.lib
     - glut32.lib
   - include files (place in the **include\GL**\ subdirectory of Visual C++)
     - gl.h
     - glu.h
     - glut.h
   - dynamically-linked libraries (place in the **Windows\System** subdirectory
     - opengl32.dll
     - glu32.dll
     - glut32.dll

Compiling OpenGL/GLUT Programs

1. Create a new project:
   - choose **File | New** from the File Menu
   - select the **Projects** tab
   - choose **Win32 Console Application**
   - fill in your Project name
2. Designate library files for the linker to use:
   - choose **Project | Settings** from the File Menu
   - under **Oject/library modules:** enter "opengl32.lib glu32.lib glut32.lib"
3. Add/Create files to the project:
   - choose **Project | Add to Project | Files** from the File menu
   - add the required program files
4. Build and Execute

1.5 Initialization

The first thing we need to do is call the glutInit() procedure. It should be called before any other GLUT routine because it initializes the GLUT library. The parameters to glutInit() should be the same as those to main(), specifically main(int argc, char** argv) and glutInit(&argc, argv), where argcp is a pointer to the program's unmodified argc variable from main. Upon return, the value pointed to by argcp will be updated, and argv is the program's unmodified argv variable from main. Like argcp, the data for argv will be updated.

The next thing we need to do is call the glutInitDisplayMode() procedure to specify the display mode for a window. You must first decide whether you want to use an RGBA (GLUT_RGBA) or color-index (GLUT_INDEX) color model. The RGBA mode stores its color buffers as red, green, blue, and alpha color components. The forth color component, alpha, corresponds to the notion of opacity. An alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 complete transparancy. Color-index mode, in contrast, stores color buffers in indices. Your decision on color mode should be based on hardware availability and what you application requires. More colors can usually be simultaneously represented with RGBA mode than with color-index mode. And for special effects, such as shading, lighting, and fog, RGBA mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is the default.

Another decision you need to make when setting up the display mode is whether you want to use single buffering (GLUT_SINGLE) or double buffering (GLUT_DOUBLE). Applications that use both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped. If you aren't using animation, stick with single buffering, which is the default.

Finally, you must decide if you want to use a depth buffer (GLUT_DEPTH), a stencil buffer (GLUT_STENCIL) and/or an accumulation buffer (GLUT_ACCUM). The depth buffer stores a depth value for each pixel. By using a "depth test", the depth buffer can be used to display objects with a smaller depth value in front of objects with a larger depth value. The second buffer, the stencil buffer is used to restrict drawing to certain portions of the screen, just as a cardboard stencil can be used with a can of spray paint to make a printed image. Finally, the accumulation buffer is used for accumulating a series of images into a final composed image. None of these are default buffers.

We need to create the characteristics of our window. A call to glutInitWindowSize() will be used to specify the size, in pixels, of your initial window. The arguments indicate the height and width (in pixels) of the requested window. Similarly, glutInitWindowPosition() is used to specify the screen location for the upper-left corner of your initial window. The arguments, x and y, indicate the location of the window relative to the entire display.

1.6 Creating a Window

To actually create a window, the with the previously set characteristics (display mode, size, location, etc), the programmer uses the glutCreateWindow() command. The command takes a string as a parameter which may appear in the title bar if the window system you are using supports it. The window is not actually displayed until the glutMainLoop() is entered.

1.7 Display Function

The glutDisplayFunc() procedure is the first and most important event callback function you will see. A callback function is one where a programmer-specified routine can be registered to be called in response to a specific type of event. For example, the argument of glutDisplayFunc() is the function that is called whenever GLUT determines that the contents of

the window needs to be redisplayed. Therefore, you should put all the routines that you need to draw a scene in this display callback function.

1.8 Reshape Function

The glutReshapeFunc() is a callback function that specifies the function that is called whenever the window is resized or moved. Typically, the function that is called when needed by the reshape function displays the window to the new size and redefines the viewing characteristics as desired. If glutReshapeFunc() is not called, a default reshape function is called which sets the view to minimize distortion and sets the display to the new height and width.

1.9 Main Loop

The very last thing you must do is call glutMainLoop(). All windows that have been created can now be shown, and rendering those windows is now effective. The program will now be able to handle events as they occur (mouse clicks, window resizing, etc). In addition, the registered display callback (from our glutDisplayFunc()) is triggered. Once this loop is entered, it is never exited!

Geometric Objects used in graphics

2.1 Point, Lines and Polygons

Each geometric object is described by a set of vertices and the type of primitive to be drawn. A vertex is no more than a point defined in three dimensional space. Whether and how these vertices are connected is determined by the primitive type. Every geometric object is ultimately described as an ordered set of vertices. We use the glVertex*() command to specify a vertex. The '*' is used to indicate that there are variations to the base command glVertex().

Some OpenGL command names have one, two, or three letters at the end to denote the number and type of parameters to the command. The first character indicates the number of values of the indicated type that must be presented to the command. The second character indicates the specific type of the arguments. The final character, if present, is 'v', indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual agreements.

For example, in the command glVertex3fv(), '3' is used to indicate three arguments, 'f' is used to indicate the arguments are floating point, and 'v' indicates that the arguments are in vector format.

Points: A point is represented by a single vertex. Vertices specified by the user as two-dimensional (only x- and y-coordinates) are assigned a z-coordinate equal to zero. To control the size of a rendered point, use glPointSize() and supply the desired size in pixels as the argument. The default is as 1 pixel by 1 pixel point. If the width specified is 2.0, the point will be a square of 2 by 2 pixels. glVertex*() is used to describe a point, but it is only effective between a glBegin() and a glEnd() pair. The argument passed to glBegin() determines what sort of geometric primitive is constructed from the vertices.

Lines: In OpenGL, the term line refers to a line segment, not the mathematician's version that extends to infinity in both directions. The easiest way to specify a line is in terms of the vertices at the endpoints. As with the points above, the argument passed to glBegin() tells it what to do with the vertices. The option for lines includes:

GL_LINES: Draws a series of unconnected line segments drawn between each set of vertices. An extraneous vertex is ignored. GL_LINE_STRIP: Draws a line segment from the first vertex to the last. Lines can intersect arbitrarily.

GL_LINE_LOOP: Same as GL_STRIP, except that a final line segment is drawn from the last vertex back to the first.

With OpenGL, the description of the shape of an object being drawn is independent of the description of its color. When a particular geometric object is drawn, it's drawn using the currently specified coloring scheme. In general, an OpenGL programmer first sets the color, using glColor*() and then draws the objects. Until the color is changed, all objects are drawn in that color or using that color scheme.

Polygons: Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points. In OpenGL, there are a few restrictions on what constitutes a primitive polygon. For example, the edges of a polygon cannot intersect and they must be convex (no indentations). There are special commands for a three-sided (triangle) and four-sided (quadrilateral) polygons, glBegin(GL_TRIANGLES) and glBegin(GL_QUADS), respectively. However, the general case of a polygon can be defined using glBegin(GL_POLYGON).

2.2 Drawing 3-D Objects

GLUT has a series of drawing routines that are used to create three-dimensional models. This means we don't have to reproduce the code necessary to draw these models in each program. These routines render all their graphics in immediate mode. Each object comes in two flavors, wire or solid. Available objects are:

2.3 Transformations

A modeling transformation is used to position and orient the model. For example, you can rotate, translate, or scale the model - or some combination of these operations. To make an object appear further away from the viewer, two options are available - the viewer can move closer to the object or the object can be moved further away from the viewer. Moving the viewer will be discussed later when we talk about viewing transformations. For right now, we will keep the default "camera" location at the origin, pointing toward the negative z-axis, which goes into the screen perpendicular to the viewing plane.

When transformations are performed, a series of matrix multiplications are actually performed to affect the position, orientation, and scaling of the model. You must, however, think of these matrix multiplications occuring in the opposite order from how they appear in the code. The order of transformations is critical. If you perform transformation A and then perform transformation B, you almost always get something different than if you do them in the opposite order.

Scaling: The scaling command glScale() multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x-, y-, and z-coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. The glScale*() is the only one of the three modeling transformations that changes the apparent size of an object: scaling with values greater than 1.0 stretches an object, and using values less than 1.0 shrinks it. Scaling with a -1.0 value reflects an object across an axis.

Translation: The translation command glTranslate() multiplies the current matrix by a matrix that moves (translates) an object by the given x-, y-, and z-values.

Rotation: The rotation command glRotate() multiplies the current matrix that rotates an object in a counter clockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees. An object that lies farther from the axis of rotation is more dramatically rotated (has a larger orbit) than an object drawn near the axis.

Sample Programs

Sample Program: 01

```
/* A basic Open GL window*/
#include<GL/glut.h>
```

```
void display (void)
{
      glClearColor (0.0,0.0,0.0,1.0);
      glClear (GL_COLOR_BUFFER_BIT);
      glLoadIdentity ();
      gluLookAt (0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0); glFlush
      ();
}
int main (int argc,char **argv)
{
      glutInit (&argc,argv);
      glutInitDisplayMode (GLUT_SINGLE);
      glutInitWindowSize (500,500);
      glutInitWindowPosition (100,100);
      glutCreateWindow ("A basic open GL window");
      glutDisplayFunc (display);
      glutMainLoop ();
      return 0;
}
```

Sample Program: 02

```
/*Program to create a teapot*/
#include<GL/glut.h>

void cube(void)
{
      glutWireTeapot(2);
}

void display(void)
{

       glClearColor(0.0,0.0,0.0,1.0);
       glClear(GL_COLOR_BUFFER_BIT);
       glLoadIdentity();
       gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
       cube();
       glFlush();
}

void reshape(int w,int h)
{
      glViewport(0,0,(GLsizei)w,(GLsizei)h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(60,(GLfloat)w/(GLfloat)h,1.0,100.0);
      glMatrixMode(GL_MODELVIEW);
}

int main(int agrc,char**agrv)

{
            glutInit(&agrc,agrv);
            glutInitDisplayMode(GLUT_SINGLE);
            glutInitWindowSize(500,500);
            glutInitWindowPosition(100,100);
            glutCreateWindow("A basic OpenGL Window");
            glutDisplayFunc(display);
            glutReshapeFunc(reshape);
            glutMainLoop();
```

```
            return 0;
}
```

Sample Program: 03

```
/* Program to create a cube*/
#include<GL/glut.h>

void cube(void)
{
      glutWireCube(2);
}

void display(void)
{

      glClearColor(0.0,0.0,0.0,1.0);
      glClear(GL_COLOR_BUFFER_BIT);
      glLoadIdentity();
      gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
      cube();
      glFlush();
}

void reshape(int w,int h)
{
      glViewport(0,0,(GLsizei)w,(GLsizei)h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(60,(GLfloat)w/(GLfloat)h,1.0,100.0);
      glMatrixMode(GL_MODELVIEW);
}

int main(int agrc,char** agrv)
{
      glutInit(&agrc,agrv);
      glutInitDisplayMode(GLUT_SINGLE);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(100,100);
      glutCreateWindow("A basic OpenGL Window");
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutMainLoop();
      return 0;
}
```

Sample Program: 04

```
/*point.c
/* Program to draw/display points */
#include<GL/glut.h>
#include<stdlib.h>
void myInit(void)
```

```
{
	glClearColor(2.0,2.0,2.0,4.0);
	glColor3f(0.0f,0.0f,0.0f);
	glPointSize(4.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0.0,640.0,0.0,480.0);
}
void display(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glBegin(GL_POINTS);
	glVertex2i(100,200);
	glVertex2i(400,200);
	glVertex2i(200,100);
	glVertex2i(200,400);
	glEnd();
	glFlush();
}
void main(int argc, char** argv)
{
	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowSize(500,500);
	glutInitWindowPosition(100,150);
	glutCreateWindow("My First Attempt");
	glutDisplayFunc(display);
	myInit();
	glutMainLoop();
}
```

Sample Program: 05

/*program to implement horizontal and vertical lines*/

```
#include<GL/glut.h>
#include<stdlib.h>
```

```
void myInit(void)
{
	glClearColor(2.0,2.0,2.0,4.0);
	glColor3f(0.0f,0.0f,0.0f);
	glLineWidth(4.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0.0,640.0,0.0,480.0);
}
void drawLineInt(GLint x1,GLint y1,GLint x2,GLint y2)
{
	glBegin(GL_LINES);
	glVertex2i(x1,y1);
	glVertex2i(x2,y2);
	glEnd();
}

void display(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glBegin(GL_LINES);
	glVertex2i(100,200);
	glVertex2i(400,200);
	glVertex2i(200,100);
	glVertex2i(200,400);
	glEnd();
	glFlush();
}
void main(int argc, char** argv)
{
	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
	glutInitWindowSize(500,500);
	glutInitWindowPosition(100,150);
	glutCreateWindow("My First Attempt");
	glutDisplayFunc(display);
	myInit();
	drawLineInt(100,200,40,60);
	glutMainLoop();
}
```

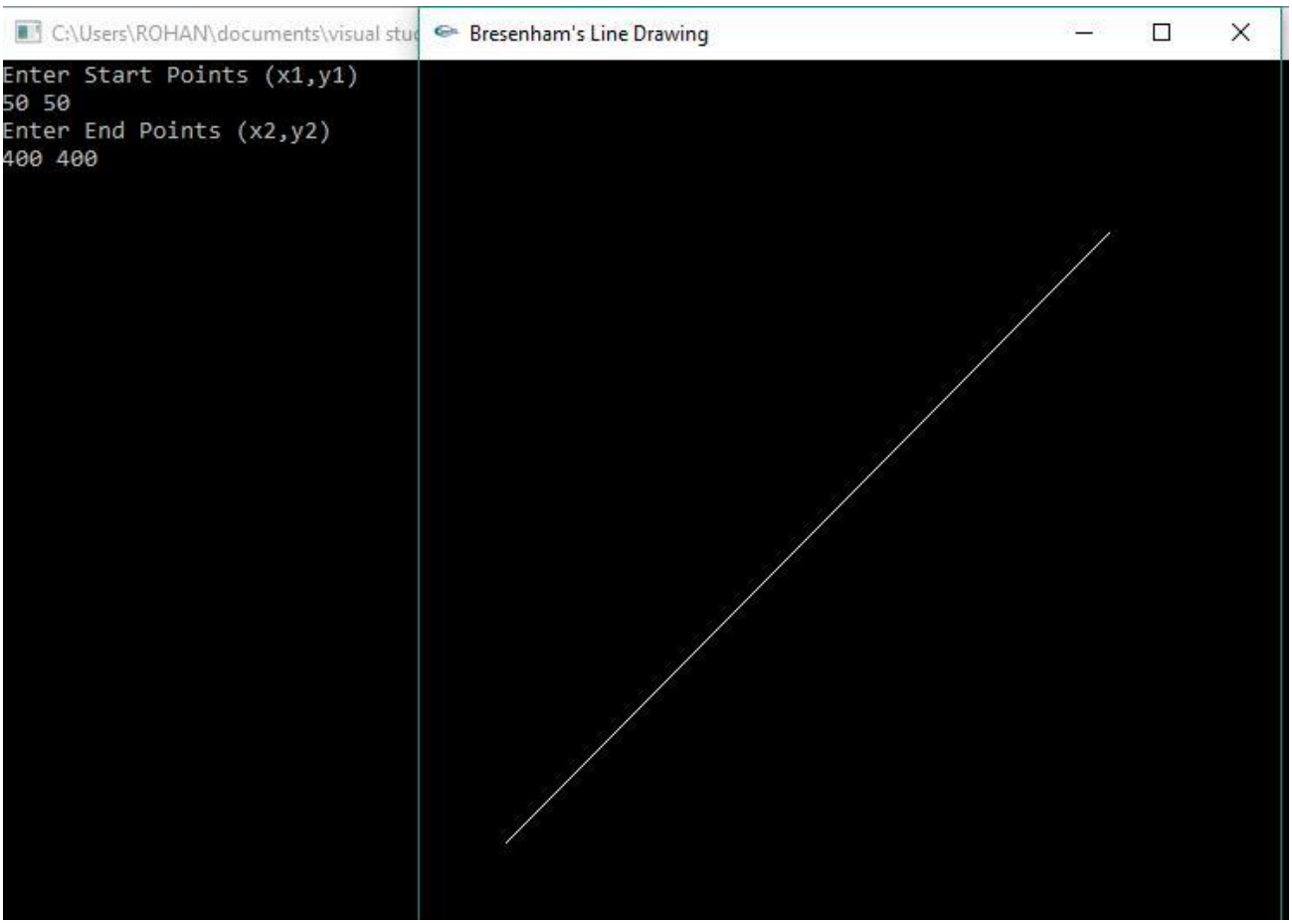
1.Implement Brenham's line drawing algorithm for all types of slope.

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

```
int x1, y1, x2, y2;

void draw_pixel(int x, int y)
{
        glColor3f(1.0,0.0,0.0);
        glBegin(GL_POINTS);
        glVertex2i(x, y);
        glEnd();
}

void brenhams_line_draw(int x1, int y1, int x2, int y2)
{
        int dx=x2-x1,dy=y2-y1;
        int p=2*dy*dx;
        int twoDy=2*dy;
        int twoDyMinusDx=2*(dy-dx);             // paranthesis are required
        int x=x1,y=y1;
        if(dx<0)
        {
                x=x2;
                y=y2;
                x2=x1;
        }
        draw_pixel(x, y);
        while(x<x2)
        {
                x++;
                if(p<0)
                        p+=twoDy;
                else
                {
                        y++;
                        p+=twoDyMinusDx;
                }
                draw_pixel(x, y);
        }
}
void myInit()
{
        glClearColor(0.0,0.0,0.0,1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, 500.0, 0.0, 500.0);
        glMatrixMode(GL_MODELVIEW);
}
void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        brenhams_line_draw(x1, y1, x2, y2);
        glFlush();
}
void main(int argc, char **argv)
{
```

```
        printf( "Enter Start Points (x1,y1)\n");
        scanf("%d %d", &x1, &y1);
        printf( "Enter End Points (x2,y2)\n");
        scanf("%d %d", &x2, &y2);

        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(0, 0);
        glutCreateWindow("Bresenham's Line Drawing");
        myInit();
        glutDisplayFunc(display);
        glutMainLoop();
}
```

OUTPUT :



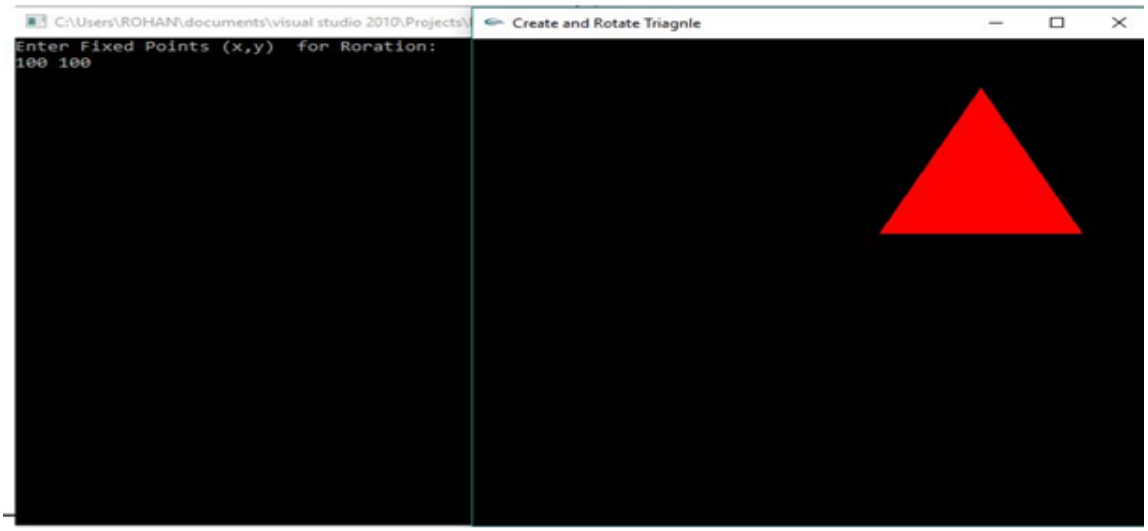
2. Create and rotate a triangle about the origin and a fixed point.

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

int x,y;
```

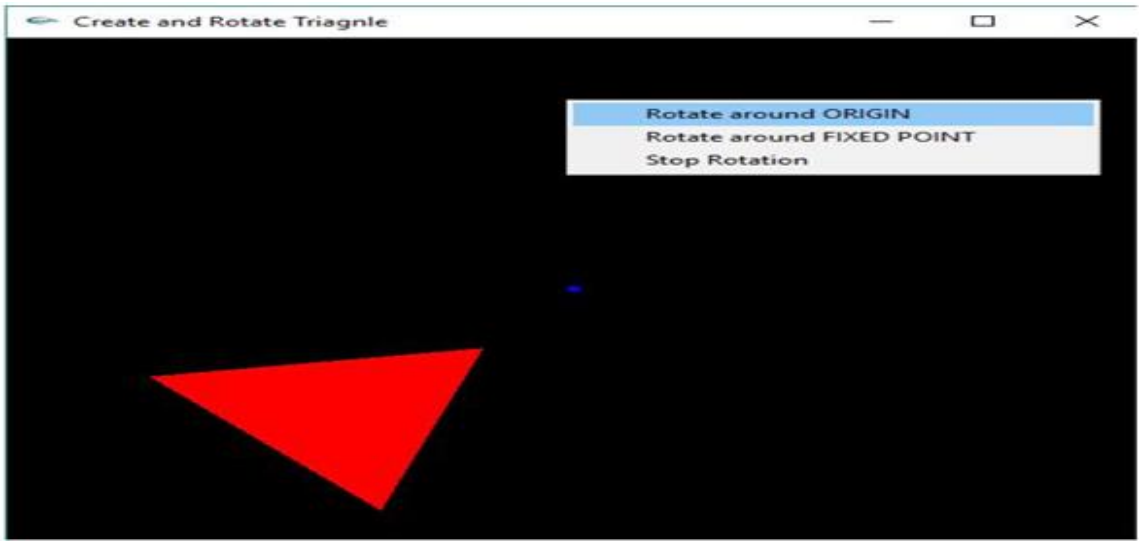
```
int rFlag=0;

void draw_pixel(float x1,float y1)
{
        glColor3f(0.0,0.0,1.0);
        glPointSize(5.0);
        glBegin(GL_POINTS);
        glVertex2f(x1,y1);
        glEnd();
}

void triangle()
{
        glColor3f(1.0,0.0,0.0);
        glBegin(GL_POLYGON);
            glVertex2f(100,100);
            glVertex2f(250,400);
            glVertex2f(400,100);
        glEnd();
}

float th=0.0;
float trX=0.0,trY=0.0;
void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glLoadIdentity();

        if(rFlag==1) //Rotate Around origin
        {
                trX=0.0;
                trY=0.0;
                th+=0.1;
                draw_pixel(0.0,0.0);
        }
        if(rFlag==2) //Rotate Around Fixed Point
        {
                trX=x;
                trY=y;
                th+=0.1;
                draw_pixel(x,y);
        }
        glTranslatef(trX,trY,0.0);
        glRotatef(th,0.0,0.0,1.0);
        glTranslatef(-trX,-trY,0.0);
           triangle();
        glutPostRedisplay();
        glutSwapBuffers();
}
void myInit()
{
        glClearColor(0.0,0.0,0.0,1.0);
        glMatrixMode(GL_PROJECTION);
```

```
        glLoadIdentity();
        gluOrtho2D(-500.0, 500.0, -500.0, 500.0);
        glMatrixMode(GL_MODELVIEW);
}


void rotateMenu (int option)
{
        if(option==1)
                rFlag=1;
        if(option==2)
                rFlag=2;
        if(option==3)
                rFlag=3;
}
void main(int argc, char **argv)
{
        printf( "Enter Fixed Points (x,y) for Roration: \n");
        scanf("%d %d", &x, &y);
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(0, 0);
        glutCreateWindow("Create and Rotate Triangle");
        myInit();
        glutDisplayFunc(display);
        glutCreateMenu(rotateMenu);
        glutAddMenuEntry("Rotate around ORIGIN",1);
        glutAddMenuEntry("Rotate around FIXED
        POINT",2); glutAddMenuEntry("Stop Rotation",3);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
        glutMainLoop();
}
```

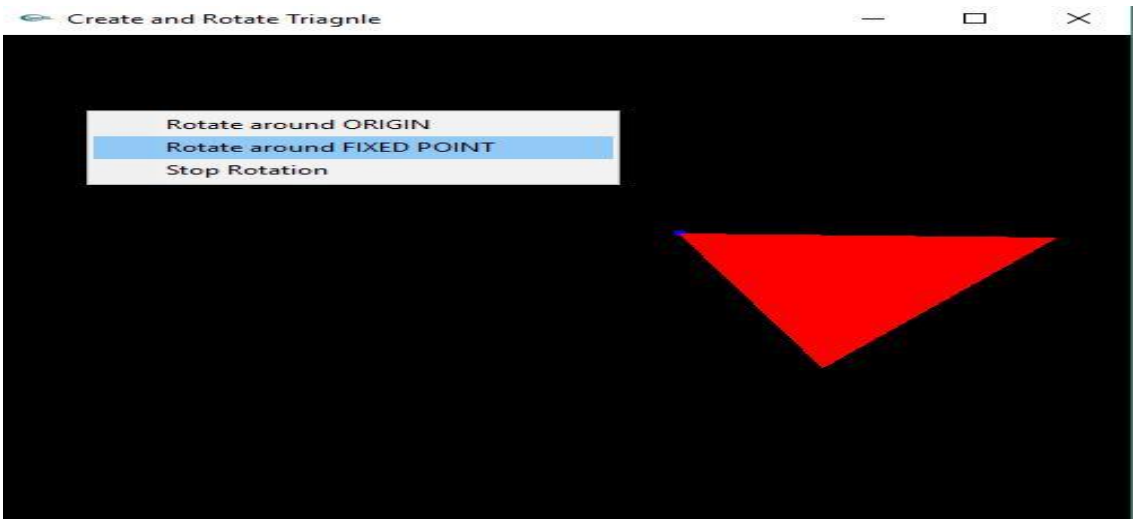
OUTPUT :



Rotation Around the ORIGIN



Rotation Around the FIXED POINT

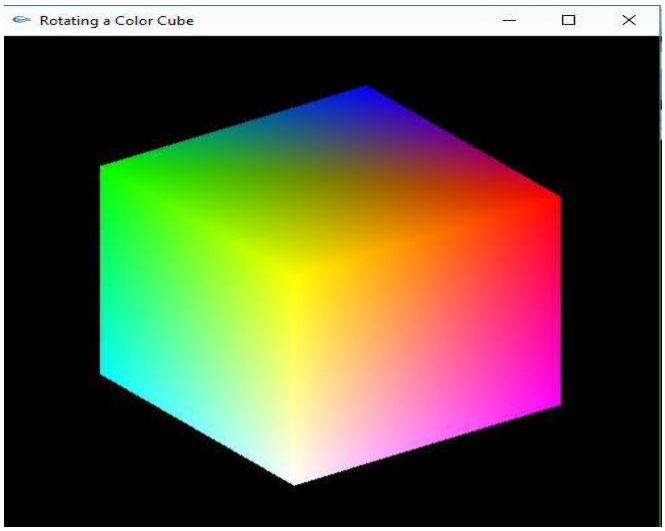


3.Program to draw a color cube and spin it using OpenGL transformation matrices.

```
#include <stdlib.h>
#include <GL/glut.h>
GLfloat vertices[][3] = {{-1,-1,-1},{1,-1,-1},{1,1,-1},{-1,1,-1}, {-1,-1,1},{1,-1,1}, {1,1,1},{-1,1,1}};
GLfloat colors[][3] = {{1,0,0},{1,1,0},{0,1,0},{0,0,1}, {1,0,1},{1,1,1},{0,1,1},{0.5,0.5,0.5}};


void polygon(int a, int b, int c , int d)
{
        glBegin(GL_POLYGON);
                glColor3fv(colors[a]);
                glVertex3fv(vertices[a]);
                glColor3fv(colors[b]);
                glVertex3fv(vertices[b]);
                glColor3fv(colors[c]);
                glVertex3fv(vertices[c]);
                glColor3fv(colors[d]);
                glVertex3fv(vertices[d]);
        glEnd();
}

void colorcube(void)
{
        polygon(0,3,2,1);
        polygon(0,4,7,3);
        polygon(5,4,0,1);
        polygon(2,3,7,6);
        polygon(1,2,6,5);
        polygon(4,5,6,7);
}

GLfloat theta[] = {0.0,0.0,0.0};
GLint axis = 2;

void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glRotatef(theta[0], 1.0, 0.0, 0.0);
        glRotatef(theta[1], 0.0, 1.0, 0.0);
        glRotatef(theta[2], 0.0, 0.0, 1.0);
        colorcube();
        glutSwapBuffers();
}

void spinCube()
{
       theta[axis] += 1.0;
        if( theta[axis] > 360.0 )
                theta[axis] -= 360.0;
        glutPostRedisplay();
}
void mouse(int btn, int state, int x, int y)
{
```

```
        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}

void myReshape(int w, int h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      if (w <= h)
      glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
      else
       glOrtho(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0,10.0);
     glMatrixMode(GL_MODELVIEW);
}

void main(int argc, char *argv[])
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(500, 500);
      glutCreateWindow("Rotating a Color Cube");
      glutReshapeFunc(myReshape);
      glutDisplayFunc(display);
      glutIdleFunc(spinCube);
      glutMouseFunc(mouse);
      glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
      glutMainLoop();
}
```

OUTPUT :

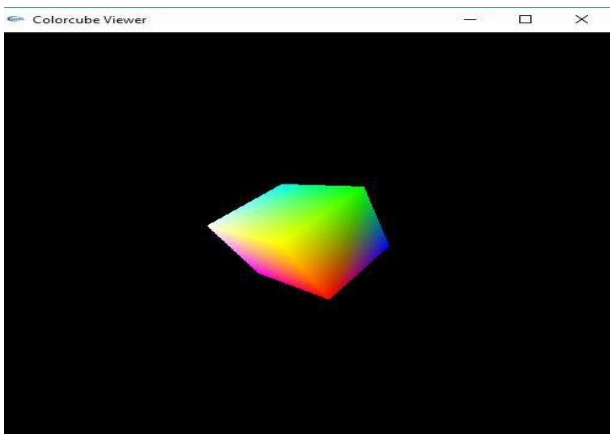


4.Program to draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.

```
#include <stdlib.h>
#include <GL/glut.h>

GLfloat vertices[][3] = {{-1,-1,-1},{1,-1,-1},{1,1,-1},{-1,1,-1}, {-1,-1,1},{1,-1,1}, {1,1,1},{-1,1,1}};
GLfloat colors[][3] = {{1,0,0},{1,1,0},{0,1,0},{0,0,1}, {1,0,1},{1,1,1},{0,1,1},{0.5,0.5,0.5}};

void polygon(int a, int b, int c , int d)
{
        glBegin(GL_POLYGON);
                glColor3fv(colors[a]);
                glVertex3fv(vertices[a]);
                glColor3fv(colors[b]);
                glVertex3fv(vertices[b]);
                glColor3fv(colors[c]);
                glVertex3fv(vertices[c]);
                glColor3fv(colors[d]);
                glVertex3fv(vertices[d]);
        glEnd();
}

void colorcube(void)
{
        polygon(0,3,2,1);
        polygon(0,4,7,3);
        polygon(5,4,0,1);
        polygon(2,3,7,6);
        polygon(1,2,6,5);
        polygon(4,5,6,7);
}

GLfloat theta[] = {0.0,0.0,0.0};
GLint axis = 2;
GLdouble viewer[]= {0.0, 0.0, 5.0}; /* initial viewer location */

void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT); glLoadIdentity();
        gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
        glRotatef(theta[0], 1.0, 0.0, 0.0);
        glRotatef(theta[1], 0.0, 1.0, 0.0);
        glRotatef(theta[2], 0.0, 0.0, 1.0);
        colorcube();
          glFlush();
        glutSwapBuffers();
}
       void mouse(int btn, int state, int x, int y)
       {
        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
        theta[axis] += 2.0;
```

```
 if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
 display();
}

void keys(unsigned char key, int x, int y)
{
        if(key == 'x') viewer[0]-= 1.0;
        if(key == 'X') viewer[0]+= 1.0;
        if(key == 'y') viewer[1]-= 1.0;
        if(key == 'Y') viewer[1]+= 1.0;
        if(key == 'z') viewer[2]-= 1.0;
        if(key == 'Z') viewer[2]+= 1.0;
        display();
}

void myReshape(int w, int h)
{
 glViewport(0, 0, w, h);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 if(w<=h)
  glFrustum(-2.0, 2.0, -2.0 * (GLfloat) h/ (GLfloat) w, 2.0* (GLfloat) h / (GLfloat) w,  2.0, 20.0);
        else
  glFrustum(-2.0, 2.0, -2.0 * (GLfloat) w/ (GLfloat) h, 2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0);
        glMatrixMode(GL_MODELVIEW);
}

 void   main(int argc, char **argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(500, 500);
        glutCreateWindow("Colorcube Viewer");
        glutReshapeFunc(myReshape);
        glutDisplayFunc(display);
        glutMouseFunc(mouse);
        glutKeyboardFunc(keys);
        glEnable(GL_DEPTH_TEST);
        glutMainLoop();
}
```

OUTPUT :



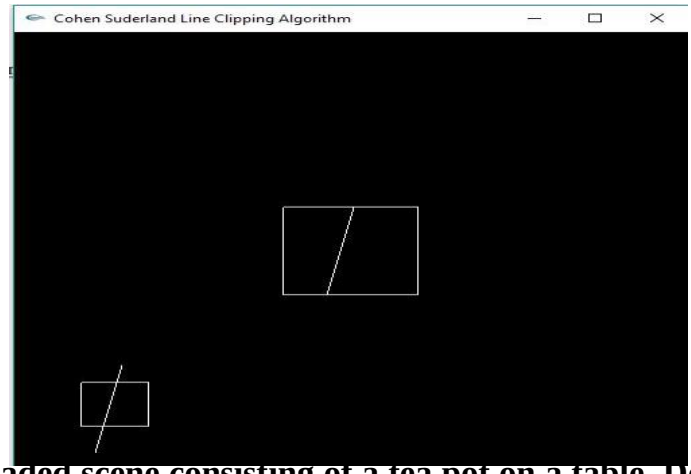
5. Program to clip a lines using Cohen-Sutherland line-clipping algorithm.

```
#include <stdio.h>
#include <GL\glut.h>

double xmin=50,ymin=50, xmax=100,ymax=100;
double
xvmin=200,yvmin=200,xvmax=300,yvmax=300;
const int RIGHT = 8;
const int LEFT = 2;
const int TOP = 4;
const int BOTTOM = 1;
int ComputeOutCode (double x, double y)
{
        int code = 0;
        if (y > ymax)                           //above the clip window
                code |= TOP;
        else if (y < ymin)                      //below the clip window
                code |= BOTTOM;
        if (x > xmax)                           //to the right of clip window
                code |= RIGHT;
        else if (x < xmin)                      //to the left of clip window
                code |= LEFT;
        return code;
}
void CohenSutherland(double x0, double y0,double x1, double y1)
{
        int outcode0, outcode1, outcodeOut;
        bool accept = false, done = false;
        outcode0 = ComputeOutCode (x0, y0);
        outcode1 = ComputeOutCode (x1, y1);
        do{
                if (!(outcode0 | outcode1))
                {
                        accept = true;
                        done = true;
                }
                else if (outcode0 & outcode1)
                        done = true;
                else    {

                        double x, y;
                        outcodeOut = outcode0? outcode0: outcode1;
                        if (outcodeOut & TOP)
                        {
                                x = x0 + (x1 - x0) * (ymax - y0)/(y1 - y0);
                                y = ymax;
                        }
                        else if (outcodeOut & BOTTOM)
                        {
                               x = x0 + (x1 - x0) * (ymin - y0)/(y1 - y0);
                                y = ymin;
                        }
                        else if (outcodeOut & RIGHT)
                        {
```

```
                        y = y0 + (y1 - y0) * (xmax - x0)/(x1 - x0);
                        x = xmax;
                }

                else
                {
                        y = y0 + (y1 - y0) * (xmin - x0)/(x1 - x0);
                        x = xmin;
                }

        if (outcodeOut == outcode0)
                {
                        x0 = x;
                        y0 = y;
                        outcode0 = ComputeOutCode (x0, y0);
                }
                else
                {
                        x1 = x;
                        y1 = y;
                        outcode1 = ComputeOutCode (x1, y1);
                }
        }
    }while (!done);

    if (accept)
    {

        double sx=(xvmax-xvmin)/(xmax-xmin);
        double sy=(yvmax-yvmin)/(ymax-ymin);
        double vx0=xvmin+(x0-xmin)*sx;
        double vy0=yvmin+(y0-ymin)*sy;
        double vx1=xvmin+(x1-xmin)*sx;
        double vy1=yvmin+(y1-ymin)*sy;

        glColor3f(1.0, 1.0, 1.0);
        glBegin(GL_LINE_LOOP);
                glVertex2f(xvmin, yvmin);
                glVertex2f(xvmax, yvmin);
                glVertex2f(xvmax, yvmax);
                glVertex2f(xvmin, yvmax);
        glEnd();

        glColor3f(1.0,1.0,1.0);
        glBegin(GL_LINES);
                glVertex2d (vx0, vy0);
        glEnd();
    }
}

void display()
{
    double x0=60,y0=20,x1=80,y1=120;
    glClear(GL_COLOR_BUFFER_BIT);
```

```
        glColor3f(1.0,1.0,1.0);
        glBegin(GL_LINES);
           glVertex2d (x0, y0);
           glVertex2d (x1, y1);
        glEnd();
        glColor3f(1.0, 1.0, 1.0);

        glBegin(GL_LINE_LOOP);
                glVertex2f(xmin, ymin);
                glVertex2f(xmax, ymin);
                glVertex2f(xmax, ymax);
                glVertex2f(xmin, ymax);
        glEnd();
        CohenSutherland(x0,y0,x1,y1);
        glFlush();
}

void myinit()
{
        glClearColor(0.0,0.0,0.0,1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,500.0,0.0,500.0);
        glMatrixMode(GL_MODELVIEW);

}
void main(int argc, char **argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Cohen Suderland Line Clipping Algorithm");
        myinit();
        glutDisplayFunc(display);
        glutMainLoop();
}
```

OUTPUT :

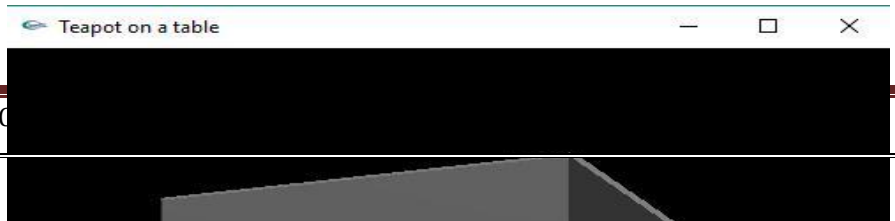


6.Program to draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the properties of the surfaces of the solid object used in the scene.

```
#include<GL/glut.h>

void teapot(GLfloat x,GLfloat y,GLfloat z)
{
        glPushMatrix();
        glTranslatef(x,y,z);
        glutSolidTeapot(0.1);
        glPopMatrix();
}

void tableTop(GLfloat x,GLfloat y,GLfloat z)
{
        glPushMatrix();
        glTranslatef(x,y,z);
        glScalef(0.6,0.02,0.5);
        glutSolidCube(1.0);
        glPopMatrix();
}

void tableLeg(GLfloat x,GLfloat y,GLfloat z)
{
        glPushMatrix();
        glTranslatef(x,y,z);
        glScalef(0.02,0.3,0.02);
        glutSolidCube(1.0);
        glPopMatrix();
}

void wall(GLfloat x,GLfloat y,GLfloat z)
{
        glPushMatrix();
        glTranslatef(x,y,z);
        glScalef(1.0,1.0,0.02);
        glutSolidCube(1.0);
        glPopMatrix();
}

void light()
{
        GLfloat mat_ambient[]={1.0,1.0,1.0,1.0}; GLfloat
        mat_diffuse[]={0.5,0.5,0.5,1.0}; GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
        GLfloat mat_shininess[]={50.0f};
        glMaterialfv(GL_FRONT,GL_AMBIENT,mat_a mbient);
        glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
        glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
        glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
        lightIntensity[]={0.7,0.7,0.7,1.0};
        glLightfv(GL_LIGHT0,GL_POSITION,light_position);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,lightIntensity);
}
void display()
{
        GLfloat teapotP=-0.07,tabletopP=-0.15,tablelegP=0.2,wallP=0.5;
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

```
        glLoadIdentity();
        gluLookAt(-2.0,2.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);

        light();                                    //Adding light source to your project

        teapot(0.0,teapotP,0.0);                                    //Create teapot
        tableTop(0.0,tabletopP,0.0);                                //Create table's top
        tableLeg(tablelegP,-0.3,tablelegP);                         //Create 1st leg
        tableLeg(-tablelegP,-0.3,tablelegP);                        //Create 2nd leg
        tableLeg(-tablelegP,-0.3,-tablelegP);                       //Create 3rd leg
        tableLeg(tablelegP,-0.3,-tablelegP);                        //Create 4th leg


        wall(0.0,0.0,-wallP);                                       //Create 1st wall
         glRotatef(90.0,1.0,0.0,0.0);
         wall(0.0,0.0,wallP);                                       //Create 2nd wall
         glRotatef(90.0,0.0,1.0,0.0);
        wall(0.0,0.0,wallP);                                        //Create 3rd wall
         glFlush();
        }
void myinit()
{
        glClearColor(0.0,0.0,0.0,1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-1.0,1.0,-1.0,1.0,-1.0,10.0);
        glMatrixMode(GL_MODELVIEW);
}
 void main(int argc,char **argv)
 {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_
        DEPTH); glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0); glutCreateWindow("Teapot
        on a table"); myinit();

        glutDisplayFunc(display);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glShadeModel(GL_SMOOTH);
        glEnable(GL_NORMALIZE);
        glEnable(GL_DEPTH_TEST);
        glutMainLoop();
 }
```

OUTPUT :



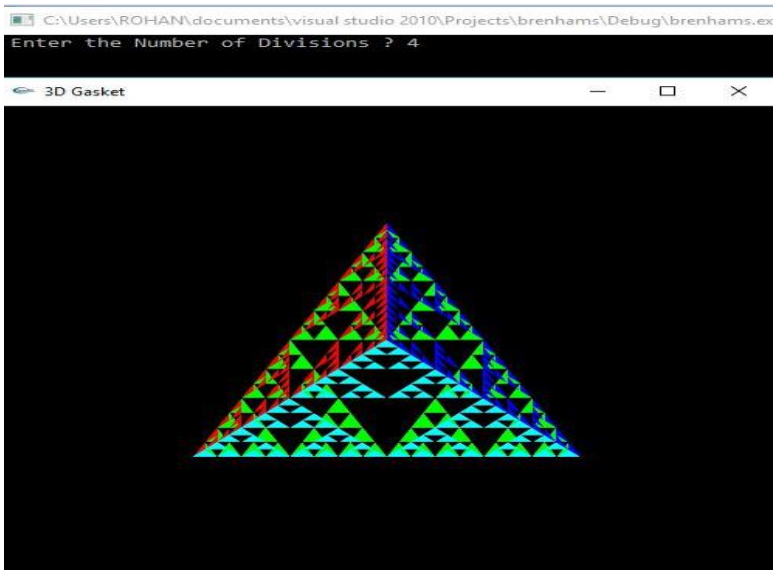
7.Program to recursively subdivide a tetrahedron to from 3D Sierpinski gasket. The number of recursive steps is to be specified by the user

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
```

```
float point v[][3]={{0.0, 0.0, 0.0}, {0.0, 1.0, -1.0},{-1.0, -1.0, -1.0}, {1.0, -1.0, -1.0}};
int n;

void triangle( point a, point b, point c)
{
      glBegin(GL_POLYGON);
         glVertex3fv(a);
          glVertex3fv(b);
          glVertex3fv(c);
      glEnd();
}

void divide_triangle(point a, point b, point c, int m)
{
      point v1, v2, v3;
      int j;
      if(m>0)
      {
            for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;
            for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;
            for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;
            divide_triangle(a, v1, v2, m-1);
            divide_triangle(c, v2, v3, m-1);
            divide_triangle(b, v3, v1, m-1);
      }
      else(triangle(a,b,c)); /* draw triangle at end of recursion */
}


void tetrahedron( int m)
{
         glColor3f(1.0,0.0,0.0);
      divide_triangle(v[0], v[1], v[2], m);
         glColor3f(0.0,1.0,0.0);
      divide_triangle(v[3], v[2], v[1], m);
         glColor3f(0.0,0.0,1.0);
      divide_triangle(v[0], v[3], v[1], m);
         glColor3f(0.0,0.0,0.0);
      divide_triangle(v[0], v[2], v[3], m);
}
void display(void)
{
         glClear(GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT);
         glLoadIdentity();
         tetrahedron(n);
         glFlush();
}
void myReshape(int w, int h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
```

```
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 *
            (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

void main(int argc, char **argv)
{
      printf(" Enter the Number of Divisions ? ");
      scanf("%d",&n);
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |
      GLUT_DEPTH); glutInitWindowSize(500, 500);
      glutCreateWindow("3D Gasket");
      glutReshapeFunc(myReshape);
      glutDisplayFunc(display);
      glEnable(GL_DEPTH_TEST);
      glClearColor (1.0, 1.0, 1.0, 1.0);
      glutMainLoop();
}
```

OUTPUT :



8. Develop a menu driven program to animate a flag using Bezier curve algorithm.

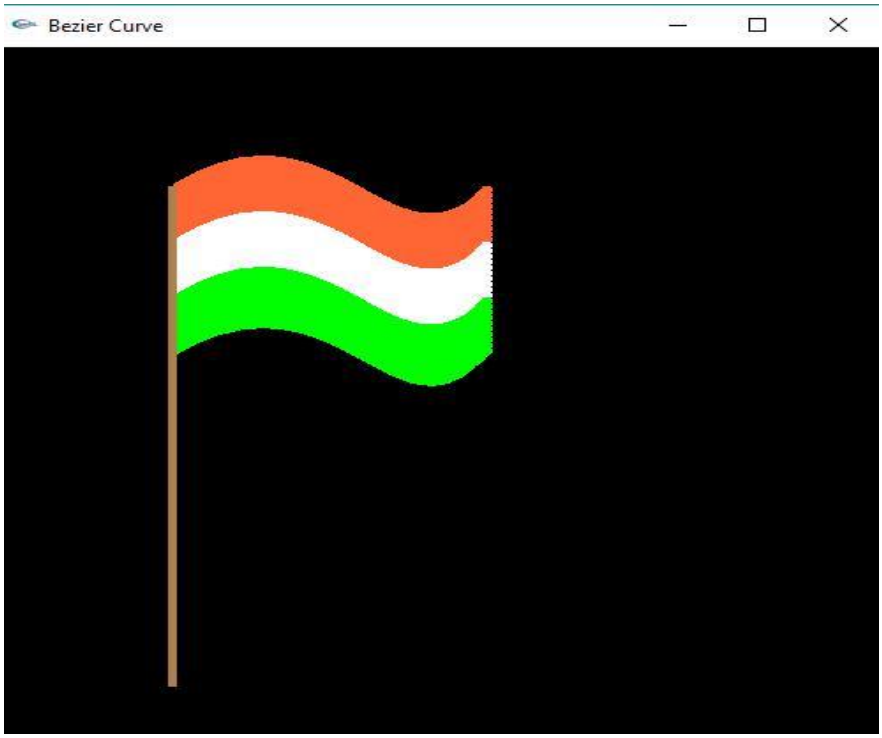
```
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
#define PI 3.1416
```

```
typedef struct point
{
        GLfloat x, y, z;
};
void bino(int n, int *C)
{
    int k, j;
    for(k=0;k<=n;k++)
        {
            C[k]=1;
            for(j=n;j>=k+1; j--)
            C[k]*=j;
            for(j=n-k;j>=2;j--)
            C[k]/=j;
        }
}
void computeBezPt(float u, point *pt1, int cPt, point *pt2, int *C)
{
        int k, n=cPt-1;
        float bFcn;
        pt1 ->x =pt1 ->y = pt1->z=0.0;
        for(k=0; k< cPt; k++)
        {
            bFcn = C[k] * pow(u, k) * pow( 1-u, n-k);
            pt1 ->x += pt2[k].x * bFcn;
            pt1 ->y += pt2[k].y * bFcn;
            pt1 ->z += pt2[k].z * bFcn;
        }
}
void bezier(point *pt1, int cPt, int bPt)
{
        point bcPt;
        float u;
        int *C, k;
        C= new int[cPt];
        bino(cPt-1, C);
        glBegin(GL_LINE_STRIP);
        for(k=0; k<=bPt; k++)
        {
            u=float(k)/float(bPt);
            computeBezPt(u, &bcPt, cPt, pt1, C);
            glVertex2f(bcPt.x, bcPt.y);
        }
        glEnd();
}
float theta = 0;
void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        int nCtrlPts = 4, nBCPts =20;
        point ctrlPts[4] = {{100, 400, 0}, {150, 450, 0}, {250, 350, 0}, {300, 400,
                                0}};
```

```
                ctrlPts[1].x +=50*sin(theta * PI/180.0);
                ctrlPts[1].y +=25*sin(theta * PI/180.0);
                ctrlPts[2].x -= 50*sin((theta+30)          * PI/180.0);
                ctrlPts[2].y -= 50*sin((theta+30)          * PI/180.0);
                ctrlPts[3].x -= 25*sin((theta)          * PI/180.0);
                ctrlPts[3].y += sin((theta-30)          * PI/180.0);
                theta+=0.2;
                glClear(GL_COLOR_BUFFER_BIT);
                glColor3f(1.0, 1.0, 1.0);
                glPointSize(5);
                glPushMatrix();
                glLineWidth(5);
                glColor3f(1, 0.4, 0.2);          //Indian flag: Orange color code
                for(int i=0;i<50;i++)
                {
                        glTranslatef(0, -0.8, 0);
                        bezier(ctrlPts, nCtrlPts, nBCPts);
                }
                glColor3f(1, 1, 1);          //Indian flag: white color code
                for(int i=0;i<50;i++)
                {
                        glTranslatef(0, -0.8, 0);
                        bezier(ctrlPts, nCtrlPts, nBCPts);
                }
                glColor3f(0, 1, 0); //Indian flag: green color code
                for(int i=0;i<50;i++)
                {
                        glTranslatef(0, -0.8, 0);
                        bezier(ctrlPts, nCtrlPts, nBCPts);
                }
                glPopMatrix();
                glColor3f(0.7, 0.5,0.3);
                glLineWidth(5);
                glBegin(GL_LINES);
                        glVertex2f(100,400);
                        glVertex2f(100,40);
                glEnd();

                glutPostRedisplay();
                glutSwapBuffers();
 }
void init()
 {
                glMatrixMode(GL_PROJECTION);
                glLoadIdentity();
                gluOrtho2D(0,500,0,500);
 }
 void main(int argc, char **argv)
 {
                glutInit(&argc, argv);
                glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
                glutInitWindowPosition(0, 0);
                glutInitWindowSize(500,500);
```

```
        glutCreateWindow("Bezier Curve");
        init();
        glutDisplayFunc(display);
        glutMainLoop();
}
```

OUTPUT :



9.**Develop a menu driven program to fill any given polygon using scan-line area filling algorithm.**

```
#include <stdlib.h>
#include <stdio.h>
#include <glut.h>
float x1,x2,x3,x4,y1,y2,y3,y4;
```

```
int fillFlag=0;
void edgedetect(float x1,float y1,float x2,float y2,int *le, int *re)
{
float mx,x,temp;
int i;
        if((y2-y1)<0){
                temp=y1;y1=y2;y2=temp;
                temp=x1;x1=x2;x2=temp;
        }
        if((y2-y1)!=0)
                mx=(x2-x1)/(y2-y1);
        else
                mx=x2-x1;
        x=x1;
        for(i=y1;i<=y2;i++)
        {
                if(x<(float)le[i])
                        le[i]=(int)x;
                if(x>(float)re[i])
                        re[i]=(int)x;
                x+=mx;
        }
}
void draw_pixel(int x,int y)
{
        glColor3f(1.0,1.0,0.0);
        glBegin(GL_POINTS);
        glVertex2i(x,y);
        glEnd();
}
void scanfill(float x1,float y1,float x2,float y2,float x3,float y3,float x4,float y4)
{
        int le[500],re[500];
        int i,y;
        for(i=0;i<500;i++)
        {
                le[i]=500;
                re[i]=0;
        }
        edgedetect(x1,y1,x2,y2,le,re);
        edgedetect(x2,y2,x3,y3,le,re);
        edgedetect(x3,y3,x4,y4,le,re);
        edgedetect(x4,y4,x1,y1,le,re);
        for(y=0;y<500;y++)

       {

                        for(i=(int)le[y];i<(int)re[y];i++)
                                draw_pixel(i,y);
        }
}
        void display()
        {
```

```
x1=200.0;y1=200.0;x2=100.0;y2=300.0;x3=200.0;y3=400.0;x4=300.0;y4=300.0;
glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(0.0, 0.0, 1.0);
        glBegin(GL_LINE_LOOP);
                        glVertex2f(x1,y1);
                        glVertex2f(x2,y2);
                        glVertex2f(x3,y3);
                        glVertex2f(x4,y4);
                glEnd();
                if(fillFlag==1)
                scanfill(x1,y1,x2,y2,x3,y3,x4,y4);
        glFlush();
        }

        void init()
        {
                glClearColor(0.0,0.0,0.0,1.0);
                glColor3f(1.0,0.0,0.0);
                glPointSize(1.0);
                glMatrixMode(GL_PROJECTION);
                glLoadIdentity();
                gluOrtho2D(0.0,499.0,0.0,499.0);
        }
        void fillMenu(int option)
        {
                if(option==1)
                        fillFlag=1;
                if(option==2)
                        fillFlag=2;
                display();
        }
        void main(int argc, char* argv[])
        {
                glutInit(&argc,argv);
                glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
                glutInitWindowSize(500,500);
                glutInitWindowPosition(0,0);
                glutCreateWindow("Filling a Polygon using Scan-line Algorithm");
                init();
                glutDisplayFunc(display);
                glutCreateMenu(fillMenu);
                glutAddMenuEntry("Fill Polygon",1);
                glutAddMenuEntry("Empty Polygon",2);
                glutAttachMenu(GLUT_RIGHT_BUTTON);
                glutMainLoop();
        }
```

OUTPUT :