Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)
(**NAAC Accredited & ISO 9001:2015 Certified Institution)**
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

Computer Graphics Laboratory with Mini Project -18CSL67

Department of Computer Science & Engineering

VI Semester

COMPUTER GRAPHICS LABORATORY WITH MINI PROJECT (Effective from the academic year 2018 -2019) SEMESTER – VI			
Course Code	**18CSL67**	**CIE Marks**	40
Number of Contact Hours/Week	0:2:2	**SEE Marks**	60
Total Number of Lab Contact Hours	36	**Exam Hours**	03
Credits – 2			

Course Learning Objectives: This course (18CSL67) will enable students to:

- Demonstrate simple algorithms using OpenGL Graphics Primitives and attributes.
- Implementation of line drawing and clipping algorithms using OpenGL functions
- Design and implementation of algorithms Geometric transformations on both 2D and 3D objects.

Descriptions (if any): --

Installation procedure of the required software must be demonstrated, carried out in groups and documented in the journal.

Programs List:

PART A

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.	Draw a colour 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

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.

Laboratory Outcomes: The student should be able to:

- Apply the concepts of computer graphics

- Implement computer graphics applications using OpenGL
- Animate real world problems using OpenGL

Conduct of Practical Examination:

- Experiment distribution
    - For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
    - For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.
- Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
- Marks Distribution *(Courseed to change in accoradance with university regulations)*
    - o) For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 = 100 Marks
    - p) For laboratories having PART A and PART B
        - i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks
        - ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

Table of Contents

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 two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

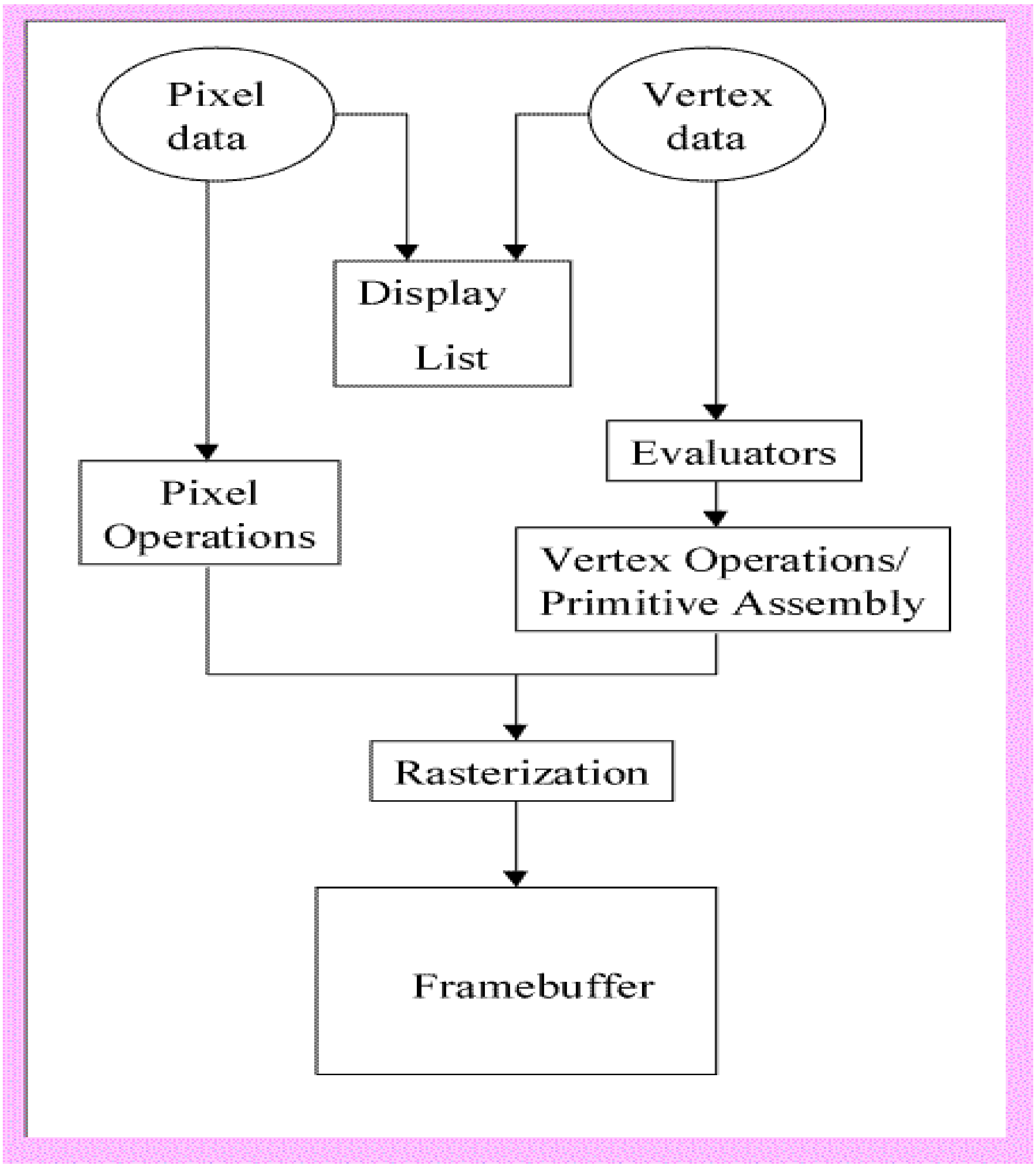
Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the OpenGL Utility Toolkit (GLUT) has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may 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.

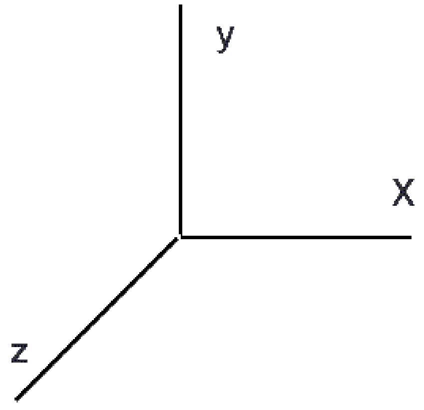
Polygons: Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points. In OpenGL, there 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 counterclockwise 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();
```

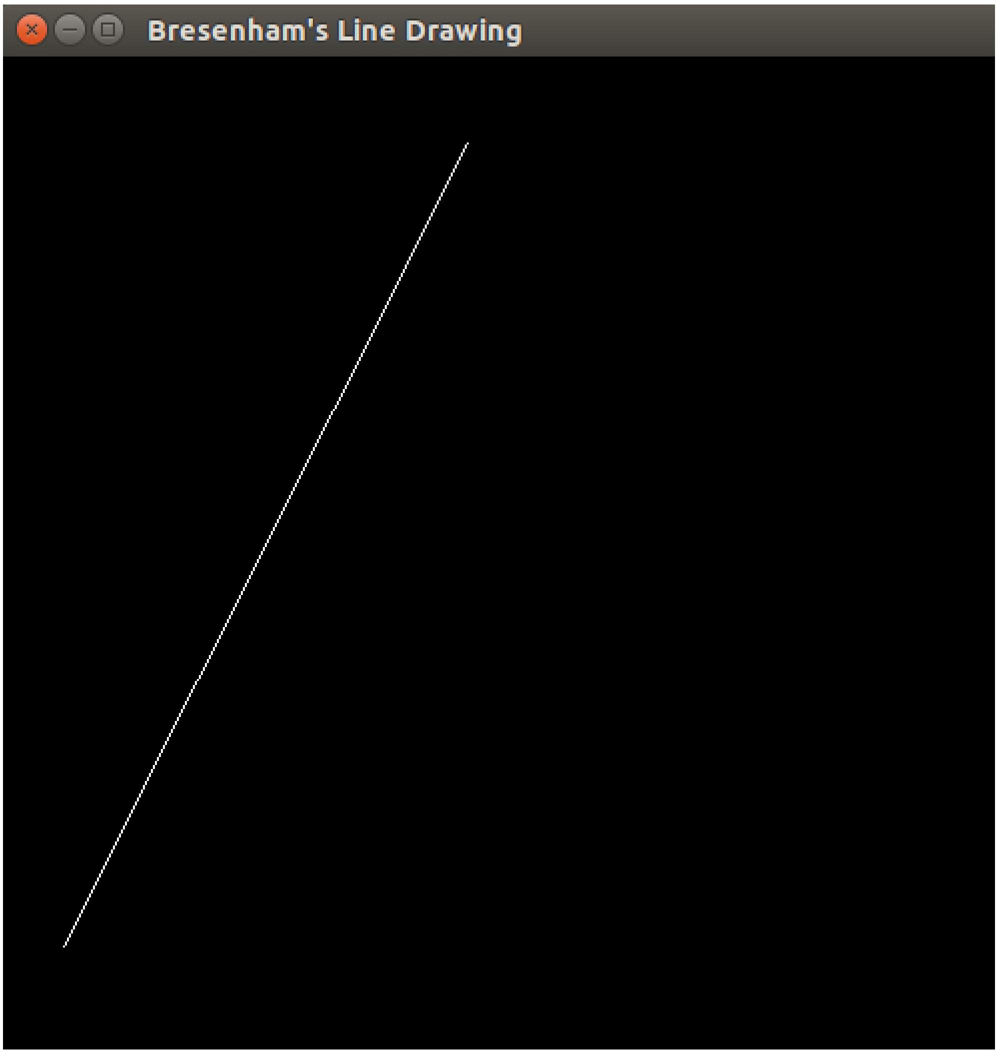
LAB PROGRAMS

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 myInit()
{
glClear(GL_COLOR_BUFFER_BIT);
glClearColor(0.0, 0.0, 0.0, 1.0);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0, 500, 0, 500);
}
void draw_pixel(int x, int y)
{
glBegin(GL_POINTS);
 glVertex2i(x, y);
glEnd();
}
void draw_line(int x1, int x2, int y1, int y2)
{
int dx, dy, i, e;
int incx, incy, inc1, inc2; int x,y;
dx = x2-x1;
dy = y2-y1;
if (dx < 0) dx = -dx;  if (dy < 0) dy = -dy; incx = 1;
if (x2 < x1) incx = -1;
incy = 1;
if (y2 < y1) incy = -1;
 x = x1; y = y1;
if (dx > dy)
{
draw_pixel(x, y);
e = 2 * dy-dx;
inc1 = 2*(dy-dx);
 inc2 = 2*dy;
for (i=0; i<dx; i++)
{
if (e >= 0)
{
y += incy; e += inc1;
}
```

```
else
e += inc2;
x += incx; draw_pixel(x, y);
}
}
else
{
draw_pixel(x, y);
e = 2*dx-dy;
inc1 = 2*(dx-dy);
inc2 = 2*dx;
for (i=0; i<dy; i++)
{
if (e >= 0)
{
x += incx; e += inc1;
}
else
e += inc2;
y += incy; draw_pixel(x, y);
}
}
}

void myDisplay()
{
draw_line(x1, x2, y1, y2); glFlush();
}
int main(int argc, char **argv)
{
printf( "Enter (x1, y1, x2, y2)\n");
scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
glutInit(&argc, argv); glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(500, 500);
glutInitWindowPosition(0, 0);
glutCreateWindow("Bresenham's Line Drawing");
myInit();
glutDisplayFunc(myDisplay);
glutMainLoop();
return 0;
}
```

Output:



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

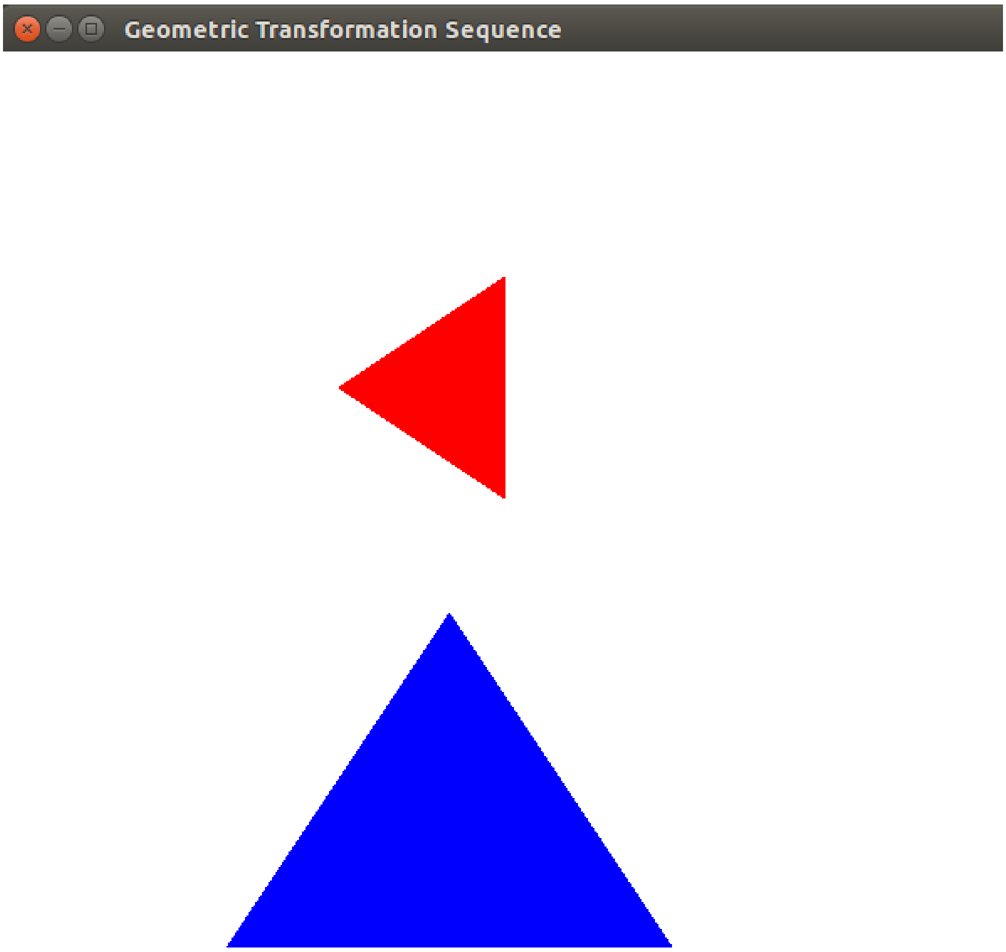
```
 #include <GL/glut.h>
 #include <stdlib.h>
 #include <math.h>
/*Set initial display-window size. */
 GLsizei winWidth = 600, winHeight = 600;
/*Set  range  for world coordinates.    */
GLfloat xwcMin = 0.0, xwcMax = 225.0;
 GLfloat ywcMin = 0.0, ywcMax = 225.0;
class wcPt2D
{
public:
GLfloat x, y;
};
typedef  GLfloat  Matrix3x3  [3][3];
 Matrix3x3 matComposite;
const GLdouble pi = 3.14159;
void init (void)
{
/*Set color of display window to white.         */
glClearColor (1.0, 1.0, 1.0, 0.0);
}
/* Construct the 3 x 3 identity matrix. */
void matrix3x3SetIdentity (Matrix3x3 matIdent3x3)
{
GLint row, col;
for (row = 0; row < 3; row++)
for (col = 0; col < 3; col++)
matIdent3x3 [row][col] = (row == col);
}
void matrix3x3PreMultiply (Matrix3x3 m1, Matrix3x3 m2)
{
GLint row, col;
Matrix3x3 matTemp;
for (row = 0; row <  3; row++)
for (col = 0; col < 3 ; col++)
matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *m2 [1][col] + m1 [row][2]
* m2  [2][col];
for (row = 0; row < 3; row++)
for (col = 0; col < 3; col++)
m2 [row][col] = matTemp [row][col];
}
void translate2D (GLfloat tx, GLfloat ty)
```

```
{
Matrix3x3 matTransl;
/*Initialize  translation  matrix  to  identity.*/
matrix3x3SetIdentity (matTransl);
matTransl [0][2] = tx; matTransl [1][2] = ty;
/*Concatenate matTransl with the  composite matrix.*/
matrix3x3PreMultiply (matTransl, matComposite);
}
void rotate2D (wcPt2D pivotPt, GLfloat theta)
{
Matrix3x3 matRot;
/*Initialize  rotation  matrix  to  identity.*/
matrix3x3SetIdentity (matRot);
matRot [0][0] = cos (theta); matRot [0][1] = -sin (theta);
matRot [0][2] = pivotPt.x * (1 - cos (theta)) + pivotPt.y * sin (theta);
matRot [1][0] = sin (theta); matRot [1][1] = cos (theta);
matRot [1][2] = pivotPt.y * (1 - cos (theta)) - pivotPt.x * sin (theta);
/*Concatenate  matRot  with  the  composite  matrix.*/
matrix3x3PreMultiply (matRot, matComposite);
}
void scale2D (GLfloat sx, GLfloat sy, wcPt2D fixedPt)
{
Matrix3x3 matScale;
/*Initialize  scaling  matrix  to  identity.*/
matrix3x3SetIdentity (matScale);
matScale  [0][0]  = sx;
matScale [0][2] = (1 - sx) * fixedPt.x; matScale [1][1] = sy;
matScale [1][2] = (1 - sy) * fixedPt.y;
/*Concatenate matScale with the  composite matrix.*/
matrix3x3PreMultiply (matScale, matComposite);
}
/* sing  the  composite  matrix,  calculate  transformed  coordinates.*/
void transformVerts2D (GLint nVerts, wcPt2D * verts)
{
GLint k;
GLfloat temp;
for (k = 0; k < nVerts; k++)
{
temp = matComposite [0][0] * verts [k].x + matComposite [0][1] * verts [k].y +
matComposite [0][2];
verts [k].y = matComposite [1][0] * verts [k].x + matComposite [1][1] * verts [k].y +
matComposite [1][2];
verts [k].x = temp;
}
```

```
}
void triangle (wcPt2D *verts)
{
GLint k;
glBegin (GL_TRIANGLES); for (k = 0; k < 3; k++)
glVertex2f (verts [k].x, verts [k].y); glEnd ( );
}
void displayFcn (void)
{
/*Define  initial  position for triangle.*/
GLint nVerts = 3;
wcPt2D verts [3] = { {50.0, 25.0}, {150.0, 25.0}, {100.0, 100.0} };
/*Calculate  position  of  triangle  centroid.*/
wcPt2D centroidPt;
GLint k, xSum = 0, ySum  =  0;
for  (k  =  0;  k <  nVerts;       k++)
{
xSum += verts [k].x;
ySum  +=  verts [k].y;
}
centroidPt.x = GLfloat (xSum) / GLfloat (nVerts);
centroidPt.y = GLfloat (ySum) / GLfloat (nVerts);
/*Set geometric transformation parameters.*/
wcPt2D pivPt,fixedPt;
pivPt = centroidPt; fixedPt = centroidPt;
GLfloat tx = 0.0, ty = 100.0;
GLfloat sx = 0.5, sy = 0.5;
GLdouble theta = pi/2.0;
glClear (GL_COLOR_BUFFER_BIT);        //         Clear display window.
glColor3f (0.0, 0.0, 1.0);        //         Set initial fill color to blue. triangle (verts);   //
          Display blue triangle.
/*Initialize  composite  matrix to identity.*/
matrix3x3SetIdentity (matCompositе);
/*Construct composite matrix for transformation sequence.*/
scale2D  (sx, sy, fixedPt);      //         First transformation: Scale. rotate2D  (pivPt,  theta);//
          Second transformation: Rotate translate2D  (tx,  ty); //         Final         transformation:
Translate.
/*Apply composite matrix to  triangle vertices.*/
 transformVerts2D (nVerts, verts);
glColor3f  (1.0, 0.0, 0.0);       // Set color for transformed triangle. triangle (verts);
glFlush ( );
}
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
```

```
glMatrixMode (GL_PROJECTION); glLoadIdentity ( );
gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax); glClear (GL_COLOR_BUFFER_BIT);
}
int main (int argc, char ** argv)
{
glutInit (&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowPosition (50, 50);
glutInitWindowSize (winWidth, winHeight);
glutCreateWindow ("Geometric Transformation Sequence");
init ( );
glutDisplayFunc (displayFcn);
glutReshapeFunc (winReshapeFcn);
glutMainLoop ( );
return 0;
}
```

Output:

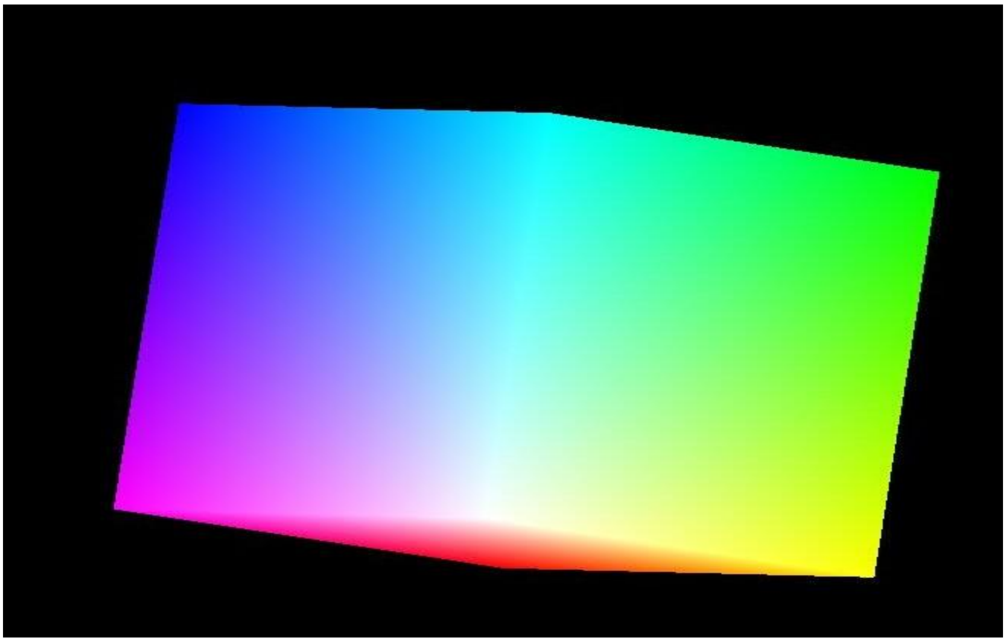


3. Draw a color cube and spin it using OpenGL transformation matrices

```
#include <stdio.h>
#include <stdlib.h>
GLfloat  vertices[  ]={ -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,-1.0,
1.0,1.0,-1.0, 1.0,1.0,  1.0, 1.0,-1.0, 1.0, 1.0 };
GLfloat  normals[  ]={ -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,-1.0, 1.0,1.0,-1.0, 1.0,1.0,  1.0, 1.0,-1.0, 1.0, 1.0  };
GLfloat  colors[  ]={0.0,0.0,0.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0,
                        1.0,0.0,1.0,1.0,1.0,1.0,0.0,1.0,1.0};
GLubyte   cubeIndices[]={0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4 };
static GLfloat theta[]={0.0,0.0,0.0};
static 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);
glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE,cubeIndices);
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_RIGHT_BUTTON && state==GLUT_DOWN) axis=1;
if(btn==GLUT_MIDDLE_BUTTON && state==GLUT_DOWN) axis=2;
}
void spincube()
{
theta[axis]+=2.0;
if(theta[axis]>360.0)
theta[axis]-=360.0;
glutPostRedisplay();
}
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);
}
int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500);
glutCreateWindow("color cuce");
glutReshapeFunc(myReshape);
glutDisplayFunc(display);
glutMouseFunc(mouse);
glutIdleFunc(spincube);
glEnable(GL_DEPTH_TEST);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glVertexPointer(3,GL_FLOAT,0,vertices);
glColorPointer(3,GL_FLOAT,0,colors);
glNormalPointer(GL_FLOAT,0,normals);
glColor3f(1.0,1.0,1.0);
glutMainLoop();
}
```

Output:



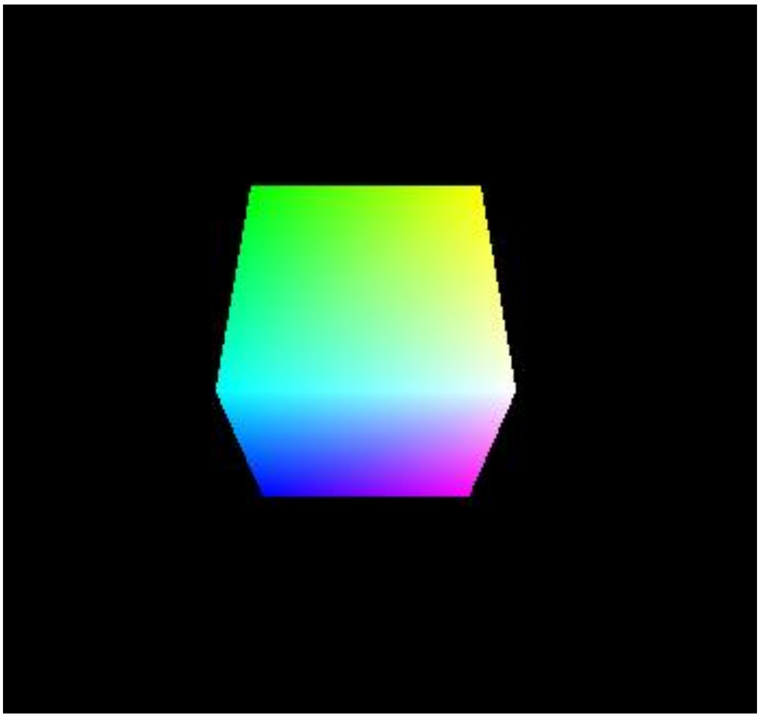
4. Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.

```
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
GLfloat vertices[ ]={ -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,-1.0, 1.0,
1.0,-1.0, 1.0,
1.0,  1.0, 1.0,
-1.0, 1.0, 1.0 };

GLfloat normals[ ] ={ -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,-1.0, 1.0,
1.0,-1.0, 1.0,
1.0,  1.0, 1.0,
-1.0, 1.0, 1.0  };

GLfloat colors[ ]={0.0,0.0,0.0,
1.0, 0.0, 0.0,
1.0,1.0,0.0,
0.0,1.0,0.0,
0.0,0.0,1.0,
1.0,0.0,1.0,
1.0,1.0,1.0,
0.0,1.0,1.0};

GLubyte cubeIndices[]={0,3,2,1, 2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0, 1, 5, 4 };
static GLfloat theta[]={0.0,0.0,0.0};
static GLint axis=2;
static GLdouble viewer[]={0.0,0.0,5.0};
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);
glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE,cubeIndices);
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_RIGHT_BUTTON && state==GLUT_DOWN) axis=1;
if(btn==GLUT_MIDDLE_BUTTON && state==GLUT_DOWN) axis=2;
theta[axis]+=2.0;
if(theta[axis]>360.0) theta[axis]-=360.0;
glutPostRedisplay();
}
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;
glutPostRedisplay();
}
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);
}
int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
 glutInitWindowSize(500,500);
glutCreateWindow("color cuce");
glutReshapeFunc(myReshape);
glutDisplayFunc(display);
glutKeyboardFunc(keys);
glutMouseFunc(mouse);
glEnable(GL_DEPTH_TEST);
```

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glVertexPointer(3,GL_FLOAT,0,vertices);
glColorPointer(3,GL_FLOAT,0,colors);
glNormalPointer(GL_FLOAT,0,normals);
glColor3f(1.0,1.0,1.0);
glutMainLoop();
}
```

Output:



5. Clip a line using Cohen-Sutherland algorithm

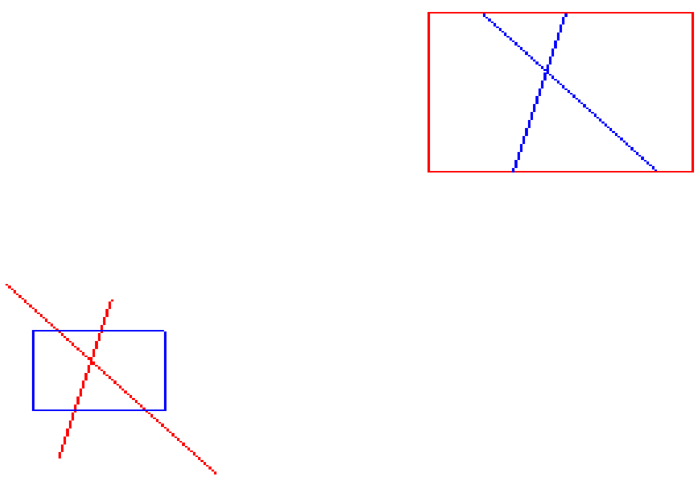
```
#include<stdio.h>
#include<stdbool.h>
#include<GL/glut.h>
#define outcode int
#define true 1
#define false 0
double xmin=50,ymin=50, xmax=100,ymax=100; // Window boundaries
double xvmin=200,yvmin=200,xvmax=300,yvmax=300; // Viewport boundaries
//int x1, x2, y1, y2;
//bit codes for the right, left, top, & bottom const int RIGHT = 8;
const int LEFT = 2; const int TOP = 4; const int BOTTOM = 1;
//used to compute  bit  codes  of  a  point outcode
ComputeOutCode (double x, double y);
//Cohen-Sutherland clipping algorithm clips a line from
//P0 = (x0, y0) to P1 = (x1, y1) against a rectangle with
//diagonal from (xmin, ymin) to (xmax, ymax).
void CohenSutherlandLineClipAndDraw (double x0, double y0,double x1, double y1)
{
//Outcodes for P0, P1, and whatever point lies outside the clip rectangle outcode
outcode0, outcode1, outcodeOut;
bool accept = false, done = false;
 //compute outcodes
outcode0 = ComputeOutCode (x0, y0); outcode1 = ComputeOutCode (x1, y1); do
{
if (!(outcode0 | outcode1))      //logical or is 0 Trivially accept & exit
{
accept = true; done = true;
}
else if (outcode0 & outcode1)//logical and is not 0. Trivially reject and exit done =
true;
else
{
//failed both tests, so calculate the line segment to clip
//from an outside point to an intersection with clip edge double x, y;

//At least one endpoint is outside the clip rectangle; pick it. outcodeOut = outcode0?
outcode0: outcode1;

//Now find the intersection point;
//use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope)* (y - y0) if(outcodeOut &
TOP)   //point is above the clip rectangle
```

```
{
x = x0 + (x1 - x0) * (ymax - y0)/(y1 - y0);y = ymax;
}
else  if(outcodeOut & BOTTOM)      //point is below the clip rectangle
{
x = x0  +  (x1  - x0)  *  (ymin  - y0)/(y1  - y0); y = ymin;
}
else  if(outcodeOut & RIGHT)          //point is to the right of clip rectangle
{
y = y0 + (y1 - y0) * (xmax - x0)/(x1 - x0);x = xmax;
}
else      //point is to the left of clip rectangle
{
y = y0  +  (y1  - y0)  *  (xmin  - x0)/(x1  - x0); x = xmin;
}

//Now we move outside point to intersection point to clip
//and get ready for next pass. if (outcodeOut == outcode0)
{
 x0 = x; y0 = y;
outcode0 = ComputeOutCode (x0, y0);
}
else
{
x1 = x; y1 = y;
outcode1 = ComputeOutCode (x1, y1);
}
}
}while (!done);

if (accept)
{
// Window to viewport mappings
double sx=(xvmax-xvmin)/(xmax-xmin); // Scale parameters
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;
//draw a red colored viewport
glColor3f(1.0, 0.0, 0.0); glBegin(GL_LINE_LOOP);
glVertex2f(xvmin, yvmin);
glVertex2f(xvmax, yvmin);
glVertex2f(xvmax, yvmax);
```

```
glVertex2f(xvmin, yvmax);
glEnd();
glColor3f(0.0,0.0,1.0); // draw blue colored clipped line glBegin(GL_LINES);
glVertex2d (vx0, vy0);
glVertex2d (vx1, vy1);
glEnd();
}
}

//Compute the bit code for a point (x, y) using the clip rectangle
//bounded diagonally by (xmin, ymin), and (xmax, ymax) outcode ComputeOutCode
(double x, double y)
{
outcode 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 display()
{
double x0=120,y0=10,x1=40,y1=130;
glClear(GL_COLOR_BUFFER_BIT);
//draw the line with red color
glColor3f(1.0,0.0,0.0);
//bres(120,20,340,250);
glBegin(GL_LINES);
glVertex2d (x0, y0);
glVertex2d (x1, y1);
glVertex2d (60,20);
glVertex2d (80,120);
glEnd();

//draw a blue colored window
glColor3f(0.0, 0.0, 1.0);
glBegin(GL_LINE_LOOP);
glVertex2f(xmin, ymin);
glVertex2f(xmax, ymin);
glVertex2f(xmax, ymax);
```

```
glVertex2f(xmin, ymax);
glEnd();
CohenSutherlandLineClipAndDraw(x0,y0,x1,y1);
CohenSutherlandLineClipAndDraw(60,20,80,120);
glFlush();
}
void myinit()
{
glClearColor(1.0,1.0,1.0,1.0);
glColor3f(1.0,0.0,0.0);
glPointSize(1.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,499.0,0.0,499.0);
}
int main(int argc, char** argv)
{
//printf("Enter End points:");
//scanf("%d%d%d%d", &x1,&x2,&y1,&y2);
glutInit(&argc,argv); glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
glutCreateWindow("Cohen Suderland Line Clipping Algorithm");
glutDisplayFunc(display);
myinit();
glutMainLoop();
}
```

Output:



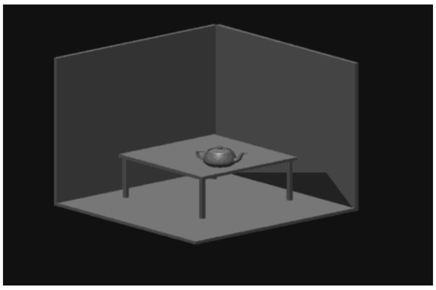
6. To draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the surfaces of the solid object used in the scene

```
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
void  wall(double  thickness)
{
glPushMatrix();
glTranslated(0.5,0.5*thickness,0.5);
glScaled(1.0,thickness,1.0);
glutSolidCube(1.0);
glPopMatrix();
}
void tableleg(double thick,double len)
{
glPushMatrix();
glTranslated(0,len/2,0);
glScaled(thick,len,thick);
glutSolidCube(1.0);
glPopMatrix();
}
void table(double topw,double topt,double legt,double legl)
{
glPushMatrix();
glTranslated(0,legl,0);
glScaled(topw,topt,topw);
glutSolidCube(1.0);
glPopMatrix();
double dist=0.95*topw/2.0-legt/2.0;
glPushMatrix();
glTranslated(dist,0,dist);
tableleg(legt,legl);
```

```
glTranslated(0,0,-2*dist);
tableleg(legt,legl);
glTranslated(-2*dist,0,2*dist);
tableleg(legt,legl);
glTranslated(0,0,-2*dist);
tableleg(legt,legl);
glPopMatrix();
}
void displaysolid(void)
{
GLfloat mat_ambient[]={0.7f,0.7f,0.7f,1.0f};
GLfloat mat_diffuse[]={0.5f,0.5f,0.5f,1.0f};
GLfloat mat_specular[]={1.0f,1.0f,1.0f,1.0f};
GLfloat mat_shininess[]={50.0f};
glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
GLfloat lightint[]={0.7f,0.7f,0.7f,1.0f};
GLfloat lightpos[]={2.0f,6.0f,3.0f,0.0f};
glLightfv(GL_LIGHT0,GL_POSITION,lightpos);
glLightfv(GL_LIGHT0,GL_DIFFUSE,lightint);
glMatrixMode(GL_PROJECTION); glLoadIdentity();
double winht=1.0;
glOrtho(-winht*64/48.0,winht*64/48.0,-winht,winht,0.1,100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.3,1.3,2.0,0.0,0.25,0.0,0.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glPushMatrix(); glRotated(90.0,0.0,0.0,1.0);
wall(0.02); glPopMatrix();
wall(0.02); glPushMatrix();
glRotated(-90.0,1.0,0.0,0.0);
wall(0.02); glPopMatrix();
```

```
glPushMatrix();
glTranslated(0.4,0,0.4); table(0.6,0.02,0.02,0.3);
glPopMatrix();
glPushMatrix();
glTranslated(0.6,0.38,0.5);
glRotated(30,0,1,0);
glutSolidTeapot(0.08);
glPopMatrix(); glFlush();
}
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500);
glutInitWindowPosition(0,0);
glutCreateWindow("teapot");
glutDisplayFunc(displaysolid);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glShadeModel(GL_SMOOTH);
glEnable(GL_DEPTH_TEST);
glEnable(GL_NORMALIZE);
glClearColor(0.1,0.1,0.1,0.0);
glViewport(0,0,640,480);
glutMainLoop();
}
```

Output:



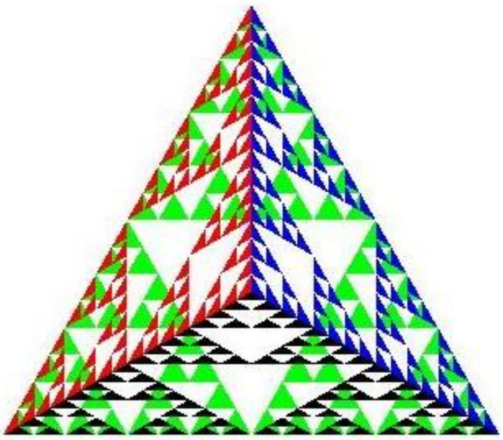
7. Design, develop and implement recursively subdivide a tetrahedron to form 3D sierpinski gasket. The number of recursive steps is to be specified by the user

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
typedef float point[3];
/* initial tetrahedron */
point v[]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},{-0.816497, -0.471405, -0.333333},
{0.816497, -0.471405, -0.333333}};
static GLfloat theta[] = {0.0,0.0,0.0};
int n;
void triangle( point a, point b, point c)
/* display one triangle using a line loop for wire frame, a single
normal for constant shading, or three normals for interpolative shading */
{
glBegin(GL_POLYGON);
 glNormal3fv(a);
glVertex3fv(a);
glVertex3fv(b);
glVertex3fv(c);
glEnd();
}
void divide_triangle(point a, point b, point c, int m)
{        /* triangle subdivision using vertex numbers
Righth and rule applied to create outward pointing faces */
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)
{        /* Apply triangle subdivision to faces of tetrahedron */
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);
glutPostRedisplay();
}
void main(int argc, char **argv)
{
printf(" No. 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:


"C:\Program Files\Microsoft Vis...
Enter the number of recursion
4

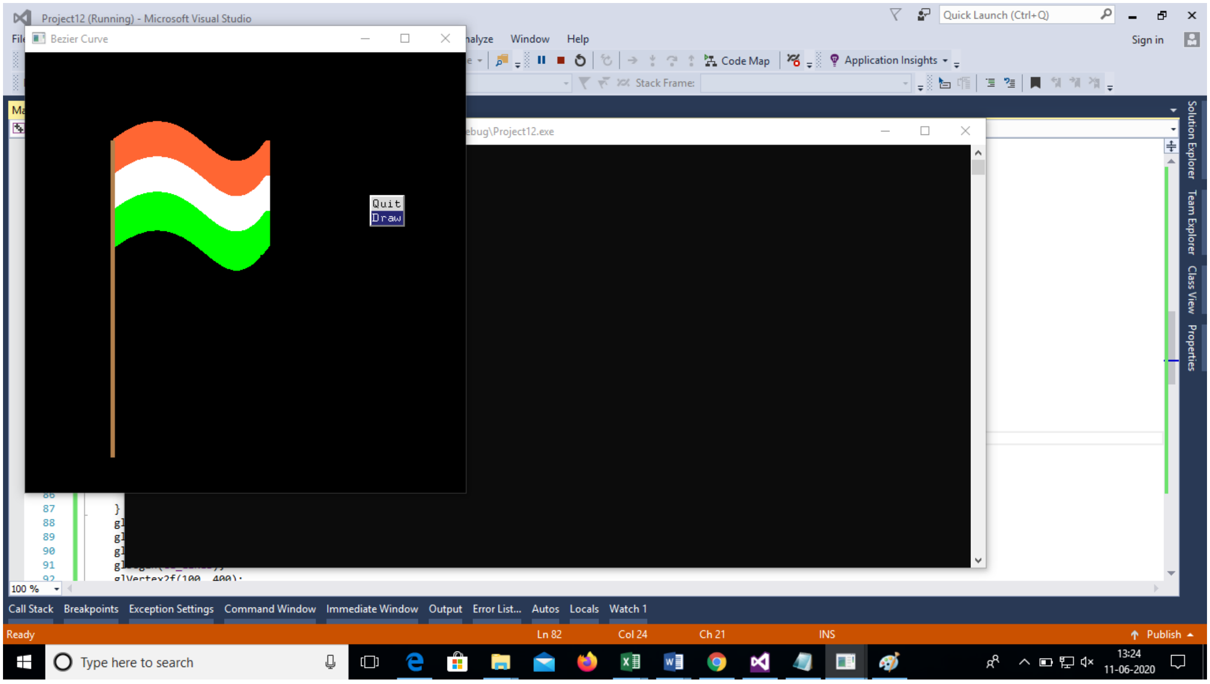


8. Develop a menu driven program to animate flag using Bezier Curve algorithm

```
#include<GL/glut.h>
#include<math.h>
#include<stdio.h>
void bezierCoefficients(int n,int *c)
{
int k,i; for(k=0;k<=n;k++)
{
c[k]=1;
for(i=n;i>=k+1;i--) c[k]*=i;
for(i=n-k;i>=2;i--) c[k]/=i;
}
}
void display()
{
int cp[4][2]={{10,10},{100,200},{200,50},{300,300}};
int c[4],k,n=3;
float x,y,u,blend;
bezierCoefficients(n,c);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
glLineWidth(5.0);
glBegin(GL_LINE_STRIP);
for(u=0;u<1.0;u+=0.01)
{
x=0;y=0;
for(k=0;k<4;k++)
{
blend=c[k]*pow(u,k)*pow(1-u,n-k);
x+=cp[k][0]*blend;
y+=cp[k][1]*blend;
}
glVertex2f(x,y);
}
glEnd();
glFlush();
}
void myinit()
{
glClearColor(1.0,1.0,1.0,1.0);
glColor3f(1.0,0.0,0.0);
glPointSize(5.0);
gluOrtho2D(0.0,600,0.0,600.0);
}
int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
```

```
glutInitWindowSize(600,600);
glutCreateWindow("Bezier Curve");
glutDisplayFunc(display);
myinit();
glutMainLoop();
return 0;
}
```

Output:

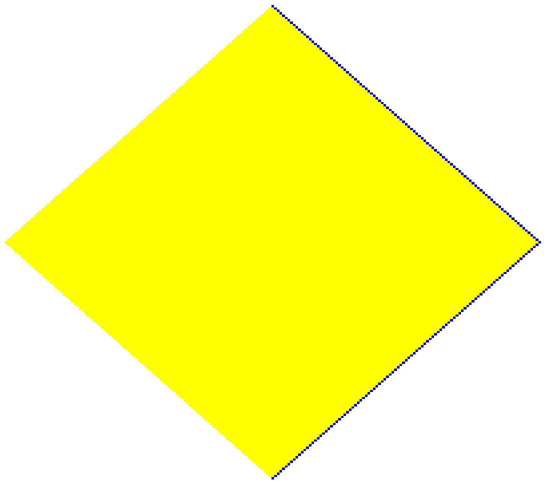


9. Develop a menu driven program to fill the polygon using scan line algorithm

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
float x1,x2,x3,x4,y1,y2,y3,y4;
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++)
{
if(le[y]<=re[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();
scanfill(x1,y1,x2,y2,x3,y3,x4,y4);
glFlush();
}
void myinit()
{
glClearColor(1.0,1.0,1.0,1.0);
glColor3f(1.0,0.0,0.0);
glPointSize(1.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,499.0,0.0,499.0);
}
int 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");
glutDisplayFunc(display);
myinit();
glutMainLoop();
}
```

Output:



Viva Questions & Answers

1. Define Computer graphics?
   Computer graphics are graphics created by computers and, more generally, the representation and manipulation of pictorial data by a computer

2. Define Computer Animation?
   Computer animation is the art of creating moving images via the use of computers. It is a subfield of **computer graphics** and animation

3. Define Pixel?
   The word *pixel* is based on a contraction of *pix* ("pictures") and *el* (for "element"). Pixels are normally arranged in a 2-dimensional grid, and are often represented using dots, squares, or rectangles

4. Define Raster graphics?
   Raster Graphics, which is the representation of images as an array of pixels, as it is typically used for the representation of photographic images.

5. Define Image?
   An **image** (from Latin *imago*) is an artifact, usually two-dimensional (a **picture**), that has a similar appearance to some subject—usually a physical object or a person.

6. Define Rendering?
   Rendering is the process of generating an image from a model, by means of computer programs.

7. Define Ray Tracing?
   Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane.

8. Define Projection?
   Projection is a method of mapping points to a plane.

9. Define 3D Projection?
   3D projection is a method of mapping three dimensional points to a two dimensional plane.

10. Define Texture Mapping?
    Texture mapping is a method for adding detail, surface texture, or color to a computer-generated graphic or 3D model.

11. Define Vector Graphics?
    Vector graphics formats are complementary to raster graphics, which is the representation of images as an array of pixels, as it is typically used for the representation of photographic images.

12. Define Pinhole camera model?
    The **pinhole camera model** describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane of an *ideal* pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light.

13. Define digital image?
    A **digital image** is a representation of a two-dimensional image using ones and zeros (binary). Without qualifications, the term "digital image" usually refers to raster images.

14. Define Grayscale?
    A **grayscale** or **greyscale** digital image is an image in which the value of each pixel is a single sample, that is, it carries only intensity information.

15. Define Pinhole Camera?
    A **pinhole camera** is a very simple camera with no lens and a single very small aperture. Simply explained, it is a light-proof box with a small hole in one side.

16. Define CCD?
    A **charge-coupled device** (**CCD**) is an analog shift register that enables the transportation of analog signals (electric charges) through successive stages (capacitors), controlled by a clock signa.

17. Define Image Resolution?
    Image resolution describes the detail an image holds. The term applies equally to digital images, film images, and other types of images. Higher resolution means more image detail.

18. Define Wavelength?
    Wavelength is the distance between repeating units of a propagating wave of a given frequency.

19. What is an Electromagnetic Spectrum?
    The **electromagnetic (EM) spectrum** is the range of all possible electromagnetic radiation frequencies. The "electromagnetic spectrum" (usually just *spectrum*) of an object is the characteristic distribution of electromagnetic radiation from that particular object.

20. Define Halftone?
    Halftone is the reprographic technique that simulates continuous tone imagery through the use of dots, varying either in size or in spacing. 'Halftone' can also be used to refer specifically to the image that is produced by this process.

21. Define WCS?
    World Coordinate Systems. While every point in the FITS data array can be located in the coordinate system determined by the array axes. **World Coordinate** Systems (WCS) are any **coordinate** systems that describe the physical **coordinate** associated with a data array, such as sky **coordinates.**

22. What is OpenGL?
    OpenGL(R) is the software interface for graphics hardware that allows graphics programmers to produce high-quality color images of 3D objects. OpenGL is a rendering only, vendor neutral API providing 2D and 3D graphics functions, including modeling, transformations, color, lighting, smooth shading, as well as advanced features like texture mapping, NURBS, fog, alpha blending and motion blur. OpenGL works in both immediate and retained (display list) graphics modes.
    OpenGL is window system and operating system independent. OpenGL has been integrated with Windows NT and with the X Window System under UNIX. Also, OpenGL is network transparent. A defined common extension to the X Window System allows an OpenGL client on one vendor's platform to run across a network to another vendor's OpenGL server

23. What hardware supports the OpenGL API?

    Many vendors have developed or are developing implementations of the OpenGL API for a variety of embedded hardware devices including aircraft avionics, PDAs (personal digital assistants such as Palm™), cellular phones, game consoles (Sony Playstation® 2), television set-top boxes, and display devices (X-Terms and network computers). The small size of the OpenGL API, its open nature, and now free use of the sample implementation make the OpenGL API an ideal graphics library for these types of applications.

24. What are the benefits of OpenGL for hardware and software developers?

    - Industry standard
    - Reliable and portable.
    - Evolving.
    - Scalable.
    - Easy to use.
    - Well-documented.

25. What is the GLUT Toolkit?
    GLUT is a portable toolkit which performs window and event operations to support OpenGL rendering

26. How does the camera work in OpenGL?
    As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0.0, 0.0, 0.0). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation by placing it on the MODELVIEW matrix. This is commonly referred to as the viewing transformation. In practice this is mathematically equivalent to a camera transformation but more efficient because model transformations and camera transformations are concatenated to a single matrix. As a result though, certain operations must be performed when the camera and only the camera is on the MODELVIEW matrix. For example to position a light source in world space it most be positioned while the viewing transformation and only the viewing transformation is applied to the MODELVIEW matrix.

27. How do I implement a zoom operation?

    A simple method for zooming is to use a uniform scale on the ModelView matrix. However, this often results in clipping by the zNear and zFar clipping planes if the model is scaled too large. A better method is to restrict the width and height of the view volume in the Projection matrix. For example, your program might maintain a zoom factor based on user input, which is a floating-point number. When set to a value of 1.0, no zooming takes place. Larger values result in greater zooming or a more restricted field of view, while smaller values cause the opposite to occur.

28. How do I get a specified point (XYZ) to appear at the center of the scene?

    gluLookAt() is the easiest way to do this. Simply set the X, Y, and Z values of your point as the fourth, fifth, and sixth parameters to gluLookAt ().

29. Define Instruction Pipeline?

    An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

30. Define Geometric Pipeline?

    Geometric manipulation of modeling primitives, such as that performed by a **Geometry Pipeline**, is the first stage in computer graphics systems which perform image generation based on geometric models.

31. Define Perspective Projection?

    Perspective (from Latin *perspicere*, to see through) in the graphic arts, such as drawing, is an approximate representation, on a flat surface (such as paper), of an image as it is perceived by the eye. The two most characteristic features of perspective are that objects are drawn.