Q No. 1: Implement Brenham's line drawing algorithm for all types of slope.

DESCRIPTION

To display screen where the straight line segments are to be drawn. The vertical axes show scan-line position and the horizontal axes identify pixel columns

OBJECTIVE

- To understand Brenham's line drawing algorithm

OUTCOME

- Students will be able to draw a line for all types of slope.

ALGORITHM

Bresenham's Line-Drawing Algorithm for $|m| < 1.0$

1. Input the two line endpoints and store the left endpoint in ($x0$, $y0$).
2. Set the color for frame-buffer position ($x0$, $y0$); i.e., plot the first point.
3. Calculate the constants _x, _y, 2_y, and 2_y − 2_x, and obtain the starting value for the decision parameter as

$$p0 = 2\_y - \_x$$

4. At each xk along the line, starting at $k = 0$, perform the following test:
If $pk < 0$, the next point to plot is (xk + 1, yk) and

$$pk{+}1 = pk + 2\_y$$

Otherwise, the next point to plot is (xk + 1, yk + 1) and

$$pk{+}1 = pk + 2\_y - 2\_x$$

5. Repeat step 4 _x − 1 more times.

PROGRAM

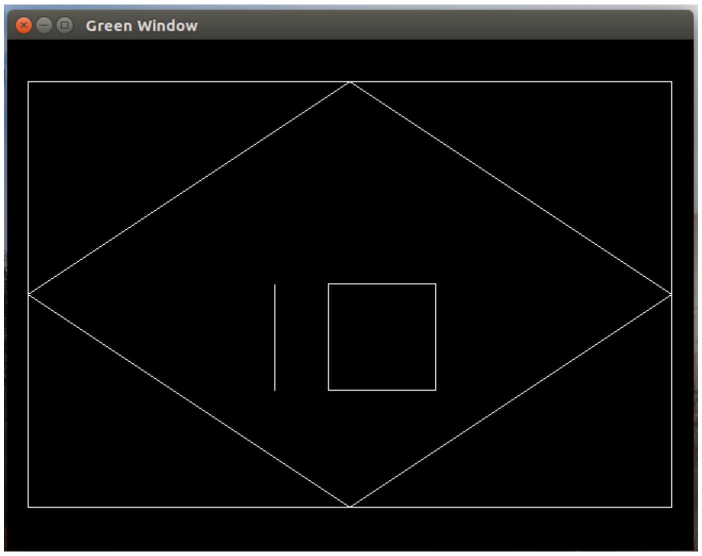
```
#include<stdio.h>
#include<GL/gl.h>
#include<GL/glut.h>

/* Function that returns -1,0,1 depending on whether x */
/* is <0, =0, >0 respectively */
#define sign(x) ((x>0)?1:((x<0)?-1:0))
```

```
/* Function to plot a point */
void setPixel(GLint x, GLint y) {
  glBegin(GL_POINTS);
  glVertex2i(x,y);
  glEnd();
}

/* Generalized Bresenham's Algorithm */
void bres_general(int x1, int y1, int x2, int y2)
{
  int dx, dy, x, y, d, s1, s2, swap=0, temp;

  dx = abs(x2 - x1);
  dy = abs(y2 - y1);
  s1 = sign(x2-x1);
  s2 = sign(y2-y1);

  /* Check if dx or dy has a greater range */
  /* if dy has a greater range than dx swap dx and dy */
  if(dy > dx){temp = dx; dx = dy; dy = temp; swap = 1;}

  /* Set the initial decision parameter and the initial point */
  d = 2 * dy - dx;
  x = x1;
  y = y1;

  int i;
  for(i = 1; i <= dx; i++)
  {
    setPixel(x,y);

    while(d >= 0)
    {
      if(swap) x = x + s1;
      else
      {
        y = y + s2;
        d = d - 2* dx;
      }
    }
    if(swap) y = y + s2;
    else x = x + s1;
    d = d + 2 * dy;
  }
  glFlush();
}
```

```
/* Function to draw a rhombus inscribed in a rectangle and roll */
/* number printed in it */
void draw(void)
{
  glClear(GL_COLOR_BUFFER_BIT);

  /* Draw rectangle */
  bres_general(20,40,620,40);
  bres_general(620,40,620,440);
  bres_general(620,440,20,440);
  bres_general(20,440,20,40);

  /* Draw rhombus */
  bres_general(320,440,20,240);
  bres_general(20,240,320,40);
  bres_general(320,40,620,240);
  bres_general(620,240,320,440);

  /* 1 */
  bres_general(250,150,250,250);
  /* 0 */
  bres_general(300,150,300,250);
  bres_general(300,250,400,250);
  bres_general(400,250,400,150);
  bres_general(400,150,300,150);

  glFlush();
}

void init() {
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  glutInitWindowPosition(0,0);
  glutInitWindowSize(640, 480);
  glutCreateWindow("Green Window");
  glClearColor(0.0,0.0,0.0,0);
  glColor3f(1.0,1.0,1.0);
  gluOrtho2D(0,640,0,480);

}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  init();
  glutDisplayFunc(draw);
  glutMainLoop();
  return 0;
}
```

SAMPLE INPUT AND OUTPUT

```
 gcc -o 1 1.c -lGL -lGLU -lglut
./1
```



Q No. 2: Create and rotate a triangle about the origin and a fixed point.

DESCRIPTION

A triangle is transformed from its original position to the desired transformation using the composite-matrix calculations in procedure transformVerts2D.

OBJECTIVE

- To understand rotation of a triangle about the origin and a fixed point
- To understand the method of drawing triangle object using simple GL primitives.

OUTCOME

- Students will be able to rotate triangle.

PROGRAM

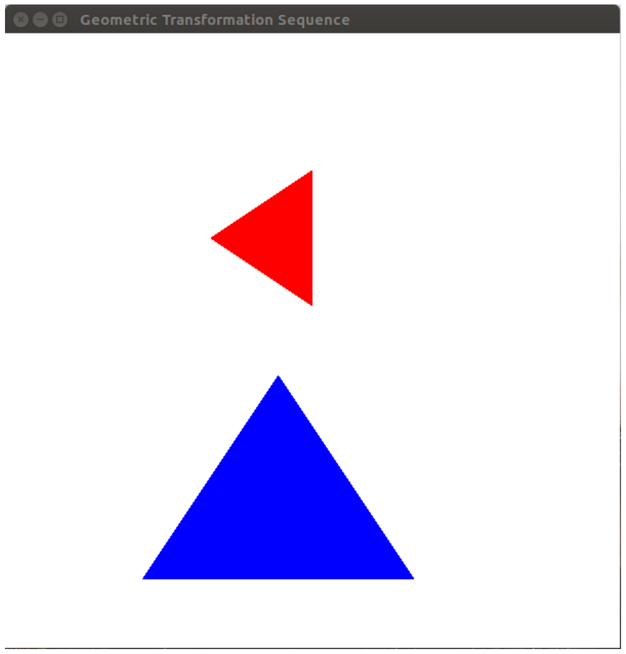
```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/* 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);
}

/* Premultiply matrix m1 times matrix m2, store result in m2. */
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);
}
/* Using 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(matComposite);
/* 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); // Display red transformed.
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 ( );
}
```

SAMPLE INPUT AND OUTPUT

```
g++ -o 2 2.c -lGL -lGLU -lglut
./2
```



Q No. 3: Draw a colour cube and spin it using OpenGL transformation matrices.

DESCRIPTION

The program draws a cube using GL_POLYGON primitive. For each vertex of the cube different color is given so that the cube becomes colorful. Whenever mouse left, right, or middle buttons are pressed the cube will spin along x axis, y axis or z axis respectively.

OBJECTIVE

- To understand rotation transformation
- To understand the method of drawing 3D objects using simple GL primitives.

OUTCOME

- Students will be able to animate 3D colored objects.

ALGORITHM

1. Initialize vertices, normal's and colors of the color cube. Also initialize theta and axis.
2. Define main()
    a. Enable display mode as double and depth buffer.
    b. Register mouse and idle callback function.
    c. Enable depth test.
3. Define reshape () enabling glOrtho.
4. Define mouse callback mouse(), coding for left, middle and right buttons for x, y and z axis respectively.
5. Define idle callback spincube()
    a. Increment theta for every click.
    b. Reinitialize theta
        If theta > 360 then theta → 0
    c. Call display()
6. Define callback function display()
    a. Load identity matrix.
    b. Define rotation function for x, y & z axis.
    c. Draw the color cube using right hand rule.
    d. Swap buffers for there is animation in the scene.

PROGRAM

```
/* rotating cube with color interpolation

 Demonstration of use of homogeneous coordinate transformations and simple data structure for
representing cube from Chapter 4
```

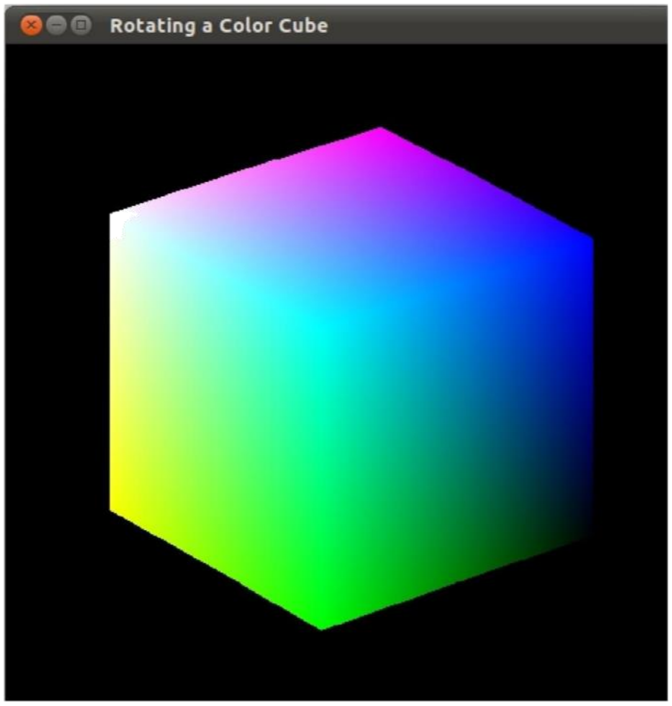
Both normals and colors are assigned to the vertices. Cube is centered at origin so (unnormalized) normals are the same as the vertex values */

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

GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-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[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-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[][3] = {{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}};

void polygon(int a, int b, int c , int d)
{
/* draw a polygon via list of vertices */
        glBegin(GL_POLYGON);
                glColor3fv(colors[a]);
                glNormal3fv(normals[a]);
                glVertex3fv(vertices[a]);
                glColor3fv(colors[b]);
                glNormal3fv(normals[b]);
                glVertex3fv(vertices[b]);
                glColor3fv(colors[c]);
                glNormal3fv(normals[c]);
                glVertex3fv(vertices[c]);
                glColor3fv(colors[d]);
                glNormal3fv(normals[d]);
                glVertex3fv(vertices[d]);
        glEnd();
}
void colorcube(void)
{
/* map vertices to faces */
        polygon(0,3,2,1);
        polygon(2,3,7,6);
        polygon(0,4,7,3);
        polygon(1,2,6,5);
        polygon(4,5,6,7);
        polygon(0,1,5,4);
}

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

void display(void)
{
/* display callback, clear frame buffer and z buffer, rotate cube and draw, swap buffers */

        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();
        glFlush();
        glutSwapBuffers();
}

void spinCube()
{
/* Idle callback, spin cube 1 degrees about selected axis */

        theta[axis] += 1.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
/* mouse callback, selects an axis about which to rotate */
        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);

/* need both double buffering and z buffer */
   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();
}
```

SAMPLE INPUT AND OUTPUT

```
cc -o 3 3.c -lGL -lGLU -lglut
./3
```



Q No. 4: Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.

DESCRIPTION

The program creates a cube using GL_POLYGON primitive. For each vertex of the cube different color is given so that the cube looks colorful. x, X, y, Y, z and Z keys are used to move camera for perspective viewing. Whenever mouse left, right, or middle buttons are pressed the cube will rotate along x axis, y axis or z axis respectively.

OBJECTIVE

- Understand perspective viewing model with respect to camera movements.

OUTCOME

- Students will be able to implement perspective viewing model with respect to camera movements.

ALGORITHM

1. Initialize vertices, normals and colors of the color cube. Also initialize theta, axis and viewer position.
2. Define main()
   a. Enable display mode as double and depth buffer.
   b. Register mouse and keyboard callback function.
   c. Enable depth test.
3. Define reshape(). Enable either glFrustum or gluPerspective for perspective viewing.
4. Define mouse callback mouse()
   a. Program for left, middle and right buttons for x, y and z axis respectively.
   b. Increment theta for every click.
   c. Reinitialize theta
      If theta > 360 then theta → 0
   d. Call display()
5. Define keyboard callback keys()
   Program for moving the viewer towards or away from the object by incrementing or decrementing by 1, for x, y & z keys for respective directions accordingly.
6. Define callback function display()
   a. Load identity matrix and gluLookAt for viewer.
   b. Define rotation function for x, y & z axis.
   c. Draw the colorcube using righthand rule.
   d. Swap buffers for there is animation in the scene.

PROGRAM

```
/* Rotating cube with viewer movement */
/* Use Lookat function in the display callback to point the viewer, whose position can be altered by the
x,X,y,Y,z, and Z keys. The perspective view is set in the reshape callback */

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

GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-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[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-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[][3] = {{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}};

void polygon(int a, int b, int c , int d)
{
        glBegin(GL_POLYGON);
                glColor3fv(colors[a]);
                glNormal3fv(normals[a]);
                glVertex3fv(vertices[a]);
                glColor3fv(colors[b]);
                glNormal3fv(normals[b]);
                glVertex3fv(vertices[b]);
                glColor3fv(colors[c]);
                glNormal3fv(normals[c]);
                glVertex3fv(vertices[c]);
                glColor3fv(colors[d]);
                glNormal3fv(normals[d]);
                glVertex3fv(vertices[d]);
        glEnd();
}

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

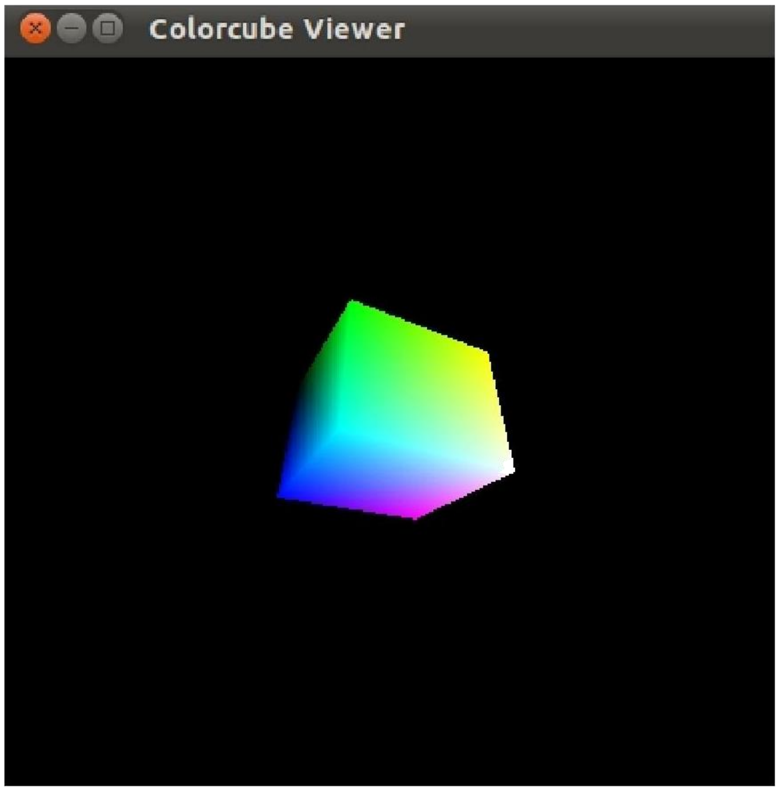
```
}

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

void display(void)
{

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* Update viewer position in modelview matrix */

glLoadIdentity();
gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

/* rotate cube */

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)
{
/* Use x, X, y, Y, z, and Z keys to move viewer */
```

```
   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);

/* Use a perspective view */

 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);

/* Or we can use gluPerspective */
 /* gluPerspective(45.0, w/h, -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("Colorcube Viewer");
 glutReshapeFunc(myReshape);
 glutDisplayFunc(display);
           glutMouseFunc(mouse);
           glutKeyboardFunc(keys);
           glEnable(GL_DEPTH_TEST);
 glutMainLoop();
```

```
}
```

SAMPLE INPUT AND OUTPUT

```
cc -o 4 4.c -lGL -lGLU -lglut
./4
```



Q No. 5: Clip a lines using Cohen-Sutherland algorithm

DESCRIPTION

The program draws a clipping window with the line which has to be clipped and calls Cohen-Sutherland line-clipping algorithm. The algorithm divides a two-dimensional space into 9 regions, and then efficiently determines the lines and portions of lines that are visible in the center region of interest (the viewport). The clipped line and clipping window will be scaled and displayed as output.

OBJECTIVE

- To understand the concept of clipping
- Learning Cohen-Sutherland line-clipping algorithm

OUTCOME

- Students will be able to implement Cohen Sutherland line clipping algorithm and compare the same with Liang barsky algorithm.

ALGORITHM

1. Given a line segment with endpoint $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$
2. Compute the 4-bit codes for each endpoint.

   If both codes are **0000**,(bitwise OR of the codes yields 0000) line lies completely **inside** the window: pass the endpoints to the draw routine.

   If both codes have a 1 in the same bit position (bitwise AND of the codes is **not** 0000), the line lies **outside** the window. It can be trivially rejected.

3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be **clipped** at the window edge before being passed to the drawing routine.
4. Examine one of the endpoints, say $P_1 = (x_1, y_1)$. Read P_1's 4-bit code in order: **Left**-to-**Right**, **Bottom**-to-**Top**.
5. When a set bit (1) is found, compute the **intersection I** of the corresponding window edge with the line from P_1 to P_2. Replace P_1 with **I** and repeat the algorithm.

PROGRAM

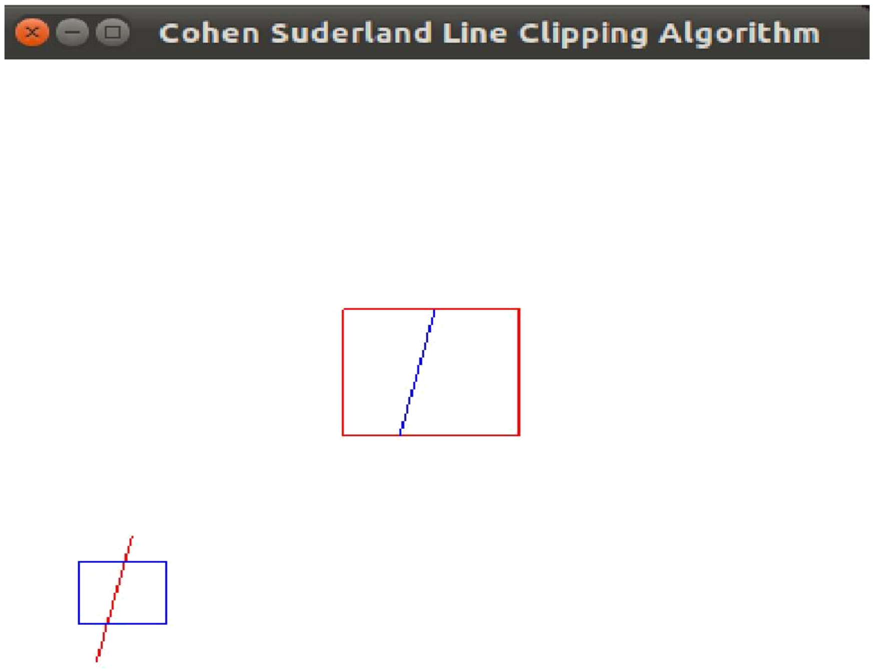
```
// Cohen-Suderland Line Clipping Algorithm with Window to viewport Mapping */
#include <stdio.h>
#include <GL/glut.h>
```

```
#include<stdbool.h>

#define outcode int
double xmin=50,ymin=50, xmax=100,ymax=100; // Window boundaries
double xvmin=200,yvmin=200,xvmax=300,yvmax=300; // Viewport boundaries
//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=60,y0=20,x1=80,y1=120;
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);
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);
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);
}

void main(int argc, char** argv)
{
        //int x1, x2, y1, y2;
        //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();
}
```

SAMPLE INPUT AND OUTPUT

```
  cc -o 5 5.c -lGL -lGLU -lglut
./5
```



Q No. 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.

DESCRIPTION

The cube is scaled in different axis's to form walls, table legs and table top. The Tea pot is placed on the table. Color and shading for all the objects and Lightning is provided with the help of openGL functions.

OBJECTIVE

- Understand the complexities of Lighting and shading
- Learn to use openGL function to draw 3D objects.

OUTCOME

- Students will be able to demonstrate how real-world lighting conditions are approximated by OpenGL
- Able to render illuminated objects by defining the different properties of light and material.

ALGORITHM

7. Initialize ambient, diffuse, specular & shininess material properties. Also initialize light intensity & position.
8. Define main()
   a. Enable display mode as single, RGB and depth buffer.
   b. Register shading with GL_SMOOTH.
   c. Use viewport for (0,0,xmax,ymax)
   d. Enable LIGHTING, LIGHT0, depth test & normalize.
9. Define reshape by gluLookAt() & clear() with color & depth buffer bit.
10. With the help of glutSolidCube () form a table (legs &top) and 3D wall. Use transformation functions translate, rotate and scale prefixed and suffixed by glPushMatrix() and PopMatrix() respectively.
11. Define display()
    a. Set ambient, diffuse, specular & shininess for glMaterials ().
    b. Set light intensity & position for glLightfv()
    c. Call draw_wall() & draw_table().
    d. Use glutSolidTeapot() to place it on the table.

PROGRAM

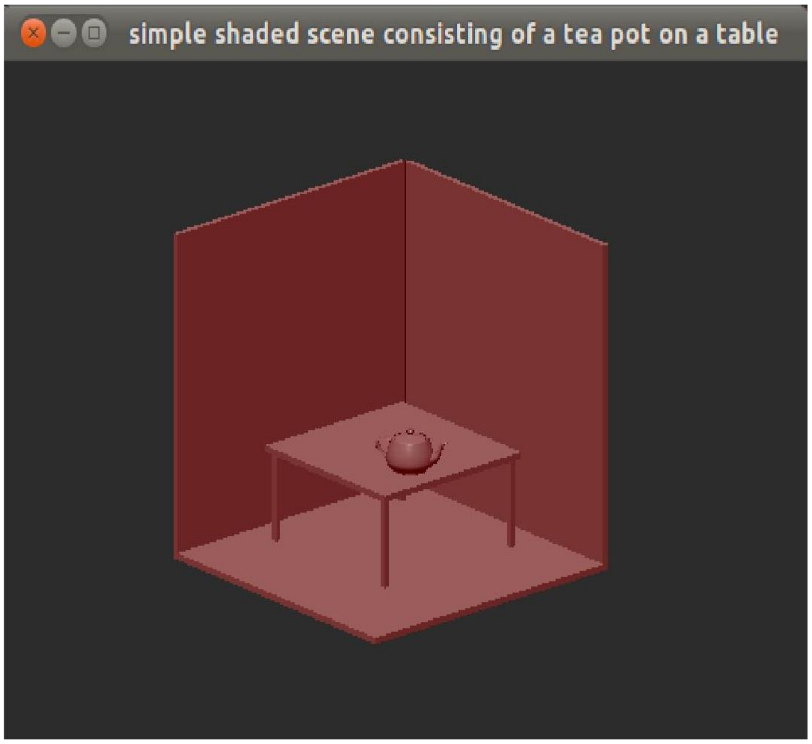
```
/* simple shaded scene consisting of a tea pot on a table */
#include <GL/glut.h>

void wall (double thickness)
{
        //draw thin wall with top = xz-plane, corner at origin
        glPushMatrix();
                glTranslated (0.5, 0.5 * thickness, 0.5);
                glScaled (1.0, thickness, 1.0);
                glutSolidCube (1.0);
        glPopMatrix();
}

//draw one table leg
void tableLeg (double thick, double len)
{
        glPushMatrix();
                glTranslated (0, len/2, 0);
                glScaled (thick, len, thick);
                glutSolidCube (1.0);
        glPopMatrix();
}

void table (double topWid, double topThick, double legThick, double legLen)
{
        //draw the table - a top and four legs
        //draw the top first
        glPushMatrix();
                glTranslated (0, legLen, 0);
                glScaled(topWid, topThick, topWid);
                glutSolidCube (1.0);
        glPopMatrix();

        double dist = 0.95 * topWid/2.0 - legThick/2.0;
        glPushMatrix();
                glTranslated (dist, 0, dist);
                tableLeg (legThick, legLen);
                glTranslated (0.0, 0.0, -2 * dist);
                tableLeg (legThick, legLen);
                glTranslated (-2*dist, 0, 2 *dist);
                tableLeg (legThick, legLen);
                glTranslated(0, 0, -2*dist);
                tableLeg (legThick, legLen);
        glPopMatrix();
}

void displaySolid (void)
```

```
{
        //set properties of the surface material
        GLfloat mat_ambient[] = {1.0f, 0.0f, 0.0f, 1.0f}; // red
        GLfloat mat_diffuse[] = {.5f, .5f, .5f, 1.0f}; //white
        GLfloat mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f}; //white
        GLfloat mat_shininess[] = {75.0f};

        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);

        //set the light source properties
        GLfloat lightIntensity[] = {0.7f, 0.7f, 0.7f, 1.0f};
        GLfloat light_position[] = {2.0f, 6.0f, 3.0f, 0.0f};
        glLightfv (GL_LIGHT0, GL_POSITION, light_position);
        glLightfv (GL_LIGHT0, GL_DIFFUSE, lightIntensity);

        //set the camera
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity();
        double winHt = 1.0; //half-height of window
        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);

        //start drawing
        glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glPushMatrix();
                glTranslated (0.6, 0.38, 0.5);
                glRotated (30, 0, 1, 0);
                glutSolidTeapot (0.08);
        glPopMatrix ();

        glPushMatrix();
                glTranslated (0.4, 0, 0.4);
                table (0.6, 0.02, 0.02, 0.3);
        glPopMatrix();
```

```
	wall (0.02);
	glPushMatrix();
		glRotated (90.0, 0.0, 0.0, 1.0);
		wall (0.02);
	glPopMatrix();

	glPushMatrix();
		glRotated (-90.0, 1.0, 0.0, 0.0);
		wall (0.02);
	glPopMatrix();

	glFlush();
}

void main (int argc, char ** argv)
{
	glutInit (&argc, argv);
	glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
	glutInitWindowSize (640, 480);
	glutInitWindowPosition (100, 100);
	glutCreateWindow ("simple shaded scene consisting of a tea pot on a table");
	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();
}
```

SAMPLE INPUT AND OUTPUT-

```
cc -o 6 6.c -lGL -lGLU -lglut

./6
```



Q No. 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.

DESCRIPTION:

When the user provides a number as input, this number specifies the number of times tetrahedron has to be subdivided to form 3D Sierpinski gasket. This algorithm finds midpoint of each edge of tetrahedron and with the help of one vertex of tetrahedron and midpoints it divides the tetrahedron. The process of finding midpoints and subdividing tetrahedron is a recursive process.

OBJECTIVE:

- To analyze and draw 3D objects
- Learning 3D Sierpinski gasket algorithm

OUTCOME: At the end of this lab session, the student will be able to

- Students will be able create 3D geometric figures using openGL function.

ALGORITHM

1. Initialize 4 vertex points of tetrahedron.
2. Define main()
   a. Enable display mode as single and depth buffer.
   b. Enable depth test.
3. Define Reshape() with glOrtho.
4. Define display()
   a. Call tetrahedron() where 4 different colors are given for each face and call divide_triangle() for each face.
   b. In divide triangle each face is recursively sub-divided into 3 triangles using midpoint of edges, n number of times.
   c. In triangle() a triangle is plotted using polygon primitives.

PROGRAM

```
#include <stdlib.h>
 #include <stdio.h>
#include<GL/glut.h>
typedef float point[3]; * initial etrahedron */

point v[]={{0.0, 0.0, 1.0},{0.0, 0.9, -0.3}, {-0.8, -0.5, -0.3}, {0.8, -0.5, -0.3};
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 right hand 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(0.0,0.0,0.0);
            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(0.0,1.0,0.0);
            divide_triangle(v[0], v[2], v[3], m);
}

void display(void)
{
           glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
        tetrahedron(n);
         glFlush();
}

void myReshape(int w, int h)
{
         glClearColor (1.0, 1.0, 1.0, 1.0);
          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();
}

int 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);
          glutMainLoop();
         return 1;
}
```

SAMPLE INPUT AND OUTPUT

```
cc -o 7 7.c -lGL -lGLU -lglut

./7

No. of Divisions ? 2
```