



BLDEAs V. P. Dr. P. G. HALAKATTI COLLEGE  
OF ENGINEERING & TECHNOLOGY,  
VIJAYAPUR 586103

DEPARTMENT OF COMPUTER SCIENCE &  
ENGINEERING

**LABORATORY MANUAL**



## COMPUTER GRAPHICS LABORATORY WITH MINI PROJECT MANUAL

Subject Code	18CSL68	IA Marks	40
Hours/Week	0:2:2	Exam Hours	03
Total Hours	36	Exam Marks	60
CREDITS		02	

Sl. No.	Particulars
<b>PART A</b>	
Design, develop, and implement the following programs in C/C++ using OpenGL API	
1.	Implement Brenham's line drawing algorithm for all types of slope.
2.	Create and rotate a triangle about the origin and a fixed point.
3.	Draw a colour cube and spin it using OpenGL transformation matrices.
4.	Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.
5.	Clip a line using Cohen-Sutherland algorithm.
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.
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.
8.	Develop a menu driven program to animate a flag using Bezier Curve algorithm.
9.	Develop a menu driven program to fill the polygon using scan line algorithm.
<b>PART B (MINI-PROJECT)</b>	
	Students should develop a mini project on the topics mentioned below or similar applications using OpenGL API. Consider all types of attributes like color, thickness, styles, font, background, speed etc., while doing mini project. During the practical exam, students should demonstrate and answer viva-voce. Simulation of concepts of OS, Data Structures, Algorithms etc.

**COURSE OBJECTIVES:** This course will enable students to

Demonstrate simple algorithms using OpenGL 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.

**COURSE OUTCOMES:** The students should be able to

Apply the concepts of computer graphics

Implement computer graphics applications using OpenGL

Animate real world problems using OpenGL

**CONDUCTION OF PRACTICAL EXAMINATION::****Experiment distribution**

1. For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
2. 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.
3. Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
4. Marks Distribution (Courseed to change in accordance with university regulations)  
For laboratories having only one part - Procedure + Execution + Viva-Voce:  $15+70+15 = 100$  Marks  
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

**REFERENCE BOOKS:**

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

## Contents

<b>1</b>	<b>PART A</b>	<b>5</b>
1.1	Bresenham's Line Drawing .....	5
1.2	Triangle Rotation .....	11
1.3	Color cube and spin .....	18
1.4	Color cube with perspective viewing .....	25
1.5	Cohen-Sutherland .....	32
1.6	Tea pot.....	41
1.7	3D Sierpinski gasket.....	48
1.8	Bezier Curve .....	54
1.9	Scan-line.....	60
<b>2</b>	<b>PART B</b>	<b>67</b>
<b>3</b>	<b>OpenGL</b>	<b>68</b>
3.1	Introduction.....	68
3.2	Introduction to frequently used OpenGL commands.....	78
<b>4</b>	<b>Viva questions</b>	<b>83</b>
<b>5</b>	<b>Viva questions and answers</b>	<b>87</b>

## 1 PART A

Design, develop, and implement the following programs in C/C++ using OpenGL API.

### 1.1 Bresenham's Line Drawing

Date: \_\_ / \_\_ / \_\_ \_\_ \_\_

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

#### 1.1.1 PREAMBLE

Bresenham's line algorithm is an algorithm that determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw line primitives in a bitmap image (e.g. on a computer screen), as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is an incremental error algorithm. It is one of the earliest algorithms developed in the field of computer graphics. An extension to the original algorithm may be used for drawing circles .

While algorithms such as Wu's algorithm are also frequently used in modern computer graphics because they can support antialiasing, the speed and simplicity of Bresenham's line algorithm means that it is still important. The algorithm is used in hardware such as plotters and in the graphics chips of modern graphics cards. It can also be found in many software graphics libraries. Because the algorithm is very simple, it is often implemented in either the firmware or the graphics hardware of modern graphics cards.

#### Concepts Used:

- + Line equations
- + Integer arithmetic

#### Algorithm: Pseudo Code OpenGL commands Familiarised:

- + glutInit
- + glutInitDisplayMode
- + glClearColor
- + glColor
- + glPointSize
- + glOrtho

**1.1.2 DESIGN:**

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 1.cpp"

**1.1.3 CODE:**

```
# Date: January 27, 2018
# File 1.cpp
# Bresenham's Line Drawing

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

**RUN:**

```
gcc 1.cpp -lglut -lGL -lGLUT  
./a.out
```



#### 1.1.4 SAMPLE OUTPUT:



### 1.1.5 STUDENT'S OBSERVATIONS

## 1.2 Triangle Rotation

Date: \_\_ / \_\_ / \_\_ \_\_

Create and rotate a triangle about the origin and a xed point.

### 1.2.1 PREAMBLE

In linear algebra, a rotation matrix is a matrix that is used to perform a rotation in Euclidean space. For example the matrix rotates points in the xy-Cartesian plane counterclockwise through an angle  $\vartheta$  about the origin of the Cartesian coordinate system. To perform the rotation using a rotation matrix  $R$ , the position of each point must be represented by a column vector  $v$ , containing the coordinates of the point. A rotated vector is obtained by using the matrix multiplication  $Rv$ . Since matrix multiplication has no effect on the zero vector (i.e., on the coordinates of the origin), rotation matrices can only be used to describe rotations about the origin of the coordinate system. Rotation matrices provide a simple algebraic description of such rotations, and are used extensively for computations in geometry, physics, and computer graphics. In 2-dimensional space, a rotation can be simply described by an angle  $\vartheta$  of rotation, but it can be also represented by the 4 entries of a rotation matrix with 2 rows and 2 columns. In 3-dimensional space, every rotation can be interpreted as a rotation by a given angle about a single xed axis of rotation (see Euler's rotation theorem), and hence it can be simply described by an angle and a vector with 3 entries. However, it can also be represented by the 9 entries of a rotation matrix with 3 rows and 3 columns. The notion of rotation is not commonly used in dimensions higher than 3; there is a notion of a rotational displacement, which can be represented by a matrix, but no associated single axis or angle.

Concepts Used: Algorithm: Pseudo Code OpenGL commands Familiarised:

```
+ glutInit
+ glutInitDisplayMode
+ glClearColor
+ glColor
+ glPointSize
+ glOrtho
```

### 1.2.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.

3. "cd 1ATXXCSXXX"

4. "gedit 2.cpp"

### 1.2.3 CODE:

```
# Date: January 27, 2018
# File 2.cpp
# Triangle Rotation

#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);
}
/* 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);
glFlush ( );
}

```

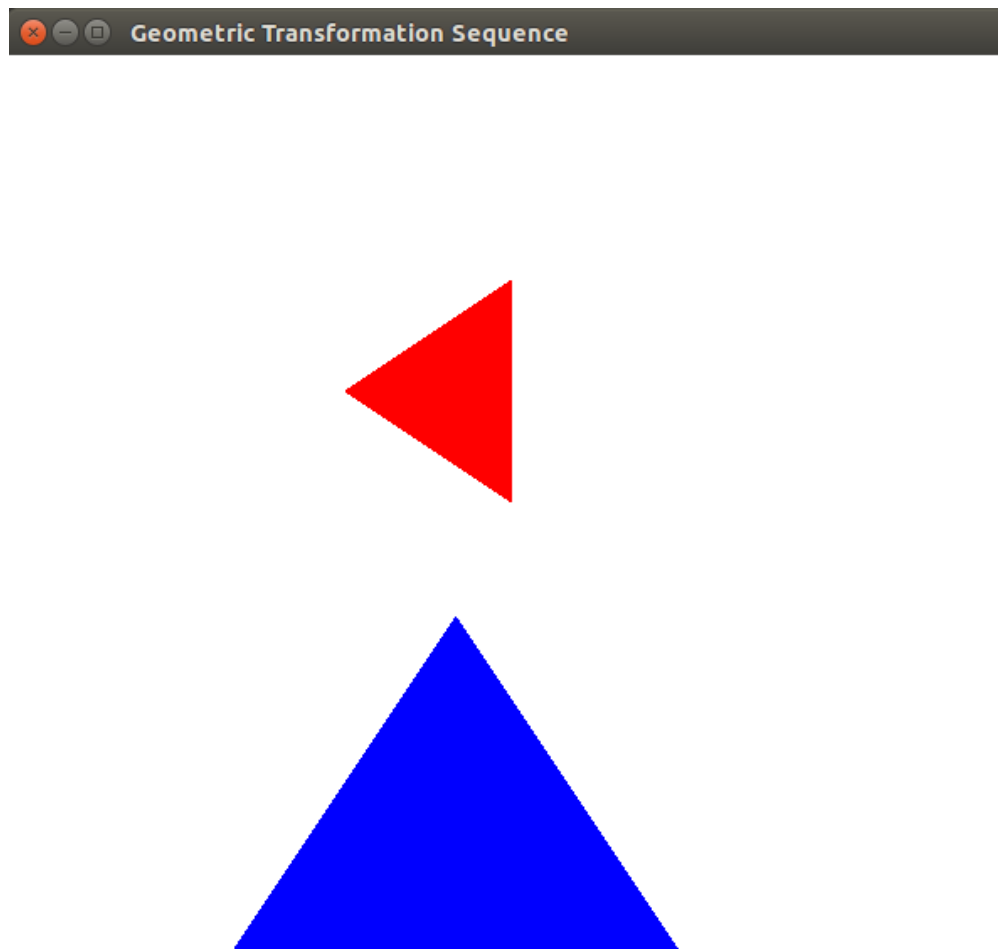
```
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);
    glClear (GL_COLOR_BUFFER_BIT);
}

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;
}
```

**RUN:**

```
gcc 2.cpp -lglut -lGL -lGLUT
./a.out
```

#### 1.2.4 SAMPLE OUTPUT:





### 1.2.5 STUDENT'S OBSERVATIONS

### 1.3 Color cube and spin

Date: \_\_ / \_\_ / \_\_ \_\_

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

#### 1.3.1 PREAMBLE

##### Concepts Used:

- + Data structures for representing a cube
- + Rotation Transformation

##### Algorithm:

Modeling a color cube with simple data structures

- + Define vertices with centre of cube as origin
- + Define normals to identify faces
- + Define colors

Rotating the cube

- + Define angle of rotation [ or accept it as input ]
- + Define/Input rotation axis
- + Use swap-buffer to ensure smooth rotation

Using callback functions to indicate either or both of the following:

- + Angle of rotation
- + Axis of rotation

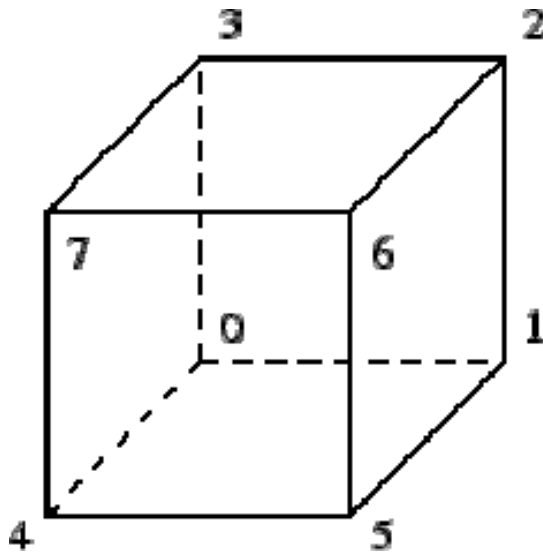
1. Define global arrays for vertices and colors

```
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 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}};
```

2. Draw a polygon from a list of indices into the array vertices and use color corresponding to first index

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

```
glEnd();
}
3. Draw cube from faces.
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);
}
4. Define the Display function to clear frame buffer , Z-buffer
    , rotate cube and draw, swap buffers.
void display(void)
{
    /* display callback, clear frame buffer and z buffer,
    rotate cube and draw, swap buffers */
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}
5. Define the spincube function to spin cube 2 degrees about selected
    axis.
6. Define mouse callback to select axis about which to rotate.
7. Define reshape function to maintain aspect ratio.
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);
}
8. Define main function to create window and to call all function.
```



OpenGL Commands Familiarized:

- + glRotate
- + glMatrixMode
- + glMouseFunc. and/or glKeyboardFunc
- + glEnable
- + glutSwapBuffer
- + glutIdleFunc

### 1.3.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 3.c"

### 1.3.3 CODE:

```
# Date: January 27, 2018
# File 3.c
# Color cube and spin

#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;

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);
}

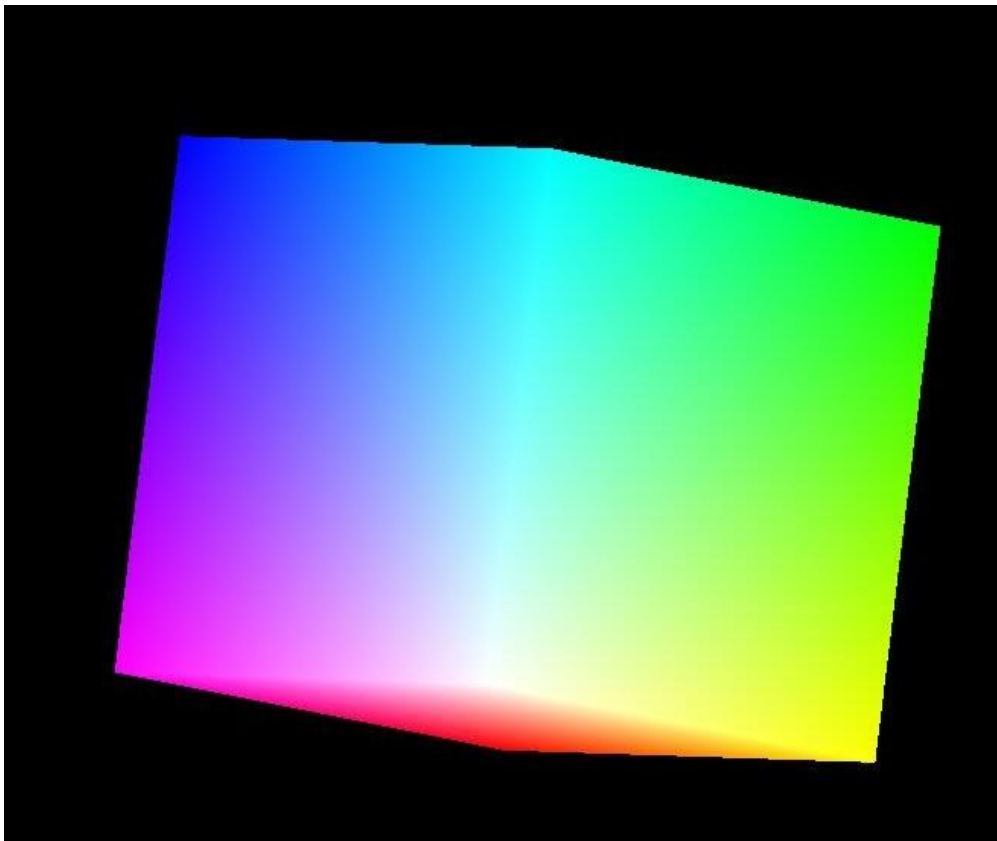
int main(int argc,char **argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutCreateWindow("color cuce");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutIdleFunc(spincube);
    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();  
}
```

**RUN:**

```
gcc 3.c -lglut -lGL -lGLUT  
./a.out
```

**1.3.4 SAMPLE OUTPUT:**



### 1.3.5 STUDENT'S OBSERVATIONS



## 1.4 Color cube with perspective viewing

Date: \_\_ / \_\_ / \_\_ \_\_

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

### 1.4.1 PREAMBLE

#### Concepts Used:

- + Data structures for representation of a cube
- + Perspective viewing
- + Defining and moving the camera
- + Input functions using Keyboard and mouse

#### Algorithm:

1. Modeling a color cube with simple data structures
  - + Define vertices with centre of cube as origin
  - + Define normals to identify faces
  - + Define colors
2. Camera Manipulations
  - + Define initial camera position - take care to define outside the cube.
3. Callback Function
  - + Handling mouse inputs - use the 3 mouse buttons to define the 3 axes of rotation
  - + Handling Keyboard Inputs - use the 3 pairs of keys to move the viewer along +ive and -ive directions of the X and Y axes respectively.

#### Pseudo Code

1. Define global arrays for vertices and colors

```
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 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}};
```
2. Draw a polygon from a list of indices into the array vertices and

- ```

    use color corresponding to first index
void polygon(int a, int b, int c, int d)
{
    glBegin(GL_POLYGON);
    glColor3fv(colors[a]);
    glVertex3fv(vertices[a]);
    glVertex3fv(vertices[b]);
    glVertex3fv(vertices[c]);
    glVertex3fv(vertices[d]);
    glEnd();
}

```
3. Draw cube from faces.
 

```

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);
}

```
  4. Initialize the theta value and initial viewer location and axis
 

```

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

```
  5. Define display function to update viewer position in model view matrix
 

```

void display(void)
{
    /* display callback, clear frame buffer and z buffer,
    rotate cube and draw, swap buffers */
    glLoadIdentity();
    gluLookAt(viewer[0], viewer[1], viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}

```
  6. Define mouse callback function to rotate about axis.
  7. Define key function to move viewer position .use x,X,y,Y z,Z keys to move viewer position.
  8. Define reshape function to maintain aspect ratio and then use perspective view.
  9. Define main function to create window and to call all function.

### OpenGL Commands Familiarized:

```
+ glutInit
+ glutInitDisplayMode
+ glClearColor
+ glColor
+ glPointSize
+ gluOrtho2D
```

#### 1.4.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 4.c"

#### 1.4.3 CODE:

```
# Date: January 27, 2018
# File 4.c
# Color cube 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();
    }

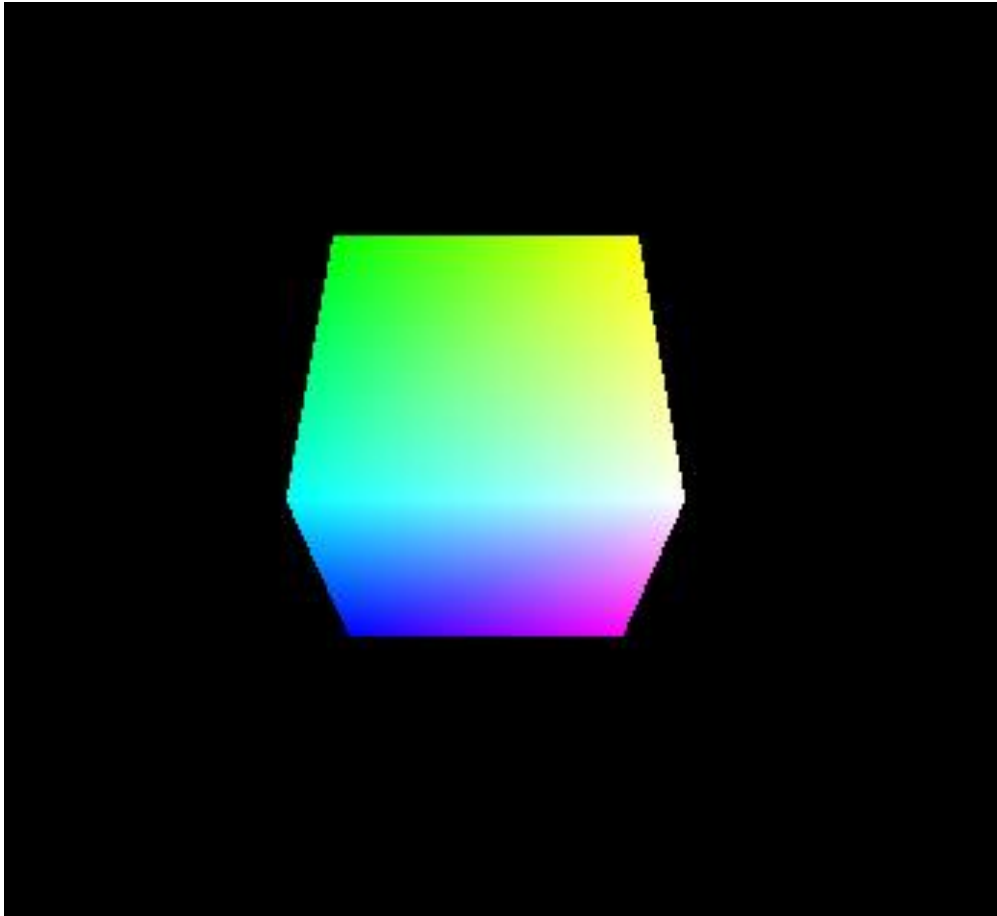
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if(w<=h)
        glFrustum(-2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w, 2.0*(GLfloat)h/(GLfloat)w, 2.0, 20.0);
    else
        glFrustum(-2.0, 2.0, -2.0*(GLfloat)w/(GLfloat)h, 2.0*(GLfloat)w/(GLfloat)h,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

**RUN:**

```
gcc 4.c -lglut -lGL -lGLUT
./a.out
```

#### 1.4.4 SAMPLE OUTPUT:



#### 1.4.5 STUDENT'S OBSERVATIONS

## 1.5 Cohen-Sutherland

Date: \_\_ / \_\_ / \_\_ \_\_

Clip a line using Cohen-Sutherland algorithm.

### 1.5.1 PREAMBLE

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivially accepted or rejected. If the line cannot be trivially accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted. To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

Concepts Used:

+ Data structures for representing a square and line.

**Algorithm:**

Every line endpoint is assigned a 4 bit Region code. The appropriate bit is set depending on the location of the endpoint with respect to that window component as shown below:

|      |      |      |
|------|------|------|
| 1001 | 1000 | 1010 |
| 0001 | 0000 | 0010 |
| 0101 | 0100 | 0110 |

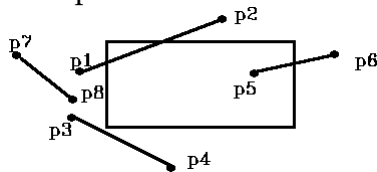
Endpoint Left of window then set bit 1 Endpoint Right of window then set bit 2 Endpoint Below window then set bit 3 Endpoint Above window then set bit 4

1. Given a line segment with endpoint  $P1=(x1,y1)$  and  $P2=(x2,y2)$
  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 . Read '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 to . Replace with I and repeat the algorithm.

**Example1:**

Can determine the bit code by testing the endpoints with window as follows:

If x is less than Xwmin then set bit 1  
 If x is greater than Xwmax then set bit 2  
 If y is less than Ywmin then set bit 3  
 If y is greater than Ywmax then set bit 4

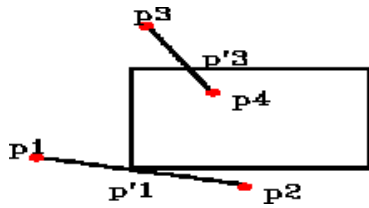
Note: can use 4 element Boolean matrix and set C[Left] = true / false (1/0). If both endpoints = 0000 (in window) then display line. If both endpoints have a bit set in same position (P7, P8) then the line is completely outside the window and is rejected.

So: can do logical AND of region codes and reject if result is 0000  
 Can do logical OR of region codes and accept if result = 0000

For the rest of the lines we must check for intersection with window. May still be outside, e.g. P3 - P4 in the above image. If point is to Left of window then compute intersection with Left window boundary. Do the same for Right, Bottom, and Top. Then recompute region code retest. So the algorithm is as follows:

1. Compute region code for endpoints
2. Check for trivial accept or reject
3. If step 2 unsuccessful then compute window intersections in order:\\ Left, Right, Bottom, Top (only do 1)
4. Repeat steps 1, 2,3 until done.

**Example2:**



I. 1)  $P_1 = 0001$  2) no 3)  $P_1 = P'_1$

$P_2 = 1000$

II. 1)  $P'_1 = 0000$  2) no 3)  $P_2 = P'_2$

$P_2 = 1000$

III. 1)  $P'_1 = 0000$  2) Yes - accept & display

$P'_2 = 0000$

Look at how to compute the line intersections

for  $P'_1$ :  $m = dy/dx = (y_1 - y'_1)/(x_1 - x'_1)$   $P_1(x_1, y_1)$

$P'_1(x'_1, y'_1)$

or  $y'_1 = y_1 + m(x'_1 - x_1)$

but for Left boundary  $x'_1 = X_{wmin}$

for Right boundary  $x'_1 = X_{wmax}$

Similarly for Top / Bottom, e.g.  $P'_3$

$x'_3 = x_3 + (y'_3 - y_3) / m$

for Top  $y'_3 = Y_{wmax}$  for Bottom  $y'_3 = Y_{wmin}$

### OpenGL Commands Familiarized:

```
+ glMatrixMode
+ glLoadIdentity
+ gluOrtho2D
+ glFlush
+ glColor3f
```

```
+ glBegin
```

### 1.5.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 5.c"

### 1.5.3 CODE:

```
# Date: January 27, 2018
# File 5.c
# Cohen-Sutherland

#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  $y = y0 + \text{slope} * (x - x0)$ ,  $x = x0 + (1/\text{slope}) * (y - y0)$ 
        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);

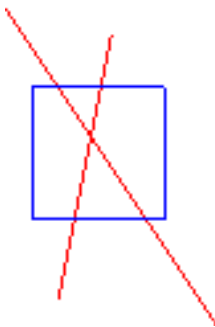
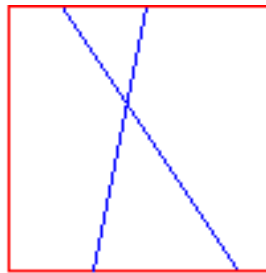
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
```

```
glutInitWindowPosition(0, 0);  
glutCreateWindow("Cohen Suderland Line Clipping Algorithm");  
glutDisplayFunc(display);  
myinit();  
glutMainLoop();  
}
```

**RUN:**

```
gcc 5.c -lglut -lGL -lGLUT  
./a.out
```

**1.5.4 SAMPLE OUTPUT:**



### 1.5.5 STUDENT'S OBSERVATIONS



## 1.6 Tea pot

Date: \_\_ / \_\_ / \_\_ \_\_ \_\_

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.

### 1.6.1 PREAMBLE

#### Concepts Used:

- + Translation, Scaling
- + Material Properties, Light Properties
- + Depth in 3D Viewing
- + Using predefined GLU library routines

#### Algorithm:

Algorithm[High Level]:

1. Recognise the objects in the problem domain and their components
  - + Teapot
  - + Table
    - Table top
    - 4 legs
  - + Walls
    - Left wall
    - Right wall
    - Floor
2. If depth testing is NOT used ensure that the background is drawn first
3. Enable lighting first – since lighting is applied at the time objects are drawn.
4. Draw the objects.

Algorithm[Low Level]:

- + Main Function:
  - + Initialization for Display, Mode , Window
  - + Enable lighting, shading, depth test
  - + Register display and callback function
  - + Define viewport appropriately
  - + Call mainloop
- + Display function:
  - + Define Material properties
  - + Define lighting properties
  - + Set the camera by defining

- projection parameters
- camera parameters
- + Plan the required centre position for each of the components
- + Draw each component using translation, scaling and rotation as required

## Pseudo Code

1. Include glut header files.
2. Define wall function as following
 

```
void wall(double thickness)
{
    glPushMatrix();
    glTranslated(0.5, 0.5*thickness, 0.5);
    glScaled(1.0, thickness, 1.0);
    glutSolidCube(1.0);
    glPopMatrix();
}
```
3. Draw one tableleg using tableLeg function as following
 

```
void tableLeg(double thick, double len)
{
    glPushMatrix();
    glTranslated(0, len/2, 0);
    glScaled(thick, len, thick);
    glutSolidCube(1.0);
    glPopMatrix();
}
```
4. Draw the table using table function.
  - i) draw the table top using glutSolidCube(1.0) function. Before this use gltranslated and glScaled function to fix table in correct place.
  - ii) Draw four legs by calling the function tableleg .before each call use gltranslated function to fix four legs in correct place.
5. Define the function displaySolid
  - i) Initialize the properties of the surface material and set the light source properties.
  - ii) Set the camera position.
  - iii) Draw the teapot using glutSolidTeapot(0.08).before this call gltranslated and glRotated.
  - iv) Call table function and wall function
  - v) Rotate wall about 90 degree and then call wall function.
  - vi) Rotate wall about -90 degree and then call wall function.
6. Define the main function to create window and enable lighting function using glEnable(GL\_LIGHTING)

## OpenGL Commands Familiarized:

- + glPushMatrix, glPopMatrix
- + glTranslate, glScale, glRotate

```
+ glMaterial, glLight
+ gluLookAt
+ glutSolidTeapot, glutSolidCube
```

### 1.6.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 6.c"

### 1.6.3 CODE:

```
# Date: January 27, 2018
# File 6.c
# Tea pot

#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 legl, double legl)
{
    glPushMatrix();
        glTranslated(0, legl, 0);
        glScaled(topw, topt, topw);
        glutSolidCube(1.0);
    glPopMatrix();

    double dist=0.95*topw/2.0-legl/2.0;

    glPushMatrix();

        glTranslated(dist, 0, dist);
        tableleg(legl, legl);

        glTranslated(0, 0, -2*dist);
        tableleg(legl, legl);

        glTranslated(-2*dist, 0, 2*dist);
        tableleg(legl, legl);

        glTranslated(0, 0, -2*dist);
        tableleg(legl, 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();

}

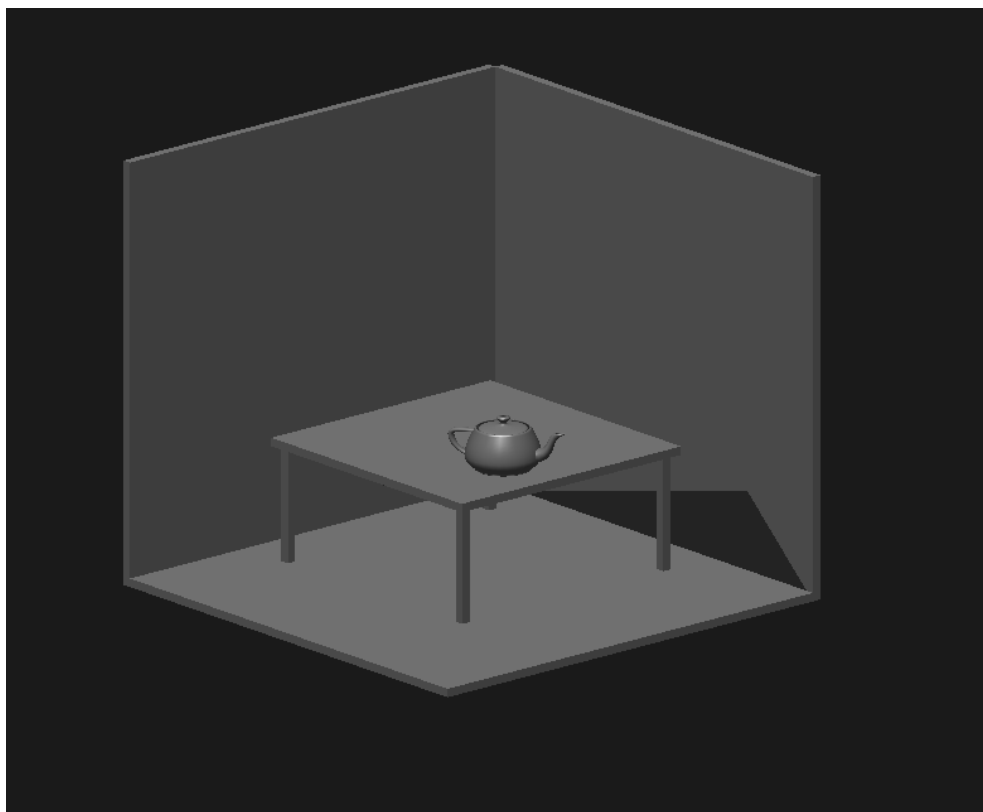
int main(int argc, char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("teapot");
    glutDisplayFunc(displaysolid);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    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();  
}
```

**RUN:**

```
gcc 6.c -lglut -lGL -lGLUT  
./a.out
```

#### 1.6.4 SAMPLE OUTPUT:



### 1.6.5 STUDENT'S OBSERVATIONS

## 1.7 3D Sierpinski gasket

Date: \_\_ / \_\_ / \_\_ \_\_

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.

### 1.7.1 PREAMBLE

Sierpinski's Triangle is a very famous fractal that's been seen by most advanced math students. This fractal consists of one large triangle, which contains an infinite amount of smaller triangles within. The infinite amount of triangles is easily understood if the fractal is zoomed in many levels. Each zoom will show yet more previously unseen triangles embedded in the visible ones.

Creating the fractal requires little computational power. Even simple graphing calculators can easily make this image. The fractal is created pixel by pixel, using random numbers; the fractal will be slightly different each time due to this. Although, if you were to run the program repeatedly, and allow each to use an infinite amount of time, the results would be always identical. No one has an infinite amount of time, but the differences in the finite versions are very small.

A fractal is generally "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole".

To generate this fractal, a few steps are involved.

We begin with a triangle in the plane and then apply a repetitive scheme of operations to it (when we say triangle here, we mean a blackened, 'filled-in' triangle). Pick the midpoints of its three sides. Together with the old vertices of the original triangle, these midpoints define four congruent triangles of which we drop the center one. This completes the basic construction step. In other words, after the first step we have three congruent triangles whose sides have exactly half the size of the original triangle and which touch at three points which are common vertices of two contiguous triangles. Now we follow the same procedure with the three remaining triangles and repeat the basic step as often as desired. That is, we start with one triangle and then produce 3, 9, 27, 81, 243, triangles, each of which is an exact scaled down version of the triangles in the preceding step.

#### Concepts Used:

- + Data structures for representing 3D vertices
- + Tetrahedron sub-division using mid-points

#### Algorithm:

- + Input: Four 3D vertices of tetrahedron, no. of divisions
- + Recursively sub-divide the each triangle by finding the mid-point



## Pseudo Code

1. Define and initializes the array to hold the vertices as follows:  
GLfloat vertices[4][3] =  
{ {0.0, 0.0, 0.0}, {25.0, 50.0, 10.0}, {50.0, 25.0, 25.0}, {25.0, 10.0, 25.0} };
2. Define and initialize initial location inside tetrahedron.  
GLfloat p[3] = {25.0, 10.0, 25.0};
3. Define Triangle function that uses points in three dimensions to display one triangle  
Use glBegin(GL\_POLYGON)  
    glVertex3fv(a)  
    glVertex3fv(b)  
    glVertex3fv(c) to display points.  
    glEnd();
4. Subdivide a tetrahedron using divide\_triangle function
  - i) if no of subdivision(m) > 0 means perform following functions
  - ii) Compute six midpoints using for loop.
  - iii) Create 4 tetrahedrons by calling divide\_triangle function
  - iv) Else draw triangle at end of recursion.
5. Define tetrahedron function to apply triangle subdivision to faces of tetrahedron by Calling the function divide\_triangle.
6. Define display function to clear the color buffer and to call tetrahedron function.
7. Define main function to create window and to call display function.

## OpenGL commands Familiarised:

```
+ glutInit  
+ glutInitDisplayMode  
+ glClearColor  
+ glColor  
+ glPointSize  
+ glOrtho
```

### 1.7.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 7.c"

### 1.7.3 CODE:

```
# Date: January 27, 2018
# File 7.c
# 3D Sierpinski gasket

#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();
}

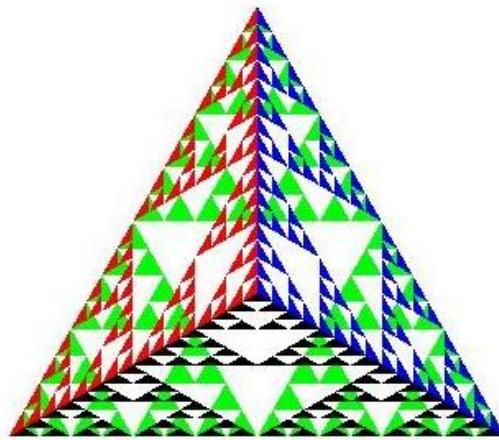
void main(int argc, char **argv)
{
    printf(" No. of Divisions ? ");
    scanf("%d", &n);
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("3D Gasket");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

**RUN:**

```
gcc 7.c -lglut -lGL -lGLUT
```

./a.out

#### 1.7.4 SAMPLE OUTPUT:



### 1.7.5 STUDENT'S OBSERVATIONS

## 1.8 Bezier Curve

Date: \_\_ / \_\_ / \_\_

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

### 1.8.1 PREAMBLE

Bezier curve is discovered by the French engineer Pierre Bezier. These curves can be generated under the control of other points. Approximate tangents by using control points are used to generate curve. The Bezier curve can be represented mathematically as

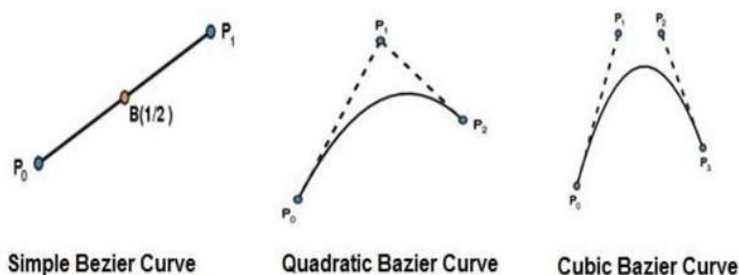
$$\sum_{k=0}^n P_i B_i^n(t)$$

Where  $p_i$  is the set of points and  $B_i^n(t)$  represents the Bernstein polynomials which are given by

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

Where  $n$  is the polynomial degree,  $i$  is the index, and  $t$  is the variable.

The simplest Bezier curve is the straight line from the point  $P_0$  to  $P_1$ . A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is determined by four control points.



Bezier curves generally follow the shape of the control polygon, which consists of the segments joining the control points and always pass through the first and last control points. They are contained in the convex hull of their defining control points. The

degree of the polynomial defining the curve segment is one less than the number of defining polygon points. Therefore, for 4 control points, the degree of the polynomial is 3, i.e. cubic polynomial. A Bezier curve generally follows the shape of the defining polygon. The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments. The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points. No straight line intersects a Bezier curve more times than it intersects its control polygon. They are invariant under an affine transformation. Bezier curves exhibit global control means moving a control point alters the shape of the whole curve. A given Bezier curve can be subdivided at a point  $t=t_0$  into two Bezier segments which join together at the point corresponding to the parameter value  $t=t_0$ .

Concepts Used: Algorithm: OpenGL Commands Familiarized:

```
+ glMatrixMode
+ glLoadIdentity
+ gluOrtho2D
+ glFlush
+ glColor3f
+ glBegin
```

### 1.8.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 8.c"

### 1.8.3 CODE:

```
# Date: January 27, 2018
# File 8.cpp
# Bezier Curve

#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);
```

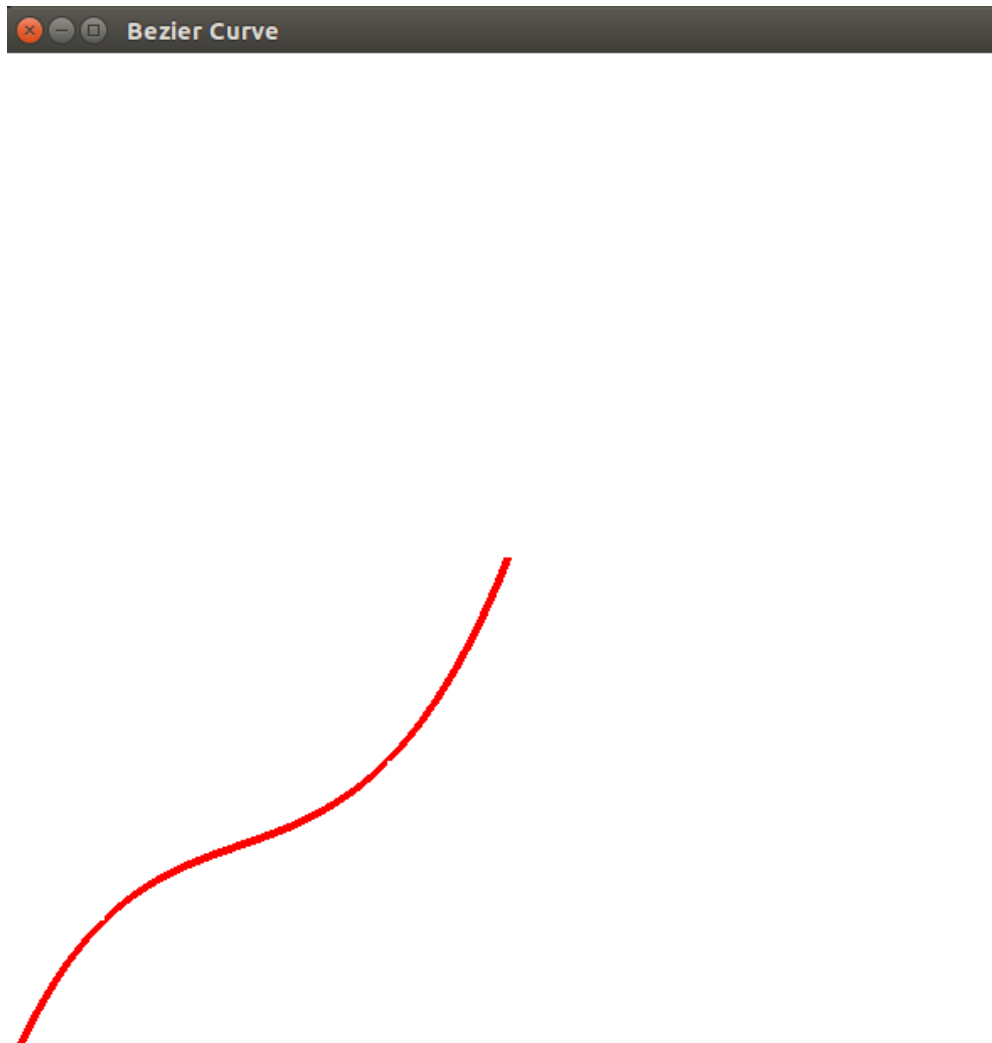


```
    glutCreateWindow("Bezier Curve");  
    glutDisplayFunc(display);  
    myinit();  
  
    glutMainLoop();  
    return 0;  
  
}
```

**RUN:**

```
gcc 8.cpp -lglut -lGL -lGLUT  
./a.out
```

#### 1.8.4 SAMPLE OUTPUT:



### 1.8.5 STUDENT'S OBSERVATIONS

## 1.9 Scan-line

Date: \_\_ / \_\_ / \_\_ \_\_ \_\_

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

### 1.9.1 PREAMBLE

The scan conversion algorithm works as follows i. Intersect each scanline with all edges ii. Sort intersections in x iii. Calculate parity of intersections to determine in/out iv. Fill the "in" pixels

Special cases to be handled: i. Horizontal edges should be excluded ii. For vertices lying on scanlines, i. count twice for a change in slope. ii. Shorten edge by one scanline for no change in slope

Coherence between scanlines tells us that

Edges that intersect scanline  $y$  are likely to intersect  $y + 1$

$X$  changes predictably from scanline  $y$  to  $y + 1$

We have 2 data structures: Edge Table and Active Edge Table

Traverse Edges to construct an Edge Table

Eliminate horizontal edges

Add edge to linked-list for the scan line corresponding to the lower vertex.

Store the following:

$y_{upper}$ : last scanline to consider

$x_{lower}$ : starting x coordinate for edge

$1/m$ : for incrementing  $x$ ; compute

Construct Active Edge Table during scan conversion. AEL is a linked list of active edges on the current scanline,  $y$ . Each active edge line has the following information

$y_{upper}$ : last scanline to consider

$x_{lower}$ : edge's intersection with current  $y$

$1/m$ :  $x$  increment

The active edges are kept sorted by x Concepts Used:

- + Use the straight line equation  $y_2 - y_1 = m(x_2 - x_1)$  to compute the x values corresponding to lines increment in y value.
- + Determine which are the left edge and right edges for the the closed polygon. for each value of y within the polygon.
- + Fill the polygon for each value of y within the polygon from x=left edge to x=right edge.

**Algorithm:**

1. Set y to the smallest y coordinate that has an entry in the ET;  
i.e, y for the first nonempty bucket.
2. Initialize the AET to be empty.
3. Repeat until the AET and ET are empty:
  - 3.1 Move from ET bucket y to the AET those edges whose  $y_{min} = y$  (entering edges).
  - 3.2 Remove from the AET those entries for which  $y = y_{max}$  (edges not involved in the next scanline), then sort the AET on x (made easier because ET is presorted).
  - 3.3 Fill in desired pixel values on scanline y by using pairs of x coordinates from AET.
  - 3.4 Increment y by 1 (to the coordinate of the next scanline).
  - 3.5 For each nonvertical edge remaining in the AET, update x for the new y.

Extensions:

1. Multiple overlapping polygons – priorities
2. Color, patterns Z for visibility

// Function scanfill

Inputs : vertices of the polygon.

Output : filled polygon.

Processing :

1. Initialize array LE to 500 and RE to 0 for all values of y(0-->499)
2. Call function EDGEDETECT for each edge of the polygon one by one to set the value of x for each value of y within that line.
3. For each value of y in the screen draw the pixels for every value of x provided . It is greater than right edge value of x.

//function Display

Inputs : Globally defined vertices of Polygon

Output : Filled polygon display

Processing :

1. Draw the polygon using LINE, LOOP
2. Fill the polygon using SCANFILL

//Function EDGEDETECT:

Inputs :

1. End co-ordinates of edge
2. Address of LE and RE

Output :

The updated value of x for left edge of the polygon and the right edge of the polygon.

Processing :

1. Find the inverse of the slope.
2. Starting from the lesser integer value of y to the greater integer value of y.
3. Compute the value of x for left edge and right edge and update the value of x for both the edges.

### OpenGL Commands Familiarized:

```
+ glutInit
+ glutInitDisplayMode
+ glClearColor
+ glColor
+ glPointSize
+ gluOrtho2D
```

#### 1.9.2 DESIGN:

1. Open the terminal.
2. Create your own directory using "mkdir 1ATXXCSXXX". THIS IS ONE TIME INSTRUCTION.
3. "cd 1ATXXCSXXX"
4. "gedit 9.c"

#### 1.9.3 CODE:

```
# Date: January 27, 2018
# File 9.c
# Scan-line

#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);
}

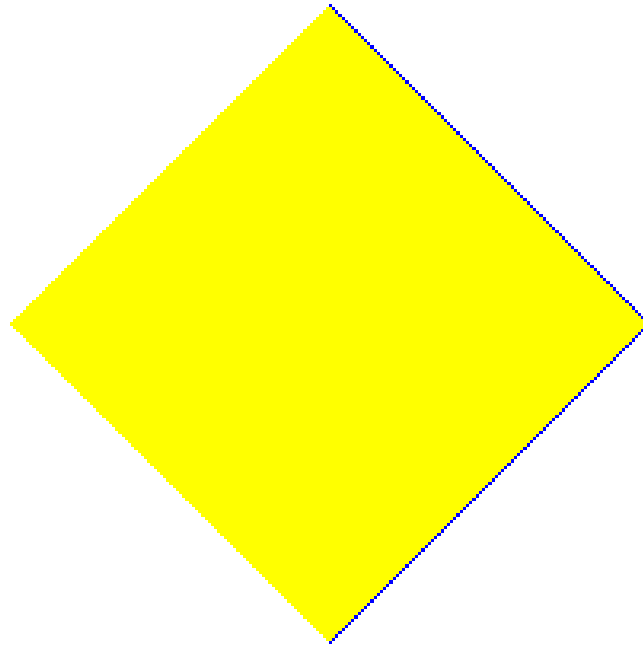
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Filling a Polygon using Scan-line Algorithm");
    glutDisplayFunc(display);
    myinit();
    glutMainLoop();
}
```

**RUN:**

```
gcc -w 9.c -lglut -lGL -lGLUT
./a.out
```



#### 1.9.4 SAMPLE OUTPUT:



### 1.9.5 STUDENT'S OBSERVATIONS

## 2 PART B

Develop a suitable Graphics package to implement the skills learnt in the theory and the exercises indicated in PART A. Use the OpenGL.

### Criteria for CG Project

Students per batch should be THREE or less.

One Three-Dimensional OpenGL Graphics Project using features from at least THREE CATEGORIES listed below:

#### Category I

Input and Interaction

Menus, Display Lists

#### Category II

Transformations

Camera Movement

#### Category III

Coloring

Texturing

Lighting/ Shading

#### Category IV

Animation

Hidden Surface Removal

## 3 OpenGL

### 3.1 Introduction

OpenGL, or the Open Graphics Library, is a 3D graphics language developed by Silicon Graphics. Before OpenGL was available, software developers had to write unique 3D graphics code for each operating system platform as well as different graphics hardware. However, with OpenGL, developers can create graphics and special effects that will appear nearly identical on any operating system and any hardware that supports OpenGL. This makes it much easier for developers of 3D games and programs to port their software to multiple platforms.

When programmers write OpenGL code, they specify a set of commands. Each command executes a drawing action or creates a special effect. Using hundreds or even thousands of these OpenGL commands, programmers can create 3D worlds which can include special effects such as texture mapping, transparency (alpha blending), hidden surface removal, antialiasing, fog, and lighting effects. An unlimited amount of viewing and modeling transformations can be applied to the OpenGL objects, giving developers an infinite amount of possibilities.

GLUT gives you the ability to create a window, handle input and render to the screen without being Operating System dependent.

The first things you will need are the OpenGL and GLUT header files and libraries for your current Operating System.

Once you have them setup on your system correctly, open your first c++ file and include them at the start of your file like so:

```
#include <GL/gl.h> //include the gl header file
#include <GL/glut.h> //include the GLUT header file
```

Now, just double check that your system is setup correctly, and try compiling your current file.

If you get no errors, you can proceed. If you have any errors, try your best to fix them. Once you are ready to move onto the next step, create a main() method in your current file.

Inside this is where all of your main GLUT calls will go.

The first call we are going to make will initialize GLUT and is done like so:  
`glutInit(&argc, argv);` //initialize the program.

Keep in mind for this, that argc and argv are passed as parameters to your main method. You can see how to do this below.

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("");
    glutDisplayFunc(display);
    myinit();
}
```

```
glutMainLoop();
}
```

Once we have GLUT initialized, we need to tell GLUT how we want to draw. There are several parameters we can pass here, but we are going to stick with the most basic GLUT\_SINGLE, which will give us a single buffered window.

```
glutInitDisplayMode(GLUT_SINGLE); //set up a basic display buffer (only singular for now)
```

The next two methods we are going to use, simply set the size and position of the GLUT window on our screen:

```
glutInitWindowSize(500, 500); //set the width and height of the window
glutInitWindowPosition(100, 100); //set the position of the window
```

And then we give our window a caption/title, and create it.

```
glutCreateWindow("A basic OpenGL Window"); //set the caption for the window
```

We now have a window of the size and position that we want. But we need to be able to draw to it. We do this, by telling GLUT which method will be our main drawing method. In this case, it is a void method called display()

```
glutDisplayFunc(display); //call the display function to draw our world
```

Finally, we tell GLUT to start our program. It does this by executing a loop that will continue until the program ends.

```
glutMainLoop(); //initialize the OpenGL loop cycle
```

So thus far, we have a window. But the display method that I mentioned is needed. Let's take a look at this and dissect it.

```
void display(void)
{
    glClearColor(0.0, 0.0, 0.0, 1.0); //clear the color of the window
    glClear(GL_COLOR_BUFFER_BIT); //Clear the Color Buffer (more buffers later on)
    glLoadIdentity(); //load the Identity Matrix
    gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //set the view
    glFlush(); //ush it all to the screen
}
```

The first method, glClearColor will set the background color of our window. In this example, we are setting the background to black (RGB 0, 0, 0). The 1.0 value on the end is an alpha value and makes no difference at this stage as we don't have alpha enabled.

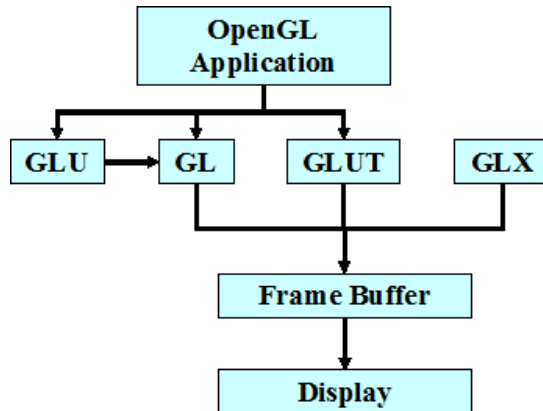
The next thing we want to do, is erase everything currently stored by OpenGL. We need to do this at the start of the method, as the method keeps looping over itself and if we draw something, we need to erase it before we draw our next frame. If we don't do this, we can end up with a big mess inside our buffer where frames have been drawn over each other.

The third method, glLoadIdentity resets our model view matrix to the identity matrix, so that our drawing transformation matrix is reset. From then, we will set the 'camera' for our scene. I am placing it 5 units back into the viewer so that anything we draw at 0,0,0 will be seen in front of us. The final thing we need to do is then flush

the current buffer to the screen so we can see it. This can be done with `glFlush()` as we are only using a single buffer.

OpenGL is an API. OpenGL is nothing more than a set of functions you call from your program (think of as collection of .h files).

OpenGL Libraries



OpenGL Hierarchy: Several levels of abstraction are provided GL

Lowest level: vertex, matrix manipulation

`glVertex3f(point.x, point.y, point.z)`

GLU

Helper functions for shapes, transformations

`gluPerspective(fovy, aspect, near, far )`

`gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);`

GLUT

Highest level: Window and interface management

`glutSwapBuffers()`

`glutInitWindowSize(500, 500);`

OpenGL Implementations: OpenGL IS an API (think of as collection of .h files):

`#include <GL/gl.h>`

`#include <GL/glu.h>`

`#include <GL/glut.h>`

Windows, Linux, UNIX, etc. all provide a platform specific implementation.

Windows: `opengl32.lib glu32.lib glut32.lib`

Linux: `-l GL -l GLU -l GLUT`

Event Loop:

OpenGL programs often run in an event loop:

Start the program  
 Run some initialization code  
 Run an infinite loop and wait for events such as Key press Mouse move,  
 click Reshape window Expose event

#### OpenGL Command Syntax (1) :

OpenGL commands start with "gl"  
 OpenGL constants start with "GL\_"  
 Some commands end in a number and one, two or three letters at the end  
 (indicating number and type of arguments)  
 A Number indicates number of arguments  
 Characters indicate type of argument

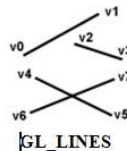
#### OpenGL Command Syntax (2)

'f' float  
 'd' double float  
 's' signed short integer  
 'i' signed integer  
 'b' character  
 'ub' unsigned character  
 'us' unsigned short integer  
 'ui' unsigned integer

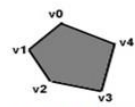
#### Ten gl primitives



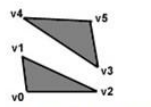
GL\_POINTS



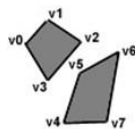
GL\_LINES



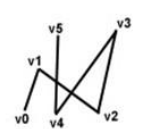
GL\_POLYGON



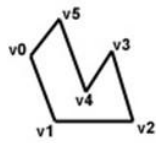
GL\_TRIANGLES



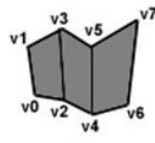
GL\_QUADS



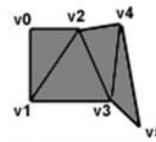
GL\_LINE\_STRIP



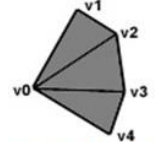
GL\_LINE\_LOOP



GL\_QUAD\_STRIP



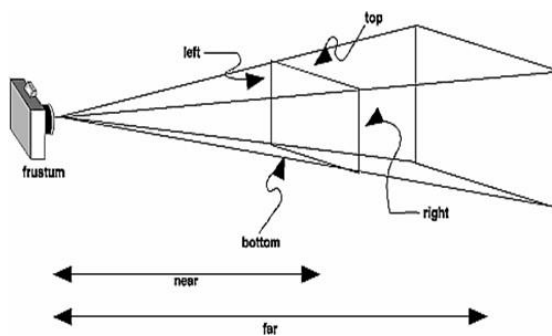
GL\_TRIANGLE\_STRIP



GL\_TRIANGLE\_FAN

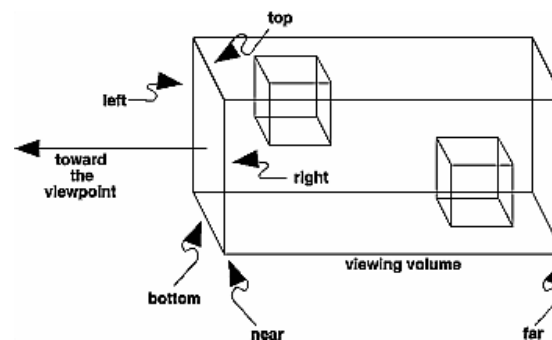
### Projections in OpenGL

#### Perspective projection



```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far);
```

#### Orthographic projection



```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
```



## INTRODUCTION

Computer graphics are graphics created using computers and, more generally, the representation and manipulation of image data by a computer hardware and software. The development of computer graphics, or simply referred to as CG, has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized the animation and video game industry. 2D computer graphics are digital images-mostly from two-dimensional models, such as 2D geometric models, text (vector array), and 2D data. 3D computer graphics in contrast to 2D computer graphics are graphics that use a three-dimensional representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering images.

## OPEN GL

OpenGL is the most extensively documented 3D graphics API(Application Program Interface) to date. Information regarding OpenGL is all over the Web and in print. It is impossible to exhaustively list all sources of OpenGL information. OpenGL programs are typically written in C and C++. One can also program OpenGL from Delphi (a Pascal-like language), Basic, Fortran, Ada, and other languages. To compile and link OpenGL programs, one will need OpenGL header files. To run OpenGL programs one may need shared or dynamically loaded OpenGL libraries, or a vendor-specific OpenGL Installable Client Driver (ICD).

## GLUT

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres, and cylinders. GLUT even has some limited support for creating pop-up menus. The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. All GLUT functions start with the glut prefix (for example, glutPostRedisplay marks the current window as needing to be redrawn).

## KEY STAGES IN THE OPENGL RENDERING PIPELINE:

### 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 - also known as immediate mode.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

### Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can

produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

#### Per-Vertex Operations

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

#### Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines.

The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

#### Pixel Operations

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

There are special pixel copy operations to copy data in the frame buffer to other parts of the frame buffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the frame buffer.

#### Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

### Rasterization

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

### Fragment Operations

Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled.

The first operation which may be encountered is texturing, where a texel (texture element) is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test (the depth buffer is for hidden-surface removal). Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed. Finally, the thoroughly processed fragment is drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

### OpenGL-Related Libraries

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following:

The 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, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. Portions of the GLU are described in the OpenGL.

For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix `glX`. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix `wgl`. For IBM OS/2, the PGL is the Presentation Manager to OpenGL interface, and its routines use the prefix `pgl`.

The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs.

Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

#### GLUT, the OpenGL Utility Toolkit

As you know, OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it's impossible to write a complete graphics program without at least opening a window, and most interesting programs require a bit of user input or other services from the operating system or window system. In many cases, complete programs make the most interesting examples, so this book uses GLUT to simplify opening windows, detecting input, and so on. If you have an implementation of OpenGL and GLUT on your system, the examples in this book should run without change when linked with them.

In addition, since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), GLUT includes several routines that create more complicated three-dimensional objects such as a sphere, a torus, and a teapot. This way, snapshots of program output can be interesting to look at. (Note that the OpenGL Utility Library, GLU, also has quadrics routines that create some of the same three-dimensional objects as GLUT, such as a sphere, cylinder, or cone.)

GLUT may not be satisfactory for full-featured OpenGL applications, but you may find it a useful starting point for learning OpenGL. The rest of this section briefly describes a small subset of GLUT routines so that you can follow the programming examples in the rest of this book.

#### OBJECTIVE AND APPLICATION OF THE LAB

The objective of this lab is to give students hands on learning exposure to understand and apply computer graphics with real world problems. The lab gives the direct experience to Visual Basic Integrated Development Environment (IDE) and GLUT toolkit. The students get a real world exposure to Windows programming API. Applications of this lab are profoundly felt in gaming industry, animation industry and Medical Image Processing Industry. The materials learned here will be useful in Programming at the Software Industry.

Setting up GLUT - main() GLUT provides high-level utilities to simplify OpenGL programming, especially in interacting with the Operating System (such as creating a window, handling key and mouse inputs). The following GLUT functions were used in the above program:

`glutInit`: initializes GLUT, must be called before other GL/GLUT functions. It takes the same arguments as the `main()`. `void glutInit(int *argc, char **argv)`  
`glutCreateWindow`: creates a window with the given title. `int glutCreateWindow(char *title)`

`glutInitWindowSize`: specifies the initial window width and height, in pixels.  
`void glutInitWindowSize(int width, int height)`

`glutInitWindowPosition`: positions the top-left corner of the initial window at (x, y). The coordinates (x, y), in term of pixels, is measured in window coordinates, i.e., origin (0, 0) is at the top-left corner of the screen; x-axis pointing right and y-axis pointing down. `void glutInitWindowPosition(int x, int y)`

`glutDisplayFunc`: registers the callback function (or event handler) for handling window-paint event. The OpenGL graphic system calls back this handler when it receives a window re-paint request. In the example, we register the function `display()` as the handler. `void glutDisplayFunc(void (*func)(void))`

`glutMainLoop`: enters the infinite event-processing loop, i.e, put the OpenGL graphics system to wait for events (such as re-paint), and trigger respective event handlers (such as `display()`). `void glutMainLoop()`

`glutInitDisplayMode`: requests a display with the specified mode, such as color mode (`GLUT_RGB`, `GLUT_RGBA`, `GLUT_INDEX`), single/double buffering (`GLUT_SINGLE`, `GLUT_DOUBLE`), enable depth (`GLUT_DEPTH`), joined with a bit OR '|'. `void glutInitDisplayMode(unsigned int displayMode)`

`void glMatrixMode (GLenum mode)`; The `glMatrixMode` function specifies which matrix is the current matrix.

`void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar)` The `glOrtho` function multiplies the current matrix by an orthographic matrix.

`void glPointSize (GLfloat size)`; The `glPointSize` function specifies the diameter of rasterized points.

`void glutPostRedisplay(void)`; `glutPostRedisplay` marks the current window as needing to be redisplayed.

`void glPushMatrix (void)`; `void glPopMatrix (void)`; The `glPushMatrix` and `glPopMatrix` functions push and pop the current matrix stack.

`GLint glRenderMode (GLenum mode)`; The `glRenderMode` function sets the rasterization mode.

`void glRotatef (GLfloat angle, GLfloat x, GLfloat y, GLfloat z)`; The `glRotatef` functions multiply the current matrix by a rotation matrix.

`void glScalef (GLfloat x, GLfloat y, GLfloat z)`; The `glScalef` functions multiply the current matrix by a general scaling matrix.

`void glTranslatef (GLfloat x, GLfloat y, GLfloat z)`; The `glTranslatef` functions multiply the current matrix by a translation matrix.

`void glViewport (GLint x, GLint y, GLsizei width, GLsizei height)`; The `glViewport` function sets the viewport.

`void glEnable, glDisable()`; The `glEnable` and `glDisable` functions enable or disable OpenGL capabilities.

`glutBitmapCharacter()`; The `glutBitmapCharacter` function used for font style.

### 3.2 Introduction to frequently used OpenGL commands

#### glutInit

glutInit is used to initialize the GLUT library.

##### Usage

```
void glutInit(int *argc, char **argv);
```

##### argc

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

##### argv

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

##### Description

glutInit will initialize the GLUT library. During this process, glutInit may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized.

#### glutInitWindowPosition, glutInitWindowSize

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

##### Usage

```
void glutInitWindowSize(int width, int height);
```

```
void glutInitWindowPosition(int x, int y);
```

Width in pixels.

##### height

Height in pixels.

##### x

Window X location in pixels.

##### y

Window Y location in pixels.

##### Description

Windows created by glutCreateWindow will be requested to be created with the current initial window position and size.

#### glutInitDisplayMode

glutInitDisplayMode sets the initial display mode.

Usage

```
void glutInitDisplayMode(unsigned int mode);  
mode
```

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

**GLUT\_RGB**

An alias for GLUT\_RGBA.

**GLUT\_INDEX**

Bit mask to select a color index mode window. This overrides GLUT\_RGBA if it is also specified.

**GLUT\_SINGLE**

Bit mask to select a single buffered window. This is the default if neither GLUT\_DOUBLE or GLUT\_SINGLE are specified.

**GLUT\_DOUBLE**

Bit mask to select a double buffered window. This overrides GLUT\_SINGLE if it is also specified.

**GLUT\_DEPTH**

Bit mask to select a window with a depth buffer.

Description

The initial display mode is used when creating top-level windows.

**glutMainLoop**

glutMainLoop enters the GLUT event processing loop.

Usage

```
void glutMainLoop(void);
```

Description

glutMainLoop enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

**glutCreateWindow**

glutCreateWindow creates a top-level window.

Usage

```
int glutCreateWindow(char *name);
```

name

ASCII character string for use as window name.

Description

glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

**glutPostRedisplay**

`glutPostRedisplay` marks the current window as needing to be redisplayed.

Usage

```
void glutPostRedisplay(void);
```

Description

Mark the normal plane of current window as needing to be redisplayed. The next iteration through `glutMainLoop`, the window's display callback will be called to redisplay the window's normal plane. Multiple calls to `glutPostRedisplay` before the next display callback opportunity generates only a single redisplay callback.

### `glutReshapeWindow`

`glutReshapeWindow` requests a change to the size of the current window.

Usage

```
void glutReshapeWindow(int width, int height);
```

`width`

New width of window in pixels.

`height`

New height of window in pixels.

Description

`glutReshapeWindow` requests a change in the size of the current window. The width and height parameters are size extents in pixels. The width and height must be positive values. The requests by `glutReshapeWindow` are not processed immediately. The request is executed after returning to the main event loop. This allows multiple `glutReshapeWindow`, `glutPositionWindow`, and `glutFullScreen` requests to the same window to be coalesced.

### `glutDisplayFunc`

`glutDisplayFunc` sets the display callback for the current window.

Usage

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

`func`

The new display callback function.

Description

`glutDisplayFunc` sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. Before the callback, the current window is set to the window needing to be redisplayed and (if no overlay display callback is registered) the layer in use is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback (this includes ancillary buffers if your program depends on their state).



### glutReshapeFunc

glutReshapeFunc sets the reshape callback for the current window.

#### Usage

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

The new reshape callback function.

#### Description

glutReshapeFunc sets the reshape callback for the current window. The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established. The width and height parameters of the callback specify the new window size in pixels. Before the callback, the current window is set to the window that has been reshaped.

### glFlush

glFlush - force execution of GL commands in real time

#### Description

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. glFlush empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in real time.

### glMatrixMode

glMatrixMode - specify which matrix is the current matrix

#### Usage

```
void glMatrixMode(GLenum mode)
```

#### Parameters

mode- Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: GL\_MODELVIEW, GL\_PROJECTION, and GL\_TEXTURE. The default value is GL\_MODELVIEW.

#### Description

glMatrixMode sets the current matrix mode. mode can assume one of three values: GL\_MODELVIEW - Applies subsequent matrix operations to the model matrix stack. GL\_PROJECTION - Applies subsequent matrix operations to the projection matrix stack.

### gluOrtho2D

#### Usage

```
gluOrtho2D(left, right, bottom, top)
```

Specifies the 2D region to be projected into the viewport. Any drawing outside the region will be automatically clipped away.

## 4 Viva questions

1. Explain all the OpenGL functions used in this program?
2. What is the principle of Sierpinski gasket?
3. Difference between additive and subtractive color?
4. What is the Graphics Architecture used in OpenGL?
5. What is Rasterisation?
6. Explain sierpinski Gasket (using points).
7. Explain sierpinski Gasket (using polye )
8. Difference between 2D Tetrahedron and 3D Tetrahedron.
9. What do you mean by clipping?
10. How is Liang-Barsky clipping different from CohenSutherland Clipping Algorithm?
11. How do you set the color attributes in Opengl?
12. What is the command for clearsreen?
13. What is Event callback function?
14. Explain Liang-Barsky line clipping algorithm?
15. Explain window to viewpoint mapping?
16. What are vertex arrays?
17. How are the faces of the color cube modeled?
18. How do you consider the inward and outward pointing of the faces?
19. Explain the OpenGL function used to rotate the color cube?
20. What is the difference between 2D and 3D orthographic projection statements?
21. Explain the Inward and Outward pointing face?
22. Explain the data structure for object representation?
23. Explain the vertex list representation of a cube?
24. What is transformation?
25. Explain the OpenGL functions used for translation, rotation and scaling?
26. What is the order of transformation?
27. State the difference between modelview and projection?
28. What is Homogeneous-coordinate representation?
29. Define the rotation matrix and object matrix.
30. Explain the procedure to obtain the resultant matrix using rotation matrix and object matrix.
31. What is the principle of Cohen-Sutherland Algorithm?
32. State the advantages and disadvantages of Cohen-Sutherland Algorithm?
33. What is an outcode?
34. What is Synthetic Camera Model?
35. What are the Camera Specifications?
36. Explain the cases of outcodes in Cohen-Sutherland algorithm.
37. Explain the cohen-sutherland line clipping algorithm
38. Mention the difference between Liang-Barsky and Cohen-Sutherland line clipping algorithm.
39. Explain about gluLookAt(...), glFrustum(...), gluPerspective?
40. Explain the different types of projections?
41. Explain Z-buffer algorithm?

42. What is antialiasing?
43. What is Center Of Projection(COP), Direction Of Projection(DOP)?
44. What is midpoint circle drawing algorithm
45. How do you get the equation  $d = 2x + 3$ ,  $d = 2(x - y) + 5$
46. What is gluOrtho2D function
47. Explain plot pixel function
48. Why do we use GLFlush function in Display
49. Explain Specular, Diffuse and Translucent surfaces.
50. What is ambient light?
51. What is umbra, penumbra?
52. Explain Phong lighting model.
53. Explain glLightfv(...), glMaterialfv(...).
54. What is glutSolidCube function ? what are its Parameters
55. What are the parameters to glScale function
56. Explain Push & Pop matrix Functions
57. What is Materialfv function & its Parameters
58. Explain GLULookAt Function
59. Explain the keyboard and mouse events used in the program?
60. What is Hidden surface Removal? How do you achieve this in OpenGL?
61. What are the functions for creating Menus in OpenGL?
62. Explain about fonts in GLUT?
63. Explain about glutPostRedisplay()?
64. Explain how the cube is constructed
65. Explain rotate function
66. What is GLFrustum function what are its Parameters
67. What is viewport
68. What is glutKeyboard Function what are its Parameters
69. Explain scanline filling algorithm?
70. What is AspectRatio,Viewport?
71. How do you use timer?
72. What are the different frames in OpenGL?
73. What is fragment processing?
74. Explain Polygon Filling algorithm
75. What is slope
76. How the edges of polygon are detected
77. Why you use GL\_PROJECTION in MatrixMode Function
78. What is dx & dy
79. What is maxx & maxy
80. What is glutPostRedisplay
81. Why do we use glutMainLoop function
82. What do you mean by GL\_LINE\_LOOP in GL\_Begin function
83. Define Computer Graphics.
84. Explain any 3 uses of computer graphics applications.
85. What are the advantages of DDA algorithm?
86. What are the disadvantages of DDA algorithm?
87. Define Scan-line Polygon fill algorithm.
88. What are Inside-Outside tests?
89. Define Boundary-Fill algorithm.
90. Define Flood-Fill algorithm.

91. Define attribute parameter. Give examples.
92. What is a line width? What is the command used to draw the thickness of lines.
93. What are the three types of thick lines? Define.
94. What are the attribute commands for a line color?
95. What is color table? List the color codes.
96. What is a marker symbol and where it is used?
97. Discuss about inquiry functions.
98. Define translation and translation vector.
99. Define window and view port.
100. Define viewing transformation.
101. Give the equation for window to viewport transformation.
102. Define view up vector.
103. What is meant by clipping? Where it happens?
104. What is point clipping and what are its inequalities?
105. What is line clipping and what are their parametric representations?
106. How is translation applied?
107. What is referred to as rotation?
108. Write down the rotation equation and rotation matrix.
109. Write the matrix representation for scaling, translation and rotation.
110. Draw the block diagram for 2D viewing transformation pipeline.
111. Mention the equation for homogeneous transformation.
112. What is known as composition of matrix?
113. Write the composition transformation matrix for scaling, translation and Rotation.
114. Discuss about the general pivot point rotation?
115. Discuss about the general fixed point scaling.
116. Explain window, view port and window – to – view port transformation
117. Mention the three raster functions available in graphics packages.
118. What is known as region codes?
119. Why Cohen Sutherland line clipping is popular?
120. Mention the point clipping condition for the liang-barsky line clipping.
121. What is called as an exterior clipping?
122. How is the region code bit values determined?
123. Why liang-barsky line clipping is more efficient than Cohen Sutherland line Clipping?
124. Differentiate uniform and differential scaling
125. Explain soft fill procedures.
126. Explain the three primary color used in graphics
127. Explain in detail about color and grey scale levels?
128. Explain color and grey scale levels.
129. Explain the area fill attributes and character attributes.
130. Explain character attributes in detail.
131. Briefly discuss about basic transformations.
132. Explain matrix representations.
133. Discuss about composite transformations.
134. Explain about reflection and shear.
135. Explain the following transformation with the matrix representations.  
Give suitable diagram for illustration translation .ii scaling.iii rotation.
136. How the rotation of an object about the pivot point is performed?
137. How window-to-viewport coordinate transformation happens.
138. Explain clipping with its operation in detail.

139. Explain Cohen- Sutherland line clipping.
140. Discuss the logical classifications of input devices.
141. Explain the details of 2d viewing transformation pipeline.
142. Explain point, line, curve, text, exterior clipping?
143. Discuss the properties of light.
144. Define chromaticity, complementary colors, color gamut and primary colors.
145. What is color model?
146. Define hue, saturation and value.
147. Explain XYZ color model.
148. Explain RGB color model.
149. Explain YIQ color model.
150. Explain CMY color model.
151. Explain HSV color model.
152. Give the procedure to convert HSV & RGB color model.
153. What is the use of chromaticity diagram?
154. What is illumination model?
155. What are the basic illumination models?
156. What is called as lightness?
157. Explain the conversion of CMY to RGB representation.
158. What is animation?
159. Define Morphing.
160. What are the steps involved in designing an animation sequence?
161. How to draw dots using OPENGL?
162. How to draw lines using OPENGL?
163. How to draw convex polygons using OPENGL?
164. What is the command used in OPENGL to clear the screen?
165. What are the various OPENGL data types?

## 5 Viva questions and answers

1. What is Computer Graphics?  
Answer: Computer graphics are graphics created using computers and, more generally, the representation and manipulation of image data by a computer.
2. What is OpenGL?  
Answer: OpenGL is the most extensively documented 3D graphics API (Application Program Interface) to date. It is used to create Graphics.
3. What is GLUT?  
Answer: The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system.
4. What are the applications of Computer Graphics?  
Answer: Gaming Industry, Animation Industry and Medical Image Processing Industries. The sum total of these industries is a Multi Billion Dollar Market. Jobs will continue to increase in this arena in the future.
5. Explain in brief 3D Sierpinski gasket?  
Answer: The Sierpinski triangle (also with the original orthography Sierpinski), also called the Sierpinski gasket or the Sierpinski Sieve, is a fractal named after the Polish mathematician Waclaw Sierpinski who described it in 1915. Originally constructed as a curve, this is one of the basic examples of self-similar sets, i.e. it is a mathematically generated pattern that can be reproducible at any magnification or reduction.
6. What is Liang-Barsky line clipping algorithm?  
Answer: In computer graphics, the Liang-Barsky algorithm is a line clipping algorithm. The Liang-Barsky algorithm uses the parametric equation of a line and inequalities describing the range of the clipping box to determine the intersections between the line and the clipping box. With these intersections it knows which portion of the line should be drawn.
7. Explain in brief Cohen-Sutherland line-clipping algorithm?  
Answer: The Cohen-Sutherland line clipping algorithm quickly detects and dispenses with two common and trivial cases. To clip a line, we need to consider only its endpoints. If both endpoints of a line lie inside the window, the entire line lies inside the window. It is trivially accepted and needs no clipping. On the other hand, if both endpoints of a line lie entirely to one side of the window, the line must lie entirely outside of the window. It is trivially rejected and needs to be neither clipped nor displayed.
8. Explain in brief scan-line filling algorithm?  
Answer: The scanline fill algorithm is an ingenious way of filling in irregular polygons. The algorithm begins with a set of points. Each point is connected to the next, and the line between them is considered to be an edge of the polygon. The points of each edge are adjusted to ensure that the point with the smaller y value appears first. Next, a data structure is created that contains a list of edges that begin on each scanline of the image. The program progresses from the first scanline upward. For each line, any pixels that contain an intersection between this scanline and an edge of the polygon are filled in. Then, the algorithm progresses along the scanline, turning on when it reaches a polygon pixel and turning off when it reaches another one, all the way across the scanline.

## 9. Explain Midpoint Line algorithm

Answer: The Midpoint line algorithm is an algorithm which determines which points in an n-dimensional raster should be plotted in order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures.

## 10. What is a Pixel?

Answer: In digital imaging, a pixel (or picture element) is a single point in a raster image. The Pixel is the smallest addressable screen element; it is the smallest unit of picture which can be controlled. Each Pixel has its address. The address of Pixels corresponds to its coordinate. Pixels are normally arranged in a 2-dimensional grid, and are often represented using dots or squares.

## 11. What is Graphical User Interface?

Answer: A graphical user interface (GUI) is a type of user interface item that allows people to interact with programs in more ways than typing such as computers; hand-held devices such as MP3 Players, Portable Media Players or Gaming devices; household appliances and office equipment with images rather than text commands.

## 12. What is the general form of an OpenGL program?

Answer: There are no hard and fast rules. The following pseudocode is generally recognized as good OpenGL form.

```
program_entrypoint
{
    // Determine which depth or pixel format should be used.
    // Create a window with the desired format.
    // Create a rendering context and make it current with the window.
    // Set up initial OpenGL state.
    // Set up callback routines for window resize and window refresh.
}
handle_resize
{
    glViewport(...);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Set projection transform with glOrtho, glFrustum, gluOrtho2D, gluPerspective, etc.
}
handle_refresh
{
    glClear(...);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // Set view transform with gluLookAt or equivalent
    // For each object (i) in the scene that needs to be rendered:
    // Push relevant stacks, e.g., glPushMatrix, glPushAttrib.
    // Set OpenGL state specific to object (i).
    // Set model transform for object (i) using glTranslatef, glScalef, glRotatef, and/or equivalent.
    // Issue rendering commands for object (i).
```



```
// Pop relevant stacks, (e.g., glPopMatrix, glPopAttrib.)  
// End for loop.  
// Swap buffers.  
}
```

13. What support for OpenGL does Open,Net,FreeBSD or Linux provide?  
Answer: The X Windows implementation, XFree86 4.0, includes support for OpenGL using Mesa or the OpenGL Sample Implementation. XFree86 is released under the XFree86 license. <http://www.xfree86.org/>
14. What is the AUX library?  
Answer: The AUX library was developed by SGI early in OpenGL's life to ease creation of small OpenGL demonstration programs. It's currently neither supported nor maintained. Developing OpenGL programs using AUX is strongly discouraged. Use the GLUT instead. It's more flexible and powerful and is available on a wide range of platforms. Very important: Don't use AUX. Use GLUT instead.
15. How does the camera work in OpenGL?  
Answer: 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.). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation.
16. How do I implement a zoom operation?  
Answer: 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.
17. What are OpenGL coordinate units?  
Answer: Depending on the contents of your geometry database, it may be convenient for your application to treat one OpenGL coordinate unit as being equal to one millimeter or one parsec or anything in between (or larger or smaller). OpenGL also lets you specify your geometry with coordinates of differing values. For example, you may find it convenient to model an airplane's controls in centimeters, its fuselage in meters, and a world to fly around in kilometers. OpenGL's ModelView matrix can then scale these different coordinate systems into the same eye coordinate space. It's the application's responsibility to ensure that the Projection and ModelView matrices are constructed to provide an image that keeps the viewer at an appropriate distance, with an appropriate field of view, and keeps the zNear and zFar clipping planes at an appropriate range. An application that displays molecules in micron scale, for example, would probably not want to place the viewer at a distance of 10 feet with a 60 degree field of view.
18. What is Microsoft Visual Studio?  
Answer: Microsoft Visual Studio is an integrated development environment (IDE) for developing windows applications. It is the most popular IDE for developing windows applications or windows based software.
19. What does the .gl or .GLE format have to do with OpenGL?  
Answer: .gl files have nothing to do with OpenGL, but are sometimes confused with it. .gl is a file format for images, which has no relationship to OpenGL.

20. Who needs to license OpenGL? Who doesn't? Is OpenGL free software?  
Answer: Companies which will be creating or selling binaries of the OpenGL library will need to license OpenGL. Typical examples of licensees include hardware vendors, such as Digital Equipment, and IBM who would distribute OpenGL with the system software on their workstations or PCs. Also, some software vendors, such as Portable Graphics and Template Graphics, have a business in creating and distributing versions of OpenGL, and they need to license OpenGL. Applications developers do NOT need to license OpenGL. If a developer wants to use OpenGL that developer needs to obtain copies of a linkable OpenGL library for a particular machine. Those OpenGL libraries may be bundled in with the development and/or run-time options or may be purchased from a third-party software vendor, without licensing the source code or use of the OpenGL trademark.
21. How do we make shadows in OpenGL? Answer: There are no individual routines to control neither shadows nor an OpenGL state for shadows. However, code can be written to render shadows.
22. What is the use of GlutInit?  
Answer: `void glutInit(int *argc, char **argv);`  
`glutInit` will initialize the GLUT library and negotiate a session with the window system. During this process, `glutInit` may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized.
23. Describe the usage of `glutInitWindowSize` and `glutInitWindowPosition`?  
Answer: `void glutInitWindowSize(int width, int height);`  
`void glutInitWindowPosition(int x, int y);`  
Windows created by `glutCreateWindow` will be requested to be created with the current initial window position and size. The intent of the initial window position and size values is to provide a suggestion to the window system for a window's initial size and position. The window system is not obligated to use this information. Therefore, GLUT programs should not assume the window was created at the specified size or position. A GLUT program should use the window's reshape callback to determine the true size of the window.
24. Describe the usage of `glutMainLoop`?  
Answer: `void glutMainLoop(void);`  
`glutMainLoop` enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.