PESIT Bangalore South Campus

10CSL67-COMPUTER GRAPHICS AND VISUALIZATION LABORATORY

Faculty: Mrs.Sarasvathi V/Archana Mathur

PART - A

IMPLEMENT THE FOLLOWING PROGRAMS IN C / C++

1. Program to recursively subdivide a tetrahedron to from 3D Sierpinski gasket. The number of recursive steps is to be specified by the user.
2. Program to implement Liang-Barsky line clipping algorithm.
3. Program to draw a color cube and spin it using OpenGL transformation matrices.
4. Program to create a house like figure and rotate it about a given fixed point using OpenGL functions.
5. Program to implement the Cohen-Sutherland line-clipping algorithm. Make provision to specify the input line, window for clipping and view port for displaying the clipped image.
6. Program to create a cylinder and a parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral.
7. Program, using OpenGL functions, to draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the properties of the surfaces of the solid object used in the scene.
8. Program to draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing. Use OpenGL functions.
9. Program to fill any given polygon using scan-line area filling algorithm. (Use appropriate data structures.)
10. Program to display a set of values { f_{ij} } as a rectangular mesh.

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.

REFERENCE BOOKS:

1. **Computer Graphics Using OpenGL** – F.S. Hill,Jr. – 2nd Edition, Pearson education, 2001.
2. **Interactive Computer Graphics A Top-Down Approach with OpenGL Edward Angel** – 2nd Edition, Addison-Wesley, 2000.

WEEK NO	**PROGRAMS TO BE COVERED**
1	Introduction to simple OpenGL programs
2	Part A: Program 1
3	Part A: Program 6 and 10
4	Part A: Program 2
5	Part A: Program 5
6	Part A: Program 4
7	Part A: Program 3
8	Part A: Program 8
9	Part A: Program 9
10	Part A: Program 7
11	Part B: Project
12	Part B: project

PART A	
S.NO	***IMPLEMENT THE FOLLOWING PROGRAMS IN C / C++***
	Introduction to simple OPENGL programs.
01.	Program to recursively subdivide a tetrahedron to from 3D Sierpinski gasket. The number of recursive steps is to be specified by the user.
02.	Program to implement Liang-Barsky line clipping algorithm.
03.	Program to draw a color cube and spin it using OpenGL transformation matrices.
04	Program to create a house like figure and rotate it about a given fixed point using OpenGL functions.
05.	Program to implement the Cohen-Sutherland line-clipping algorithm. Make provision to specify the input line, window for clipping and view port for displaying the clipped image.
06.	Program to create a cylinder and a parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral.

07.	Program, using OpenGL functions, to draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the properties of the surfaces of the solid object used in the scene.
08.	Program to draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing. Use OpenGL functions.
09.	Program to fill any given polygon using scan-line area filling algorithm. (Use appropriate data structures.)
10.	Program to display a set of values {fij} as a rectangular mesh.
	PART B - PROJECT

OPENGL

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.

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 the with most basic GLUT_SINGLE, which will give use 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 whe 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. Lets 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 teh 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(); //flush 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 user so that anything we draw at 0,0,0 will be seen infront 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.

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

Sierpinski's Triangle

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.

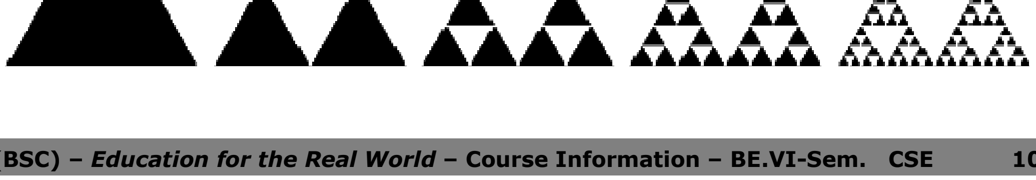
Creating the fractal requires little computational power. Even simple graphing calculators can easily make this image. The fractal is created pixel by pixel, using random numbers; the fractal will be slightly different each time due to this. Although, if you were to run the program repeatedly, and allow 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.

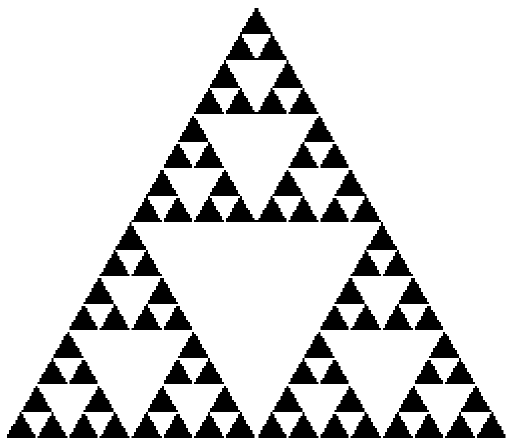
We begin with a triangle in the plane and then apply a repetitive scheme of operations to it (when we say triangle here, we mean a blackened, 'filled-in' triangle). Pick the midpoints of its three sides. Together with the old verticies of the original triangle, these midpoints define four congruent triangles 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 verticies of two contiguous trianges. 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 preceeding step.

ALGORITHM



- Start with a triangle
- Connect bisectors of sides and remove central triangle
- Repeat

FINAL TRIANGLE



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 triagle 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.
```

2. Program to implement Liang-Barsky line clipping algorithm.

Point and Line Clipping

Clipping:

Remove points outside a region of interest.

- Discard (parts of) primitives outside our window...

Point clipping:

Remove points outside window.

- A point is either entirely inside the region or not.

Line clipping:

Remove portion of line segment outside window.

- Line segments can straddle the region boundary.
- Liang-Barsky algorithm efficiently clips line segments to a halfspace.
- Halfspaces can be combined to bound a convex region.
- Use **outcodes** to better organize combinations of halfspaces.
- Can use some of the ideas in Liang-Barsky to clip points.

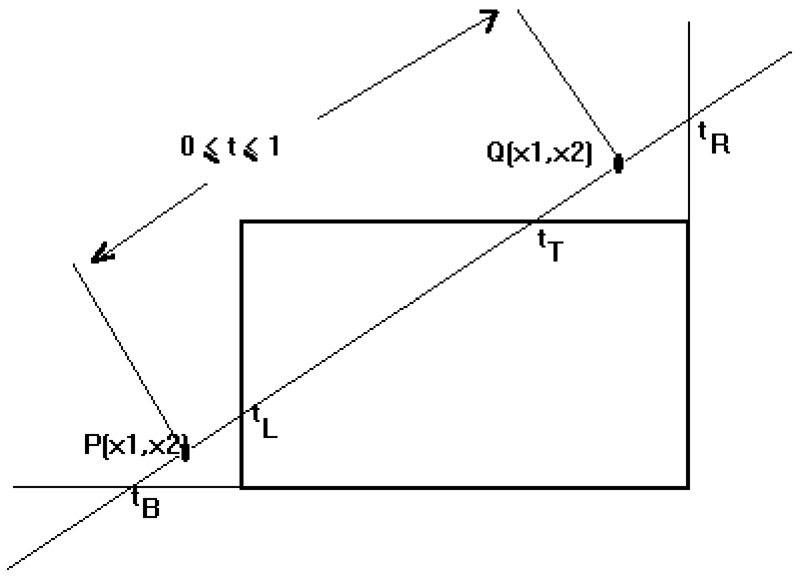
Parametric representation of line:

$$L(t) = (1 - t)A + tB$$

or equivalently

$$L(t) = A + t(B - A)$$

Liang and Barsky have created an algorithm that uses floating-point arithmetic but finds the appropriate end points with at most four computations. This algorithm uses the parametric equations for a line and solves four inequalities to find the range of the parameter for which the line is in the viewport.



Let $P(x_1,y_1)$, $Q(x_2,y_2)$ be the line which we want to study. The **parametric equation of the line segment** from gives x-values and y-values for every point in terms of a **parameter t** that ranges from 0 to 1. The equations are

$x = x_1 + (x_2 - x_1)*t = x_1 + dx*t$ and $y = y_1 + (y_2 - y_1)*t = y_1 + dy*t$

We can see that when t = 0, the point computed is P(x1,y1); and when t = 1, the point computed is Q(x2,y2).

Algorithm

1. Set $t_{min} = 0$ and $t_{max} = 1$
2. Calculate the values of tL, tR, tT, and tB .
   - if $t < t_{min}$ or $t > t_{max}$ ignore it and go to the next edge
   - otherwise classify the **t**value as entering or exiting value (using inner product to classify)
   - if **t** is entering value set $t_{min} = t$; if **t** is exiting value set $t_{max} = t$
3. If $t_{min} < t_{max}$ then **draw a line** from (x1 + dx*tmin, y1 + dy*tmin) to (x1 + dx*tmax, y1 + dy*tmax)
4. If the line crosses over the window, you will see (x1 + dx*tmin, y1 + dy*tmin) and (x1 + dx*tmax, y1 + dy*tmax) are intersection between line and edge.

More simple algorithm

1. Let P(x1,y1) and Q(x2,y2) be the outer points of initial line

2. Set Tmin = 0 and Tmax = 1 (Tmin represents the starting position and Tmax represents ending position of actual phase of clipping the line relatively to initial line; at beginning positions of initial line)

3. Use coordinates of clipping edges to find values of individual inequalities (generally t or individualy tL,tR,tT,tB)

From parametric equation x = x1 + (x2-x1)*t, y = y1 + (y2-y1)*t (0<=t<=1):
- let R (R = x1 + t*(x2 - x1)) is x coordinate of right edge - tL = (L-x1)/(x2-x1)
- let L (L = x1 + t*(x2 - x1))is x coordinate of left edge - tR = (R-x1)/(x2-x1)
- let T is y coordinate of top edge - tT = (T-y1)/(y2-y1)
- let B is y coordinate of bottom edge - tB = (B-y1)/(y2-y1)

If ttmax ignore it and go to the next edge otherwise classify the value of t as entering or exiting value (using inner product to classify). If t is entering value set tmin = t. If t is exiting value set tmax =t.

4. If tmin < tmax then draw a line from (x1 + dx*tmin, y1 + dy*tmin) to (x1 + dx*tmax, y1 + dy*tmax).

5. If the line crosses over the rectangle, you will see
(x1 + dx*tmin, y1 + dy*tmin) and (x1 + dx*tmax, y1 + dy*tmax)
are intersection between line and edge.

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

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);
        polygo n(0,1,5,4);
}
```

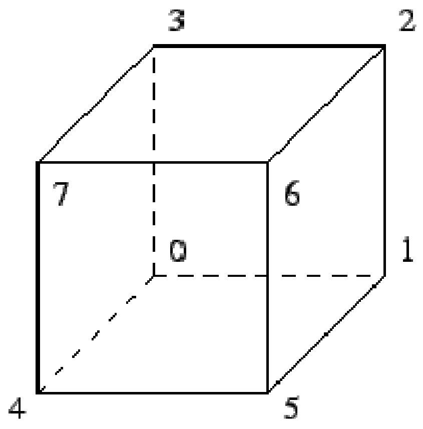
4. Define the Display function to clear frame buffer , Z-buffer ,rotate cube and draw,swap
buffers.

```
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.



4. Program to create a house like figure and rotate it about a given fixed point using OpenGL functions.

PSEUDO CODE

```
1. Initialize the house array to draw the house
      GLfloat  house[3][9] =
   {{100.0,100.0,175.0,250.0,250.0,150.0,150.0,200.0,200.0},{100.0,300.0,400.0,300.0,
   100.0,100.0,150.0,150.0,100.0},{1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}};
2. Initialize the rotation matrix as follow:
   GLfloat rot_mat[3][3]={{0},{0},{0}};
3. Initialize the resultant matrix
   GLfloat result[3][9] = {{0},{0},{0}};
4. Initialize the pivot point h,k as 100,100
5. Define the rotate function as follows:
   i) Calculate m and n value m=-h*(cos(theta)-1)+k*(sin(theta));
   n=-k*(cos(theta)-1)-h*(sin(theta));
   ii) Rotation matrix is
                  cos(theta)    -sin(theta)    m
                  sin(theta)     cos(theta)    n
                  0               0            1
   iii) Call Multiply function to multiply the two matrices Rotation matrix and object
        matrix.
6. Define the multiply function to multiply Rotation matrix and object matrix to give
   resultant transformed house.
7. Define the function drawhouse to draw house using GL_LINE_LOOP.
8. Define the function Drawrotatehouse to draw house after rotating.
9. Define the display function which is used to call all function.
10. Define the myinit function to specify matrixmode and gluortho.
11. Define main to get theta value from user and to create window.
```

5. **Program to implement the Cohen-Sutherland line-clipping algorithm. Make provision to specify the input line, window for clipping and view port for displaying the clipped image.**

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivally accepted or rejected. If the line cannot be trivally 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.

Algorithm

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

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

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

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

6. **Program to create a cylinder and a parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral.**

Parallelepiped

In geometry, a **parallelepiped** is a three-dimensional figure formed by six parallelograms. (The term rhomboid is also sometimes used with this meaning.) It is to a parallelogram as a cube is to a square: Euclidean geometry supports all four notions but affine geometry admits only parallelograms and parallelepipeds. Three equivalent definitions of *parallelepiped* are

- a polyhedron with six faces (hexahedron), each of which is a parallelogram,
- a hexahedron with three pairs of parallel faces, and
- a prism of which the base is a parallelogram.

The rectangular cuboid (six rectangular faces), cube (six square faces), and the rhombohedron (six rhombus faces) are all specific cases of parallelepiped.

DEFINITION

Solid figure, having six faces, all parallelograms; all opposite faces being similar and parallel .

In geometry, a **quadrilateral** is a polygon with four 'sides' or **edges** and four **vertices** or **corners**. Sometimes, the term **quadrangle** is used, for analogy with triangle, and sometimes **tetragon** for consistency with pentagon (5-sided), hexagon (6-sided) and so on. The word **quadrilateral** is made of the words **quad** and **lateral**. **Quad** means four and **lateral** means sides. The interior angles of a quadrilateral add up to 360 degrees of arc.

PSEUDO CODE

```
1. Include glut header files.
2. Define draw_pixel function to plot points.
      Void  draw_pixel(Glint cx,Glint cy)
      {

              glcolor3f(1.0,0.0,0.0);
              glBegin(GL_POINTS);
              glVertex2i(cx,cy);
              glEnd();
         }

3. Define plotpixel function  to plot eight points in perimeter of circle.
```

```
  Void plotpixel(Glint h,GLint k,Glint x,Glint y)
 {
         draw_pixel(x+h,y+k);
         draw_pixel(-x+h,y+k);
         draw_pixel(x+h,-y+k);
         draw_pixel(-x+h,-y+k);
         draw_pixel(y+h,x+k);
         draw_pixel(-y+h,x+h);
         draw_pixel(y+h,-x+h);
         draw_pixel(-y+h,-x+h);
}
```

4. Define circle_draw function to draw circle using mid point circle algorithm by calling plotpixel function.
5. Define cylinder_draw function and initialize mid point and radius value.
    i) initialize n value as 50
    ii) for(i=10,i<n;i+=3)

```
         {
  Circle_draw(xc,yc+I,r);
         }
```

6. Define parallelepiped function to draw 4 sides of parallelepiped.

```
  Void parallelepiped(int x1,int x2,int y1,int y2,int y3,int y4)
  {
          glcolor3f(0.0,0.0,1.0);
          glpointSize(2.0);
          glBegin(GL_LINE_LOOP);
          glVertex2i(x1,y1);
          glVertex2i(x2,y3);
          glVertex2i(x2,y4);
          glVertex2i(x1,y2);
          glEnd();
   }
```

7. Define parallelepiped_draw function.
    i) Initialize all vertices value and n=40;

```
          for(i=0;i<n,i+=2)
          {
            parallelepiped(x1+i,x2+i,y1+i,y2+i,y3+i,y4+i)
           }
```

8. Define display function to call cylinder_draw and parallelepiped_draw function.
9. Define main function to create window and to call all function.

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

PSEUDO CODE

1. Include glut heaeder 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);
    gPopMatrix();
}
```

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)

8. Program to draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing. Use OpenGL functions.

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);
        polygo n(0,1,5,4);
}
```

1. Intialize 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}
```

2. Define display function to upadate 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();
}
```

3. Define mouse callback function to rotate about axis.

7. Define key function to move viwer position .use x,X,y,Y z,Z keys to move viewer position.

8. Define reshape function to maintain aspect ratioand then use perspective view.
9. Define main fuction to create window and to call all function.

9. **Program to fill any given polygon using scan-line area filling algorithm. (Use appropriate data structures.)**

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
  1. Eliminate horizontal edges
  2. 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

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), the 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

10. **Program to display a set of values { f_{ij} } as a rectangular mesh.**

PSEUDO CODE

1. Include the header files.
2. Define the maxx ,maxy,n value and dx,dy value
3. Initialize the values for x0 and y0 value and declare array x[maxx],y[maxy].
4. Define the init function to clear color ,defining point size ,to specify matrix mode,to request redisplay..

```
    Void init()
    {
                    glClearColor(1.0,1.0,1.0,1.0);
                    glColor3f(1.0,0.0,0.0);
                    glPointSize(5.0);
                    glMatrixMode(GL_PROJECTION);
                    glLoadIdentity();
                    gluOrtho2D(0.0,499.0,0.0,499.0);
                    glutPostRedisplay();
    }
```

5. Define display function

    i) clear the window using glClear(GL_COLOR_BUFFER_BIT)
    ii) set the color
    iii) compute x[i] and y[i] value by adding dx ,dy value to x0 and y0 repectively.
    iv) Using GL_LINE_LOOP draw the lines.

6. Define main function to create window and to call display and init function