```
/* 2. Program to recursively subdivide a tetrahedron to from 3D Sierpinski gasket.
      The number of recursive steps is to be specified by the user. */

/* A fractal is an abstract object used to describe and simulate naturally occurring objects,
     fractals include the idea of a detailed pattern that repeats itself
   Artificially created fractals commonly exhibit similar patterns at increasingly small scales
   It is also known as expanding symmetry or evolving symmetry
   If the replication is exactly the same at every scale, it is called a self-similar pattern
   It is a mathematically generated pattern that can be reproducible at any magnification
     or reduction, named after the Polish mathematician Wacław Sierpiński

   Polish ! Warsaw,  Maria Skłodowska-Curie, Henryk Sienkiewicz, Bagels,
                  Łukasiewicz notation

   Attractive fixed set : is an element of the function's domain that is mapped to itself by the
     function , c is a fixed point of the function f(x) if f(c) = c

   An equilateral triangle is a triangle in which all three sides are equal
   Equilateral triangles are also equiangular, all three internal angles are also congruent to
     each other and are each 60°

   Sierpinski triangle (orthography Sierpiński), Sierpinski gasket or the Sierpinski Sieve
     is a fractal and attractive fixed set with the overall shape of an equilateral triangle,
     subdivided recursively into smaller equilateral triangles

   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 */

/* A fractal is 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

   Example : Consider triangle, on a 2D plane

                          ( x3, y3 )
                              /\
                             /  \
                            /    \
                           /      \
                          /        \
                         /          \
                        /            \
                       /______________\

                 ( x1, y1 )              ( x2, y2 )     */

/* Sierpiński gasket :
     Repeat for all the verticies :
       Creating gasket, i.e. start with a vertex of the object and get all the mid points
         to the other verticies
       This results in smaller strucure of the original geometric object

       For another iteration take the smaller objects created in last setp and perform above steps
      Creating gasket, i.e. start with a vertex of the object and get all the mid points
        to the other verticies
                                   ( x3, y3 )
                                       /\
                                      /    \
                                     /       \
                                    /          \
     (x1+x3)/2, (y1+y3)/2 *                  * (x2+x3)/2, (y2+y3)/2
                                /                 \
                               /                   \
                              /______*______\

                ( x1, y1 )        (x1+x2)/2,        ( x2, y2 )
                                  (y1+y2)/2


      This results in smaller strucure of the original geometric object

                                   ( x3, y3 )
                                       /\
                                      /    \
                                     /       \
                                    /          \
     (x1+x3)/2, (y1+y3)/2 * - - - - - - - - * (x2+x3)/2, (y2+y3)/2
                                /   \          /   \
                               /       \      /       \
                              /_____\/_____\

                ( x1, y1 )              *           ( x2, y2 )
                                  (x1+x2)/2,
                                  (y1+y2)/2



      The gasket is perfectly self similar, an attribute of many fractal images, any portion
         is an exact replica of the phase of the gasket

      The construction of the 3 dimensional version of the gasket follows similar rules for the
        2D case except that the building blocks are square based pyramids instead of triangles */

/* Algorithm flow :
     Start with a triangle in the plane
     Apply repetitive scheme of operations to it:
        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 triangle

     Follow the same procedure with the three new triangles  */

// Algorithm generates 1, 3, 9, 27, 81, 243, ... 3^m triangles for m number of divisions

/* Algorithm:
 Given m, number of steps

 Start with a tetrahedron
   Take the edges, calculate their mid points
   From each original vertex construct a new tetrahedron consisting of
     vertex(original,midpoint,midpoint,midpoint)
 repeat it recursively ( till m > 0 )

 At m = 0 draw all the leaf nodes of the recursive tree

OR

 Start with a triangle
 Connect bisectors of sides and remove central triangle
 Repeat m times  */

/* Pseudocode:
 Given : m number of divisions
 Initialize array to hold four vertices of tetrahedron

 draw_triangle( three points )
   {
     Using three points display one triangle;
   }

 divide_tetrahedron( four points , m ) // Subdivide a tetrahedron
   {
     if m > 0
       {
         Compute six midpoints ( Three visible joined triangles of tetrahedron have three
                                   common edges, hence three common mid points
                                 Three uncommon edges, so three more mid points ) ;
         make four divide_tetrahedron(  ) calls; // Create 4 tetrahedrons
       }
     else
         draw_triangle( three points );
   }

 display( )
   {
     Clear the color buffer;
     tetrahedron( );
   }

 main( )
   {
     register display function;
     create window;
   }  */
 #include <stdio.h>
 #include <GL/glut.h>

 typedef float point[3];        // Why 3 ? Can it be any arbitary number ?

 point v[] = { { 0.0, 0.0, 1.0 }, // Consider point as 3 dimension, plot / mark the points
               { 0.0, 0.942809, -0.333333 }, // What is it ?
               { -0.816497, -0.471405, -0.333333 },
               { 0.816497, -0.471405, -0.333333 }    }; // Why 4 points ?

/*  Three dimension / co ordinates , ( x , y , z ) system

                       + y
                        ^
                         |
                         |
                         |
                         |
                        / \
                    /       \
                 /             \
              /                   \
           + z                      + x
*/

/*  Tetrahedron :  (0, 0, 1) , (0, 0.9, -.3) , (-.8, -.4, -.3) , (.8, -.4, -.3)

                       + y
                        ^
                         |  *  (0, 0.9, -.3)
                         |
            (-.8, -.4, -.3)|
                *        |
                        / \
                    /       \
                 /             \
     (0, 0, 1)*                     \ +x
           + z                      *  (.8, -.4, -.3)
*/

/*  Tetrahedron :  (0, 0, 1) , (0, 0.9, -.3) , (-.8, -.4, -.3) , (.8, -.4, -.3)

                       +y
                          *
                          ^
                       / /  \
                    /   /    \
                *     /        \
                /   /            \
               /  /                \
             /  /                    \
            * --------_______         \  +x
         + z                -------- *
*/
```

```
 int n;

 void draw_triangle( point a, point b, point c ) // Draw triangle given three points a, b, c
  {
   glBegin( GL_POLYGON );//delimit the vertices of a primitive or a group of like primitives
// GL_POLYGON : Draws a single, convex polygon. Vertices 1 through N define this polygon
   glVertex3fv( a );  // specify a vertex
   glVertex3fv( b );
   glVertex3fv( c );
   glEnd(); // glBegin and glEnd delimit the vertices that define a primitive or a group of like
primitives
  }

 void midpoint( point a, point b, point save ) // Find mid point of line ab, given end
  { // points : a and b; coordinates of mid points = ∀ coordinates: ∑ of coordinates / 2
   save[0]=( a[0] + b[0] ) / 2; //
   save[1]=( a[1] + b[1] ) / 2;
   save[2]=( a[2] + b[2] ) / 2;
  }
 // Divide the triangles forming tetrahedron m times, given four end points of tetrahedron
 void divide_tetrahedron( point a, point b, point c, point d, int m )
  {
   point mab, mac, mad, mbc, mbd, mcd; // Six edges => hence six mid points

   if( m > 0 ) // Stopping condition, if m > 0
    { // Find mid point of point p1 and p2 and save it in mp1p2 , using mid point formulae
     midpoint( a, b, mab );
     midpoint( a, c, mac );
     midpoint( a, d, mad );
     midpoint( b, c, mbc );
     midpoint( b, d, mbd );
     midpoint( c, d, mcd );

     // consider midpoints as new vertices and now divide the bigger tetrahedron into
     divide_tetrahedron( a, mab, mac, mad, m-1 ); // four tertahedron
     divide_tetrahedron( mab, b, mbc, mbd, m-1 );
     divide_tetrahedron( mac, mbc, c, mcd, m-1 );
     divide_tetrahedron( mad, mbd, mcd, d, m-1 );
    }
   else
    { //draw the traingles forming tetrahedron in different color
     glColor3f( 1.0, 0.0, 0.0 );  draw_triangle( a, b, c ); // Color = Red
     glColor3f( 0.0, 0.0, 1.0 );  draw_triangle( a, c, d ); // Color = Blue
     glColor3f( 0.0, 1.0, 0.0 );  draw_triangle( c, b, d ); // Color = ?
     glColor3f( 0.0, 0.0, 0.0 );  draw_triangle( a, b, d ); // Color = ?
    } // glColor3f : Specify new red, green, and blue values for the current color
  }

 void display( void )
  { //Called everytime the display is refreshed, draws a tetrahedron
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glLoadIdentity();
   divide_tetrahedron( v[0], v[1], v[2], v[3], n ); // Divide tetrahedron n times
   // Indirectly - divide triangles forming tetrahedron n times
   glFlush();
  }

 void myReshape(int w,int h) // Three dimension being displayed in 2 dimension view
  { // Executed when window size is changed
   glViewport( 0, 0, w, h ); // set the viewport
   glMatrixMode( GL_PROJECTION ); // specify which matrix is the current matrix
   glLoadIdentity( ); // replace the current matrix with the identity matrix , 4 x 4 - Why ?

   if( w <= h )   //Get exact aspect ratio , what is aspect ratio ?
     glOrtho( -2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w, 2.0*(GLfloat)h/(GLfloat)w, -10.0, 10.0 );
   else // glOrtho - multiply the current matrix with an orthographic matrix
     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();  // marks the current window as needing to be redisplayed
  }

 int main( int argc, char **argv )
  {
   printf("\n Please enter number of divisions : "); // Try with 0 , 1 , 2 . . .
   scanf("%d",&n);

   glutInit( &argc, argv );  // initialize the GLUT library
   glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH ); // set initial displ
ay mode
   glutInitWindowSize( 500, 500 ); // set initial window size
   glutCreateWindow( "3DGasket" ); //window with a title
   glutReshapeFunc( myReshape );   // sets reshape callback for the current window
   glutDisplayFunc( display );     // register display drawing function
   glEnable( GL_DEPTH_TEST );
   glClearColor( 1.0, 1.0, 1.0, 1.0 );
   glutMainLoop();
  }

/* Output: compile as
   g++ -o SierpinskiGasket2 SierpinskiGasket2.c -lGLU -lGL -lglut

   If no errors, run as
   ./SierpinskiGasket2 */

/* glutReshapeFunc : sets the reshape callback for the current window

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

        func : the new reshape callback function

   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

   If a reshape callback is not registered for a window or NULL is passed to glutReshapeFunc
```

```
    (to deregister a previously registered callback), the default reshape callback is used

   This default callback will simply call glViewport(0,0,width,height) on the normal plane
    (and on the overlay if one exists)

   If an overlay is established for the window, a single reshape callback is generated

   It is the callback's responsibility to update both the normal plane and overlay for the
     window (changing the layer in use as necessary)

   It is up to the GLUT program to manage the size and positions of subwindows within a
     top-level window
   Still, reshape callbacks will be triggered for subwindows when their size is changed using
     glutReshapeWindow */

/* glViewport — set the viewport

   void glViewport(GLint x,  GLint y,  GLsizei width,  GLsizei height);

      x, y : Specify the lower left corner of the viewport rectangle, in pixels
             The initial value is (0,0)

      width, height : Specify the width and height of the viewport, when a GL context is
          first attached to a window, width and height are set to the dimensions of that window

   glViewport specifies the affine transformation of x and y from normalized device
     coordinates to window coordinates, if  xnd  and  ynd are normalized device coordinates

   Then the window coordinates xw and yw are :

   xw = ( xnd + 1 )( width / 2 ) + x
   yw = ( ynd + 1 )( height / 2 ) + y    */

/* glutReshapeWindow : requests a change to the size of the current window

   void glutReshapeWindow(int width, int height);

      width  = New width of window in pixels
      height = New height of window in pixels */

/* glMatrixMode — specify which matrix is the current matrix

   void glMatrixMode(GLenum mode);

      mode : Specifies which matrix stack is the target for subsequent matrix operations
       Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE

      GL_MODELVIEW: Applies subsequent matrix operations to the modelview matrix stack
      GL_PROJECTION: Applies subsequent matrix operations to the projection matrix stack
      GL_TEXTURE      : Applies subsequent matrix operations to the texture matrix stack  */
```

```
/* glLoadIdentity — replace the current matrix with the identity matrix

  void glLoadIdentity( void);

  It is semantically equivalent to calling glLoadMatrix with the identity matrix
     [ 1 0 0 0
       0 1 0 0
       0 0 1 0
       0 0 0 1 ]    */

/* glOrtho — multiply the current matrix with an orthographic matrix

  void glOrtho( GLdouble left,  GLdouble right,  GLdouble bottom,  GLdouble top,
                GLdouble nearVal,  GLdouble farVal);

     left, right : Specify the coordinates for the left and right vertical clipping planes
     bottom, top : Specify the coordinates for the bottom and top horizontal clipping planes
     nearVal, farVal : Specify the distances to the nearer and farther depth clipping planes

  These values are negative if the plane is to be behind the viewer

  glOrtho describes a transformation that produces a parallel projection
  The current matrix is multiplied by this matrix and the result replaces the current matrix
    as if glMultMatrix were called with the following matrix as its argument:
      [ 2 / right - left         0                  0                  tx
             0         2 / top - bottom             0                  ty
             0                   0         -2 / farVal - nearVal       tz
             0                   0                  0                  1  ]
       where
           tx = -  right+left         /   right-left
           ty = -  top+bottom         /   top-bottom
           tz = -  farVal+nearVal /   farVal-nearVal    */

/* glutPostRedisplay - marks the current window as needing to be redisplayed

  void glutPostRedisplay(void);

  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
  glutPostRedisplay may be called within a window's display or overlay display callback to
    re-mark that window for redisplay    */

/* glColor3f - Specify new red, green, and blue values for the current color

  void glColor3f( GLfloat red,    GLfloat green,    GLfloat blue );

  Parameters : red, green, blue  ; and values of 1 for full and 0 for no intensity

  glColor3f( 1, 1, 1); will set color to ?       */
```

```
/* glBegin and glEnd — delimit the vertices of a primitive or a group of like primitives

   void glBegin(GLenum mode);

   mode : Specifies the primitive or primitives that will be created from vertices
          presented between glBegin and the subsequent glEnd

   Ten symbolic constants are accepted: GL_POINTS, GL_LINES, GL_LINE_STRIP,
       GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN,
       GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON

   Only a subset of GL commands can be used between glBegin and glEnd
   Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn

   void glEnd(void);    */

/* glVertex — specify a vertex

   void glVertex3fv(const GLfloat * v);

      v : Specifies a pointer to an array of two, three, or four elements
         The elements of a two-element array are x and y; of a three-element array, x, y, and z;
               and of a four-element array, x, y, z, and w

   glVertex commands are used within glBegin/glEnd pairs to specify point, line, and polygon
      vertices

   When only x and y are specified, z defaults to 0 and w defaults to 1
      When x, y, and z are specified, w defaults to 1    */
```