THE SIERPINSKI GASKET

We use as a sample problem the drawing of the Sierpinski gasket—an interesting shape that has a long history and is of interest in areas such as fractal geometry. The Sierpinski gasket is an object that can be defined recursively and randomly; in the limit, however, it has properties that are not at all random. We start with a two dimensional version, but as we will later, the three-dimensional version is almost identical.

Suppose that we start with three points in space. As long as the points are not collinear, they are the vertices of a unique triangle and also define a unique plane. We assume that this plane is the plane $z = 0$ and that these points, as specified in some convenient coordinate system, are $(x_1,y_1,0)$, $(x_2,y_2,0)$, and $(x_3,y_3,0)$ the construction proceeds as follows:

**Gasket 2D algorithm**:
1. Pick an initial point $(x, y, z)$ at random inside the triangle.
2. Select one of the three vertices at random.
3. Find the location halfway between the initial point and the randomly selected vertex.
4. Display this new point by putting some sort of marker, such as a small circle, at the corresponding location on the display.
5. Replace the point at $(x, y, z)$ with this new point.
6. Return to step 2.

Thus, each time that we generate a new point, we display it on the output device. This process is illustrated in Figure, where $p_0$ is the initial location, and $p1$ and $p2$ are the first two locations generated by our algorithm.
Sierpinski Gasket
void display()
{
    GLfloat vertices[3][3] = {
        {0.0, 0.0, 0.0}, {25.0, 50.0, 0.0},
        {50.0, 0.0, 0.0}};
    /* an arbitrary triangle in the plane z=0 */
    GLfloat p[3] = {7.5, 5.0, 0.0};
    /* or set to any desired initial point inside the triangle */
    int j, k;
    int rand(); /* standard random-number generator */

    glBegin(GL_POINTS);
    for (k = 0; k < 5000; k++)
    {
        /* pick a random vertex from 0,1,2 */
        j = rand() % 3;

        /* compute new location */
        p[0] = (p[0] + vertices[j][0]) / 2;
        p[1] = (p[1] + vertices[j][1]) / 2;

        /* display new point */
        glVertex3fv(p);
    }
    glEnd();
    glFlush();
}

PRIMITIVES AND ATTRIBUTES

OpenGL supports two classes of primitives: geometric primitives and image, or raster primitives. Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves, and surfaces. Because geometric primitives exist in a two- or three-dimensional space, they can be manipulated by operations such as rotation and translation. In addition, they can be used as building blocks for other geometric objects using these same operations. Raster primitives, such as arrays of pixels, lack geometric properties and cannot be manipulated in space in the same way as geometric primitives.
The primitives and their type specifications include the following:

Points (GL_POINTS) Each vertex is displayed at a size of at least one pixel.

Line segments (GL_LINES) The line-segment type causes successive pairs of vertices to be interpreted as the endpoints of individual segments. Note that successive segments usually are disconnected because the vertices are processed on a pairwise basis.

Polylines (GL_LINE_STRIP, GL_LINE_LOOP) If successive vertices (and line segments) are to be connected, we can use the line strip, or polyline form. Many curves can be approximated via a suitable polyline. If we wish the polyline to be closed, we can locate the final vertex in the same place as the first, or we can use the GL_LINE_LOOP type, which will draw a line segment from the final vertex to the first, thus creating a closed path.

Polygon Basics:

Line segments and polylines can model the edges of objects, but closed objects also may have interiors. Usually we reserve the name polygon for an object that has a border that can be described by a line loop but also has a well-defined interior. Polygons play a special role in computer graphics because we can display them rapidly and use them to approximate arbitrary surfaces. The performance of graphics systems is characterized by the number of polygons per second that can be rendered. We can render a polygon in a variety of ways: We can render only its edges; we can render its interior with a solid color or a pattern; and we can render or not render the edges. Although the outer edges of a polygon are defined easily by an ordered list of vertices, if the interior is not well defined, then the list of vertices may not be rendered at
all or rendered in an undesirable manner. Three properties will ensure that a polygon will be displayed correctly: It must be simple, convex, and flat.

**Polygon Types in OpenGL**

**Polygons** (GL_POLYGON) The edges are the same as they would be if we used line loops. Successive vertices define line segments, and a line segment connects the final vertex to the first. The interior is filled according to the state of the relevant attributes.

**Triangles and Quadrilaterals** (GL_TRIANGLES, GL_QUADS) These objects are special cases of polygons. Successive groups of three and four vertices are interpreted as triangles and quadrilaterals, respectively. Using these types may lead to a rendering more efficient than that obtained with polygons.

**Strips and Fans** (GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN) These objects are based on groups of triangles or quadrilaterals that share vertices and edges. In the triangle strip, for example, each additional vertex is combined with the previous two vertices to define a new triangle (Figure 2.14). For the quad_strip, we combine two new vertices with the previous two vertices to define a new quadrilateral. A triangle fan is based on one fixed point. The next two points determine the first triangle, and subsequent triangles are formed from one new point, the previous point, and the first (fixed) point.
POLYGONS AND RECURSION

The output from our gasket program above shows considerable structure. If we were to run the program with more iterations, then much of the randomness in the image would disappear. Examining this structure, we see that regardless of how many points we generate, there are no points in the middle. If we draw line segments connecting the midpoints of the sides of the original triangle, then we divide the original triangle into four triangles, the middle one containing no points (see Figure below).
Looking at the other three triangles, we see that we can apply the same observation to each of them; that is, we can subdivide each of these triangles into four triangles by connecting the midpoints of the sides, and each middle triangle will contain no points. This structure suggests a second method for generating the Sierpinski gasket—one that uses polygons instead of points and does not require the use of a random number generator. One advantage of using polygons is that we can fill solid areas on our display. Our strategy is to start with a single triangle, to subdivide it into four smaller triangles by bisecting the sides, and then to remove the middle triangle from further consideration. We repeat this procedure on the remaining triangles until the size of the triangles that we are removing is small enough—about the size of one pixel—that we can draw the remaining triangles.

We can implement the process that we just described through a recursive program. We start its development with a simple function that draws a single triangular polygon given three arbitrary vertices:

```c
void triangle(GLfloat *a, GLfloat *b, GLfloat *c)
{
    glVertex2fv(a);
    glVertex2fv(b);
    glVertex2fv(c);
}
```

Suppose that the vertices of our original triangle are given by the following array:

```c
GLfloat v[3][2];
```

Then the midpoints of the sides are given by the array m [3] [3], which can be computed using the following code:
for (j=0; j<2; j++) m[0][j] = (v[0][j]+v[1][j])/2.0;
for (j=0; j<2; j++) m[1][j] = (v[0][j]+v[2][j])/2.0;
for (j=0; j<2; j++) m[2][j] = (v[1][j]+v[2][j])/2.0;

With these six locations, we can use triangle to draw the three triangles formed by \((v[0], m[0], m[1]); (v[2], m[1], m[2]); \) and \((v[1], m[2], m[0])\). We do not simply want to draw these triangles; we want to subdivide them. Hence, we make the process recursive. We define a recursive function `divide_triangle(float *a, float *b, float *c, int k)` that will draw the triangles only if \(k\) is zero. Otherwise, it will subdivide the triangle specified by \(a\), \(b\), and \(c\) and decrease \(k\).

```c
void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *c, int k) {
    GLfloat ab[2], ac[2], bc[2];
    int j;
    if(k>0) {
        /* compute midpoints of sides */
        for (j=0; j<2; j++) ab[j]=(a[j]+b[j])/2;
        for (j=0; j<2; j++) ac[j]=(a[j]+c[j])/2;
        for (j=0; j<2; j++) bc[j]=(b[j]+c[j])/2;

        /* subdivide all but inner triangle */
        divide_triangle(a, ab, ac, k-1);
        divide_triangle(c, ac, bc, k-1);
        divide_triangle(b, bc, ab, k-1);
    }
    else triangle(a,b,c); /* draw triangle at end of recursion */
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        divide_triangle(v[0], v[1], v[2], n);
    glEnd();
    glFlush();
}
```