Chapter 8

Three-Dimensional Object Representations

This chapter examines the methods that have been devised to describe and visualize various kinds of objects and data sets. In addition, programming projects explore the application of OpenGL functions for different representations, as well as the implementation details involved in the algorithms for generating displays of splines, fractals, and other objects.

Exercises

8-1. Input for this algorithm is the sphere radius, assuming that the sphere is centered at the coordinate origin. A polygon-mesh representation can be constructed using longitude and latitude lines to obtain vertex positions. Thus, two additional input parameters could be provided to specify the number of latitude and longitude lines that are to be used in subdividing the sphere surface.

The parametric equations for either a longitude line or a latitude line can be obtained from Eqs. 8-2. Along each longitude line, angle θ has a constant value. Similarly, angle ϕ is constant along any latitude line. For example, the coordinates along the longitude line in the xz plane ($\theta = 0$) are calculated with the equations

$$x = r\cos\phi, \qquad y = 0, \qquad z = r\sin\phi$$

and the equations for computing coordinate positions along the latitude line in the xy plane ($\phi = 0$) are

$$x = r\cos\theta, \qquad y = r\sin\theta, \qquad z = 0$$

If we denote the required number of latitude lines as n_{lat} and the required number of longitude lines as n_{long}, the algorithm can obtain vertex coordinates by calculating n_{long} positions along n_{lat} latitude lines (or vice versa). Thus, angle θ is assigned n_{long} values over the range from $-\pi$ to $+\pi$, and angle ϕ is assigned n_{lat} values over the range from $-\pi/2$ to $+\pi/2$.

The algorithm should take sphere symmetries into account to reduce the number of trigonometric calculations. Coordinate positions need to be computed from the parametric sphere equations only for the latitude lines in the northern (or southern) hemisphere. Positions in the other hemisphere are obtained using a reflection about the xy plane. Also, the algorithm need only calculate positions within one octant of each circular latitude (or longitude) line. The remaining positions in the other octants for each circle are obtained by symmetry (Fig. 3-18). Furthermore, the latitude "lines" at the top and bottom of the sphere (the poles) are points.

At the top of the sphere, polygon edges are defined from the pole position radially down to the coordinate positions on the first latitude line below the pole, and a polygon edge is defined between each pair of adjacent positions along that latitude line. A similar set of polygons is constructed at the bottom of the sphere. Thus, the top and bottom of the spherical surface are each represented with a triangular mesh.

For all other latitude lines, adjacent pairs of coordinate positions are used to define a horizontal edge of a quadrilateral, and adjacent positions along a longitude line joining two latitude lines are used to define a vertical edge of a quadrilateral. To facilitate processing and to ensure that the vertex positions for each surface facet are coplanar, additional calculations can be performed to split each quadrilateral into two triangles. The final step in the algorithm is to store the information for vertex coordinates, polygon edges, and surface facets in polygon tables (Section 3-15).

8-2. A polygon-mesh representation for an ellipsoid can be constructed using Eqs. 8-4 and the sphere algorithm from the preceding exercise. The primary differences between the two algorithms are that the parametric equations involve a radius value for each axis and input values are needed for r_x, r_y, and r_z, with the ellipsoid centered at the coordinate origin. Also, coordinate positions along an ellipse are symmetric between quadrants, but not between octants.

8-3. Assuming that the cylinder parameters are specified relative to the coordinate origin, input for this algorithm can be the cylinder radius r and the height h above the xy plane. Thus, the cylinder axis is along the z axis, the center position for the bottom of the cylinder is (0, 0, 0), and the center position for the top of the cylinder is (0, 0, h). If some other position is to be given for the cylinder, calculations for the polygon-mesh vertex coordinates can be carried out relative to the given cylinder reference position. Or, the cylinder can be translated and rotated to place its base in the xy plane and its axis along the z axis.

First, compute a set of equally spaced points around the perimeter of the cylinder top. This can be accomplished using the midpoint circle algorithm (Section 3-9) or the parametric circle equations 3-28. In either case, we need only calculate coordinates for the points in the first quadrant, and the remaining points are determined by symmetry (Fig. 3-18). These points provide the vertices for a convex polygon representation of the top of the cylinder, and form the basis for obtaining the polygon representation for the cylinder base. For every position (x, y, h) along the top of the cylinder, there is a corresponding point (x, y, 0) at the base of the cylinder, which are the remaining vertex positions

needed for the polygon at the base and for the rectangles along the side of the cylinder. The final step in the algorithm is to store the information for vertex coordinates, polygon edges, and surface facets in polygon tables (Section 3-15).

The number of circle subdivisions can be an additional input parameter. Or a fixed number of subdivisions, such as 8 or 10, can be used for all input cylinders.

To facilitate processing and to ensure that the vertex positions for each surface facet are coplanar, additional calculations can be performed to convert the polygon representation into a triangular mesh. This is accomplished for the top and bottom of the cylinder by adding radial lines from the circle center to the vertex positions on the circle perimeter. For each rectangular strip along the side of the cylinder, add a diagonal line. Also, the size of the polygons along the side of the cylinder can be reduced by first dividing each rectangle into a number of smaller rectangles, then splitting the reduced rectangles into triangles. Thus, another option for this algorithm is the specification of the number of horizontal slices to be used in creating the rectangles along the side of the cylinder.

8-4. A polygon-mesh representation for a superellipsoid can be constructed using Eqs. 8-12 and the ellipsoid algorithm (Exercise 8-2), along with the additional parameters s_1 and s_2.

8-5. An algorithm for generating a polygon-mesh representation for the surface of a meta-ball object can be set up using techniques similar to those for the sphere and ellipsoid algorithms (Exercises 8-1 and 8-2). Given values for parameters b, d, and the surface constant for $f(r)$, convert the object description to a parametric form involving surface parameters u and v. Then, subdivide the surface into a number of subintervals in the two orthogonal directions to obtain a set of polygon facets.

8-6. Input for this routine can be a data file that includes a value for the tension parameter t, as well as a set of four control-point positions. An additional input can be either the number of coordinate positions to plot along the curve or a step size Δu for the parametric parameter u.

The parametric representation 8-35 can be rewritten in standard polynomial form for each of the x and y coordinates, and positions along the curve between the middle two control points are calculated for parameter values $u_{j+1} = u_j + \Delta u$, where $u_0 = 0$, $u_n = 1$, and $j = 0, 1, 2, \ldots, n$. These coordinate positions could be connected with straight line segments, or forward differences can be used to determine points along the cardinal-spline path.

To display the curve section, the example program in Section 8-10 can be modified so that the cardinal-spline routine replaces procedure `bezier`. As an option, a closed curve as in Fig. 8-29 can be displayed using a cyclic permutation of the four control points. Another option is to specify more that four control points so that multiple connected cardinal-spline sections can be generated between all control points except the first point and the last point.

8-7. The parametric equations for a Kochanek-Bartels spline can be derived from boundary conditions 8-36 using the same procedures discussed for cardinal splines. And the curve

can be displayed using the cardinal-spline procedures in Exercise 8-6. For this program, an input file can include values for the tension, bias, and continuity parameters, in addition to the control-point positions and a value for the parametric step size Δu.

8-8. As noted in Section 8-10, a Bézier curve generated with three distinct, noncollinear control points is a parabola. The three blending functions are

$$BEZ_{0,2} = (1-u)^2, \qquad BEZ_{1,2} = 2u(1-u), \qquad BEZ_{2,2} = u^2$$

Function $BEZ_{0,2}$ has a minimum value of 0 at $u = 1.0$ and a maximum value of 1.0 at $u = 0$. Function $BEZ_{1,2}$ has its minimum value of 0 at both $u = 0$ and $u = 1.0$, and $BEZ_{1,2}$ has a maximum value of 0.5 at $u = 0.5$. Function $BEZ_{2,2}$ has a minimum value of 0 at $u = 0$ and a maximum value of 1.0 at $u = 1.0$.

8-9. The five Bézier blending functions for $n = 4$ are

$$\begin{array}{lcll} BEZ_{0,4} & = & (1-u)^4, & \text{min} = 0 \text{ at } u = 1; \quad \text{max} = 1 \text{ at } u = 0 \\ BEZ_{1,4} & = & 4u(1-u)^3, & \text{min} = 0 \text{ at } u = 0, 1; \quad \text{max} = 27/64 \text{ at } u = 1/4 \\ BEZ_{2,4} & = & 6u^2(1-u)^2, & \text{min} = 0 \text{ at } u = 0, 1; \quad \text{max} = 3/8 \text{ at } u = 1/2 \\ BEZ_{3,4} & = & 4u^3(1-u), & \text{min} = 0 \text{ at } u = 0, 1; \quad \text{max} = 27/64 \text{ at } u = 3/4 \\ BEZ_{4,4} & = & u^4, & \text{min} = 0 \text{ at } u = 0; \quad \text{max} = 1 \text{ at } u = 1 \end{array}$$

8-10. For this exercise, replace the set of four control-point coordinates given in procedure `displayFcn` with input from a data file. Other input values could also be included in the data file, such as the curve width, the curve color, and details for displaying the control points and control graph. For example, the control points could be displayed in a specified size and color, with connecting dashed lines.

8-11. For this exercise, replace the fixed values for variable `nCtrlPts` and the control-point coordinates given in procedure `displayFcn` with input from a data file that contains the control-point positions and the number of positions. Various other data can be included in the input, such as the number of points to be plotted along the curve path, the width and color of the curve, and details for displaying the control points and control graph. For example, the control points could be displayed in a specified size and color, with connecting dashed lines. Other options include displaying the curve as set of straight-line segments, or a set of points obtained with forward-difference calculations.

8-12. This programming exercise is a variation of the program for Exercise 8-10. Modify the programming example in Section 8-10 so that the OpenGL routines from the example in Section 8-18 are used to generate and display the Bézier curve, instead of the explicit calculations for the blending functions and curve coordinates.

8-13. Expand the program from the preceding exercise so that the control points are specified in three-dimensional Cartesian coordinates and a three-dimensional Bézier curve is generated. Viewing routines for displaying the curve can be taken from previous programming

examples. Various projection planes could be used to view the curve from different positions. For example, a curve defined in the xz plane projects to a straight line if the projection plane is parallel to the xy plane. Also, the four input control points might not be coplanar.

8-14. Input for each curve section can be limited to three or four control points. Any number of connected curve sections could be displayed, or the routine could be limited to the display of just two connecting curve sections. An input file can be used to specify the coordinates for each set of control points, the number of control points in each set, and the number of curve sections to be displayed.

For the second curve section, the positions for the first two control points could be calculated from the continuity conditions, or all control-point positions could be specified as input. If all control-point positions are included in the input file, tests should be performed to be sure that the first-order continuity conditions are satisfied. Thus, the coordinates for the first control point of the second curve section should be the same as the coordinates for the last control point of the first curve section. And the coordinates for the second control point should satisfy Eq. 8-47. Either the program from Exercise 8-10 or the program from Exercise 8-12 could be modified to display the two curve sections.

8-15. This is an extension of the preceding exercise, with an added calculation for the coordinates of $\mathbf{p}_{2'}$. For this routine, the third control point of the second curve section is positioned so that the first and second derivatives of the first and second curve sections match at the boundary. Thus, the position for the first control point of the second curve section is the same as the position for the last control point of the first curve section, the position for the second control point of the second curve section is calculated from Eq. 8-47, and the position for the third control point of the second curve section is calculated from Eqs. 8-45 as

$$\mathbf{p}_{2'} = \mathbf{p}_n + \frac{2n}{n'}(\mathbf{p}_n - \mathbf{p}_{n-1}) + \frac{n(n-1)}{n'(n'-1)}\left[\mathbf{p}_{n-2} - 2\mathbf{p}_{n-1} + \mathbf{p}_n\right]$$

8-16. The subdivisions methods given in Section 8-17 are to be used to calculate curve positions in the program for Exercise 8-10.

8-17. The forward-difference methods given in Section 8-17 are to be used to calculate curve positions in the program for Exercise 8-10.

8-18. For a given value of n, a uniform knot vector with integer values from 0 to $n+d+1 = n+6$ can be constructed. Then the knot vector values are used in relations 8-53 to obtain the first blending function. The remaining blending functions can then be obtained from Eqs. 8-55. Also, the blending functions $B_{k,3}(u)$ from Example 8-1 in Section 8-12 can be used in the determination of blending functions $B_{k,5}(u)$.

As an example, if $n = d = 5$, the knot vector is

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

And blending function $B_{0,5}(u)$ is evaluated from blending functions $B_{0,3}(u)$ in Example 8-1.

8-19. For a given value of n, a uniform knot vector with integer values from 0 to $n+d+1 = n+7$ can be constructed. Then the knot vector values are used in relations 8-53 to obtain the first blending function. The remaining blending functions can then be obtained from Eqs. 8-55. Also, the blending functions $B_{k,3}(u)$ from Example 8-1 in Section 8-12 can be used in the determination of blending functions $B_{k,6}(u)$.

As an example, if $n = d = 6$, the knot vector is

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

And blending function $B_{0,6}(u)$ is evaluated from blending functions $B_{0,3}(u)$ in Example 8-1.

8-20. In general, the set of blending functions could be determined for specified values of the knot vector and parameters d and n. These blending functions would then replace the Bézier blending functions in the programming example of Section 8-10. A simpler assignment for this problem is to implement the blending functions in Exercise 8-1 of Section 8-12 to display the quadratic B-spline curve. And the forward-difference methods from Section 8-17 are used to calculate coordinate positions along the B-spline curve, using the procedures discussed for Exercise 8-17.

8-21. As in Exercise 8-12, programming examples from Section 8-10 and 8-18 can be combined to display the curve. In this case, the GLU routines are to be used to generate a B-spline.

8-22. For this exercise, input can be specified in a data file that contains the coordinates for three control points and the values for the weight factors ω_k. These input values are then used in Eq. 8-69, along with the Bézier blending functions, to generate a conic section, using the methods given in Section 8-15.

8-23. The routine from the preceding exercise is modified to implement Eq. 8-71, using the methods described in the Section 8-15 example.

8-24. Coordinates for a position on a Bézier surface are calculated from Eq. 8-51, given a set of control points and the values for the blending functions. At any point $\mathbf{P}(u, v)$ on the surface, we can calculate two surface vectors that are tangent to the surface along the directions of the surface parameters:

$$\mathbf{r}_u = \frac{\partial \mathbf{P}}{\partial u}, \qquad \mathbf{r}_v = \frac{\partial \mathbf{P}}{\partial v}$$

The unit normal vector at point $\mathbf{P}$ is then calculated as

$$\mathbf{n} = \frac{\mathbf{r}_u \times \mathbf{r}_v}{|\mathbf{r}_u \times \mathbf{r}_v|}$$

8-25. For quadratic equations of the form

$$x(u) = a_0 + a_1 u + a_2 u^2$$

with $0 \leq u \leq 1$ and increments $u_{k+1} = u_k + \delta$, the forward-difference calculations are

$$\begin{aligned} x_{k+1} &= x_k + \Delta x_k \\ \Delta x_{k+1} &= \Delta x_k + \Delta^2 x_k \end{aligned}$$

with initial values

$$\begin{aligned} x_0 &= a_0 \\ \Delta x_0 &= a_1\delta + a_2\delta^2 \\ \Delta^2 x_0 &= 2a_2\delta^2 \end{aligned}$$

and the second forward difference $\Delta^2 x_k$ is a constant for all values of k. The above equations are then applied repeatedly to obtain the (x, y) coordinate positions along the curve from $u = 0$ to $u = 1$. This algorithm can be implemented to display a quadratic curve, specified by input parameters.

8-26. Forward-difference calculations for a cubic curve are given in Section 8-17. These calculations are then applied repeatedly to obtain the (x, y) coordinate positions along the curve from $u = 0$ to $u = 1$. As in the preceding exercise, this algorithm could be implemented to display a cubic curve, specified by input parameters.

8-27. An input file for this problem can include the components of the translation vector, specifying the sweep direction relative to the coordinate origin, a translation distance, and a shape that is defined in the xy plane. The translation distance (Section 8-19) can be assumed to specify the third dimension for the object relative to its definition in the xy plane. (Otherwise, starting and ending positions along the sweep direction can be specified.) For example, a translation distance in the $+z$ direction for a circle specifies the height of a circular cylinder above the xy plane.

The surface for the three-dimensional object is to be defined with a polygon mesh. This is accomplished by dividing both the perimeter of the two-dimensional object and the translation distance into a number of intervals, which are connected to form the boundaries of the surface facets, as illustrated in Fig. 8-55. The size of the intervals (or the number of intervals) in each direction can be specified in the input file. A complete description of the surface is then given in a set of polygon tables that specify the vertex and edge information for each surface facet.

8-28. Methods for this exercise are similar to those in the preceding exercise. The difference is that rotational sweep parameters are specified in the input file instead of translational sweep parameters. A rotation axis can be specified in various ways. It could be defined with an axis vector or with two three-dimensional coordinate positions (Section 5-11). Or the rotation axis could be assumed to be in the xy plane at a specified distance from the center of the object, as in Fig. 8-56. Also, a rotation angle can be given in the input

file, which specifies the amount of rotation about the axis from the defined position of the object in the xy plane. Or, a 360° sweep could be assumed.

The perimeter of the two-dimensional object is divided in subintervals, which are rotated incrementally through the sweep angle to form a set of points on the three-dimensional surface. Surface facet information is then stored in tables as in Exercise 8-27.

8-29. The ray-casting method discussed in Section 8-20 can be used to generate the CSG description of an object formed from two overlapping three-dimensional shapes using a specified Boolean operation. For this algorithm, two simple shapes, such as a cube and a sphere or a cube and a tetrahedron, can be defined relative to the coordinate origin. For instance, each object can be defined with its center at the coordinate origin. The parameters for the two objects, such as the sphere radius and the cube vertex coordinates, could be specified in an input file, along with the new center position and orientation for each object, the surface color for each object, and the type of Boolean operation to be applied.

Geometric transformations are applied to place the two objects in some region of space, given the input positioning and orientation information. Then ray intersection calculations are performed, as indicated in Figs. 8-60 and 8-61, for each pixel within the firing-plane projection of the coordinate extents of the scene.

To compute the intersection position for a ray and a sphere, substitute the equation for the line along the path of the orthogonal ray into the sphere equation and solve for the distance from the firing plane. For example, if a pixel is at position $\mathbf{P}_{pix}$ on the firing plane and the center of a sphere with radius r is at position $\mathbf{P}_c$, then the intersection position is calculated from the sphere equation as

$$|(\mathbf{P}_{pix} + s\,\mathbf{u}) - \mathbf{P}_c|^2 - r^2 = 0$$

where s is the distance along the ray from the firing plane and $\mathbf{u}$ is the unit direction vector for the ray. If the firing plane is the xy plane, as in Fig. 8-61, then all rays are parallel to the z axis and $\mathbf{u} = \mathbf{u}_z$. If the discriminant in the solution for s is negative, the ray does not intersect the sphere. Otherwise, the intersection point nearest the firing plane is the visible surface intersection position.

Methods for computing the intersection position for a ray and a polygon surface of a polyhedron are similar to those for computing the intersection of a polygon edge with a clipping plane (Exercise 3-17). First determine whether the ray intersects the bounding sphere for the polyhedron. If it does, then test the surface normal vector for each polygon to identify those polygons that are visible to the firing plane. The normal vector $\mathbf{N}$ for a visible polygon satisfies the inequality

$$\mathbf{N}\cdot\mathbf{u} < 0$$

For each visible polygon, substitute the equation for the pixel ray into the plane equation to obtain

$$\mathbf{N}\cdot(\mathbf{P}_{pix} + s\,\mathbf{u}) + D = 0$$

Solving for s, we obtain the intersection position on the plane of the polygon. Next perform an inside-outside test to determine if this intersection point is inside the polygon boundaries. If the point is outside, the ray does not intersect that surface facet.

After the intersection calculations are completed for a pixel, sort the intersection positions from left to right along the ray path. Then generate the CSG description according to the kind of Boolean operation to be performed.

8-30. An octree representation (section 8-21) can be constructed for each object, starting with the coordinate extents of the object or with a bounding cube. Repeatedly subdivide the octant box into octants to obtain the voxel regions that are completely filled with the object. If the octant is initially a cube, subdivide to obtain those suboctants that are within the coordinate extents of the object. Next, the vertex positions for the octants are compared to the surface boundaries of the object. An octant that is not completely inside or outside the object surfaces is divided into suboctants. This testing and subdivision process continues until each suboctant is either completely inside the object surfaces or completely outside. An octant is completely inside a sphere with radius r, for example, if the distance squared from each octant vertex to the circle center is less than or equal to r^2. For a polyhedron surface facet, inside-outside tests can be performed for each octant vertex. An octant is completely inside a polyhedron if its vertices are inside all surface facets, and an octant is completely outside of a convex polyhedron if its vertices are outside any one surface facet. The complete tree representation is then constructed with a series of nodes and pointers down to the homogeneous leaf nodes.

Once object descriptions are stored as octree representations, the CSG tree is constructed by testing the nodes in the two octrees to be combined at each step. This involves comparing the same octants in the two octree representations to determine which surface points should be projected to the firing plane. The octree nodes are searched from the front of the octree to the back to identify the visible surfaces.

8-31. First, locate the four vertex positions for the coordinate extents of the two-dimensional scene. These four positions can define the initial boundaries of the quadtree representation (Section 8-21), or a square quadtree region can be constructed around the bounding rectangle for the scene. Next, subdivide the quadtree region into four quadrants and compare the vertex positions for each quadrant to the boundaries of the objects in the scene. For a scene composed entirely of polygons, inside-outside polygon tests can be used to determine the relative positions of the quadrant vertices. If all vertices of a quadrant are not inside a single object, subdivide that quadrant and test the subdivisions. Continue subdividing until all subquadrants are homogeneous. The complete quadtree representation is then constructed with a series of nodes and pointers down to the homogeneous leaf nodes, as illustrated in Fig. 8-65.

Instead of using the coordinate extents for the scene, the display-window limits could form the quadtree boundaries. Also, a clipping window could be defined for the scene, so that a quadtree representation is constructed within the clipping window.

8-32. For a specified display window, determine the region of the display that is occupied by the quadtree representation, as constructed in the preceding exercise. All pixel positions in the display window that are outside the quadtree representation are set to the window background color. Starting at the top of the tree, check the contents of data fields for each quadrant. The color for a homogeneous quadrant is assigned to the pixels along the scan lines within that quadrant. For heterogeneous quadrants, follow the pointers in the tree representation to the homogeneous regions.

As an option, a clipping region could be defined for the scene. The quadtree representation is then constructed around the cliped portion of the scene.

8-33. The coordinate extents of the three-dimensional object can be used to define the limits of the octree spatial region (Section 8-21), or a cube can be constructed around the bounding parallelepiped for the object. Divide the octree box into octants and test the position of the octant vertices. An octant is completely inside the object if all eight vertices are inside all surfaces of the polyhedron. And an octant is completely outside the object if all octant vertices are outside any one surface facet, assuming that the object is a convex polyhedron. Other octants are subdivided until each suboctant is homogeneous. The complete octree representation is then constructed with a series of nodes and pointers down to the homogeneous leaf nodes.

8-34. An outline similar to Fig. 8-90(b) can be generated from a horizontal line using Eq. 8-109. Input parameters could be used to specify the length and position of the line, as well as the number of line subdivisions. The value of the random parameter r can be selected from a Gaussian distribution or some other specified distribution.

8-35. An array of random elevations z_{ij} above a ground plane can be generated using the calculations discussed for Figs. 8-91 and 8-92.

8-36. A midpoint-displacement method can be used to construct the Koch snowflake. Starting with the unit equilateral triangle, displace the midpoint of each edge by a distance of $\frac{1}{2\sqrt{3}}$ away from the triangle center along a line that is perpendicular to that edge. (The perpendicular line has a slope that is equal to the negative of the edge slope.) This displaced midpoint position is then joined with straight-line segments to the two positions along that edge that are at a distance of 1/3 from the edge endpoints. The middle third of each edge is then deleted to produce the generator shape shown in Fig. 8-70.

The displaced midpoint positions of the three edges and the three triangle vertices form a set of vertices for a regular hexagon. If one edge of the unit equilateral triangle is along the x axis with its left endpoint at the coordinate origin of the xy plane, the hexagon vertex set is

$$\{\ (0,\ 0),\ \ (0,\ \frac{1}{\sqrt{3}}),\ \ (\frac{1}{2},\ \frac{\sqrt{3}}{2}),\ \ (1,\ \frac{1}{\sqrt{3}}),\ \ (1,\ 1),\ \ (\frac{1}{2},\ \frac{-1}{2\sqrt{3}})\ \}$$

One way to generate the coordinate position for the displaced midpoint of an edge is to place the unit edge on the x axis with its left endpoint at the coordinate origin. Then the

perpendicular to the edge is a vertical line, and the coordinates for the displaced midpoint are $(\frac{1}{2}, \frac{-1}{2\sqrt{3}})$. Thus an iterative procedure can be developed that successively transforms each edge of the equilateral triangle to the x axis, then generates the coordinates for the displaced midpoint of the edge. This procedure is repeated for each new edge for the specified number of iterations, and the final shape is displayed as a set of straight-line segments.

Another method for constructing the Koch curve is to apply a sequence of translate-rotate transformations to the generator shape shown in Fig. 8-70. The generator can be constructed from a unit line on the x axis by displacing its midpoint position and forming a list of the four line-segment endpoints for the polyline. Then, each edge of the equilateral triangle is replaced with this generator pattern. This can be accomplished by translating the left end of the generator to the left endpoint position of each triangle edge, and then rotating the generator onto the line segment. At each iteration, the generator is scaled by a factor of $\frac{1}{3}$, and the process is repeated for each new edge.

Alternatively, the translate-rotate operations could be applied to the triangle as a whole. Scale the unit triangle by a factor of $\frac{1}{3}$, then place the scaled triangle onto the center third of each edge of the unit triangle and delete the bottom edge of the scaled triangle. At each iteration, scale the reduced triangle from the previous step and transform it to each of the new edges. This procedure is repeated for each new edge for the specified number of iterations, and final shape is displayed as a set of straight-line segments.

8-37. One of the generators from Fig. 8-73 or Fig. 8-74 can be assigned for this problem, or the choice of pattern can be optional. Or a different pattern could be used. The definition for the pattern can include a list of segment endpoints and the fractal dimension, calculated from Eq. 8-107 or Eq. 8-108.

As in the preceding exercise, the geometric pattern for the generator can be defined so that its endpoints are on the x axis. Then the generator is scaled to the size of the segment length and mapped to each straight line segment in the pattern, using a sequence of translation and rotation transformations. Repeat this process for the specified number of iterations and display the final shape as a set of straight-line segments.

8-38. Procedure `selfSqTransf` in the self-squaring programming example of Section 8-23 can be modified to display this fractal curve, using input values for the real and imaginary parts of the complex constant λ. The curve points are plotted for the inverse function

$$z = f^{-1}(z') = \pm\sqrt{z' - \lambda}$$

The real and imaginary parts for z can be obtained from Eqs. 8-114 as

$$x = Re(z) = \pm\sqrt{\frac{|discr| + Re(discr)}{2}}$$

$$y = Im(z) = \pm\sqrt{\frac{|discr| - Re(discr)}{2}}$$

where

$$discr = z' - \lambda \;=\; (x' - \lambda_x) + i(y' - \lambda_y)$$

8-39. As in the preceding exercise, procedure `selfSqTransf` in the self-squaring programming example of Section 8-23 can be modified to display this fractal curve. The curve points are plotted for the inverse function

$$z = f^{-1}(z') = \pm i\sqrt{1 + i\,z'}$$

The real and imaginary parts for z can be obtained from Eqs. 8-114 as

$$x = Re(z) = \pm\sqrt{\frac{|discr| - Re(discr)}{2}}$$

$$y = Im(z) = \pm\sqrt{\frac{|discr| + Re(discr)}{2}}$$

where

$$discr = 1 + i\,z' \;=\; (1 - y') + i\,x'$$

8-40. For this modification of the Mandelbrot programming example in Section 8-23, a set of colors could be assigned. Or the number of color levels could be assigned without specifying the color for each interval of the iteration count.

8-41. The program in the preceding exercise can be modified to include a procedure to input a data file containing values for the number of color levels and the color for each interval of the iteration count. Also, procedure `mandelbrot` must be revised so that a variable number of color levels (as specified in the input file) can be displayed.

8-42. Interactive keyboard input could be used to specify the coordinate limits for the rectangular zoom region in the complex plane. The selected rectangle could be displayed as in Fig. 8-102. Then the rectangular region is mapped to the size of the display window to produce an enlarged view of the region details. Also, other color-coding schemes could be used in the display of this rectangular section.

8-43. An input data file, containing specifications for the reference circle and the object to be inverted, can be provided for this routine. A reference circle is defined by its center coordinates and radius, and the object to be inverted can be a set of point positions, a circle, a triangle, or some other geometric object. The reference circle, input object, and inverted object can be displayed in different colors or with different attributes.

8-44. Any set of substitution rules similar to those shown in Fig. 8-107 could be used for this problem, and an algorithm or description can be given for implementing the transformation of the triangle.

8-45. Implement the algorithm from the previous exercise and show the various transformed shapes within a single display window using different viewports to obtain the simultaneous views of the altered triangle (see Fig. 6-10). Each transformed object can be displayed in a different color.

8-46. A cross section of a solid sphere in the xy plane can exploded by breaking the circular shape into small triangular "particles", and moving the particles radially outward from the circle center. At intervals, each section can be further subdivided to produce smaller sections, with sections disappearing when they become too small.

Also, random variables can be used to control the particle motions and disintegration times, and particles could change colors over time. The motion can be stopped at a set time, or particles can continue to move until they travel beyond the limits of the display window.

Alternatively, the sphere could be exploded in three-dimensions, by specifying its surface as a triangular mesh and expanding the triangles radially outward and disintegrating the triangles over time. Or, the sphere surface could be broken into a number of small spheres as it explodes. An orthogonal projection can be used to display the particle movement.

8-47. The cylinder can be defined with its base centered at the coordinate origin in the xy plane, so that its axis is along the coordinate z axis, and it can be exploded using methods similar to those in the preceding exercise. Surface sections along the side of the cylinder can move radially outward from the cylinder axis, and surface sections at the top and bottom can move along vertical paths outward from the cylinder center.

8-48. For this exercise, the size of the rectangular cloth and the number of subdivisions for the spring network could be specified as input. The area of the rectangle is then divided into the specified number of subsections, with each of the four sides of a rectangular subsection as a spring, similar to the square spring network shown in Fig. 8-118. For example, if a cloth section is divided into a 2 by 3 (or 3 by 2) grid of springs, the edges of the cloth are represented with ten springs and the interior with seven springs. Other schemes are possible, such as including a spring along a diagonal of each rectangular subsection.

8-49. A set of n color values could be specified as input for this routine, along with a set of data values and their associated coordinate positions. Search the data set to determine the minimum and maximum input values. Then divide the data range into n intervals, with one color assigned to each interval. Each input value can then be displayed as a color-coded point.

If the input values are specified at some coordinate interval from each other, values could be displayed as color-coded geometric figures. For instance, each value could be displayed as a small square or circle.

8-50. For this exercise, a rectangular grid of data values can be specified at fixed intervals Δx and Δy in the x and y directions, so that the positions for the four corners of a grid cell are specified as

$$(x_k, y_k), \quad (x_k, y_k + \Delta y), \quad (x_k + \Delta x, y_k + \Delta y), \quad (x_k + \Delta x, y_k)$$

for $k = 1, 2, \ldots, n$ and with data values specified row-by-row from the top of the grid to the bottom. The input can also include either a value n_c for the number of contour lines to be plotted or a set of contour-line data values.

If n_c is given as input, search the data set to determine the minimum and maximum input values. Then select n_c evenly spaced data values, from the minimum to the maximum.

Data values should be specified so that a contour line cannot cross any cell edge more than once. To locate the contour paths, cells could be processed row-by-row to determine which contour sections are in each cell, but it is usually best to trace one complete contour path before proceeding to the next contour value.

A cell table can be constructed that lists coordinate and data information. For each cell, the table information can include the coordinates for the four cell vertices, the four vertex data values, and the minimum and maximum data values for that cell.

Select a contour data value V_c and check the cell table row-by-row to find the first cell that satisfies the condition

$$V_{cellmin} \le V_c < V_{cellmax}$$

Next, test the four cell data values to determine if one or more vertex data values are equal to V_c (a contour line could be coincident with a cell edge). If not, locate the two cell edges that are intersected by this contour line and calculate the intersection coordinates using a linear interpolation. For example, if

$$V_{j,k} < V_c < V_{j,k+1}$$

for the left edge of the cell, then the contour-line intersection coordinates at that edge are

$$x_c = x_j, \qquad y_c = \frac{(V_{j,k+1} - V_c)y_k + (V_c - V_{j,k})y_{k+1}}{V_{j,k+1} - V_{j,k}}$$

The two cell intersections can then be joined with a straight-line segment. (A spline curve could also be used to approximate the contour path through the cell, but this can result in intersecting contour lines and misinterpretations of the contour path. A polyline contour clearly shows the boundaries of the data grid, which is usually a finite sampling of a continuous data field.)

Once the contour segment has been determined for the first cell, the adjacent cells are processed to locate the next contour segments. This cell processing is repeated until all cells have been examined or the contour forms a closed polyline. Then, the next contour line is plotted.

8-51. The grid of vector values for this exercise can be input as in Exercise 8-50, and all vectors could be two-dimensional (in the xy plane), with magnitudes less than the grid spacing. Input could also include a set of n color values for encoding the visualization.

Compute the magnitude squared for each vector, and divide the range of magnitudes squared into n subintervals. Each vector is then assigned a color value, and vector directions are displayed as in Fig. 8-133.

Alternatively, the input could be a grid of three-dimensional vector data. The visualization for the vector field could then be set up in three-dimensions and viewed from selected positions.