Chapter 9

Visible-Surface Detection Methods

Methods for identifying visible portions of a scene are explored in the exercises for this chapter, as well as the OpenGL functions for back-face removal and depth-buffer visibility testing.

Exercises

9-1. Input files could be used to specify the polyhedron vertices and the viewing parameters. Also, polygon tables could be constructed and the normal vector computed for each surface of the object. For a convex polyhedron, all back faces are identified using the inequality test 9-2.

9-2. Assuming that the polyhedron is completely contained within the view volume, no clipping operations are needed for this procedure. Therefore, the visible surfaces can be orthogonally projected (Sections 7-6 and 7-10) to the view plane once the back faces have been eliminated.

9-3. Assuming that the polyhedron is completely contained within the view volume, no clipping operations are needed for this procedure. Therefore, the visible surfaces can be mapped to the view plane, using a perspective projection (Sections 7-8 and 7-10) once the back faces have been eliminated.

9-4. A rotation axis can be established through any selected point on the polyhedron, such as a vertex or the centroid, with the axis perpendicular to the view-plane normal vector. Assuming that the polyhedron is completely contained within the view volume, no clipping operations are needed for this procedure. Therefore, the visible surfaces can be orthogonally projected (Sections 7-6 and 7-10) to the view plane once the back faces have been eliminated each time an incremental rotation (Sections 5-11 and 5-17) is applied to the object. The polyhedron can be rotated continuously until a termination input is given, using a small pause interval after each step of the rotation.

9-5. An input file could be used to specify the polyhedron vertices, and polygon tables can be constructed for the surface facets of the polyhedron. Set up an array that is large enough to contain positions for each projected point of the polyhedron on the view plane, and implement the depth-buffer algorithm given in Section 9-3. Storage requirements

for the depth-buffer array can be reduced by considering the coordinate extents of the polyhedron on the view plane. The size of this array is determined by the number of projected pixel positions and the number of bits per pixel needed for the depth values.

9-6. Input files could be used to specify the polyhedron vertices and the viewing parameters. Also, polygon tables could be constructed for each object, and a depth-buffer array is established for all objects in the scene. As in previous exercises, the scene can be assumed to be completely contained within the view volume.

9-7. The A-buffer stores a linked list for all surfaces that overlap each pixel (Section 9-4). Thus the depth-buffer array from the preceding exercise is modified to store a pointer value for each pixel position, instead of a depth value, and a linked list of surfaces is established for each pixel.

9-8. For this extension, antialiasing procedures from either Section 4-17 or Section 4-18 can be used to smooth edges of surface areas.

9-9. An input file could be used to specify the polyhedron vertices and the viewing parameters, and polygon tables can be constructed for the surface facets. Along each scan line for the scene, locate surface intersections and compare the depths of overlapping surfaces to determine visibility, as in Fig. 9-10. All surfaces could be opaque, or some could be transparent. If any surfaces are designated as transparent, blending procedures (Section 4-9) can be used to compute displayed surface characteristics. Depending on the shape of the polyhedron, the back side of a surface facet may be visible through a transparent surface.

9-10. The input file from the preceding exercise can be modified to contain the descriptions for several objects in a scene. Scan lines are processed using the methods from Section 9-5, and all objects could be opaque.

9-11. Modify Exercise 9-9 to display the visible polyhedron surfaces using the depth-sorting algorithm (Section 9-6). For a convex polyhedron, the program need apply only the simple depth checks.

9-12. Extend the program of the preceding exercise to process any input polyhedron, which, in general, will require more extensive depth checks.

9-13. Modify Exercise 9-10 to display the scene using the depth-sorting algorithm (Section 9-6). If all polyhedrons are convex, the program need apply only the simple depth checks.

9-14. An input file can be used to specify the polyhedron vertices, and polygon tables can be set up that contain the plane parameters for each surface facet of the polyhedron. Determine the coordinate extents for the polyhedron, and construct a binary tree, as in Fig. 9-19, using binary partitioning planes in the region containing the polyhedron. At each step in the tree construction, select one of the object surface facets and use the plane containing that facet to divide the space into two subspaces. The plane parameters for each facet can also be used to identify surfaces that are behind and in front of the plane.

For a convex polyhedron, all surface facets are on one side of any selected plane, but a nonconvex polyhedron could be input to the program.

9-15. Depth calculations for test 3 of the area-subdivision method are performed using the plane equations and the bounding coordinates of the area under consideration, and not the actual vertex coordinates of surfaces. For some cases, the calculated depth values will not correspond to actual depths of the surfaces within the area. In that case, the relative depths of surfaces will be incorrectly represented.

9-16. Initial tests can be made using the bounding rectangle (coordinate extents) of the surface, assuming that all surfaces are polygons. This bounding rectangle can be tested against the coordinate limits of the rectangular area to determine whether the surface is completely outside or inside. If the coordinate extents of the surface overlap the area, the individual edges of the surface are then tested.

9-17. The routines needed here are similar to those discussed for Exercise 8-31. Each node in the quadtree representation contains four fields. Divide the projected area of the object on the view plane into four quadrants, one for each quadtree node below the root node. If a quadrant is completely within the bounds of a single surface, the color for that surface is stored in the corresponding quadrant field. Otherwise, the quadrant is subdivided and a pointer to the next quadtree node is stored in the quadrant field. The quadtree is complete when all subdivisions of the space are homogeneous regions.

9-18. As in the algorithm for Exercise 8-32, the quadtree is searched for homogeneous regions, starting at the top of the tree. At each heterogeneous region, follow the pointers in the tree representation to the homogeneous regions. The color for a homogeneous region of the quadtree is assigned to the pixels along the scan lines within that region.

9-19. Assume that the octree (Section 8-21) has been constructed so that the viewing direction is toward the front nodes. The octree nodes can then be processed from front-to-back along scan lines until homogeneous regions are encountered, which specify either a visible surface color or the background color. The color areas can be transferred to a quadtree, or they can be loaded directly into the frame buffer. Different views of the object can be obtained by rotating the octree representation in front of the view plane.

9-20. A ray from any pixel position is a straight line that is perpendicular to the view plane. Set up an equation for each pixel ray, and calculate surface intersection positions along the rays. The closest intersection to the view plane is the visible surface position for a pixel.

A unit normal vector $\mathbf{u}$ for the view plane is in the opposite direction to the orthogonal projection direction from the scene. Thus, a ray from any pixel position $\mathbf{P}_0$ on the view plane can be described with the vector equation

$$\mathbf{P} = \mathbf{P}_0 + s\,\mathbf{u}$$

where s is the distance along the ray from position $\mathbf{P}_0$ to position $\mathbf{P}$.

A vector sphere equation can be written in the form

$$|\mathbf{P} - \mathbf{P}_c|^2 - r^2 = 0$$

where $\mathbf{P}$ is any surface position, $\mathbf{P}_c$ is the center position, and r is the radius of the sphere. Combining the pixel ray equation with the sphere equation yields

$$s = \mathbf{u}\cdot\Delta\mathbf{P} \pm \sqrt{(\mathbf{u}\cdot\Delta\mathbf{P})^2 - |\Delta\mathbf{P}|^2 + r^2}$$

with $\Delta\mathbf{P} = \mathbf{P}_c - \mathbf{P}_0$. The ray does not intersect the sphere if the discriminant of this equation is negative. Otherwise, the closer of the two ray intersection positions is on the visible side of the sphere surface.

For a scene containing a single sphere and no other objects, the ray-casting procedure can be simplified by performing intersection calculations only for those pixels along the interior of scan lines that intersect the projection of the sphere on the view plane. All other pixels are set to the background color. And if the sphere surface is a single color, with no textures or patterns, then intersection calculations are unnecessary.

9-21. In the scan-line algorithm, multiple scan lines can be generated per pixel. The pixel values across the various scan lines can then be averaged (Section 4-17) to produce the final intensity for each pixel. And either scan-line or area-sampling methods can be used in the depth-buffer, area-subdivision, and other visible-surface detection algorithms.

9-22. Contour curves in the xz plane can be plotted using the representation

$$z = f(x, y)$$

given the form of the surface function, a specified range of y values, and an increment Δy. Take unit steps in x and calculate the corresponding z values for each y contour, using the visibility checks discussed in Section 13-12.

9-23. Input for this algorithm can be the projected view of a scene, giving the projection coordinates for line endpoints. For example, input could be a set of polygon tables that specify the positions for surface facets after they have been projected to the view plane. For each line segment in the projected scene, first test the positions of its two endpoints relative to the coordinate extents of each polygon surface. If one or both endpoints are within the coordinate extents of a surface facet, locate the intersection positions, if any, for the line segment and each edge of the surface facet. Then calculate the depths of the line segment and the surface edges at the intersection positions to determine visibility.

Alternatively, a scene definition and viewing parameters could be specified in an input file. Then the scene is projected to a view plane, and visible line segments are identified.

9-24. Most methods for identifying the visible parts of a scene can be applied to wire-frame displays, although some are more efficient than others. Back-face elimination methods, however, cannot be used to locate visible line segments.

In the depth-buffer algorithm, depth calculations can be performed for positions along a line segment as it is scan converted. These depth values are then compared to the values previously stored in the depth buffer.

The depth-sorting method can be used to sort line segments along with the surfaces in a scene. Both lines and surfaces are then loaded into the frame buffer in depth order, from back to front.

In the area-subdivision method, the coordinate extents of a projected line segment could be used to determine overlap regions, if any, with surfaces or other lines. Then subdivision methods are applied to locate the homogeneous regions. The subdivision region for an isolated or foreground vertical line is a vertical column of pixels. Similarly, the subdivision region for an isolated or foreground horizontal line is a horizontal row of pixels. Other lines are subdivided into a set of pixel-size regions.

Octree methods could be applied to scenes containing both individual line segments and surface facets. A surrounding cube for a scene is repeatedly divided into octants until homogeneous regions are obtained. The voxel regions representing the line segment in the octree each contain just one pixel.

And ray-casting procedures can also be used to locate visible line segments in a scene. First, a pixel ray is tested for intersection with the bounding box of a line segment. If the ray hits this box, calculate the intersection position for the ray and line, using parametric equations for both the pixel ray and the line segment. If the intersection position is between the two endpoints for the line segment, compare the depth of the line-segment intersection position with the depths of other objects intersected by that ray.

9-25. The algorithm for Exercise 9-23 can be modified to display hidden sections of line segments as dashed lines, rather than eliminating them. The sizes for the dashes and spaces along a line segment could be specified as input. And the dashes could be generated using a modified line-drawing algorithm (Section 3-5) or OpenGL routines (Section 4-8).

9-26. An input file can be used to specify the polyhedron vertices, the colors for the surface facets, and the viewing parameters. And keyboard input could be used to indicate whether front or back faces are to be removed. The viewing position could be moved inside the polyhedron, with back faces projected for viewing instead of the front faces. And, the scene could be expanded to include isolated line segments and other objects, such as a point set, and all polygon surfaces could be selected for removal so that only the other objects are displayed.

9-27. The program in Exercise 9-26 is to be altered by eliminating the culling routines and replacing them with the OpenGL depth-buffer functions (Section 9-14). Also, interactive input could be used to change the viewing position, so that any front face of the polyhedron could be viewed. As an option, both the culling and depth-buffer routines could be applied to the polyhedron.

9-28. Interactive input could be used to specify the depth range and test condition for this modification of the program in Exercise 9-27.

9-29. An input file can be used to specify the polyhedron vertices and the viewing parameters. The values for the foreground and background colors in the depth-offset method could also be specified as input.

9-30. The default minimum and maximum depth values can be used for the depth cueing calculations, or other values could be assigned.

9-31. Polygon tables could be specified for each of the polyhedrons, or the input file could simply contain the vertex coordinates for each object, so that the polygon tables can be constructed after the input is received. And interactive keyboard input could be used to specify the depth range.