Chapter 10

Illumination Models and Surface-Rendering Methods

The exercises for this chapter explore various aspects of light sources, lighting models, scan-line surface rendering, ray tracing, radiosity, and texture mapping, in addition to applications of the OpenGL functions for illumination effects, surface rendering, and texturing.

Exercises

10-1. All input could be specified in data files, including the polygon tables for the tetrahedron, the background lighting level I_a, the light-source position and intensity I_l, and the coefficients k_a and k_d. For each surface of the tetrahedron, one of the vertices can be selected to determine the unit vector **L** to the light source. (If the light source is sufficiently far from the surface, **L** is approximately constant for that surface.) Normal vectors from the polygon tables are used to calculate the unit normal **N** for each surface, and a single value for I_{diff} is calculated for each surface.

10-2. The input polygon tables for Exercise 10-1 could be modified to provide the vertex coordinates for the surface facets, or a sphere could be defined by simply specifying its radius (relative to the coordinate origin or a designated center position) and the number of polygon patches that are to be used to approximate the sphere surface. In the latter case, the polygon tables for the surface facets can be constructed using the methods discussed for Exercise 8-1.

10-3. For Gouraud surface rendering: Calculate a surface normal vector for each polygon facet, identify those facets with one or more common vertices, obtain the average unit normal vector at each vertex of a surface patch, calculate the diffuse intensity at each vertex of a surface patch, and linearly interpolate the intensities across the surface patches. A parallel projection could be used to map the surfaces onto a specified view plane.

Incremental intensity calculations along a scan line are performed using Eq. 10-53, and the incremental calculations along a polygon edge are performed as indicated in Fig. 10-49. For scan line y, the first scan line below the vertex with an intensity value of I_1, the

intensity at the intersection with the left edge is calculated as

$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2$$

And for scan line $y - 1$, the intensity at the intersection with the left edge is calculated as

$$I' = I + \frac{I_2 - I_1}{y_1 - y_2}$$

10-4. For Phong surface rendering: Calculate a surface normal vector for each polygon facet, identify those facets with one or more common vertices, obtain the average unit normal vector at each vertex of a surface patch, linearly interpolate the normal vectors across the surface patches, and apply the diffuse-intensity calculations at each surface pixel position. A parallel projection could be used to map the surfaces onto a specified view plane. The incremental calculations are performed using the vector equivalents of the equations in Exercise 10-3.

10-5. This exercise is an extension of Exercises 10-2 and 10-3. Surface illumination is now calculated as a combination of diffuse and specular reflection terms.

10-6. This exercise is an extension of Exercises 10-2 and 10-4. Surface illumination is now calculated as a combination of diffuse and specular reflection terms.

10-7. Values for parameters a_0 and a_1 could be specified as input ($a_2 = 0$), and the value for the radial intensity attenuation function is calculated using Eq. 10-1. The intensity at each surface position is now computed using Eq. 10-19 with one light source, no surface emissions, and the angular intensity attenuation set to the value 1.0.

10-8. The position and intensity for each of the two point light sources could be specified as input, and the intensity at each surface position is computed using Eq. 10-19 with no surface emissions and the angular intensity attenuation set to the value 1.0.

10-9. Parameters for the pane of glass, such as its reflection and transparency coefficients, could be specified as input. The light intensity can be calculated from Eq. 10-30, or refraction effects (Section 10-4) could be used to obtain the projected view of the sphere.

10-10. In general, changes in $\mathbf{N}\cdot\mathbf{H}$ are smaller than the corresponding changes in $\mathbf{V}\cdot\mathbf{R}$ as the viewing position changes. Also, $\mathbf{V}\cdot\mathbf{R}$ contributions are negative when $\phi > 90°$.

10-11. If all vectors in Figs. 10-16 and 10-22 are coplanar, the angle between $\mathbf{H}$ and $\mathbf{L}$ is $\theta + \alpha$ and the angle between $\mathbf{H}$ and $\mathbf{V}$ is $\theta - \alpha + \phi$. But $\mathbf{H}$ is the halfway vector between $\mathbf{L}$ and $\mathbf{V}$, so these two angles are equal and $2\alpha = \phi$. However, if $\mathbf{V}$ is not in the plane defined by $\mathbf{L}$ and $\mathbf{N}$, then $\mathbf{H}$ is also not in that plane, and the angle between $\mathbf{L}$ and $\mathbf{H}$ may not be $\theta + \alpha$.

10-12. Back-face detection can be used only to eliminate those surfaces that are not facing toward the viewer. The depth-buffer method directly employs an illumination model,

as noted in the algorithm of Section 9-3. Similarly, all other visibility detection algorithms invoke an illumination model once the visible surfaces have been identified. And ray-casting procedures are fundamental to a ray-tracing routine, which also invokes an illumination model to compute surface intensities.

10-13. The basic depth-buffer algorithm cannot generate transparency effects. But the A-buffer method, which is an extension of the depth-buffer approach, can handle transparent surfaces. Scan-line methods can be used to calculate transparency effects by obtaining intensities from objects behind a transparency surface. Ray-casting methods form the initial phase in a ray-tracing algorithm, which processes both reflected and refracted rays through a scene.

10-14. A visible-surface detection method, such as the scan-line algorithm (Section 9-5), can be used to identify shadow areas by determining which parts of the scene are not visible from the light-source position. Shadow areas that are visible to the viewer are illuminated with ambient light only.

10-15. Each pixel in the grid can be assigned $m - 1$ intensity values above 0, and there are n^2 pixels in the grid. Therefore, the total number of intensity levels that can be assigned to the grid is

$$(m - 1)n^2 + 1$$

where the grid intensity is the sum of the individual pixel intensities.

10-16. For a 3 by 3 pixel grid on a bilevel *RGB* system, there are ten possible settings for each phosphor color (Fig. 10-40) and three phosphor color dots per pixel. Therefore, the total number of color combinations for the grid is 10^3.

10-17. Given a surface facet, such as a rectangle or triangle, project the facet to the view plane and tile it with a set of 3 by 3 pixel grids. The surface facet can be tiled starting at some position on or near the facet, as illustrated in Fig. 4-31. Apply an illumination model to the center pixel of each grid cell that overlaps the facet, and normalize the calculated intensities to the range from 0.0 to 1.0.

Ten intensities, labeled 0 through 9, can be represented with the 3 by 3 pixel grids, and the calculated intensities can be distributed over the ten halftone approximation patterns as in Fig. 10-40. Other schemes could also be used for distributing the intensity values. For example, an intensity value less than 0.05 could be mapped to pattern 0, an intensity in the interval from 0.05 to 0.15 could be mapped to pattern 1, and an intensity above 0.95 could be mapped to level 9.

10-18. Input for this routine is a value for n that is a multiple of 2. Matrix D_n is then obtained from D_2, using a series of recurrence calculations.

10-19. Each pixel position $(x,\ y)$ is mapped to a matrix position using Eqs. 10-47. The pixel is turned on if the value of the matrix element is less than the array intensity value for that pixel.

10-20. Values for the error-diffusion parameters can be input to this routine, or the routine could use the values shown in Fig. 10-45.

10-21. The checkerboard and sphere positions are to be specified at some position in front of the projection plane, with the projection reference point behind the plane, as illustrated in Fig. 10-52. Pixel rays are then generated at positions along the scan lines within the limits of the projected scene. The ray-tracing tree is constructed as shown in Fig. 10-55. Intersection positions on the sphere surface are calculated using Eq. 10-71, and intersection positions on the plane of the checkerboard are calculated using Eq. 10-76 and inside-outside polygon tests.

The checkerboard colors could be specified as input or the board can be constructed with alternating black and white squares. Similarly, the color for the spherical surface could be given as input RGB values, and the illumination effects are calculated at the surface intersection positions.

10-22. Modify the program in the preceding exercise to include multiple objects, with or without the checkerboard. A bounding sphere around each object can be used for the initial ray-tracing tests.

10-23. Modify the program in the preceding exercise so that the entire scene is enclosed within a bounding cube. Repeatedly subdivide the cube until each subdivision of the cube contains no more than n objects, where integer n can be predefined or specified as input. Then trace each ray through the various subdivisions, testing for object intersections within those cells that contain one or more objects.

10-24. A scene description can be input in a data file, and bounding spheres or space subdivision can be used in the algorithms for the initial ray-intersection tests. Initial positions for the pixel rays are generated as in Fig. 10-79, directions for scattered reflections and transmissions are generated as in Fig. 10-81, and shadow rays are generated over the coordinate extents of the light sources as in Fig. 10-82.

10-25. For this routine, a textured sphere can be defined with its center at some initial position, and a motion path can be specified as a series of displacement positions from the initial position. The position displacements can be translational offsets or angular increments about a rotation axis. One ray per pixel can be generated at some random (jittered) time within the total time frame for the motion. Jittered times can be obtained by randomly assigning pixel rays to particular positions along the motion path. A pixel ray is then processed as in Exercise 10-21 (without the checkerboard), with the sphere placed at the proper position corresponding to the time assigned to that pixel.

10-26. Apply the algorithm for the basic model discussed in Section 10-12 to a specified rectangular box with one face of the box open to the viewer. Input parameters for this routine include the form factors, energy emission from the surface light source, and the surface reflectivities.

10-27. Given a set of surfaces, set $B_k = E_k$ for all surface patches. Then, select the patch with the highest radiosity value and approximate the radiosity for all other patches. Repeat this process, as specified in the algorithm steps given in Section 10-12.

10-28. An input file can be used to specify viewing parameters and the environment map as a box or a spherical surface around a given opaque sphere. From the projection reference point, each pixel area on the view plane is then reflected from the sphere in the scene to the environment map (Fig. 10-98). A color is obtained for the pixel by averaging the environment-map colors that are within the limits of the pixel projection. The input file could also be used to describe a group of objects, instead of a single sphere.

10-29. Set up a surface texture pattern, as described in Section 10-17. An input file can be used to provide polygon tables for the cube, and interactive input could be used to select one of the cube faces. The four corners of the rectangular pattern are then mapped to the four corners of the selected cube surface, and the texture colors are mapped to the corresponding pixel positions on the surface. An orthogonal projection can be used to view the textured cube surface.

10-30. The surface texture pattern can be mapped to one face of the tetrahedron by aligning two adjacent corners of the pattern with two vertices of the selected face. Pixel positions across the surface scan lines are then mapped to positions within the texture array.

10-31. A section of the sphere surface can be specified with four coordinate positions, using a spherical-coordinate representation for each position. Then the surface section can be transformed into the rectangular texture pattern, so that the surface points are assigned corresponding texture colors.

10-32. The program of Exercise 10-29 can be modified so that two diagonally opposite corners are chosen on the cube face, and these two vertex positions are mapped to the end positions of the one-subscript texture array. The texture pattern is then superimposed on the selected diagonal line, and the cube is projected to the view plane.

10-33. Two input positions on the sphere surface can be specified in spherical coordinates, and an input file can be used to provide the viewing parameters and the sphere radius and position. A surface point along the arc between the two input positions is then assigned a texture color corresponding to its distance from the arc endpoints. Spherical-coordinate calculations can be used to determine positions along the arc, which has a radius r relative to the sphere center. A ray-tracing algorithm could be used to view the sphere and the texture stripe, or a scan-line method on the view plane could be used to determine the spherical surface colors.

10-34. For a given spherical surface, the surface normal vector at any position is along the line from the sphere center through the surface point. This normal vector is then modified using Eq. 10-116. For this procedure, a sinusoidal bump function $b(u, v)$ could be defined in parametric coordinates over the sphere surface.

10-35. Modify the procedure from the preceding exercise so that a file of bump values is input and a table lookup is used to obtain values for the bump function. Linear interpolation can be used to obtain the bump values, and finite-difference calculations (Appendix A) can be used to compute the partial derivatives of the bump function. A bump-function table can be generated by evaluating an analytic bump function at selected increments for the independent variables.

10-36. An input file could be used to give the polygon tables for the tetrahedron and the various illumination and viewing parameters, including the positions of the light sources. Section 10-20 provides example applications for the OpenGL functions to be used in this program.

10-37. Parameters for the spotlights could be included in the input file, or they could be given interactively.

10-38. A smoky atmosphere can be generated by specifying a light gray color in the `glFog` function.

10-39. Parameters for the `glBlendFunc` could be specified in the input file.

10-40. Complete the line-texture programming example in Section 10-21, and specify endpoint coordinates for a set of diagonal lines. An interactive routine could be used to select color choices and texture patterns.

10-41. Expand the surface-texture programming example in Section 10-21 to map an 8 by 8 pattern of black and white squares onto a square of any selected size.

10-42. Set the background color for the display window to white and provide a menu of color choices, allowing any two colors to be selected for the checkerboard.

10-43. Set up a two-dimensional pattern of diagonal stripes using the OpenGL surface-texture functions (Section 10-21), and map the pattern onto a white rectangle of any selected size.

10-44. An input file can be used to provide the viewing parameters and the sphere color, radius, and position. Four coordinate positions on the sphere surface can be specified using a spherical-coordinate representation for each position. Then the projected surface section can be transformed into the rectangular texture pattern, so that the surface points are assigned corresponding texture colors.

Alternatively, the texture pattern could be applied to the entire sphere surface. This could be accomplished using the function `glutSolidSphere`, discussed in Section 8-6, and the `gluQuadricTexture` function, discussed in Section 10-21.

10-45. Use the `glutSolidTeapot` function (Section 8-6) to generate the teapot, and use the `gluQuadricTexture` function (Section 10-21) to map the pattern onto the teapot surface.