Angel and Shreiner: Interactive Computer Graphics, Sixth Edition

Chapter 6 Solutions

6.1 First, consider the problem in two dimensions. We are looking for an α and β such that both parametric equations yield the same point, that is

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2 = (1-\beta)x_3 + \beta x_4,$$

$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2 = (1-\beta)y_3 + \beta y_4.$$

These are two equations in the two unknowns α and β and, as long as the line segments are not parallel (a condition that will lead to a division by zero), we can solve for α β. If *both* these values are between 0 and 1, the segments intersect.
If the equations are in 3D, we can solve two of them for the α and β where x and y meet. If when we use these values of the parameters in the two equations for z, the segments intersect if we get the same z from both equations.

6.2 We can form the equation of the plane of the first polygon using any three of its vertices. We can next put each of the vertices of the second polygon into this equation. If we get the same sign for all, then the second polygon cannot intersect the first. If two successive vertices have different signs, the corresponding edge intersects the plane of the first polygon. We can then determine the point of intersection and see if this point lies inside the first polygon. Note that we also must test of the first polygon intersects the second.

6.3 If we clip a convex region against a convex region, we produce the intersection of the two regions, that is the set of all points in both regions, which is a convex set and describes a convex region. To see this, consider any two points in the intersection. The line segment connecting them must be in both sets and therefore the intersection is convex.

6.4 The object–oriented approach has an outer loop that goes over objects.We can try to parallelize the rendering of objects. However, we would need some method of combining the objects at the end and performing hidden surface removal. One possibility is to have parallel rendering modules that share a frame buffer and a depth buffer.
The image–oriented approach has an outer loop over pixels. Thus, one approach to parallelization would be to have processor that could process

pixels independently. One example of this strategy is a parallel ray tracer. However, in this case, each processer must have access to all the objects so we would probably want to put the objects into shared memory.

6.5 See Problem 5.22. Nonuniform scaling will not preserve the angle between the normal and other vectors.

6.6 First we want to move the window to the center with the translation $T(-(x_{max}+x_{min})/2, -(y_{max}+y_{min})/2, 0)$. Next we want to scale the sides to be the size of the viewport by $S(\frac{u_{max}-u_{min}}{x_{max}-x_{min}}, \frac{v_{max}-v_{min}}{y_{max}-y_{min}}, 1)$. Finally we move the origin to the center of the viewport by
$T((u_{max}+u_{min})/2, (v_{max}+v_{min})/2, 0)$.

6.7 Note that we could use OpenGL to, produce a hidden line removed image by using the z buffer and drawing polygons with edges and interiors the same color as the background. But of course, this method was not used in pre–raster systems.
Hidden–line removal algorithms work in object space, usually with either polygons or polyhedra. Back–facing polygons can be eliminated. In general, edges are intersected with polygons to determine any visible parts. Good algorithms (see Foley or Rogers) use various coherence strategies to minimize the number of intersections.

6.8 We can use our ability to render filled polygons by forming quadrilaterals. Consider the back (furthest from the viewer) two sets of data $\{f(x_i, z_N)\}$ and $\{f(x_i, z_{N-1})\}$. We form quadrilaterals from the values $\{f(x_i, z_N)\}$,$\{f(x_{i+1}, z_N)\}$, $\{f(x_i, z_{N-1})\}$, and $\{f(x_{i-1}, z_{N-1})\}$ for $i = 0, 1, 2, ...$ Once these are rendered, we go on to form quadrilaterals from rows $N-1$ and $N-2$ and so on. Note that even if we do not have a z buffer available, we achieve hidden surface removal by the painting the polygons back to front.
We can derive a hidden line removal by rendering the curves front to back. Consider the curve $\{f(x_i, z_0)\}$. Because it is the front curve, it cannot be obscured from the viewer by any other curve. We can from a polygon from these points and a line $y = c$ below the curve in the plane $z = z_0$. The second curve can only be obscured from the viewer by the first curve (the *visible top*) and the curve $y = c$ (the *visible bottom*). We can draw the parts of the curve above the visible top and the visible bottom, updating the visible top and visible bottom as necessary. We can also form a mesh by connecting corresponding points of the curve with the visible top and bottom. We then proceed with the successive curves, front to back. Note

that part of the calculation involves intersecting the curve with the visible top and bottom. This calculation must be carried out to the resolution of the display to avoid dangling line segments. This algorithm does not require fill and was used before raster displays became available.

6.9 The $O(k)$ was based upon computing the intersection of rays with the planes containing the k polygons. We did not consider the cost of filling the polygons, which can be a large part of the rendering time. If we consider a scene which is viewed from a given point there will be some percentage of the area of the screen that is filled with polygons. As we move the viewer closer to the objects, fewer polygons will appear on the screen but each will occupy a larger area on the screen, thus leaving the area of the screen that is filled approximately the same. Thus the rendering time will be about the same even though there are fewer polygons displayed.

6.10 Each convex polyhedra can be culled because none of its back faces are visible. If the polyhedra are solid, then there can be no intersecting surfaces and an object space algorithm such as depth sort may work well.

6.11 There are a number of ways we can attempt to get $O(k \log k)$ performance. One is to use a better sorting algorithm for the depth sort. Other strategies are based on divide and conquer such a binary spatial partitioning.

6.12 We can find the equation of the plane of one the polygons from any three of its vertices. We can then successively test the values of all the vertices of the other in the this equation. If we get the same sign for each vertex, then the second polygon does not intersect the first. Note that we must also test if the first polygon intersects the second by forming the equation of the plane from three vertices of the second.
We can also test the sign of the determinant

$$\begin{vmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_i & y_i & z_i & 1 \end{vmatrix},$$

where the first three lines are from three vertices of the first (second) polygon and the last vertex is each vertex of the second (first) polygon.

6.13 If we consider a ray tracer that only casts rays to the first intersection and does not compute shadow rays, reflected or transmitted rays, then the image produced using a Phong model at the point of intersection will be the same image as produced by our pipeline renderer. This approach is sometimes called ray casting and is used in volume rendering and CSG. However, the data are processed in a different order from the pipeline renderer. The ray tracer works ray by ray while the pipeline renderer works object by object.

6.15 Consider a circle centered at the origin: $x^2 + y^2 = r^2$. If we know that a point (x, y) is on the curve than, we also know $(-x, y)$, $(x, -y)$, $(-x, -y)$, (y, x), $(-y, x)$, $(y, -x)$, and $(-y, -x)$ are also on the curve. This observation is known as the eight–fold symmetry of the circle. Consequently, we need only generate 1/8 of the circle, a 45 degree wedge, and can obtain the rest by copying this part using the symmetries. If we consider the 45 degree wedge starting at the bottom, the slope of this curve starts at 0 and goes to 1, precisely the conditions used for Bresenham's line algorithm. The tests are a bit more complex and we have to account for the possibility the slope will be one but the approach is the same as for line generation.

6.16 If we look at all the paths generated by flood fill they visit each square once. Thus if we were to draw lines between these paths we would have a maze. Alternately, we can start with cells in which all walls are present and as we flood fill, we remove a wall each time we enter a new cell. In order to generate a different maze each time we can order the recursively calls randomly.

6.17 Flood fill should work with arbitrary closed areas. In practice, we can get into trouble at corners if the edges are not clearly defined. Such can be the case with scanned images.

6.18 Suppose that the equation of the edge is $y = mx + h$ where m and h are determined from (x_1, y_1) and (x_2, y_2). For any change Δy in y, the corresponding change in x must be $\Delta x = \frac{1}{m}\Delta y$. Thus if (x_i, y_i) is the intersection of the edge with a scan line, the intersection with the next scanline must be at $(x_i + \frac{1}{m}, y_i + 1)$ because we move one unit in y and must still be on the edge. Thus, once we have the first intersection, the calculation of successive intersection requires only one addition per intersection.

6.19 Note that if we fill by scan lines vertical edges are not a problem. Probably the best way to handle the problem is to avoid it completely by never allowing vertices to be on scan lines. OpenGL does this by having vertices placed halfway between scan lines. Other systems jitter the y value of any vertex where it is an integer.

6.20 Consider scan line algorithms. Each vertex has a priority, so we can think of all the intersections of edges of all the polygons with scan lines generating a set of (x, y, p) triplets where p is the priority of a polygon. We must now sort by the three values and the order of the sort determines the fill algorithm. For example, in the $p - yx$ algorithm, we first sort by priority and then apply the yx algorithm. This is a painter's algorithm that paints the least important polygon first. The $y - px$ algorithm proceeds a scan line at a time, applying priority on each scan line.

6.21 Although each pixel uses five rays, the total number of rays has only doubled, i.e. consider a second grid that is offset one half pixel in both the x and y directions.

6.22 As long as the radius is less than 0.5, the circle is entirely within the square and the intensity is the area of the circle: πr^2. For values of the radius between 0.5 and $\frac{\sqrt{2}}{2}$ the intensity may increase slower because part of the pixel overlaps adjacent pixels. If the adjacent pixels are lit, the overlapped area will not contribute any additional brightness.

6.23 A mathematical answer can be investigated using the notion of reconstruction of a function from its samples (see Chapter 7). However, a very easy to see by simply drawing bitmap characters that small pixels lead to very unreadable characters. A readable character should have some overlap of the pixels.

6.24 The jaggedness we see in rasterized lines is caused by having to choose a single pixel for each x (or y). A line of one pixel width actually covers more than one pixel for each x (or y), something that an antialiasing algorithm takes care of. In a CRT, if the beam is not well focussed, the pixel we draw on the display is wider and covers more than the desired pixel, an effect that will make the jaggedness less apparent, although the line will be wider. Thus a poorer display, one that cannot focus well, may appear to have antialiased the display.

6.25 We want k levels between I_{min} and I_{max} that are distributed exponentially. Then $I_0 = I_{min}$, $I_1 = I_{min}r$,
$I_2 = I_{min}r^2, ..., I_{k-1} = I_{max} = I_{min}r^{k-1}$. We can solve the last equation for the desired $r = (\frac{I_{max}}{I_{min}})^{\frac{1}{k-1}}$

6.26 We can use a simple threshold test where the threshold changes each time based on the random number generator. Thus, for each pixel we compare it to a random number between 0 and 1. If the pixel exceeds the random number we color it white, if it is less we color it black. In most case there are fewer pixels in the image than binary locations on the display so we must repeat the experiment. Thus if we want to display a 512 x 512 gray scale image on a 2048 x 2048 binary display, each pixel should be compared to 16 random numbers to generate a 4 x 4 bit pattern.

6.27 If there are very few levels, we cannot display a gradual change in brightness. Instead the viewer will see steps of intensity. A simple rule of thumb is that we need enough gray levels so that a change of one step is not visible. We can mitigate the problem by adding one bit of random noise to the least significant bit of a pixel. Thus if we have 3 bits (8 levels), the third bit will be noise. The effect of the noise will be to break up regions of almost constant intensity so the user will not be able to see a step because it will be masked by the noise. In a statistical sense the jittered image is a noisy (degraded) version of the original but in a visual sense it appears better.

6.28 Consider the trapezoid formed by the an edge and the x axis or the four points (x_i, y_i), (x_{i+1}), $(x_i, 0)$, and $(x_{i+1}, 0)$. The area of this trapezoid is $\frac{1}{2}\frac{y_{i+1}+y_i}{x_{i+1}-x_i}$. Suppose that all the y_i are positive (an assumption which is not really necessary). Then if $x_{i+1} > x_i$, the area is positive and if $x_{i+1} < x_i$ the area is negative. If we sum these areas over all i (modulo the number of edges) then all we are taking the area under the upper edges of the polygon and subtracting the area under the lower edges, leaving the interior area of polygon.