Chapter 7 Solutions

7.1 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 segements 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.

7.3 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 therfore the intersection is convex.

7.5 See Problem 6.22. Nonuniform scaling will not preserve the angle between the normal and other vectors.
7.7 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.
7.9 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 apporximately the same. Thus the rendering time will be about the same even though there are fewer polygons displayed.

7.11 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.

7.13 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.

7.15 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.

7.17 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.

7.19 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.

7.21 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.

7.23 A mathematical answer can be investigated using the notion of reconstruction of a function from its samples (see Chapter 10). 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.

7.25 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 $r = (\frac{I_{max}}{I_{min}})^{\frac{1}{k-1}}$

7.27 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.