Lecture 22: Hidden Surface Algorithms

thou didst hide thy face, and I was troubled. Psalm 30:7

1. Hidden Surface Algorithms

Surfaces can be hidden from view by other surfaces. *The purpose of hidden surface algorithms is to determine which surfaces are obstructed by other surfaces in order to display only those surfaces visible to the eye*. In theory, hidden surface algorithms are required for all types of surfaces; in practice, we shall restrict our attention to polygonal models.

In this lecture we shall assume that all the surfaces are planar polygons and that all the polygons are opaque. We shall not assume that the polygons necessarily enclose a solid nor that the polygons form a manifold. Our objective is: given any collection of polygons to display only those polygons visible to the eye.

Polygonal models of complicated shapes may contain millions of polygons, so just like shading algorithms, the emphasis in hidden surface algorithms is on speed. A good hidden surface algorithm must be fast as well as accurate. Sorting, tailored data structures, and pixel coherence are all employed to speed up hidden surface algorithms.

There are two standard types of hidden surface algorithms: image space algorithms and object space algorithms. Image space algorithms work in pixel space (the frame buffer) and are applied after pseudoperspective; object space algorithms work in model space and are applied before pseudoperspective. These algorithms have the following generic structures:

Image Space Algorithm

```
For each pixel:
    Find the closest polygon.
    Render the pixel with the color
    and intensity of this polygon.
```

Object Space Algorithm

```
For each polygon:
    Find the unobstructed part of the polygon.
    Render this part of the polygon.
```

The speed of image space algorithms is $O(nN)$, where N is the number of pixels and n is number of polygons, because for each pixel we must examine every polygon. The speed of object space algorithms is $O(n^2)$, where n is the number of polygons, because to determine which part of each polygon is visible, we need to compare every polygon with every other polygon.

Many hidden surface algorithms have been developed, each with their own advantages and disadvantages. We shall examine five of the most common hidden surface algorithms: z-buffer,

scan line, ray casting, depth sort, and bsp-tree. The *z*-buffer and scan line algorithms are image space algorithms; the depth sort and bsp-tree algorithms are object space algorithms. Ray casting can work both in image space and in object space. We will compare and contrast the relative merits and limitations of each of these algorithms, and we shall see that there is no single solution to the hidden surface problem that is optimal in all situations.

2. The Heedless Painter

The heedless painter displays the polygons in a scene in the order in which they occur in some list. Polygons that appear early in the list can be overpainted by polygons that appear later in the list.

The heedless painter is slow because the same pixel may be painted many times, once for each polygon that lies over the pixel. Worse the scene displayed by the heedless painter is often incorrect because polygons that appear later in the list may overpaint polygons that appear earlier in the list, even though the later polygons actually lie behind the earlier polygons.

The heedless painter procedure is not a true hidden surface algorithm. Nevertheless, valid hidden surface algorithms can be generated by fixing the problems in the heedless painter procedure. We begin with two such hidden surface algorithms: *z*-buffer and scan line.

3. *Z*-Buffer (Depth Buffer)

The *z*-buffer or depth buffer algorithm employs a special data structure called the *z-buffer* or *depth buffer*. The *z*-buffer is a large memory array which is the same size as the frame buffer -- that is, the *z*-buffer stores an entry for each pixel. This entry consists of the current depth as well as the current color or intensity of the corresponding pixel.

Using this data structure, the *z*-buffer algorithm proceeds much as the heedless painter visiting each polygon in turn, but with one crucial difference: a pixel is overpainted -- that is, a new color or intensity is stored in the *z*-buffer -- only if

depth of current polygon at pixel < current depth of pixel in z-buffer.

The *z*-buffer begins with the color or intensity of each pixel initialized to the background color or intensity, and the depth of each pixel initialized to infinity. To make the *z*-buffer algorithm as fast as possible, the depth of the pixels in each polygon can be computed incrementally.

To compute the depth of each pixel incrementally, we proceed almost exactly as we did in

Lecture 21 for Gouraud and Phong shading, where we computed the intensities and normal vectors at the pixels of a polygon incrementally.

i. First, compute the depth at the vertices of the polygon. After pseudoperspective, this depth is typically just the value of the z-coordinate at each vertex.
ii. Next, compute the depth along the edges of the polygon.
iii. Finally, compute the depth along scan lines for the pixels in the interior of the polygon.

The explicit details of this computation are given below.

Recall that for a line $L(t)$ passing through the points P_1, P_2

$$L(t) = (1-t)P_1 + tP_2 = P_1 + t(P_2 - P_1)$$
$$L(t + \Delta t) = P_1 + (t + \Delta t)(P_2 - P_1).$$

Subtracting the first equation from the second equation yields

$$\Delta L = \Delta t(P_2 - P_1)$$

or equivalently

$$\Delta x = \Delta t(x_2 - x_1) \qquad \Delta y = \Delta t(y_2 - y_1) \qquad \Delta z = \Delta t(z_2 - z_1).$$

Thus along any line we can compute depth incrementally by setting

$$z_{next} = z_{current} + \Delta z.$$

Along a scan line

$$\Delta x = 1 \Rightarrow \Delta t = \frac{1}{x_2 - x_1} \Rightarrow \Delta z = \frac{z_2 - z_1}{x_2 - x_1},$$

and when we move to the next scan line along a polygonal edge

$$\Delta y = 1 \Rightarrow \Delta t = \frac{1}{y_2 - y_1} \Rightarrow \Delta z = \frac{z_2 - z_1}{y_2 - y_1}.$$

If these computations are not completely familiar, you should review Gouraud shading in Lecture 21, Section 3.

The advantages of the z-buffer algorithm are that the z-buffer is simple to understand and easy to implement; the disadvantages are that the z-buffer is memory intensive and relatively slow, since this algorithm may repaint the same pixel many times. The next algorithm we shall study avoids both of these drawbacks by introducing more complicated data structures.

4. Scan Line

The scan line algorithm paints the pixels scan line by scan line. To decide which polygon to paint for each pixel, the scan line algorithm for hidden surfaces maintains two special data structures: an *active edge list* and an *edge table*.

An edge is said to be *active* if the edge intersects the current scan line. For each active edge with end points $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$, the a*ctive edge list* stores the following data:

$$Edge\,Data = (Y_{\max}, current\,X_{\text{int}}, current\,Z_{\text{int}}, \Delta x, \Delta z, pointer\ to\ polygon),$$

where

$Y_{max} = max(y_1, y_2)$ = maximum value of y along the edge;

$current\,X_{\text{int}}$ = x-coordinate of the point where the edge intersects the current scan line;

$current\,Z_{\text{int}}$ = z-coordinate (depth) of the point where the edge intersects the current scan line;

$\Delta x = \dfrac{x_2 - x_1}{y_2 - y_1}$ = change in x along the edge when moving to the next scan line;

$\Delta z = \dfrac{z_2 - z_1}{y_2 - y_1}$ = change in z along the edge when moving to the next scan line;

pointer to polygon -- points to the polygon containing the edge.

The expressions for Δx and Δz can be derived in the same way as the formula for Δz in the z-buffer algorithm.

For each scan line, we also introduce an *edge table* consisting of a list of those edges whose lower vertex lies on the scan line. The data stored for each edge in the edge table is the same as the data that is stored for each edge in the active edge list.

New edges must be introduced into the active edge list as they become active, and old edges must be removed as they become inactive. Updating the active edge list is performed by the following algorithm.

Updating the Active Edge List

For each edge in the active edge list:

If $Y_{scan\ line} > Y_{\max}$, delete the edge from the active edge list.

Otherwise update the values of $current\,X_{\text{int}}$ and $current\,Z_{\text{int}}$

$current\,X_{\text{int}} = current\,X_{\text{int}} + \Delta x$

$current\,Z_{\text{int}} = current\,Z_{\text{int}} + \Delta z$

Insert each edge in the edge table for the current scan line ($Y_{scan\ line}$) into the active edge list.

Sort the active edge list by increasing values of $current\,X_{\text{int}}$.

We can now use the following scan line algorithm to paint the scene and avoid hidden surfaces.

Scan Line Algorithm

For each scan line:

Update the active edge list (see above algorithm).

Initialize

Current Polygon = polygon corresponding to the first edge in the active edge list.

Current Pixel = *current* X_{int} for the first polygon in the active edge list.

Repeat until the scan line reaches the pixel on the last edge in the active edge list:

For each polygon *P* corresponding to an edge in the active edge list with *current* $X_{\text{int}} >$ *Current Pixel*:

Compute the *z*-value z_P of the polygon *P* at the *Current Pixel* (see discussion below).

Determine the pixel x_P (if any) where the polygon *P* crosses in front of the *Current Polygon* (see discussion below).

If no polygon crosses in front of the *Current Polygon*:

Fill the pixels along the scan line from the *Current Pixel* to the end of the *Current Polygon* with the color and intensity of the *Current Polygon*.

Set

Current Polygon = the next polygon in the active edge list with *current* X_{int} greater than or equal to the last filled pixel.

Current Pixel = *current* X_{int} for *Current Polygon*.

Otherwise

Set

x_s = the smallest value of x_P

Q = polygon corresponding to x_s.

Fill the pixels along the scan line from the *Current Pixel* to x_s with the color and intensity of the *Current Polygon*.

Set

Current Polygon = Q

Current Pixel = x_s.

To determine the *z*-value z_P of a polygon *P* at the *Current Pixel,* simply evaluate the equation of the plane of the polygon *P* at $x =$ *Current Pixel*, $y = Y_{scan\ line}$, and solve for z. Since the equation of a plane is linear, this computation involves solving one linear equation in one unknown.

The only real difficulty in the scan line algorithm is to determine where to switch polygons (see Figure 1). Mathematically our problem is: given the depths z,z^* of two polygons at the same point along a scan line, find the pixel at which the polygons intersect. We can compute depth incrementally along a scan line just as we did in the *z*-buffer algorithm by adding Δz. Let *N* be the

number of pixels from the *Current Pixel* to the pixel where the crossover occurs. At the pixel where the crossover occurs, the depths of the two polygons are equal. Therefore

$$z + N\Delta z = z^* + N\Delta z^*,$$

so

$$N = \frac{z^* - z}{\Delta z - \Delta z^*}.$$

The values z, z^* are known, since these value are the depths of the two polygons at the *Current Pixel*; the values $\Delta z, \Delta z^*$ are known because these values are stored in the edge table. The number N tells us how many more pixels we must visit before we switch to the next polygon. Note that at most one crossover can occur between any pair of polygons, so once we switch polygons we never switch back. Of course, if *Current Pixel* $+ N$ is greater than the x-coordinate of last pixel in the *Current Polygon*, then no crossover occurs.

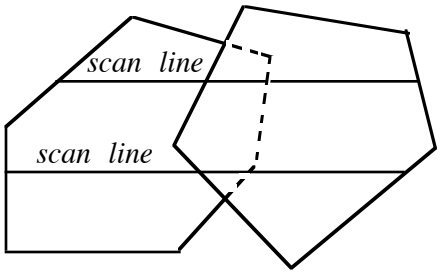

Figure 1: Crossover.

The main advantage of the scan line algorithm is speed. The scan line algorithm is faster than the z-buffer algorithm for two reasons: the scan line algorithm avoids overpainting pixels -- each pixel is visited and painted only once -- and the scan line algorithm takes advantage of pixel coherence -- adjacent pixels typically lie on the same polygon and accordingly can be painted with the same color and intensity. Another important advantage of the scan line algorithm is that the scan line algorithm is compatible with the algorithms for Gouraud and Phong shading. If we use the scan line algorithm, then hidden surfaces and shading can both be computed in the same pass, since both algorithms proceed scan line by scan line. The main disadvantage of the scan line algorithm is that this algorithm requires more complicated data structures than the z-buffer algorithm and the algorithm itself is a good deal more difficult to implement.

5. Ray Casting

The ray casting algorithm for hidden surfaces employs no special data structures. Instead, as in recursive ray tracing, a ray is fired from the eye through each pixel on the screen in order to locate the polygon in the scene closest to the eye. The color and intensity of this polygon is displayed at the pixel.

Ray Casting Algorithm

Through each pixel, fire a ray from the eye:

Intersect the ray with the plane of each polygon.

Reject intersections that lie outside the polygon.

Accept the closest remaining intersection -- that is, the intersection with the smallest value of the parameter along the line.

Ray casting is easy to implement for polygonal models because the only calculation required is the intersection of a line with a plane. Let L be the line determined by the eye point E and a pixel F, and let S be a plane determined by a unit normal N and a point Q. Then the parametric equation of the line is $L(t) = E + tv$, where $v = F - E$, and the implicit equation of the plane is $N \bullet (P - Q) = 0$. Thus the line intersects the plane when

$$N \bullet (L(t) - Q) = N \bullet (E + tv - Q) = 0.$$

Solving for t yields

$$t = \frac{N \bullet (Q - E)}{N \bullet v} = \frac{N \bullet (Q - E)}{N \bullet (F - E)}. \qquad (3.1)$$

The actual intersection point can be found by substituting this value of t into the parametric equation of the line. For polygonal models, we still need to determine if this intersection point actually lies inside the polygon. For convex polygons such tests are provided in Lecture 19, Section 3; for arbitrary polygons, inside-outside tests are discussed in Lecture 11, Section 7.

Ray casting can be applied either before or after pseudoperspective. The only difference is that after pseudoperspective the eye is mapped to infinity, so the rays are fired perpendicular to the plane of the pixels rather than through the eye. Thus in Equation (3.1) v is the normal to the pixel plane; the remainder of the algorithm is unchanged. Employed after pseudoperspective, ray casting is an image space algorithm; invoked before pseudoperspective, ray casting is an object space algorithm.

The main advantage of the ray casting algorithm for hidden surfaces is that ray casting can be used even with non-planar surfaces. All that is needed to implement the ray casting algorithm for hidden surfaces is a line/surface intersection algorithm for each distinct surface type in the model.

The main disadvantage of ray casting is that the method is slow. Ray casting is a brute force technique that makes no use of pixel coherence. For adjacent pixels it is quite likely that the same polygon is closest to the eye, but ray casting makes no attempt to test for this pixel coherence. Instead ray casting employs dense sampling, so the ray casting algorithm for hidden surfaces is typically slower than other methods that employ more sophisticated data structures.

6. Depth Sort

Depth sort is another variation on the heedless painter's algorithm. The data structure for depth sort is a list of polygons sorted by increasing depth. Two polygons that overlap in depth are said to *conflict*. Conflicts are resolved by a sorting algorithm which is done in object space, so depth sort is an object space algorithm. After all the conflicts are resolved, the polygons are painted in order from back to front.

Depth Sort Algorithm

Sort the polygons by the furthest vertex from the screen:

For each polygon, find the minimum and maximum values of z (depth) at the vertices.

Sort the polygons in order of decreasing maximum z-coordinates.

Resolve z-overlaps (see below).

Paint the polygons in order from farthest to nearest.

The main difficulty in implementing depth sort is resolving z-overlaps. Notice that without this step, the depth sort algorithm would overpaint near polygons by far polygons just like the heedless painter's algorithm.

Below is a five step procedure for resolving conflicts (z-overlaps). Here the terms *z-extents* refers to the minimum and maximum values of the z-coordinates; *x-extents* and *y-extents* are defined in an analogous manner.

Resolving Conflicts

For every pair of polygons P and Q:

If the z-extents of P and Q overlap, then perform the following five tests:

i. Do the x–extents fail to overlap?

ii. Do the y–extents fail to overlap?

iii. Does every vertex of P lie on the far side of Q?

iv. Does every vertex of Q lie on near side of P?

v. Do the xy-projections of P and Q fail to overlap?

If any test succeeds, P does not obscure Q, so draw P before Q.

Otherwise we cannot resolve the conflict, so split the polygon P by the plane of the polygon Q (see below).

The five tests are ordered by difficulty from easy to hard. The x-extents and y-extents of a polygon are easy to determine by examining the x and y coordinates of all the vertices of the polygon. If the x-extents or y-extents of P and Q fail to overlap, then it does not matter which

polygon is painted first, since the projections of P and Q on the screen will not overlap (see Figure 2).

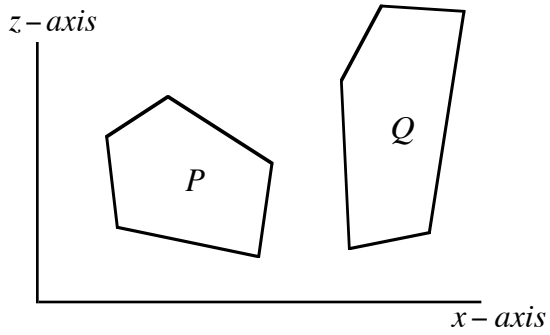

Figure 2: The z-extents of P and Q overlap, but the x-extents do not overlap. Therefore the projections of P and Q into the xy-plane will not overlap.

If every vertex of P lies on the far side of the screen from Q or if every vertex of Q lies on the near side of the screen from P, then P does not obscure Q so we can safely paint P before Q (see Figures 3,4). We can determine if P lies on the far side of Q by computing one dot product for every vertex of P. Fix a vertex V_Q of Q, and let N_Q be the outward pointing normal -- the normal pointing to the far side of the screen, the side of the screen away from the eye -- of the plane containing the polygon Q. Then P lies on the far side of Q if and only if for every vertex V_P of P, $(V_P - V_Q) \bullet N_Q > 0$ (see Figure 3). Similarly, fix a vertex V_P of P, and let N_P be the outward pointing normal of the plane containing the polygon P. Then Q lies on the near side of P if and only if for every vertex V_Q of Q, $(V_P - V_Q) \bullet N_P > 0$ (see Figure 4). Notice in Figure 4 that Q lies on the near side of P, even though P does not lie on the far side of Q, so the third and fourth tests are independent. If tests *iii* and *iv* fail, we can reverse the roles of P and Q and retest.

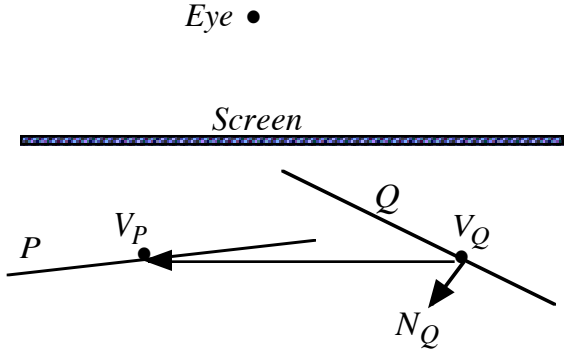

Figure 3: P lies on the far side of Q: $(V_P - V_Q) \bullet N_Q > 0$.

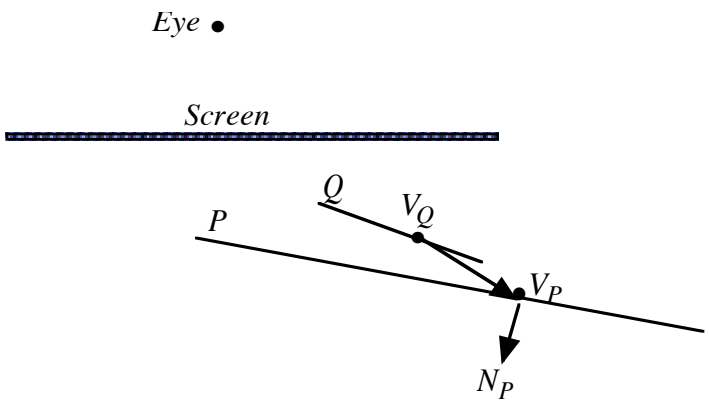

Figure 4: Q lies on the near side of P: $(V_P - V_Q) \bullet N_P > 0$. Notice that P does not lie on the far side of Q.

If the first four tests fail, then we can apply inside-outside tests to determine if the projections of P and Q on the screen overlap. If any vertex in the projection of P lies inside the projection of Q or if any vertex of Q lies inside the projection of P, then the projections of P and Q on the screen overlap. For convex polygons inside-outside tests are provided in Lecture 19, Section 3; for arbitrary polygons, inside-outside tests are discussed in Lecture 11, Section 7.

If the projections of P and Q do not overlap, then once again it does not matter which polygon is painted first. However, if the projections of P and Q do overlap, then the conflict cannot be resolved, and we must split one of the polygons, say P, by intersecting the polygon P with the plane of the polygon Q. Since splitting will be used again for bsp-trees, we present the details of this splitting procedure below in a separate subsection.

This method of resolving conflicts works for conflicts between pairs of polygons, but not all conflicts can be resolved in this manner because not all conflicts are localized to pairs of polygons. There can be cyclic conflicts where P obscures Q, Q obscures R, and R obscures P, even though there is no conflict between any pair of polygons (see Figure 5). These special situations can only be avoided by marking and splitting the troublesome polygons.

The main advantages of the depth sort algorithm for hidden surfaces is that the data structure is simple -- just an ordered list of polygons -- and the rendering algorithm is straightforward -- just paint the polygons in the order in which they appear in the list.

The main disadvantage of depth sort is that conflicts may be difficult to resolve. In the worst case if all five tests fail, then we need to resort to a splitting algorithm. For arbitrary nonconvex polygons, splitting can be a bit tricky. Moreover, finding and resolving cyclic conflicts may also be difficult to do.

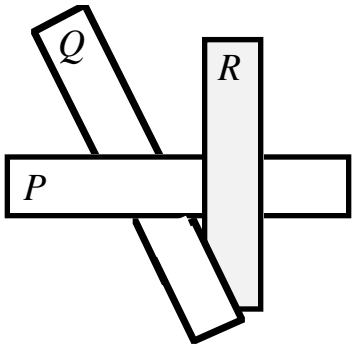

Figure 5: A cyclic conflict: P obscures Q, Q obscures R, R obscures P,

6.1 Polygon Splitting. To split a polygon that lies on both sides of a fixed plane, we must first find the line where the given plane intersects the plane of the polygon. We then intersect this line with the edges of the polygon to split the polygon into polygonal segments that lie on either side of the splitting plane.

Consider then two non-parallel planes

$$N_1 \bullet (P - Q_1) = 0$$
$$N_2 \bullet (P - Q_2) = 0.$$

These planes intersect in a line L determined by a point Q and a direction vector N. By Lecture 11, Section 5.3, we can choose

$$N = N_1 \times N_2$$
$$Q = \frac{(N_1 \bullet Q_1)\big((N_2 \bullet N_2)N_1 - (N_1 \bullet N_2)N_2\big) + (N_2 \bullet Q_2)\big((N_1 \bullet N_1)N_2 - (N_1 \bullet N_2)N_1\big)}{|N_1 \times N_2|^2}.$$

In fact, it is easy to see that the vector N is perpendicular to both planes, since $N = N_1 \times N_2 \perp N_1, N_2$. Hence N is parallel to the line L. Similarly, it is straightforward to verify that the point Q satisfies the equations of both planes (see Exercise 1), and hence the point Q lies on the line L.

Once we have a point Q and a direction vector N for the line L, we can split the polygon by intersecting L with every edge of the polygon. Two coplanar lines

$$L_1(s) = P + su$$
$$L_2(t) = Q + tv$$

intersect when $L_1(s) = L_2(t)$ -- that is, when

$$P + su = Q + tv,$$

or equivalently when

$$su - tv = Q - P.$$

Dotting both sides first with u and then with v yields two linear equations in two unknowns:

$$s(u \bullet u) - t(v \bullet u) = (Q - P) \bullet u$$
$$s(u \bullet v) - t(v \bullet v) = (Q - P) \bullet v .$$

Solving for s,t gives the parameters of the intersections points, and substituting s into the expression for L_1 or t into the expression for L_2 gives the coordinates of the intersection point. (For further details, see Lecture 11, Section 7.)

Since edges are bounded line segments, if the polygon is convex, the line L actually intersects only two edges of the polygon. If $L_1(s)$ represents the line corresponding to an edge P_iP_{i+1} of the polygon and $u = P_{i+1} - P_i$, then $L_2(t)$ intersects the edge P_iP_{i+1} if and only if at the intersection point of the two lines $0 \le s \le 1$. Thus it is easy to detect which edges of the polygon are intersected by L and to split the polygon accordingly (see Figure 6).

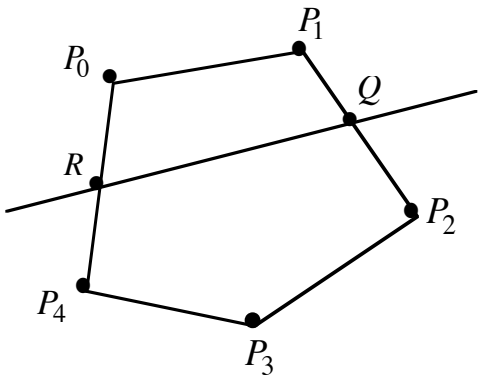

Figure 6: A polygon $P_0P_1P_2P_3P_4$ split by a straight line into two polygons P_0P_1QR and $QP_2P_3P_4R$.

7. BSP-Tree

A *binary space partitioning tree* (*bsp–tree*) is a binary tree whose nodes contain polygons. For each node in a bsp-tree all the polygons in the left subtree lie behind the polygon at the node, while all the polygons in the right subtree lie in front of the polygon at the node. Each polygon has a fixed normal vector, and front and back are measured relative to this fixed normal. Once a bsp-tree is constructed for a scene, the polygons are rendered by an in order traversal of the bsp-tree. Recursive algorithms for generating a bsp-tree and then using the bsp-tree to render a scene are presented below.

Algorithm for Generating a BSP–Tree

Select any polygon (plane) in the scene for the root.

Partition all the other polygons in the scene to the back (left subtree) or the front (right subtree).

Split any polygons lying on both sides of the root (see Section 6.1).

Build the left and right subtrees recursively.

BSP-Tree Rendering Algorithm (In Order Tree Traversal)

If the eye is in front of the root, then

Display the left subtree (behind)

Display the root

Display the right subtree (front)

If eye is in back of the root, then

Display the right subtree (front)

Display the root

Display the left subtree (back)

To generate a bsp-tree, we must be able to determine on which side of a plane a polygon lies and to split polygons that lie of both sides of a plane. Consider a plane defined by a point Q and a normal vector N. For any point P there are three possibilities:

i. P lies in front of the plane $\Leftrightarrow$ $N \bullet (P - Q) > 0$.

ii. P lies on the plane $\Leftrightarrow$ $N \bullet (P - Q) = 0$.

iii. P lies behind the plane $\Leftrightarrow$ $N \bullet (P - Q) < 0$.

If all the vertices of a polygon lie in front of a plane, then the entire polygon lies in front of the plane; if all the vertices of a polygon lie behind a plane, then the entire polygon lies behind the plane. Otherwise part of the polygon lies in front and another part lies behind the plane. In this case, we split the polygon by the plane, using the procedure presented in Section 6.1.

The main advantage of the bsp-tree algorithm for hidden surfaces is that we can use the same bsp–tree for different positions of the eye. Thus when we want to move around in a scene the bsp-tree is the preferred approach to hidden surfaces. The main disadvantage of bsp-trees is the work involved in splitting polygons.

8. Summary

We have discussed five different hidden surface algorithms: z-buffer, scan line, ray casting, depth sort, and bsp-tree. Table 1 contains a summary of the main features of each of these algorithms. Two key ideas are applied to help increase the speed of these algorithms: sorting of

edges by depth, and pixel coherence for depth and intensity. We can speed up some of these algorithms still further by storing bounding boxes for each polygonal face and avoiding interference tests when the bounding boxes do not overlap.

Algorithms for finding hidden surfaces and procedures for producing shadows use essentially the same computations: hidden surface algorithms find surfaces invisible to the eye, shadow procedures find surfaces invisible to a light source. Thus we can use the same algorithms for computing hidden surfaces and for calculating shadows. Moreover, we can often save time by reusing calculations. We can move the eye point and reuse shadow calculations; similarly, we can move light source and reuse hidden surface computations.

Algorithm	*Type*	*Sorting*	*Coherence*
Z-Buffer	*Image Space*	*None*	*Depth Calculations*
Scan Line	*Image Space*	*Edges in the Active Edge List*	*Intensity Calculations*
Ray Casting	*Image/Object Space*	*None*	*None*
Depth Sort	*Object Space*	*Polygons*	*Intensity Calculations*
BSP-Tree	*Object Space*	*Polygons in Tree*	*Intensity Calculations*

Table 1: Properties of hidden surface algorithms.

Exercise:

1. Consider two planes
$$N_1 \bullet (P - Q_1) = 0$$
$$N_2 \bullet (P - Q_2) = 0.$$
   a. Show that if
$$Q = \frac{(N_1 \bullet Q_1)\big((N_2 \bullet N_2)N_1 - (N_1 \bullet N_2)N_2\big) + (N_2 \bullet Q_2)\big((N_1 \bullet N_1)N_2 - (N_1 \bullet N_2)N_1\big)}{|N_1 \times N_2|^2}.$$
   then
$$N_1 \bullet (Q - Q_1) = 0$$
$$N_2 \bullet (Q - Q_2) = 0.$$
   b. Conclude that the point Q lies on the line representing the intersection of these two planes.

Programming Projects:

1. Implement the five hidden surface algorithms discussed in this lecture in your favorite programming language using your favorite API.
   a. Use your implementations to render several large polygonal models.
   b. Compare the relative speeds of the five algorithms for each of your models.

2. Integrate the scan line hidden surface algorithm with Gouraud or Phong shading to simultaneously remove hidden surfaces and eliminate Mach bands.

3. Implement the bsp-tree hidden surface algorithm and use this procedure to display the same models from several different points of view.