```
global_settings{radiosity{}}
```

at the top to enable radiosity computation with default settings. The difference is significant, is it not?

Figure 19.24(a) is the output without radiosity (or see it separately as the image file `sphereInBoxPOV.jpg` in our `Code` folder), while Figure 19.24(b) is the output with radiosity (`sphereInBoxPOVWithRadiosity.jpg` in `Code`). There clearly is much more light going around inside the box in the latter rendering. End

Exercise 19.12. If the lighting in a scene changes, then which steps of the radiosity algorithm need to be redone? How about if the geometry changes, e.g., with a ball looping in and out of a torus?

19.4 BSP Trees

The BSP (Binary Space Partitioning) tree algorithm for hidden surface removal, invented by Fuchs, Kedem and Naylor [47], is an implementation of the so-called *painter's method*. The painter's method draws objects from back to front toward the eye, each new object possibly drawn over earlier ones, which automatically engenders hidden surface removal. See Figure 19.25.

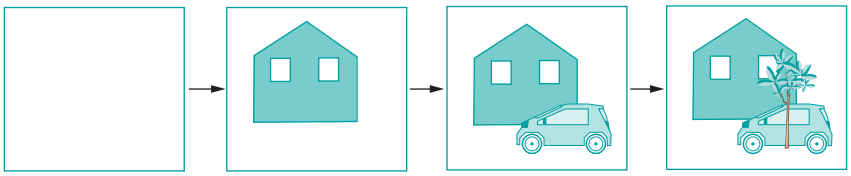

Figure 19.25: The painter's method draws the scene "toward" the eye: first the house, then the car and, finally, the tree.

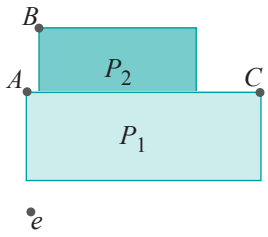

Figure 19.26: Vertex A of P_1 is closer to eye e than vertex B of P_2, while B is closer than C of P_1.

How does one implement the painter's method in a scene specified as a set of polygons? One would obviously like to sort the polygons relative to the eye e, so that those farther away appear earlier in the sorted order than nearer ones, and then draw them in this order. The first problem that one runs into along this line of thinking is determining which of two given polygons, say P_1 and P_2, is "farther from" e. Measuring the distance from e to vertices of P_1 and P_2 is not a workable approach, as there may be one vertex of P_1 that is closer to e than some vertex of P_2, while another vertex of P_1 is farther from e than one of P_2, e.g., as in Figure 19.26.

The figure itself suggests a possible solution. If one imagines P_1 and P_2 as vertical rectangles lying on parallel planes p_1 and p_2, respectively, then evidently P_2 lies *behind* p_1 in that P_2 lies on the side of p_1 opposite

the eye (i.e., in the half-space of p_1 not containing the eye). Clearly, then, P_2 should be taken to be farther than P_1 and drawn before P_1, because it cannot obscure P_1. Unfortunately, for arbitrary polygons P_1 and P_2, it may not be the case that the plane of either separates the other from the eye, as, e.g., in Figure 19.27, where the plane of each intersects the other.

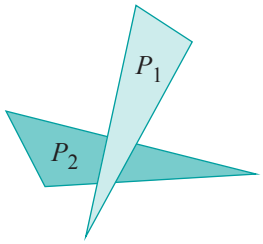

Figure 19.27: Each of P_1 and P_2 intersects the plane containing the other – the planes themselves are not drawn.

Suppose, for the moment, that a scene comprising a set S of polygons does, in fact, contain at least one, say P, such that every polygon of S, other than P, lies entirely on one side or the other of the plane p containing P (i.e., each polygon of S, other than P, lies entirely in one or other half-space of p). In this case, we say that P *splits* S. Suppose, as well, that the eye e does not lie on p but to one side as well. See Figure 19.28, where the polygon P splits the set S into two, S' and S'', and the eye e lies on the same side of p as S''. Clearly, then, a legitimate drawing order has the polygons of S' first (drawn in some appropriate order among themselves), then P and, finally, the polygons of S'' (again in some appropriate order).

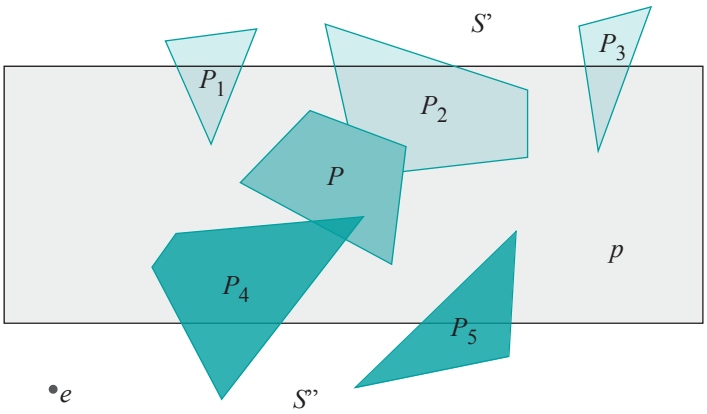

Figure 19.28: The polygon P splits the set of six polygons. P's plane p is drawn a light gray. The three to one side of p, viz., P_1, P_2 and P_3, form the subset S', while the two, P_4 and P_5, on the other form S''. The eye e lies on the same side of p as S''.

If the equation of the plane p is $ax + by + cz + d = 0$, then the sign of the quantity $ax + by + cz + d$, positive or negative, determines the side of p that a point $[x\ y\ z]^T$ lies. For the sake of compact notation we'll denote by h a function whose sign determines the side of p that a primitive lies (assuming, of course, that the primitive does lie wholly to one side or the other). For example, $h(e)$ and $h(P_1)$ have opposite signs in Figure 19.28.

Below, then, is a first cut at an algorithm to draw the polygons in S on the assumption that a splitting polygon P of S can indeed be found. Moreover, the recursive `draw()` calls are based on the premise that every time a subset of S is partitioned by a splitting polygon into two smaller subsets, a splitting polygon can *again* be found in either (provided the size of the subset is greater than 1; otherwise, of course, there is no need to

partition further). In other words, we suppose that we can keep partitioning by means of splitting polygons until we are down to singletons.

Drawing Algorithm Version 1

```
void draw(S, e) // Input S, a set of polygons, and e, eye location.
{
   if (|S| == 0) ; // Nothing to do.
   else if (|S| == 1)
   {
      P = the one member of S;
      draw(P);
   }
   else
   {
      P = a splitting polygon of S; // Can be found by assumption.
      S'  = polygons Q of S - P such that h(Q) > 0;
      S'' = polygons Q of S - P such that h(Q) < 0;
      if ( h(e) > 0 )
      {
         draw(S'', e);
         draw(P);
         draw(S', e):
      }
      else // h(e) < 0
      {
         draw(S', e);
         draw(P);
         draw(S'', e):
      }
   }
}
```

In the scenario of the preceding algorithm, the splitting polygons can be naturally arranged in the form of a binary tree, called a *BSP tree*: the children (if they exist) of each splitting polygon P being the splitting polygons of the two subsets that result from partitioning by P. A BSP tree node corresponding to a particular splitting polygon stores its data via a pointer to the polygon itself in the list of scene primitives, in addition to the obligatory left and right child pointers. Figure 19.29 shows an example of a set S of seven polygons lying on parallel planes and a corresponding BSP tree (only one pointer has been drawn from a tree node to a polygon).

Observe that the BSP tree of a given set of polygons need not be unique, but depends on the splitting polygon selected from each subset (in the situation of Figure 19.29 any member of a subset of S splits it and we have been intentionally arbitrary in making choices). Moreover, the BSP tree, once constructed, is a static data structure – provided the scene doesn't change – independent of the eye. It can, therefore, be pre-computed. It's

size is linear in the number of polygons.

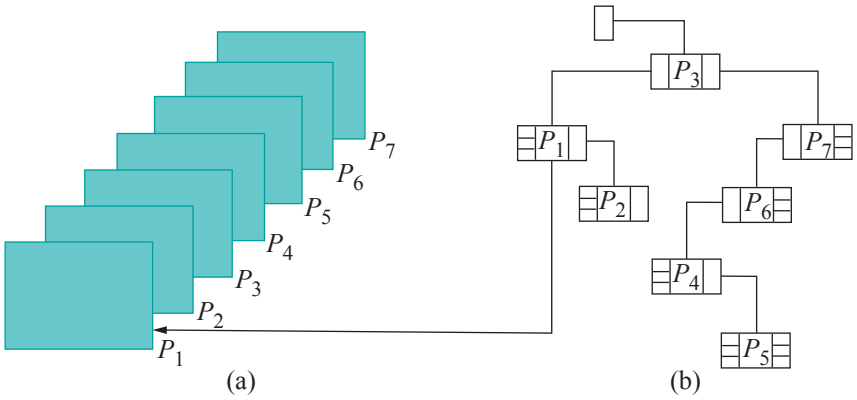

Figure 19.29: (a) Seven polygons on parallel planes (b) A corresponding BSP tree (only one polygon pointer drawn to avoid clutter).

Here is recursive pseudo-code to construct a BSP tree for an input set S of polygons, keeping the earlier assumption that splitting polygons can be repeatedly found:

BSP Tree Construction Algorithm Version 1

```
pointer makeBSPTree(S) // Input S, a set of polygons.
{
   allocate empty BSP tree node  root;

   if ( |S| == 1 )
   {
      P = the one member of S;
      root.value = P;
      root.left_child = NULL;
      root.right_child = NULL;
   }
   else
   {
      P = a splitting polygon in S; // Can be found by assumption.
      root.value = P;
      S'   = polygons Q of S - P such that h(Q) > 0;
      S''  = polygons Q of S - P such that h(Q) < 0;
      if ( |S'| == 0 ) root.left_child = NULL;
      else root.left_child = makeBSPTree(S');
      if ( |S''| == 0 ) root.right_child = NULL;
      else root.right_child = makeBSPTree(S'');
   }
```

```
    return pointer to root;
}
```

Returning to the question of drawing, we ask the reader in the next exercise to devise an algorithm to draw a scene by choosing splitting polygons given a BSP tree.

Exercise 19.13. Given input a pointer T to the root of a BSP tree constructed for the set S of polygons comprising a scene – assume that we were indeed able to split successive subsets until the tree was complete – and the location of the eye e, write a routine `draw(T, e)` to draw the scene by modifying Drawing Algorithm Version 1. For future reference, call your routine the BSP Tree Based Drawing Algorithm.

Exercise 19.14. What is the complexity of the BSP Tree Based Drawing Algorithm – does it depend on the height of the BSP tree or its number of nodes?
Hint: The drawing algorithm traverses the BSP tree, visiting each node once.

Exercise 19.15. In OpenGL, for instance, not only is the location of the camera given, but the direction in which it's pointed and its up direction as well (think `gluLookAt()`). Where would these come into play in a drawing routine which uses a BSP tree?
Hint: They are needed at the time of drawing a polygon (e.g., `draw(P)` in Version 1 of the drawing algorithm) and *not* when traversing the BSP tree. The reason is that if, say, the plane of a polygon P_1 separates the eye e from another polygon P_2, then P_2 cannot obscure P_1 no matter how a camera at e is pointed or its top aligned, implying that P_2 can be drawn ahead of P_1 regardless of these parameters.

We come now to the final piece of the puzzle. How do we assure the existence of a splitting polygon in each subset as we successively partition them? *Answer*: We cannot. Instead, we'll resort to some amount of violence, if need be, to members of the subset in order to be able to extract a splitting polygon. The idea is actually simple. Given a set S of polygons, choose one (arbitrarily), say P. If another polygon Q of S intersects the plane p of P, then subdivide Q into subpolygons that do not intersect p (mind that touching is not considered intersecting in the realm of BSP trees). We'll call this the process of *subdivision of Q by P*.

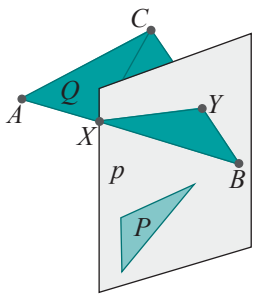

Figure 19.30: Subdividing Q by P.

For example, $Q = ABC$ in Figure 19.30, which intersects the plane p of P in the straight segment XY, can be subdivided into the quadrilateral $AXYC$ and the triangle YXB. The subdivision of Q by P need not be unique as long as the outcome is a set of polygons that don't intersect p. For example, if drawing primitives are triangles, as in OpenGL, the subdivision of Q in the figure into the three triangles AXC, CXY and YXB might be preferable. If Q does not intersect p, then its subdivision by P is, of course, a null process.

Accordingly, if every polygon of the set S, other than P, is subdivided by P, then the result is a *refinement* of S, in that each polygon in this collection is a subpolygon of some polygon of S. Evidently, from the way it was made, this refinement is split by P. From the point of view of drawing the originally specified scene, there is no loss in refining, because the union of the polygons in the refinement is exactly the same as that of those in the original set. It's just that the number of polygons has increased. Care must be taken, however, that the new polygons are consistent with the ones from which they are derived with respect to orientation, normal direction and so forth.

We obtain now our final version of the BSP tree construction algorithm, which makes no assumption on the polygon set S, by a simple modification of the first version to include refinement.

BSP Tree Construction Algorithm Version 2

```
pointer makeBSPTree(S) // Input S, a set of polygons.
{
   allocate empty BSP tree node  root;

   if ( |S| == 1 )
   {
      P = the one member of S;
      root.value = P;
      root.left_child = NULL;
      root.right_child = NULL;
   }
   else
   {
      P = a random polygon in S;
      root.value = P;
      S    = refinement of S obtained by subdividing by P each
             polygon of S other than P;
      S'   = polygons Q of S - P such that h(Q) > 0;
      S''  = polygons Q of S - P such that h(Q) < 0;
      if ( |S'| == 0 ) root.left_child = NULL;
      else root.left_child = makeBSPTree(S');
      if ( |S''| == 0 ) root.right_child = NULL;
      else root.right_child = makeBSPTree(S'');
   }

   return pointer to root;
}
```

This algorithm, together with the BSP Tree Based Drawing Algorithm, answer to Exercise 19.13, completes a routine for drawing with hidden surface removal.

Exercise 19.16. Construct a BSP tree for the "cycle" of three triangles

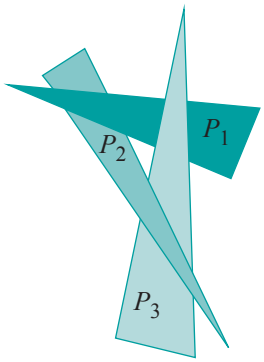

Figure 19.31: A "cycle" of triangles.

disposed as in Figure 19.31, each intersecting the plane of the other two, subdividing as necessary.

Remark 19.8. An implementation of BSP trees would have to take into account additional practical issues that we have not discussed, e.g., what to do if the eye e lies *on* the plane of a polygon in the tree (we have been tacitly assuming that it always lies to a side) and how to prevent *slivers* from arising when subdividing a polygon (e.g., if B is very close to p in Figure 19.30, though still on the side opposite to that of A and C, then the triangle YXB is undesirably long and narrow).

Remark 19.9. The complexity of the BSP tree returned by the second version of the construction algorithm depends on the choice of P used to subdivide the other members of S. However, the complexity of determining an optimal sequence of subdividing polygons is generally not worth the improvement in the tree itself. A random choice of the subdividing polygon seems to work best in practice.

Remark 19.10. BSP trees were used extensively for visibility determination, particularly in flight simulators, in the 60's and 70's (flight simulators were the killer app for high-end real-time rendering systems at that time). However, through the 1980's, BSP trees continued to be superseded by the simple-minded z-buffer by virtue of the latter's sheer speed, derived from being implemented on dedicated hardware (the first graphics cards were released in 1981).

However, the space-partitioning technique that BSP trees implement has numerous other applications: the frustum culling technique we studied in Chapter 6 is an example.

19.5 Summary, Notes and More Reading

In this chapter we went into particularly gory detail about the synthetic-camera pipeline that OpenGL implements, the fixed-functionality variant in particular. The reader should now be in a position to even implement a barebones version of her own. The synthetic-camera pipeline is based on a local illumination model. We were introduced as well to two global models, that of ray tracing and radiosity, and saw how much more realism they afford than the synthetic camera, though at hugely more computation cost. We also learned about BSP trees, a classical technique which integrates hidden surface removal into the rendering process.

The seminal work on ray tracing was by Appel [4] and Whitted [143], and on radiosity by Goral [54]. The book by Jim Blinn [16], a CG pioneer, has several insightful articles, written in his particularly entertaining style, on various pipeline-related topics. Segal-Akeley [123] is a must-read high-level overview of the OpenGL pipeline written by two members of the original