- Triangle. Three vectors per triangle, for $9n_t$ units of storage;
- IndexedMesh. One vector per vertex and three ints per triangle, for $3n_v + 3n_t$ units of storage.

The relative storage requirements depend on the ratio of n_t to n_v.

As a rule of thumb, a large mesh has each vertex connected to about six triangles (although there can be any number for extreme cases). Since each triangle connects to three vertices, this means that there are generally twice as many triangles as vertices in a large mesh: $n_t \approx 2n_v$. Making this substitution, we can conclude that the storage requirements are $18n_v$ for the `Triangle` structure and $9n_v$ for `IndexedMesh`. Using shared vertices reduces storage requirements by about a factor of two; and this seems to hold in practice for most implementations.

Is this factor of two worth the complication? I think the answer is yes, and it becomes an even bigger win as soon as you start adding "properties" to the vertices.

12.1.3 Triangle Strips and Fans

Indexed meshes are the most common in-memory representation of triangle meshes, because they achieve a good balance of simplicity, convenience, and compactness. They are also commonly used to transfer meshes over networks and between the application and graphics pipeline. In applications where even more compactness is desirable, the triangle vertex indices (which take up two-thirds of the space in an indexed mesh with only positions at the vertices) can be expressed more efficiently using *triangle strips* and *triangle fans*.

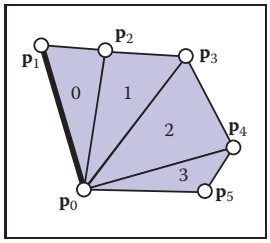


Figure 12.9. A triangle fan.

A triangle fan is shown in Figure 12.9. In an indexed mesh, the triangles array would contain [(0, 1, 2), (0, 2, 3), (0, 3, 4), (0, 4, 5)]. We are storing 12 vertex indices, although there are only six distinct vertices. In a triangle fan, all the triangles share one common vertex, and the other vertices generate a set of triangles like the vanes of a collapsible fan. The fan in the figure could be specified with the sequence [0, 1, 2, 3, 4, 5]: the first vertex establishes the center, and subsequently each pair of adjacent vertices (1-2, 2-3, etc.) creates a triangle.

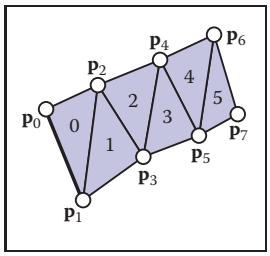


Figure 12.10. A triangle strip.

The triangle strip is a similar concept, but it is useful for a wider range of meshes. Here, vertices are added alternating top and bottom in a linear strip as shown in Figure 12.10. The triangle strip in the figure could be specified by the sequence [0 1 2 3 4 5 6 7], and every subsequence of three adjacent vertices (0-1-2, 1-2-3, etc.) creates a triangle. For consistent orientation, every other triangle needs to have its order reversed. In the example, this results in the triangles (0, 1, 2), (2, 1, 3), (2, 3, 4), (4, 3, 5), etc. For each new vertex that comes in, the oldest vertex is forgotten and the order of the two remaining vertices is swapped. See Figure 12.11 for a larger example.

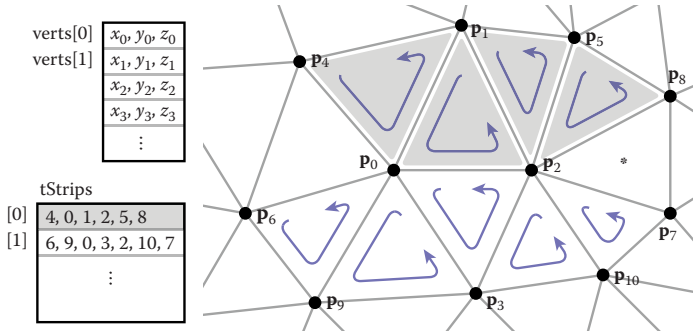


Figure 12.11. Two triangle strips in the context of a larger mesh. Note that neither strip can be extended to include the triangle marked with an asterisk.

In both strips and fans, $n + 2$ vertices suffice to describe n triangles—a substantial savings over the $3n$ vertices required by a standard indexed mesh. Long triangle strips will save approximately a factor of three if the program is vertex-bound.

It might seem that triangle strips are only useful if the strips are very long, but even relatively short strips already gain most of the benefits. The savings in storage space (for only the vertex indices) are as follows:

strip length	1	2	3	4	5	6	7	8	16	100	∞
relative size	1.00	0.67	0.56	0.50	0.47	0.44	0.43	0.42	0.38	0.34	0.33

So, in fact, there is a rather rapid diminishing return as the strips grow longer. Thus, even for an unstructured mesh, it is worthwhile to use some greedy algorithm to gather them into short strips.

12.1.4 Data Structures for Mesh Connectivity

Indexed meshes, strips, and fans are all good, compact representations for static meshes. However, they do not readily allow for meshes to be modified. In order to efficiently edit meshes, more complicated data structures are needed to efficiently answer queries such as:

- Given a triangle, what are the three adjacent triangles?
- Given an edge, which two triangles share it?

- Given a vertex, which faces share it?
- Given a vertex, which edges share it?

There are many data structures for triangle meshes, polygonal meshes, and polygonal meshes with holes (see the notes at the end of the chapter for references). In many applications the meshes are very large, so an efficient representation can be crucial.

The most straightforward, though bloated, implementation would be to have three types, `Vertex`, `Edge`, and `Triangle`, and to just store all the relationships directly:

```
Triangle {
  Vertex v[3]
  Edge e[3]
}

Edge {
  Vertex v[2]
  Triangle t[2]
}

Vertex {
  Triangle t[]
  Edge e[]
}
```

This lets us directly look up answers to the connectivity questions above, but because this information is all inter-related, it stores more than is really needed. Also, storing connectivity in vertices makes for variable-length data structures (since vertices can have arbitrary numbers of neighbors), which are generally less efficient to implement. Rather than committing to store all these relationships explicitly, it is best to define a class interface to answer these questions, behind which a more efficient data structure can hide. It turns out we can store only some of the connectivity and efficiently recover the other information when needed.

The fixed-size arrays in the `Edge` and `Triangle` classes suggest that it will be more efficient to store the connectivity information there. In fact, for polygon meshes, in which polygons have arbitrary numbers of edges and vertices, only edges have fixed-size connectivity information, which leads to many traditional mesh data structures being based on edges. But for triangle-only meshes, storing connectivity in the (less numerous) faces is appealing.

A good mesh data structure should be reasonably compact and allow efficient answers to all adjacency queries. Efficient means constant-time: the time to find neighbors should not depend on the size of the mesh. We'll look at three data structures for meshes, one based on triangles and two based on edges.

The Triangle-Neighbor Structure

We can create a compact mesh data structure based on triangles by augmenting the basic shared-vertex mesh with pointers from the triangles to the three neighboring triangles, and a pointer from each vertex to one of the adjacent triangles (it doesn't matter which one); see Figure 12.12:

```
Triangle {
   Triangle nbr[3];
   Vertex v[3];
}

Vertex {
   // ... per-vertex data ...
   Triangle t; // any adjacent tri
}
```

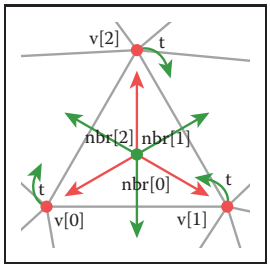


Figure 12.12. The references between triangles and vertices in the triangle-neighbor structure.

In the array `Triangle.nbr`, the kth entry points to the neighboring triangle that shares vertices k and $k+1$. We call this structure the *triangle-neighbor structure*. Starting from standard indexed mesh arrays, it can be implemented with two additional arrays: one that stores the three neighbors of each triangle, and one that stores a single neighboring triangle for each vertex (see Figure 12.13 for an example):

```
Mesh {
   // ... per-vertex data ...
   int tInd[nt][3]; // vertex indices
   int tNbr[nt][3]; // indices of neighbor triangles
   int vTri[nv]; // index of any adjacent triangle
}
```

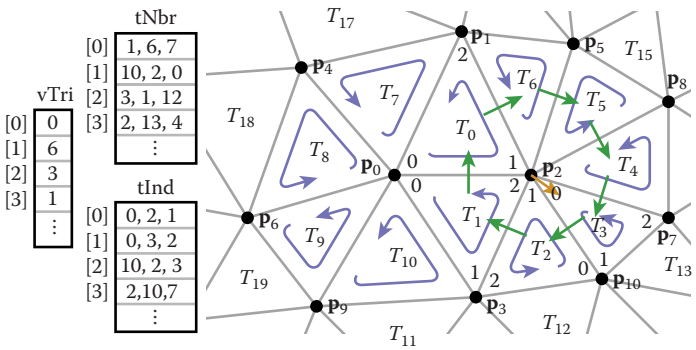


Figure 12.13. The triangle-neighbor structure as encoded in arrays, and the sequence that is followed in traversing the neighboring triangles of vertex 2.

Clearly the neighboring triangles and vertices of a triangle can be found directly in the data structure, but by using this triangle adjacency information carefully it is also possible to answer connectivity queries about vertices in constant time. The idea is to move from triangle to triangle, visiting only the triangles adjacent to the relevant vertex. If triangle t has vertex v as its kth vertex, then the triangle `t.nbr[`k`]` is the next triangle around v in the clockwise direction. This observation leads to the following algorithm to traverse all the triangles adjacent to a given vertex:

Of course, a real program would *do* something with the triangles as it found them.

```
TrianglesOfVertex(v) {
   t = v.t
   do {
      find i such that (t.v[i] == v)
      t = t.nbr[i]
   } while (t != v.t)
}
```

This operation finds each subsequent triangle in constant time—even though a search is required to find the position of the central vertex in each triangle's vertex list, the vertex lists have constant size so the search takes constant time. However, that search is awkward and requires extra branching.

A small refinement can avoid these searches. The problem is that once we follow a pointer from one triangle to the next, we don't know from which way we came: we have to search the triangle's vertices to find the vertex that connects back to the previous triangle. To solve this, instead of storing pointers to neighboring triangles, we can store pointers to specific edges of those triangles by storing an index with the pointer:

```
Triangle {
   Edge nbr[3];
   Vertex v[3];
}

Edge { // the i-th edge of triangle t
   Triangle t;
   int i;  // in {0,1,2}
}

Vertex {
   // ... per-vertex data ...
   Edge e;  // any edge leaving vertex
}
```

In practice the `Edge` is stored by borrowing two bits of storage from the triangle index t to store the edge index i, so that the total storage requirements remain the same.

In this structure the neighbor array for a triangle tells *which* of the neighboring triangles' edges are shared with the three edges of that triangle. With this extra information, we always know where to find the original triangle, which leads to an invariant of the data structure: for any jth edge of any triangle t,

$$t\texttt{.nbr[}j\texttt{].t.nbr[}t\texttt{.nbr[}j\texttt{].i].t} == t.$$

Knowing which edge we came in through lets us know immediately which edge to leave through in order to continue traversing around a vertex, leading to a streamlined algorithm:

```
TrianglesOfVertex(v) {
   {t, i} = v.e;
   do {
      {t, i} = t.nbr[i];
      i = (i+1) mod 3;
   } while (t != v.e.t);
}
```

The triangle-neighbor structure is quite compact. For a mesh with only vertex positions, we are storing four numbers (three coordinates and an edge) per vertex and six (three vertex indices and three edges) per face, for a total of $4n_v + 6n_t \approx 16n_v$ units of storage per vertex, compared with $9n_v$ for the basic indexed mesh.

The triangle neighbor structure as presented here works only for manifold meshes, because it depends on returning to the starting triangle to terminate the traversal of a vertex's neighbors, which will not happen at a boundary vertex that doesn't have a full cycle of triangles. However, it is not difficult to generalize it to manifolds with boundary, by introducing a suitable sentinel value (such as -1) for the neighbors of boundary triangles and taking care that the boundary vertices point to the most counterclockwise neighboring triangle, rather than to any arbitrary triangle.

The Winged-Edge Structure

One widely used mesh data structure that stores connectivity information at the edges instead of the faces is the *winged-edge* data structure. This data structure makes edges the first-class citizen of the data structure, as illustrated in Figures 12.14 and 12.15.

In a winged-edge mesh, each edge stores pointers to the two vertices it connects (the *head* and *tail* vertices), the two faces it is part of (the *left* and *right* faces), and, most importantly, the next and previous edges in the counterclockwise traversal of its left and right faces (Figure 12.16). Each vertex and face also stores a pointer to a single, arbitrary edge that connects to it:

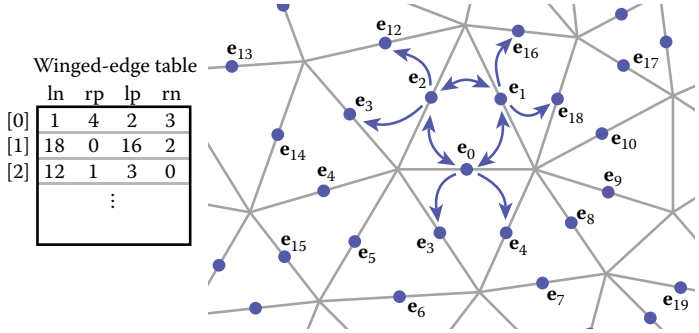
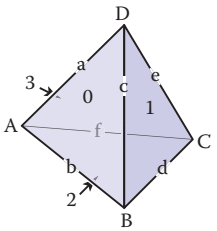


Figure 12.14. An example of a winged-edge mesh structure, stored in arrays.



Edge	*Vertex 1*	*Vertex 2*	*Face left*	*Face right*	*Pred left*	*Succ left*	*Pred right*	*Succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	e	b	d

Vertex	*Edge*
A	a
B	d
C	d
D	e

Face	*Edge*
0	a
1	c
2	d
3	a

Figure 12.15. A tetrahedron and the associated elements for a winged-edge data structure. The two small tables are not unique; each vertex and face stores any one of the edges with which it is associated.

```
Edge {
   Edge lprev, lnext, rprev, rnext;
   Vertex head, tail;
   Face left, right;
}

Face {
   // ... per-face data ...
   Edge e;  // any adjacent edge
}

Vertex {
   // ... per-vertex data ...
   Edge e;  // any incident edge
}
```

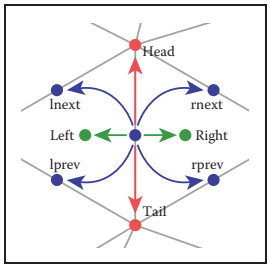


Figure 12.16. The references from an edge to the neighboring edges, faces, and vertices in the winged-edge structure.

The winged-edge data structure supports constant-time access to the edges of a face or of a vertex, and from those edges the adjoining vertices or faces can be found:

```
EdgesOfVertex(v) {
   e = v.e;
   do {
      if (e.tail == v)
         e = e.lprev;
      else
         e = e.rprev;
   } while (e != v.e);
}

EdgesOfFace(f) {
   e = f.e;
   do {
      if (e.left == f)
         e = e.lnext;
      else
         e = e.rnext;
   } while (e != f.e);
}
```

These same algorithms and data structures will work equally well in a polygon mesh that isn't limited to triangles; this is one important advantage of edge-based structures.

As with any data structure, the winged-edge data structure makes a variety of time/space tradeoffs. For example, we can eliminate the `prev` references. This makes it more difficult to traverse clockwise around faces or counterclockwise around vertices, but when we need to know the previous edge, we can always follow the successor edges in a circle until we get back to the original edge. This saves space, but it makes some operations slower. (See the chapter notes for more information on these tradeoffs).

The Half-Edge Structure

The winged-edge structure is quite elegant, but it has one remaining awkwardness—the need to constantly check which way the edge is oriented before moving to the next edge. This check is directly analogous to the search we saw in the basic version of the triangle neighbor structure: we are looking to find out whether we entered the present edge from the head or from the tail. The solution is also almost indistinguishable: rather than storing data for each edge, we store data for each *half-edge*. There is one half-edge for each of the two triangles that share an edge, and the two half-edges are oriented oppositely, each oriented consistently with its own triangle.

The data normally stored in an edge is split between the two half-edges. Each half-edge points to the face on its side of the edge and to the vertex at its head, and each contains the edge pointers for its face. It also points to its neighbor on the other side of the edge, from which the other half of the information can be found. Like the winged-edge, a half-edge can contain pointers to both the previous and next half-edges around its face, or only to the next half-edge. We'll show the example that uses a single pointer.

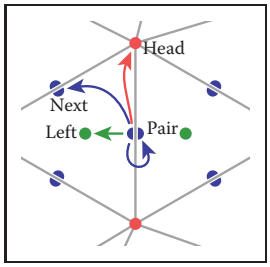


Figure 12.17. The references from a half-edge to its neighboring mesh components.

```
HEdge {
   HEdge pair, next;
   Vertex v;
   Face f;
}

Face {
   // ... per-face data ...
   HEdge h;  // any h-edge of this face
}

Vertex {
   // ... per-vertex data ...
   HEdge h;  // any h-edge pointing toward this vertex
}
```

Traversing a half-edge structure is just like traversing a winged-edge structure except that we no longer need to check orientation, and we follow the `pair` pointer to access the edges in the opposite face.

```
EdgesOfVertex(v) {
   h = v.h;
   do {
      h = h.pair.next;
   } while (h != v.h);
}
```

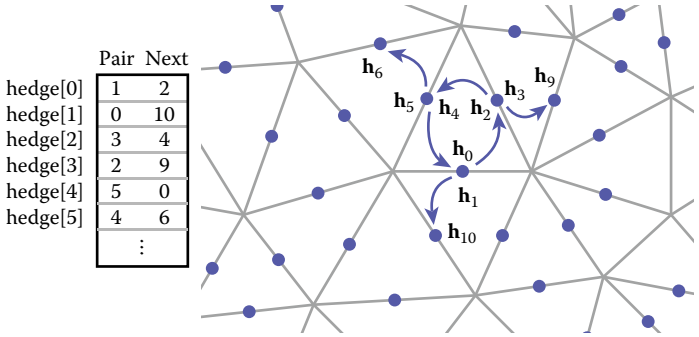


Figure 12.18. An example of a half-edge mesh structure, stored in arrays.

```
EdgesOfFace(f) {
   h = f.h;
   do {
      h = h.next;
   } while (h != f.h);
}
```

The vertex traversal here is clockwise, which is necessary because of omitting the `prev` pointer from the structure.

Because half-edges are generally allocated in pairs (at least in a mesh with no boundaries), many implementations can do away with the `pair` pointers. For instance, in an implementation based on array indexing (such as shown in Figure 12.18), the array can be arranged so that an even-numbered edge i always pairs with edge $i + 1$ and an odd-numbered edge j always pairs with edge $j - 1$.

In addition to the simple traversal algorithms shown in this chapter, all three of these mesh topology structures can support "mesh surgery" operations of various sorts, such as splitting or collapsing vertices, swapping edges, adding or removing triangles, etc.

12.2 Scene Graphs

A triangle mesh manages a collection of triangles that constitute an object in a scene, but another universal problem in graphics applications is arranging the objects in the desired positions. As we saw in Chapter 6, this is done using transformations, but complex scenes can contain a great many transformations and organizing them well makes the scene much easier to manipulate. Most scenes admit to a hierarchical organization, and the transformations can be managed according to this hierarchy using a *scene graph*.

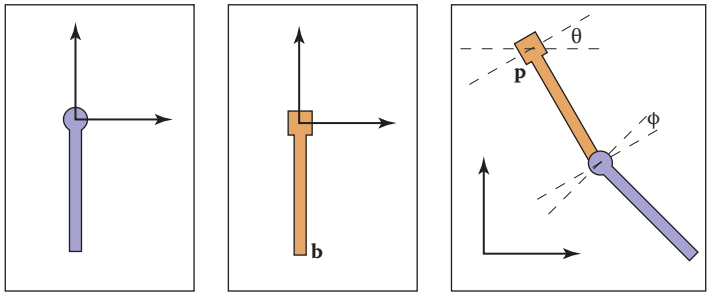


Figure 12.19. A hinged pendulum. On the left are the two pieces in their "local" coordinate systems. The hinge of the bottom piece is at point **b** and the attachment for the bottom piece is at its local origin. The degrees of freedom for the assembled object are the angles (θ,ϕ) and the location **p** of the top hinge.

To motivate the scene-graph data structure, we will use the hinged pendulum shown in Figure 12.19. Consider how we would draw the top part of the pendulum:

$$\mathbf{M}_1 = \text{rotate}(\theta)$$
$$\mathbf{M}_2 = \text{translate}(\mathbf{p})$$
$$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$$
Apply $\mathbf{M}_3$ to all points in upper pendulum

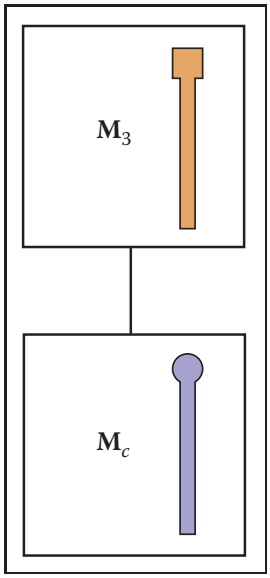


Figure 12.20. The scene graph for the hinged pendulum of Figure 12.19.

The bottom is more complicated, but we can take advantage of the fact that it is attached to the bottom of the upper pendulum at point $\mathbf{b}$ in the local coordinate system. First, we rotate the lower pendulum so that it is at an angle ϕ relative to its initial position. Then, we move it so that its top hinge is at point $\mathbf{b}$. Now it is at the appropriate position in the local coordinates of the upper pendulum, and it can then be moved along with that coordinate system. The composite transform for the lower pendulum is:

$$\mathbf{M}_a = \text{rotate}(\phi)$$
$$\mathbf{M}_b = \text{translate}(\mathbf{b})$$
$$\mathbf{M}_c = \mathbf{M}_b\mathbf{M}_a$$
$$\mathbf{M}_d = \mathbf{M}_3\mathbf{M}_c$$
Apply $\mathbf{M}_d$ to all points in lower pendulum

Thus, we see that the lower pendulum not only lives in its own local coordinate system, but also that coordinate system itself is moved along with that of the upper pendulum.

We can encode the pendulum in a data structure that makes management of these coordinate system issues easier, as shown in Figure 12.20. The appropriate matrix to apply to an object is just the product of all the matrices in the chain from

the object to the root of the data structure. For example, consider the model of a ferry that has a car that can move freely on the deck of the ferry, and wheels that each move relative to the car as shown in Figure 12.21.

As with the pendulum, each object should be transformed by the product of the matrices in the path from the root to the object:

- ferry transform using M_0;
- car body transform using M_0M_1;
- left wheel transform using $M_0M_1M_2$;
- left wheel transform using $M_0M_1M_3$.

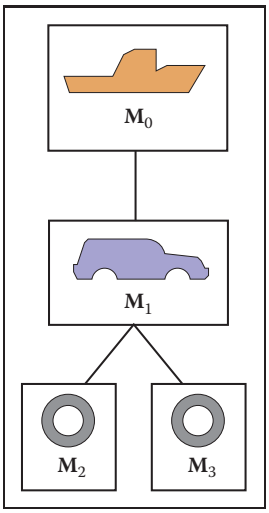


Figure 12.21. A ferry, a car on the ferry, and the wheels of the car (only two shown) are stored in a scene-graph.

An efficient implementation can be achieved using a *matrix stack*, a data structure supported by many APIs. A matrix stack is manipulated using *push* and *pop* operations that add and delete matrices from the right-hand side of a matrix product. For example, calling:

push($\mathbf{M}_0$)
push($\mathbf{M}_1$)
push($\mathbf{M}_2$)

creates the active matrix $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1\mathbf{M}_2$. A subsequent call to *pop()* strips the last matrix added so that the active matrix becomes $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1$. Combining the matrix stack with a recursive traversal of a scene graph gives us:

function traverse(node)
push($\mathbf{M}_{\text{local}}$)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()

There are many variations on scene graphs but all follow the basic idea above.

12.3 Spatial Data Structures

In many, if not all, graphics applications, the ability to quickly locate geometric objects in particular regions of space is important. Ray tracers need to find objects that intersect rays; interactive applications navigating an environment need to find the objects visible from any given viewpoint; games and physical simulations require detecting when and where objects collide. All these needs can be supported

by various *spatial data structures* designed to organize objects in space so they can be looked up efficiently.

In this section we will discuss examples of three general classes of spatial data structures. Structures that group objects together into a hierarchy are *object partitioning* schemes: objects are divided into disjoint groups, but the groups may end up overlapping in space. Structures that divide space into disjoint regions are *space partitioning* schemes: space is divided into separate partitions, but one object may have to intersect more than one partition. Space partitioning schemes can be regular, in which space is divided into uniformly shaped pieces, or irregular, in which space is divided adaptively into irregular pieces, with smaller pieces where there are more and smaller objects.

We will use ray tracing as the primary motivation while discussing these structures, though they can all also be used for view culling or collision detection. In Chapter 4, all objects were looped over while checking for intersections. For N objects, this is an $O(N)$ linear search and is thus slow for large scenes. Like most search problems, the ray-object intersection can be computed in sub-linear time using "divide and conquer" techniques, provided we can create an ordered data structure as a preprocess. There are many techniques to do this.

This section discusses three of these techniques in detail: bounding volume hierarchies (Rubin & Whitted, 1980; Whitted, 1980; Goldsmith & Salmon, 1987), uniform spatial subdivision (Cleary, Wyvill, Birtwistle, & Vatti, 1983; Fujimoto, Tanaka, & Iwata, 1986; Amanatides & Woo, 1987), and binary space partitioning (Glassner, 1984; Jansen, 1986; Havran, 2000). An example of the first two strategies is shown in Figure 12.22.

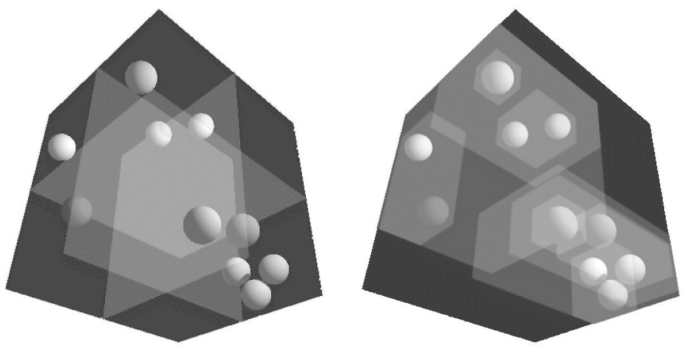


Figure 12.22. Left: a uniform partitioning of space. Right: adaptive bounding-box hierarchy. *Image courtesy David DeMarle.*

12.3.1 Bounding Boxes

A key operation in most intersection-acceleration schemes is computing the intersection of a ray with a bounding box (Figure 12.23). This differs from conventional intersection tests in that we do not need to know where the ray hits the box; we only need to know whether it hits the box.

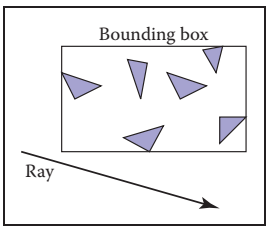


Figure 12.23. The ray is only tested for intersection with the surfaces if it hits the bounding box.

To build an algorithm for ray-box intersection, we begin by considering a 2D ray whose direction vector has positive x and y components. We can generalize this to arbitrary 3D rays later. The 2D bounding box is defined by two horizontal and two vertical lines:

$$x = x_{\text{min}},$$
$$x = x_{\text{max}},$$
$$y = y_{\text{min}},$$
$$y = y_{\text{max}}.$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}].$$

As shown in Figure 12.24, the intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\text{min}}$:

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{x_d}.$$

We then make similar computations for t_{xmax}, t_{ymin}, and t_{ymax}. The ray hits the box if and only if the intervals $[t_{\text{xmin}}, t_{\text{xmax}}]$ and $[t_{\text{ymin}}, t_{\text{ymax}}]$ overlap, i.e., their intersection is nonempty. In pseudocode this algorithm is:

$t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
$t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
$t_{\text{ymin}} = (y_{\text{min}} - y_e)/y_d$
$t_{\text{ymax}} = (y_{\text{max}} - y_e)/y_d$
if $(t_{\text{xmin}} > t_{\text{ymax}})$ or $(t_{\text{ymin}} > t_{\text{xmax}})$ **then**
  return false
else
  return true

The if statement may seem non-obvious. To see the logic of it, note that there is no overlap if the first interval is either entirely to the right or entirely to the left of the second interval.

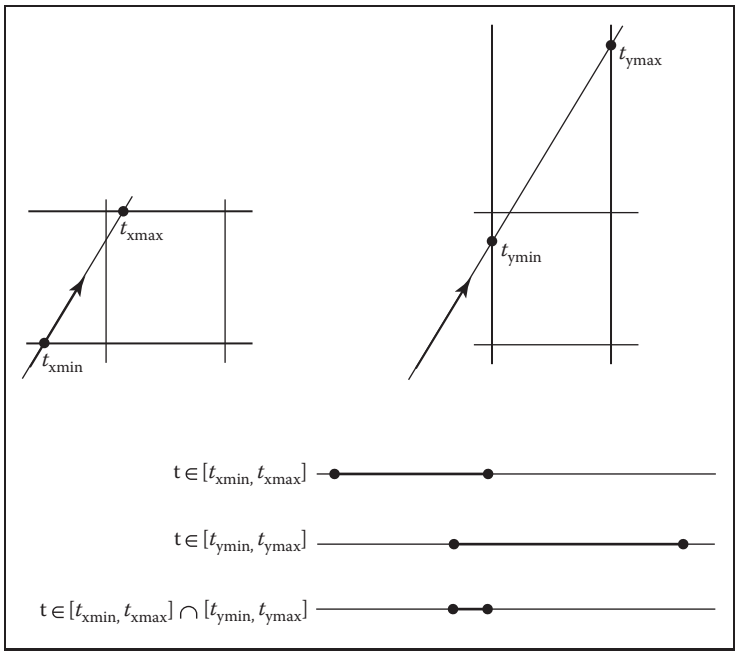


Figure 12.24. The ray will be inside the interval $x \in [x_{\text{min}}, x_{\text{max}}]$ for some interval in its parameter space $t \in [t_{\text{xmin}}, t_{\text{xmax}}]$. A similar interval exists for the y interval. The ray intersects the box if it is in both the x interval and y interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

The first thing we must address is the case when x_d or y_d is negative. If x_d is negative, then the ray will hit x_{max} before it hits x_{min}. Thus the code for computing t_{xmin} and t_{xmax} expands to:

if $(x_d \geq 0)$ **then**
  $t_{\text{xmin}} = (x_{\text{min}} - x_e)/x_d$
  $t_{\text{xmax}} = (x_{\text{max}} - x_e)/x_d$
else
  $t_{\text{xmin}} = (x_{\text{max}} - x_e)/x_d$
  $t_{\text{xmax}} = (x_{\text{min}} - x_e)/x_d$

A similar code expansion must be made for the y cases. A major concern is that horizontal and vertical rays have a zero value for y_d and x_d, respectively. This will cause divide by zero which may be a problem. However, before addressing this directly, we check whether IEEE floating point computation handles these cases gracefully for us. Recall from Section 1.5 the rules for divide by zero: for

any positive real number a,

$$+a/0 = +\infty;$$
$$-a/0 = -\infty.$$

Consider the case of a vertical ray where $x_d = 0$ and $y_d > 0$. We can then calculate

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{0};$$
$$t_{\text{xmax}} = \frac{x_{\text{max}} - x_e}{0}.$$

There are three possibilities of interest:

1. $x_e \leq x_{\text{min}}$ (no hit);
2. $x_{\text{min}} < x_e < x_{\text{max}}$ (hit);
3. $x_{\text{max}} \leq x_e$ (no hit).

For the first case we have

$$t_{\text{xmin}} = \frac{\text{positive number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (\infty, \infty)$. That interval will not overlap with any interval, so there will be no hit, as desired. For the second case, we have

$$t_{\text{xmin}} = \frac{\text{negative number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (-\infty, \infty)$ which will overlap with all intervals and thus will yield a hit as desired. The third case results in the interval $(-\infty, -\infty)$ which yields no hit, as desired. Because these cases work as desired, we need no special checks for them. As is often the case, IEEE floating point conventions are our ally. However, there is still a problem with this approach.

Consider the code segment:

if $(x_d \geq 0)$ **then**
  $t_{\text{min}} = (x_{\text{min}} - x_e)/x_d$
  $t_{\text{max}} = (x_{\text{max}} - x_e)/x_d$

else

$t_{\text{min}} = (x_{\text{max}} - x_e)/x_d$

$t_{\text{max}} = (x_{\text{min}} - x_e)/x_d$

This code breaks down when $x_d = -0$. This can be overcome by testing on the reciprocal of x_d (A. Williams, Barrus, Morley, & Shirley, 2005):

$a = 1/x_d$

if $(a \geq 0)$ **then**

$t_{\text{min}} = a(x_{\text{min}} - x_e)$

$t_{\text{max}} = a(x_{\text{max}} - x_e)$

else

$t_{\text{min}} = a(x_{\text{max}} - x_e)$

$t_{\text{max}} = a(x_{\text{min}} - x_e)$

12.3.2 Hierarchical Bounding Boxes

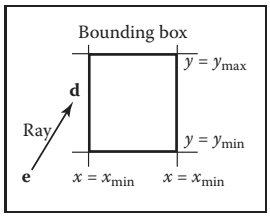


Figure 12.25. A 2D ray $\mathbf{e} + t\,\mathbf{d}$ is tested against a 2D bounding box.

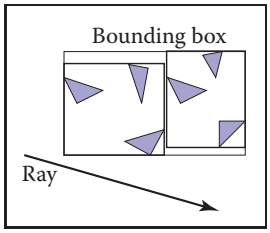


Figure 12.26. The bounding boxes can be nested by creating boxes around subsets of the model.

The basic idea of hierarchical bounding boxes can be seen by the common tactic of placing an axis-aligned 3D bounding box around all the objects as shown in Figure 12.25. Rays that hit the bounding box will actually be more expensive to compute than in a brute force search, because testing for intersection with the box is not free. However, rays that miss the box are cheaper than the brute force search. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition as shown in Figure 12.26. The data structure for the hierarchy shown in Figure 12.27 might be a tree with the large bounding box at the root and the two smaller bounding boxes as left and right subtrees. These would in turn each point to a list of three triangles. The intersection of a ray with this particular hard-coded tree would be:

if (ray hits root box) **then**

  if (ray hits left subtree box) **then**

    check three triangles for intersection

  if (ray intersects right subtree box) **then**

    check other three triangles for intersection

  if (an intersections returned from each subtree) **then**

    return the closest of the two hits

  else if (a intersection is returned from exactly one subtree) **then**

    return that intersection

  else

    return false

else

  return false

Some observations related to this algorithm are that there is no geometric ordering between the two subtrees, and there is no reason a ray might not hit both subtrees. Indeed, there is no reason that the two subtrees might not overlap.

A key point of such data hierarchies is that a box is guaranteed to bound all objects that are below it in the hierarchy, but they are *not* guaranteed to contain all objects that overlap it spatially, as shown in Figure 12.27. This makes this geometric search somewhat more complicated than a traditional binary search on strictly ordered one-dimensional data. The reader may note that several possible optimizations present themselves. We defer optimizations until we have a full hierarchical algorithm.

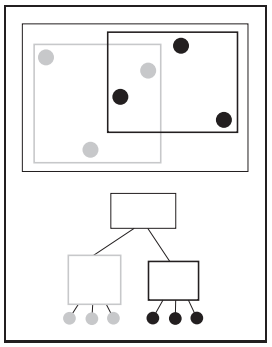


Figure 12.27. The gray box is a tree node that points to the three gray spheres, and the thick black box points to the three black spheres. Note that not all spheres enclosed by the box are guaranteed to be pointed to by the corresponding tree node.

If we restrict the tree to be binary and require that each node in the tree have a bounding box, then this traversal code extends naturally. Further, assume that all nodes are either leaves in the tree and contain a primitive, or that they contain one or two subtrees.

The `bvh-node` class should be of type surface, so it should implement `surface::hit`. The data it contains should be simple:

```
class bvh-node subclass of surface
  virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
  virtual box bounding-box()
  surface-pointer left
  surface-pointer right
  box bbox
```

The traversal code can then be called recursively in an object-oriented style:

```
function bool bvh-node::hit(ray a + tb, real t0, real t1,
                            hit-record rec)
  if (bbox.hitbox(a + tb, t0, t1)) then
    hit-record lrec, rrec
    left-hit = (left ≠ NULL) and (left → hit(a + tb, t0, t1, lrec))
    right-hit = (right ≠ NULL) and (right → hit(a+tb, t0, t1, rrec))
    if (left-hit and right-hit) then
      if (lrec.t < rrec.t) then
        rec = lrec
      else
        rec = rrec
      return true
    else if (left-hit) then
      rec = lrec
      return true
    else if (right-hit) then
```

```
        rec = rrec
        return true
      else
        return false
    else
      return false
```

Note that because `left` and `right` point to surfaces rather than `bvh-nodes` specifically, we can let the virtual functions take care of distinguishing between internal and leaf nodes; the appropriate hit function will be called. Note that if the tree is built properly, we can eliminate the check for left being NULL. If we want to eliminate the check for right being NULL, we can replace NULL right pointers with a redundant pointer to left. This will end up checking left twice, but will eliminate the check throughout the tree. Whether that is worth it will depend on the details of tree construction.

There are many ways to build a tree for a bounding volume hierarchy. It is convenient to make the tree binary, roughly balanced, and to have the boxes of sibling subtrees not overlap too much. A heuristic to accomplish this is to sort the surfaces along an axis before dividing them into two sublists. If the axes are defined by an integer with $x = 0$, $y = 1$, and $z = 2$ we have:

```
function bvh-node::create(object-array A, int AXIS)
  N = A.length
  if (N= 1) then
    left = A[0]
    right = NULL
    bbox = bounding-box(A[0])
  else if (N= 2) then
    left-node = A[0]
    right-node = A[1]
    bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
  else
    sort A by the object center along AXIS
    left= new bvh-node(A[0..N/2 − 1], (AXIS +1) mod 3)
    right = new bvh-node(A[N/2..N−1], (AXIS +1) mod 3)
    bbox = combine(left → bbox, right → bbox)
```

The quality of the tree can be improved by carefully choosing AXIS each time. One way to do this is to choose the axis such that the sum of the volumes of the bounding boxes of the two subtrees is minimized. This change compared to rotating through the axes will make little difference for scenes composed of isotopically distributed small objects, but it may help significantly in less well-behaved

scenes. This code can also be made more efficient by doing just a partition rather than a full sort.

Another, and probably better, way to build the tree is to have the subtrees contain about the same amount of space rather than the same number of objects. To do this we partition the list based on space:

```
function bvh-node::create(object-array A, int AXIS)
  N = A.length
  if (N = 1) then
    left = A[0]
    right = NULL
    bbox = bounding-box(A[0])
  else if (N = 2) then
    left = A[0]
    right = A[1]
    bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
  else
    find the midpoint m of the bounding box of A along AXIS
    partition A into lists with lengths k and (N − k) surrounding m
    left = new bvh-node(A[0..k], (AXIS +1) mod 3)
    right = new bvh-node(A[k + 1..N − 1], (AXIS +1) mod 3)
    bbox = combine(left → bbox, right → bbox)
```

Although this results in an unbalanced tree, it allows for easy traversal of empty space and is cheaper to build because partitioning is cheaper than sorting.

12.3.3 Uniform Spatial Subdivision

Another strategy to reduce intersection tests is to divide space. This is fundamentally different from dividing objects as was done with hierarchical bounding volumes:

- In hierarchical bounding volumes, each object belongs to one of two sibling nodes, whereas a point in space may be inside both sibling nodes.
- In spatial subdivision, each point in space belongs to exactly one node, whereas objects may belong to many nodes.

In uniform spatial subdivision, the scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes as shown in Figure 12.28. When an object is hit, the traversal ends.

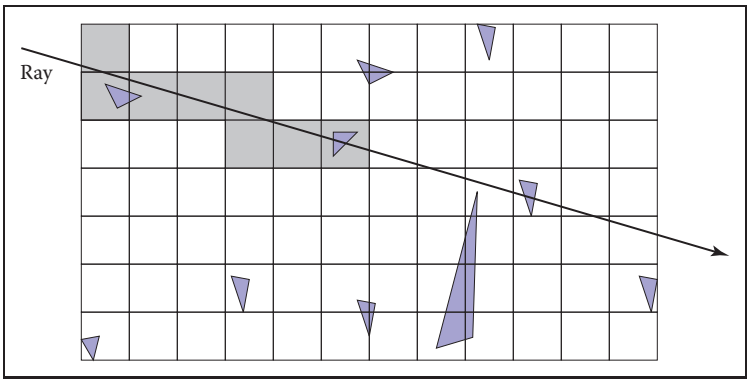


Figure 12.28. In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

The grid itself should be a subclass of surface and should be implemented as a 3D array of pointers to surface. For empty cells these pointers are NULL. For cells with one object, the pointer points to that object. For cells with more than one object, the pointer can point to a list, another grid, or another data structure, such as a bounding volume hierarchy.

This traversal is done in an incremental fashion. The regularity comes from the way that a ray hits each set of parallel planes, as shown in Figure 12.29. To see how this traversal works, first consider the 2D case where the ray direction has positive x and y components and starts outside the grid. Assume the grid is bounded by points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. The grid has $n_x \times n_y$ cells.

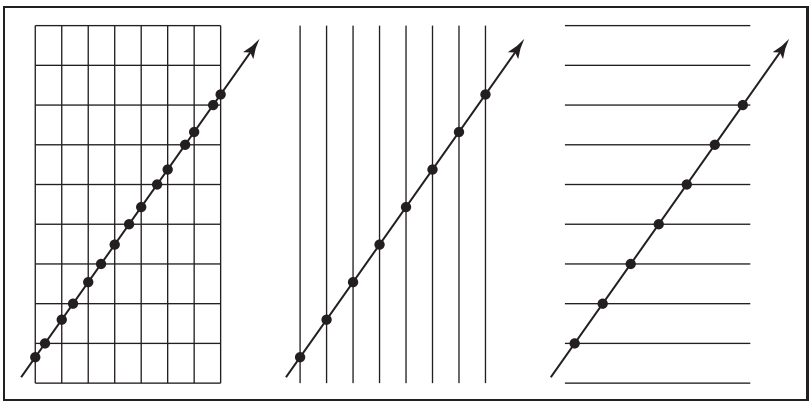


Figure 12.29. Although the pattern of cell hits seems irregular (left), the hits on sets of parallel planes are very even.

Our first order of business is to find the index (i, j) of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then, we need to traverse the cells in an appropriate order. The key parts to this algorithm are finding the initial cell (i, j) and deciding whether to increment i or j (Figure 12.30). Note that when we check for an intersection with objects in a cell, we restrict the range of t to be within the cell (Figure 12.31). Most implementations make the 3D array of type "pointer to surface." To improve the locality of the traversal, the array can be tiled as discussed in Section 12.5.

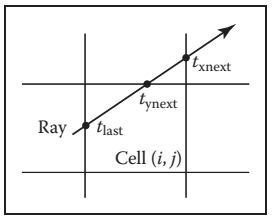


Figure 12.30. To decide whether we advance right or upward, we keep track of the intersections with the next vertical and horizontal boundary of the cell.

12.3.4 Axis-Aligned Binary Space Partitioning

We can also partition space in a hierarchical data structure such as a *binary space partitioning tree* (BSP tree). This is similar to the BSP tree used for visibility sorting in Section 12.4, but it's most common to use axis-aligned, rather than polygon-aligned, cutting planes for ray intersection.

A node in this structure contains a single cutting plane and a left and right subtree. Each subtree contains all the objects on one side of the cutting plane. Objects that pass through the plane are stored in in both subtrees. If we assume the cutting plane is parallel to the yz plane at $x = D$, then the node class is:

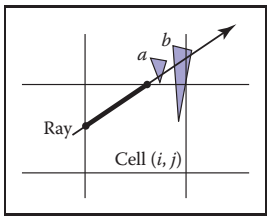


Figure 12.31. Only hits within the cell should be reported. Otherwise the case above would cause us to report hitting object *b* rather than object *a*.

```
class bsp-node subclass of surface
    virtual bool hit(ray e + td, real t0, real t1, hit-record rec)
    virtual box bounding-box()
    surface-pointer left
    surface-pointer right
    real D
```

We generalize this to y and z cutting planes later. The intersection code can then be called recursively in an object-oriented style. The code considers the four cases shown in Figure 12.32. For our purposes, the origin of these rays is a point at parameter t_0:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}.$$

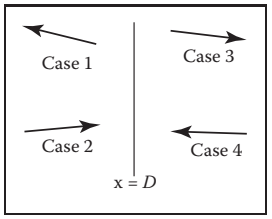


Figure 12.32. The four cases of how a ray relates to the BSP cutting plane $x = D$.

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.

2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.

3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.

4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
function bool bsp-node::hit(ray a + tb, real t0, real t1,
                            hit-record rec)
  xp = xa + t0xb
  if (xp < D) then
    if (xb < 0) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    t = (D − xa)/xb
    if (t > t1) then
      return (left ≠ NULL) and (left→hit(a + tb, t0, t1, rec))
    if (left ≠ NULL) and (left→hit(a + tb, t0, t, rec)) then
      return true
    return (right ≠ NULL) and (right→hit(a + tb, t, t1, rec))
  else
    analogous code for cases 3 and 4
```

This is very clean code. However, to get it started, we need to hit some root object that includes a bounding box so we can initialize the traversal, t_0 and t_1. An issue we have to address is that the cutting plane may be along any axis. We can add an integer index *axis* to the *bsp-node* class. If we allow an indexing operator for points, this will result in some simple modifications to the code above, for example,

$$x_p = x_a + t_0 x_b$$

would become

$$u_p = a[\text{axis}] + t_0 b[\text{axis}]$$

which will result in some additional array indexing, but will not generate more branches.

While the processing of a single bsp-node is faster than processing a bvh-node, the fact that a single surface may exist in more than one subtree means there are more nodes and, potentially, a higher memory use. How "well" the trees are built determines which is faster. Building the tree is similar to building the BVH tree. We can pick axes to split in a cycle, and we can split in half each time, or we can try to be more sophisticated in how we divide.

12.4 BSP Trees for Visibility

Another geometric problem in which spatial data structures can be used is determining the visibility ordering of objects in a scene with changing viewpoint.

If we are making many images of a fixed scene composed of planar polygons, from different viewpoints—as is often the case for applications such as games—we can use a *binary space partitioning* scheme closely related to the method for ray intersection discussed in the previous section. The difference is that for visibility sorting we use non–axis-aligned splitting planes, so that the planes can be made coincident with the polygons. This leads to an elegant algorithm known as the BSP tree algorithm to order the surfaces from front to back. The key aspect of the BSP tree is that it uses a preprocess to create a data structure that is useful for any viewpoint. So, as the viewpoint changes, the same data structure is used without change.

12.4.1 Overview of BSP Tree Algorithm

The BSP tree algorithm is an example of a *painter's algorithm*. A painter's algorithm draws every object from back-to-front, with each new polygon potentially overdrawing previous polygons, as is shown in Figure 12.33. It can be implemented as follows:

```
sort objects back to front relative to viewpoint
for each object do
    draw object on screen
```

The problem with the first step (the sort) is that the relative order of multiple objects is not always well defined, even if the order of every pair of objects is. This problem is illustrated in Figure 12.34 where the three triangles form a *cycle*.

The BSP tree algorithm works on any scene composed of polygons where no polygon crosses the plane defined by any other polygon. This restriction is then relaxed by a preprocessing step. For the rest of this discussion, triangles are assumed to be the only primitive, but the ideas extend to arbitrary polygons.

The basic idea of the BSP tree can be illustrated with two triangles, T_1 and T_2. We first recall (see Section 2.5.3) the implicit plane equation of the plane containing T_1: $f_1(\mathbf{p}) = 0$. The key property of implicit planes that we wish to take advantage of is that for all points $\mathbf{p}^+$ on one side of the plane, $f_1(\mathbf{p}^+) > 0$; and for all points $\mathbf{p}^-$ on the other side of the plane, $f_1(\mathbf{p}^-) < 0$. Using this property, we can find out on which side of the plane T_2 lies. Again, this assumes all three vertices of T_2 are on the same side of the plane. For discussion, assume

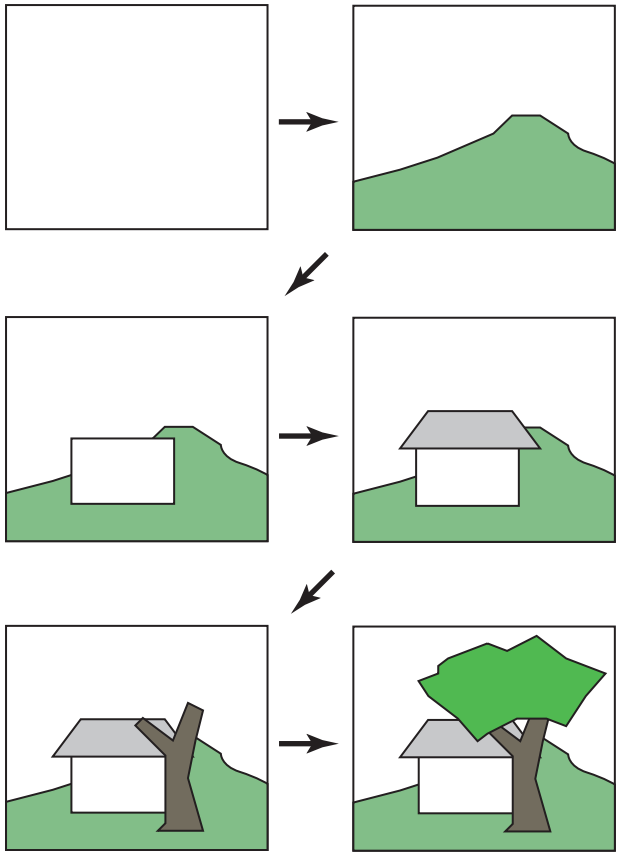


Figure 12.33. A painter's algorithm starts with a blank image and then draws the scene one object at a time from back-to-front, overdrawing whatever is already there. This automatically eliminates hidden surfaces.

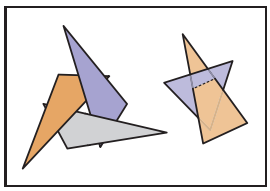


Figure 12.34. A cycle occurs if a global back-to-front ordering is not possible for a particular eye position.

that T_2 is on the $f_1(\mathbf{p}) < 0$ side of the plane. Then, we can draw T_1 and T_2 in the right order for any eyepoint $\mathbf{e}$:

if $(f_1(\mathbf{e}) < 0)$ **then**
  draw T_1
  draw T_2
else
  draw T_2
  draw T_1

The reason this works is that if T_2 and $\mathbf{e}$ are on the same side of the plane containing T_1, there is no way for T_2 to be fully or partially blocked by T_1 as seen

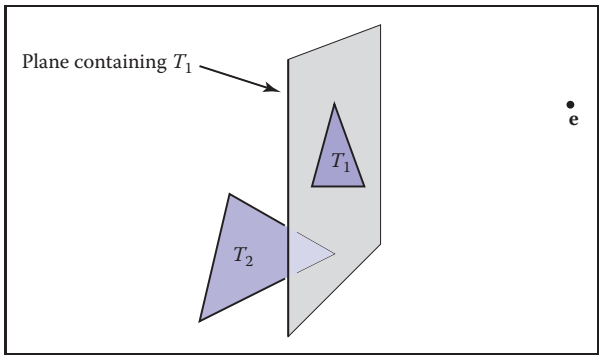


Figure 12.35. When **e** and T_2 are on opposite sides of the plane containing T_1, then it is safe to draw T_2 first and T_1 second. If **e** and T_2 are on the same side of the plane, then T_1 should be drawn before T_2. This is the core idea of the BSP tree algorithm.

from **e**, so it is safe to draw T_1 first. If **e** and T_2 are on opposite sides of the plane containing T_1, then T_2 cannot fully or partially block T_1, and the opposite drawing order is safe (Figure 12.35).

This observation can be generalized to many objects provided none of them span the plane defined by T_1. If we use a binary tree data structure with T_1 as root, the *negative* branch of the tree contains all the triangles whose vertices have $f_i(\mathbf{p}) < 0$, and the *positive* branch of the tree contains all the triangles whose vertices have $f_i(\mathbf{p}) > 0$. We can draw in proper order as follows:

```
function draw(bsptree tree, point e)
  if (tree.empty) then
    return
  if (f_tree.root(e) < 0) then
    draw(tree.plus, e)
    rasterize tree.triangle
    draw(tree.minus, e)
  else
    draw(tree.minus, e)
    rasterize tree.triangle
    draw(tree.plus, e)
```

The nice thing about that code is that it will work for any viewpoint **e**, so the tree can be precomputed. Note that, if each subtree is itself a tree, where the root triangle divides the other triangles into two groups relative to the plane containing it, the code will work as is. It can be made slightly more efficient by terminating the recursive calls one level higher, but the code will still be simple. A tree

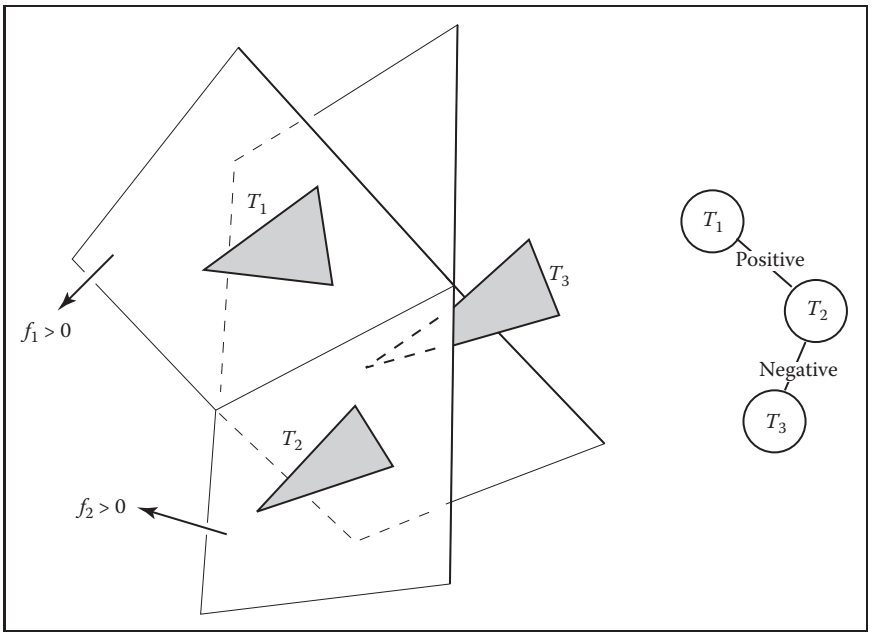


Figure 12.36. Three triangles and a BSP tree that is valid for them. The "positive" and "negative" are encoded by right and left subtree position, respectively.

illustrating this code is shown in Figure 12.36. As discussed in Section 2.5.5, the implicit equation for a point $\mathbf{p}$ on a plane containing three non-colinear points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ is

$$f(\mathbf{p}) = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) \cdot (\mathbf{p} - \mathbf{a}) = 0. \tag{12.1}$$

It can be faster to store the (A, B, C, D) of the implicit equation of the form

$$f(x, y, z) = Ax + By + Cz + D = 0. \tag{12.2}$$

Equations (12.1) and (12.2) are equivalent, as is clear when you recall that the gradient of the implicit equation is the normal to the triangle. The gradient of Equation (12.2) is $\mathbf{n} = (A, B, C)$ which is just the normal vector

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}).$$

We can solve for D by plugging in any point on the plane, e.g., $\mathbf{a}$:

$$\begin{aligned} D &= -Ax_a - By_a - Cz_a \\ &= -\mathbf{n} \cdot \mathbf{a}. \end{aligned}$$

This suggests the form:

$$\begin{aligned} f(\mathbf{p}) &= \mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{a} \\ &= \mathbf{n} \cdot (\mathbf{p} - \mathbf{a}) \\ &= 0, \end{aligned}$$

which is the same as Equation (12.1) once you recall that $\mathbf{n}$ is computed using the cross product. Which form of the plane equation you use and whether you store only the vertices, $\mathbf{n}$ and the vertices, or $\mathbf{n}$, D, and the vertices, is probably a matter of taste—a classic time-storage tradeoff that will be settled best by profiling. For debugging, using Equation (12.1) is probably the best.

The only issue that prevents the code above from working in general is that one cannot guarantee that a triangle can be uniquely classified on one side of a plane or the other. It can have two vertices on one side of the plane and the third on the other. Or it can have vertices on the plane. This is handled by splitting the triangle into smaller triangles using the plane to "cut" them.

12.4.2 Building the Tree

If none of the triangles in the dataset cross each other's planes, so that all triangles are on one side of all other triangles, a BSP tree that can be traversed using the code above can be built using the following algorithm:

```
tree-root = node(T_1)
for i ∈ {2, ..., N} do
    tree-root.add(T_i)
function add ( triangle T )
    if (f(a) < 0 and f(b) < 0 and f(c) < 0) then
        if (negative subtree is empty) then
            negative-subtree = node(T)
        else
            negative-subtree.add (T)
    else if (f(a) > 0 and f(b) > 0 and f(c) > 0) then
        if positive subtree is empty then
            positive-subtree = node(T)
        else
            positive-subtree.add (T)
    else
        we have assumed this case is impossible
```

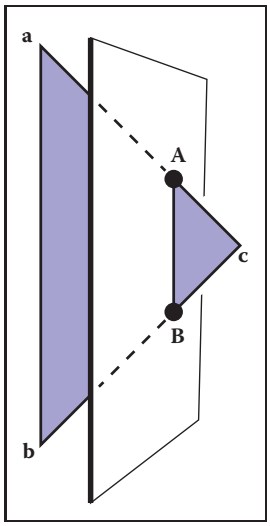


Figure 12.37. When a triangle spans a plane, there will be one vertex on one side and two on the other.

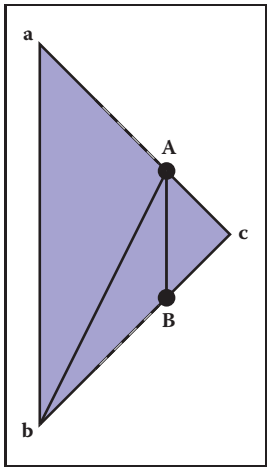


Figure 12.38. When a triangle is cut, we break it into three triangles, none of which span the cutting plane.

The only thing we need to fix is the case where the triangle crosses the dividing plane, as shown in Figure 12.37. Assume, for simplicity, that the triangle has vertices $\mathbf{a}$ and $\mathbf{b}$ on one side of the plane, and vertex $\mathbf{c}$ is on the other side. In this case, we can find the intersection points $\mathbf{A}$ and $\mathbf{B}$ and cut the triangle into three new triangles with vertices

$$T_1 = (\mathbf{a}, \mathbf{b}, \mathbf{A}),$$
$$T_2 = (\mathbf{b}, \mathbf{B}, \mathbf{A}),$$
$$T_3 = (\mathbf{A}, \mathbf{B}, \mathbf{c}),$$

as shown in Figure 12.38. This order of vertices is important so that the direction of the normal remains the same as for the original triangle. If we assume that $f(\mathbf{c}) < 0$, the following code could add these three triangles to the tree assuming the positive and negative subtrees are not empty:

positive-subtree = node (T_1)
positive-subtree = node (T_2)
negative-subtree = node (T_3)

A precision problem that will plague a naive implementation occurs when a vertex is very near the splitting plane. For example, if we have two vertices on one side of the splitting plane and the other vertex is only an extremely small distance on the other side, we will create a new triangle almost the same as the old one, a triangle that is a sliver, and a triangle of almost zero size. It would be better to detect this as a special case and not split into three new triangles. One might expect this case to be rare, but because many models have tessellated planes and triangles with shared vertices, it occurs frequently, and thus must be handled carefully. Some simple manipulations that accomplish this are:

function add(triangle T)
  fa = $f(\mathbf{a})$
  fb = $f(\mathbf{b})$
  fc = $f(\mathbf{c})$
  if $(abs(fa) < \epsilon)$ **then**
    fa = 0
  if $(abs(fb) < \epsilon)$ **then**
    fb = 0
  if $(abs(fc) < \epsilon)$ **then**
    fc = 0
  if ($fa \leq 0$ and fb ≤ 0 and fc ≤ 0) **then**
    if (negative subtree is empty) **then**
      negative-subtree = node(T)

```
    else
      negative-subtree.add(T)
  else if (fa ≥ 0 and fb ≥ 0 and fc ≥ 0) then
    if (positive subtree is empty) then
      positive-subtree = node(T)
    else
      positive-subtree.add(T)
  else
    cut triangle into three triangles and add to each side
```

This takes any vertex whose f value is within ϵ of the plane and counts it as positive or negative. The constant ϵ is a small positive real chosen by the user. The technique above is a rare instance where testing for floating-point equality is useful and works because the zero value is set rather than being computed. Comparing for equality with a computed floating-point value is almost never advisable, but we are not doing that.

12.4.3 Cutting Triangles

Filling out the details of the last case "cut triangle into three triangles and add to each side" is straightforward, but tedious. We should take advantage of the BSP tree construction as a preprocess where highest efficiency is not key. Instead, we should attempt to have a clean compact code. A nice trick is to force many of the cases into one by ensuring that $\mathbf{c}$ is on one side of the plane and the other two vertices are on the other. This is easily done with swaps. Filling out the details in the final else statement (assuming the subtrees are nonempty for simplicity) gives:

```
if (fa * fc ≥ 0) then
  swap(fb, fc)
  swap(b, c)
  swap(fa, fb)
  swap(a, b)
else if (fb * fc ≥ 0) then
  swap(fa, fc)
  swap(a, c)
  swap(fa, fb)
  swap(a, b)
compute A
compute B
T1 = (a, b, A)
T2 = (b, B, A)
```

$T_3 = (\mathbf{A}, \mathbf{B}, \mathbf{c})$
if $(fc \geq 0)$ **then**
  negative-subtree.add(T_1)
  negative-subtree.add(T_2)
  positive-subtree.add(T_3)
else
  positive-subtree.add(T_1)
  positive-subtree.add(T_2)
  negative-subtree.add(T_3)

This code takes advantage of the fact that the product of a and b are positive if they have the same sign—thus, the first if statement. If vertices are swapped, we must do two swaps to keep the vertices ordered counterclockwise. Note that exactly one of the vertices may lie exactly on the plane, in which case the code above will work, but one of the generated triangles will have zero area. This can be handled by ignoring the possibility, which is not that risky, because the rasterization code must handle zero-area triangles in screen space (i.e., edge-on triangles). You can also add a check that does not add zero-area triangles to the tree. Finally, you can put in a special case for when exactly one of fa, fb, and fc is zero which cuts the triangle into two triangles.

To compute $\mathbf{A}$ and $\mathbf{B}$, a line segment and implicit plane intersection is needed. For example, the parametric line connecting $\mathbf{a}$ and $\mathbf{c}$ is

$$\mathbf{p}(t) = \mathbf{a} + t(\mathbf{c} - \mathbf{a}).$$

The point of intersection with the plane $\mathbf{n} \cdot \mathbf{p} + D = 0$ is found by plugging $\mathbf{p}(t)$ into the plane equation:

$$\mathbf{n} \cdot (\mathbf{a} + t(\mathbf{c} - \mathbf{a})) + D = 0,$$

and solving for t:

$$t = -\frac{\mathbf{n} \cdot \mathbf{a} + D}{\mathbf{n} \cdot (\mathbf{c} - \mathbf{a})}.$$

Calling this solution t_A, we can write the expression for $\mathbf{A}$:

$$\mathbf{A} = \mathbf{a} + t_A(\mathbf{c} - \mathbf{a}).$$

A similar computation will give $\mathbf{B}$.

12.4.4 Optimizing the Tree

The efficiency of tree creation is much less of a concern than tree traversal because it is a preprocess. The traversal of the BSP tree takes time proportional to the number of nodes in the tree. (How well balanced the tree is does not matter.) There will be one node for each triangle, including the triangles that are created as a result of splitting. This number can depend on the order in which triangles are added to the tree. For example, in Figure 12.39, if T_1 is the root, there will be two nodes in the tree, but if T_2 is the root, there will be more nodes, because T_1 will be split.

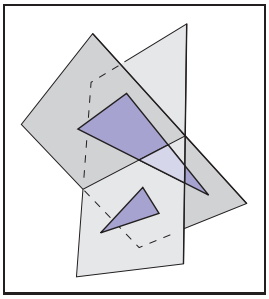


Figure 12.39. Using T_1 as the root of a BSP tree will result in a tree with two nodes. Using T_2 as the root will require a cut and thus make a larger tree.

It is difficult to find the "best" order of triangles to add to the tree. For N triangles, there are $N!$ orderings that are possible. So trying all orderings is not usually feasible. Alternatively, some predetermined number of orderings can be tried from a random collection of permutations, and the best one can be kept for the final tree.

The splitting algorithm described above splits one triangle into three triangles. It could be more efficient to split a triangle into a triangle and a convex quadrilateral. This is probably not worth it if all input models have only triangles, but would be easy to support for implementations that accommodate arbitrary polygons.

12.5 Tiling Multidimensional Arrays

Effectively utilizing the memory hierarchy is a crucial task in designing algorithms for modern architectures. Making sure that multidimensional arrays have data in a "nice" arrangement is accomplished by *tiling*, sometimes also called *bricking*. A traditional 2D array is stored as a 1D array together with an indexing mechanism; for example, an N_x by N_y array is stored in a 1D array of length $N_x N_y$ and the 2D index (x, y) (which runs from $(0, 0)$ to $(N_x - 1, N_y - 1)$) maps to the 1D index (running from 0 to $N_x N_y - 1$) using the formula

$$\text{index} = x + N_x y.$$

An example of how that memory lays out is shown in Figure 12.40. A problem with this layout is that although two adjacent array elements that are in the same row are next to each other in memory, two adjacent elements in the same column will be separated by N_x elements in memory. This can cause poor memory locality for large N_x. The standard solution to this is to use *tiles* to make memory

$j = 2$	8	9	10	11
$j = 1$	4	5	6	7
$j = 0$	0	1	2	3
	$i = 0$	$i = 1$	$i = 2$	$i = 3$

Figure 12.40. The memory layout for an untiled 2D array with $N_x = 4$ and $N_y = 3$.

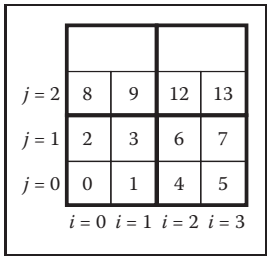


Figure 12.41. The memory layout for a tiled 2D array with $N_x = 4$ and $N_y = 3$ and 2×2 tiles. Note that padding on the top of the array is needed because N_y is not a multiple of the tile size two.

locality for rows and columns more equal. An example is shown in Figure 12.41 where 2×2 tiles are used. The details of indexing such an array are discussed in the next section. A more complicated example, with two levels of tiling on a 3D array, is covered after that.

A key question is what size to make the tiles. In practice, they should be similar to the memory-unit size on the machine. For example, if we are using 16-bit (2-byte) data values on a machine with 128-byte cache lines, 8×8 tiles fit exactly in a cache line. However, using 32-bit floating-point numbers, which fit 32 elements to a cache line, 5×5 tiles are a bit too small and 6×6 tiles are a bit too large. Because there are also coarser-sized memory units such as pages, hierarchical tiling with similar logic can be useful.

12.5.1 One-Level Tiling for 2D Arrays

If we assume an $N_x \times N_y$ array decomposed into square $n \times n$ tiles (Figure 12.42), then the number of tiles required is

$$B_x = N_x/n,$$
$$B_y = N_y/n.$$

Here, we assume that n divides N_x and N_y exactly. When this is not true, the array should be *padded*. For example, if $N_x = 15$ and $n = 4$, then N_x should be changed to 16. To work out a formula for indexing such an array, we first find

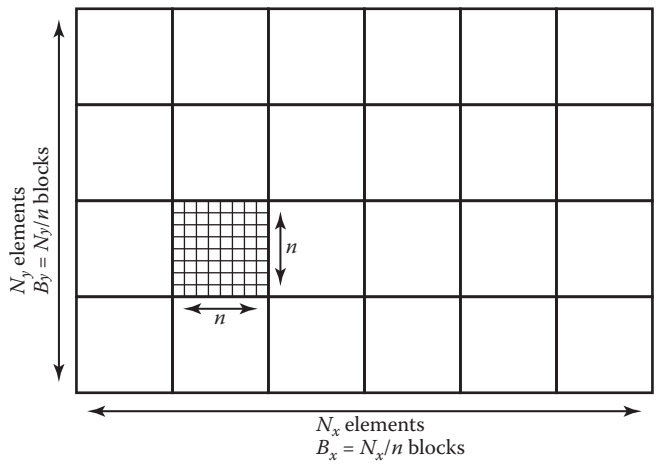


Figure 12.42. A tiled 2D array composed of $B_x \times B_y$ tiles each of size n by n.

the tile indices (b_x, b_y) that give the row/column for the tiles (the tiles themselves form a 2D array):

$$b_x = x \div n,$$
$$b_y = y \div n,$$

where $\div$ is integer division, e.g., $12 \div 5 = 2$. If we order the tiles along rows as shown in Figure 12.40, then the index of the first element of the tile (b_x, b_y) is

$$\text{index} = n^2(B_x b_y + b_x).$$

The memory in that tile is arranged like a traditional 2D array as shown in Figure 12.41. The partial offsets (x', y') inside the tile are

$$x' = x \bmod n,$$
$$y' = y \bmod n,$$

where $\bmod$ is the remainder operator, e.g., $12 \bmod 5 = 2$. Therefore, the offset inside the tile is

$$\text{offset} = y'n + x'.$$

Thus the full formula for finding the 1D index element (x, y) in an $N_x \times N_y$ array with $n \times n$ tiles is

$$\begin{aligned}\text{index} &= n^2(B_x b_y + b_x) + y'n + x', \\ &= n^2((N_x \div n)(y \div n) + x \div n) + (y \bmod n)n + (x \bmod n).\end{aligned}$$

This expression contains many integer multiplication, divide and modulus operations, which are costly on some processors. When n is a power of two, these operations can be converted to bitshifts and bitwise logical operations. However, as noted above, the ideal size is not always a power of two. Some of the multiplications can be converted to shift/add operations, but the divide and modulus operations are more problematic. The indices could be computed incrementally, but this would require tracking counters, with numerous comparisons and poor branch prediction performance.

However, there is a simple solution; note that the index expression can be written as

$$\text{index} = F_x(x) + F_y(y),$$

where

$$F_x(x) = n^2(x \div n) + (x \bmod n),$$
$$F_y(y) = n^2(N_x \div n)(y \div n) + (y \bmod n)n.$$

We tabulate F_x and F_y, and use x and y to find the index into the data array. These tables will consist of N_x and N_y elements, respectively. The total size of the tables will fit in the primary data cache of the processor, even for very large data set sizes.

12.5.2 Example: Two-Level Tiling for 3D Arrays

TLB: translation lookaside buffer, a cache that is part of the virtual memory system.

Effective TLB utilization is also becoming a crucial factor in algorithm performance. The same technique can be used to improve TLB hit rates in a 3D array by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a $40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 cannot be factored by $mn = 10$, one level of padding is needed. Empirically useful sizes are $m = 5$ for 16-bit datasets and $m = 6$ for float datasets.

The resulting index into the data array can be computed for any (x, y, z) triple with the expression

$$\begin{aligned}
\text{index} \quad = \quad & ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\
& +((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\
& +((z \div n) \div m)n^3m^3 \\
& +((x \div n) \bmod m)n^3m^2 \\
& +((y \div n) \bmod m)n^3m \\
& +((z \div n) \bmod m)n^3 \\
& +(x \bmod (n^2))n^2 \\
& +(y \bmod n)n \\
& +(z \bmod n),
\end{aligned}$$

where N_x, N_y and N_z are the respective sizes of the dataset.

Note that, as in the simpler 2D one-level case, this expression can be written as

$$\text{index} = F_x(x) + F_y(y) + F_z(z),$$

where

$$\begin{aligned}
F_x(x) \quad = \quad & ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\
& +((x \div n) \bmod m)n^3m^2 \\
& +(x \bmod n)n^2, \\
F_y(y) \quad = \quad & ((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\
& +((y \div n) \bmod m)n^3m + \\
& +(y \bmod n)n, \\
F_z(z) \quad = \quad & ((z \div n) \div m)n^3m^3 \\
& +((z \div n) \bmod m)n^3 \\
& +(z \bmod n).
\end{aligned}$$

Frequently Asked Questions

• Does tiling really make that much difference in performance?

On some volume rendering applications, a two-level tiling strategy made as much as a factor-of-ten performance difference. When the array does not fit in main memory, it can effectively prevent thrashing in some applications such as image editing.

• How do I store the lists in a winged-edge structure?

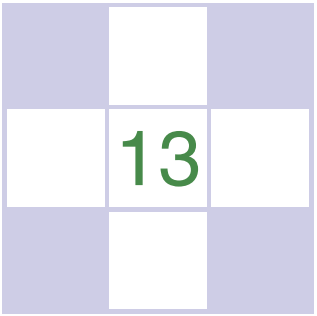
For most applications, it is feasible to use arrays and indices for the references. However, if many delete operations are to be performed, then it is wise to use linked lists and pointers.

Notes

The discussion of the winged-edge data structure is based on the course notes of *Ching-Kuang Shene* (Shene, 2003). There are smaller mesh data structures than winged-edge. The tradeoffs in using such structures is discussed in *Directed Edges—A Scalable Representation for Triangle Meshes* (Campagna, Kobbelt, & Seidel, 1998). The tiled-array discussion is based on *Interactive Ray Tracing for Volume Visualization* (S. Parker et al., 1999). A structure similar to the triangle neighbor structure is discussed in a technical report by Charles Loop (Loop, 2000). A discussion of manifolds can be found in an introductory topology text (Munkres, 2000).

Exercises

1. What is the memory difference for a simple tetrahedron stored as four independent triangles and one stored in a winged-edge data structure?
2. Diagram a scene graph for a bicycle.
3. How many look-up tables are needed for a single-level tiling of an n-dimensional array?
4. Given N triangles, what is the minimum number of triangles that could be added to a resulting BSP tree? What is the maximum number?



13 More Ray Tracing

A ray tracer is a great substrate on which to build all kinds of advanced rendering effects. Many effects that take significant work to fit into the object-order rasterization framework, including basics like the shadows and reflections already presented in Chapter 4, are simple and elegant in a ray tracer. In this chapter, we discuss some fancier techniques that can be used to ray-trace a wider variety of scenes and to include a wider variety of effects. Some extensions allow more general geometry: instancing and constructive solid geometry (CSG) are two ways to make models more complex with minimal complexity added to the program. Other extensions add to the range of materials we can handle: refraction through transparent materials, like glass and water, and glossy reflections on a variety of surfaces are essential for realism in many scenes.

This chapter also discusses the general framework of *distribution ray tracing* (Cook, Porter, & Carpenter, 1984), a powerful extension to the basic ray-tracing idea in which multiple random rays are sent through each pixel in an image to produce images with smooth edges and to simply and elegantly (if slowly) produce a wide range of effects from soft shadows to camera depth-of-field.

The price of the elegance of ray tracing is exacted in terms of computer time: most of these extensions will trace a very large number of rays for any nontrivial scene. Because of this, it's crucial to use the methods described in Chapter 12 to accelerate the tracing of rays.

If you start with a brute-force ray intersection loop, you'll have ample time to implement an acceleration structure while you wait for images to render.

13.1 Transparency and Refraction

In Chapter 4, we discussed the use of recursive ray tracing to compute specular, or mirror, reflection from surfaces. Another type of specular object is a *dielectric*—a transparent material that refracts light. Diamonds, glass, water, and air are dielectrics. Dielectrics also filter light; some glass filters out more red and blue light than green light, so the glass takes on a green tint. When a ray travels from a medium with refractive index n into one with a refractive index n_t, some of the light is transmitted, and it bends. This is shown for $n_t > n$ in Figure 13.1. Snell's Law tells us that

$$n \sin\theta = n_t \sin\phi.$$

Example values of n:
air: 1.00;
water: 1.33–1.34;
window glass: 1.51;
optical glass: 1.49–1.92;
diamond: 2.42.

Computing the sine of an angle between two vectors is usually not as convenient as computing the cosine, which is a simple dot product for the unit vectors such as we have here. Using the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$, we can derive a refraction relationship for cosines:

$$\cos^2\phi = 1 - \frac{n^2\left(1-\cos^2\theta\right)}{n_t^2}.$$

Note that if n and n_t are reversed, then so are θ and ϕ as shown on the right of Figure 13.1.

To convert $\sin\phi$ and $\cos\phi$ into a 3D vector, we can set up a 2D orthonormal basis in the plane of the surface normal, $\mathbf{n}$, and the ray direction, $\mathbf{d}$.

From Figure 13.2, we can see that $\mathbf{n}$ and $\mathbf{b}$ form an orthonormal basis for the plane of refraction. By definition, we can describe the direction of the transformed

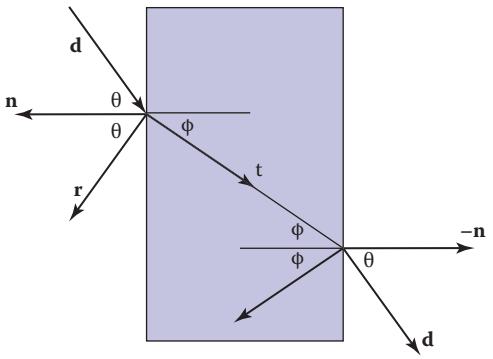


Figure 13.1. Snell's Law describes how the angle ϕ depends on the angle θ and the refractive indices of the object and the surrounding medium.

ray, $\mathbf{t}$, in terms of this basis:

$$\mathbf{t} = \sin\phi\mathbf{b} - \cos\phi\mathbf{n}.$$

Since we can describe $\mathbf{d}$ in the same basis, and $\mathbf{d}$ is known, we can solve for $\mathbf{b}$:

$$\mathbf{d} = \sin\theta\mathbf{b} - \cos\theta\mathbf{n},$$
$$\mathbf{b} = \frac{\mathbf{d} + \mathbf{n}\cos\theta}{\sin\theta}.$$

This means that we can solve for $\mathbf{t}$ with known variables:

$$\begin{aligned}\mathbf{t} &= \frac{n\left(\mathbf{d} + \mathbf{n}\cos\theta)\right)}{n_t} - \mathbf{n}\cos\phi \\ &= \frac{n\left(\mathbf{d} - \mathbf{n}(\mathbf{d}\cdot\mathbf{n})\right)}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1 - (\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}.\end{aligned}$$

Note that this equation works regardless of which of n and n_t is larger. An immediate question is, "What should you do if the number under the square root is negative?" In this case, there is no refracted ray and all of the energy is reflected. This is known as *total internal reflection*, and it is responsible for much of the rich appearance of glass objects.

The reflectivity of a dielectric varies with the incident angle according to the *Fresnel equations*. A nice way to implement something close to the Fresnel equations is to use the *Schlick approximation* (Schlick, 1994a),

$$R(\theta) = R_0 + (1 - R_0)\left(1 - \cos\theta\right)^5,$$

where R_0 is the reflectance at normal incidence:

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2.$$

Note that the $\cos\theta$ terms above are always for the angle in air (the larger of the internal and external angles relative to the normal).

For homogeneous impurities, as is found in typical colored glass, a light-carrying ray's intensity will be attenuated according to *Beer's Law*. As the ray travels through the medium it loses intensity according to $dI = -CI\,dx$, where dx is distance. Thus, $dI/dx = -CI$. We can solve this equation and get the exponential $I = k\exp(-Cx)$. The degree of attenuation is described by the RGB attenuation constant a, which is the amount of attenuation after one unit of distance. Putting in boundary conditions, we know that $I(0) = I_0$, and $I(1) =$

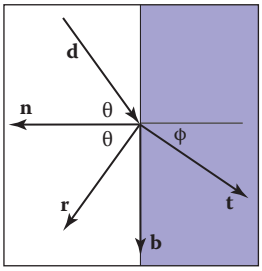


Figure 13.2. The vectors $\mathbf{n}$ and $\mathbf{b}$ form a 2D orthonormal basis that is parallel to the transmission vector $\mathbf{t}$.

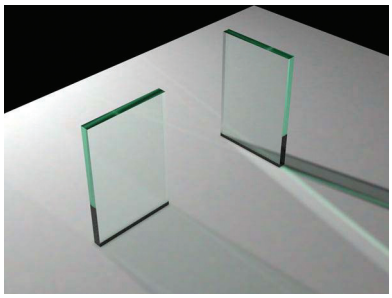


Figure 13.3. The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter 23.

$aI(0)$. The former implies $I(x) = I_0 \exp(-Cx)$. The latter implies $I_0 a = I_0 \exp(-C)$, so $-C = \ln(a)$. Thus, the final formula is

$$I(s) = I(0)e^{\ln(a)s},$$

where $I(s)$ is the intensity of the beam at distance s from the interface. In practice, we reverse-engineer a by eye, because such data is rarely easy to find. The effect of Beer's Law can be seen in Figure 13.3, where the glass takes on a green tint.

To add transparent materials to our code, we need a way to determine when a ray is going "into" an object. The simplest way to do this is to assume that all objects are embedded in air with refractive index very close to 1.0, and that surface normals point "out" (toward the air). The code segment for rays and dielectrics with these assumptions is:

```
if (p is on a dielectric) then
  r = reflect(d, n )
  if (d · n < 0) then
    refract(d, n, n, t)
    c = −d · n
    k_r = k_g = k_b = 1
  else
    k_r = exp(−a_r t)
    k_g = exp(−a_g t)
    k_b = exp(−a_b t)
    if refract(d, −n, 1/n, t) then
      c = t · n
    else
      return k ∗ color(p + tr)
  R_0 = (n − 1)^2/(n + 1)^2
```

$R = R_0 + (1 - R_0)(1 - c)^5$
return $k(R \text{ color}(\mathbf{p} + t\mathbf{r}) + (1 - R) \text{ color}(\mathbf{p} + t\mathbf{t}))$

The code above assumes that the natural log has been folded into the constants (a_r, a_g, a_b). The *refract* function returns false if there is total internal reflection, and otherwise it fills in the last argument of the argument list.

13.2 Instancing

An elegant property of ray tracing is that it allows very natural *instancing*. The basic idea of instancing is to distort all points on an object by a transformation matrix before the object is displayed. For example, if we transform the unit circle (in 2D) by a scale factor $(2, 1)$ in x and y, respectively, then rotate it by $45°$, and move one unit in the x-direction, the result is an ellipse with an eccentricity of 2 and a long axis along the $(x = -y)$-direction centered at $(0, 1)$ (Figure 13.4). The key thing that makes that entity an "instance" is that we store the circle and the composite transform matrix. Thus, the explicit construction of the ellipse is left as a future operation at render time.

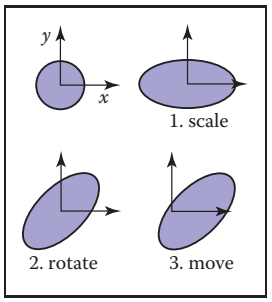


Figure 13.4. An instance of a circle with a series of three transforms is an ellipse.

The advantage of instancing in ray tracing is that we can choose the space in which to do intersection. If the base object is composed of a set of points, one of which is $\mathbf{p}$, then the transformed object is composed of that set of points transformed by matrix $\mathbf{M}$, where the example point is transformed to $\mathbf{Mp}$. If we have a ray $\mathbf{a} + t\mathbf{b}$ that we want to intersect with the transformed object, we can instead intersect an *inverse-transformed ray* with the untransformed object (Figure 13.5). There are two potential advantages to computing in the untransformed space (i.e., the right-hand side of Figure 13.5):

1. The untransformed object may have a simpler intersection routine, e.g., a sphere versus an ellipsoid.
2. Many transformed objects can share the same untransformed object thus reducing storage, e.g., a traffic jam of cars, where individual cars are just transforms of a few base (untransformed) models.

As discussed in Section 6.2.2, surface normal vectors transform differently. With this in mind and using the concepts illustrated in Figure 13.5, we can determine the intersection of a ray and an object transformed by matrix $\mathbf{M}$. If we create an instance class of type *surface*, we need to create a *hit* function:

instance::hit(ray $\mathbf{a} + t\mathbf{b}$, real t_0, real t_1, hit-record rec)
ray $\mathbf{r}' = \mathbf{M}^{-1}\mathbf{a} + t\mathbf{M}^{-1}\mathbf{b}$

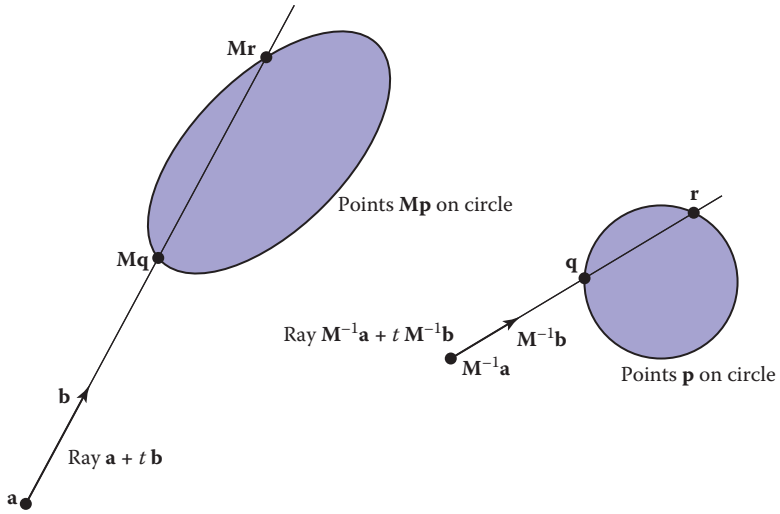


Figure 13.5. The ray intersection problem in the two spaces are just simple transforms of each other. The object is specified as a sphere plus matrix **M**. The ray is specified in the transformed (world) space by location **a** and direction **b**.

```
if (base-object→hit(r′, t0, t1, rec)) then
    rec.n = (M^-1)^T rec.n
    return true
else
    return false
```

An elegant thing about this function is that the parameter rec.t does not need to be changed, because it is the same in either space. Also note that we need not compute or store the matrix $\mathbf{M}$.

This brings up a very important point: the ray direction $\mathbf{b}$ must *not* be restricted to a unit-length vector, or none of the infrastructure above works. For this reason, it is useful not to restrict ray directions to unit vectors.

13.3 Constructive Solid Geometry

One nice thing about ray tracing is that any geometric primitive whose intersection with a 3D line can be computed can be seamlessly added to a ray tracer. It turns out to also be straightforward to add *constructive solid geometry* (CSG) to a ray

tracer (Roth, 1982). The basic idea of CSG is to use set operations to combine solid shapes. These basic operations are shown in Figure 13.6. The operations can be viewed as *set* operations. For example, we can consider C the set of all points in the circle and S the set of all points in the square. The intersection operation $C \cap S$ is the set of all points that are both members of C and S. The other operations are analogous.

Although one can do CSG directly on the model, if all that is desired is an image, we do not need to explicitly change the model. Instead, we perform the set operations directly on the rays as they interact with a model. To make this natural, we find all the intersections of a ray with a model rather than just the closest. For example, a ray $\mathbf{a} + t\mathbf{b}$ might hit a sphere at $t = 1$ and $t = 2$. In the context of CSG, we think of this as the ray being inside the sphere for $t \in [1, 2]$. We can compute these "inside intervals" for all of the surfaces and do set operations on those intervals (recall Section 2.1.2). This is illustrated in Figure 13.7, where the hit intervals are processed to indicate that there are two intervals inside the difference object. The first hit for $t > 0$ is what the ray actually intersects.

In practice, the CSG intersection routine must maintain a list of intervals. When the first hitpoint is determined, the material property and surface normal is that associated with the hitpoint. In addition, you must pay attention to precision issues because there is nothing to prevent the user from taking two objects that abut and taking an intersection. This can be made robust by eliminating any interval whose thickness is below a certain tolerance.

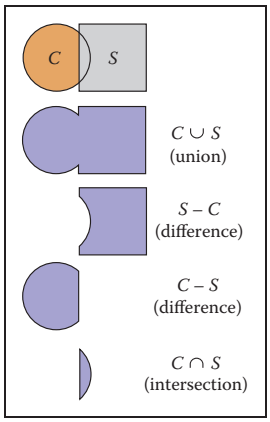


Figure 13.6. The basic CSG operations on a 2D circle and square.

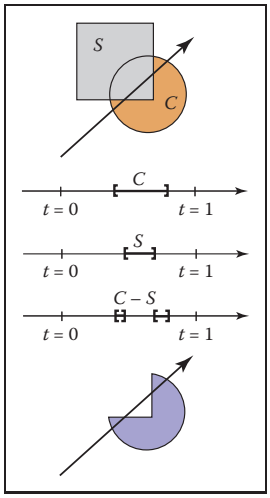


Figure 13.7. Intervals are processed to indicate how the ray hits the composite object.

13.4 Distribution Ray Tracing

For some applications, ray-traced images are just too "clean." This effect can be mitigated using *distribution ray tracing* (Cook et al., 1984). The conventionally ray-traced images look clean, because everything is crisp; the shadows are perfectly sharp, the reflections have no fuzziness, and everything is in perfect focus. Sometimes we would like to have the shadows be soft (as they are in real life), the reflections be fuzzy as with brushed metal, and the image have variable degrees of focus as in a photograph with a large aperture. While accomplishing these things from first principles is somewhat involved (as is developed in Chapter 23), we can get most of the visual impact with some fairly simple changes to the basic ray tracing algorithm. In addition, the framework gives us a relatively simple way to antialias (recall Section 8.3) the image.

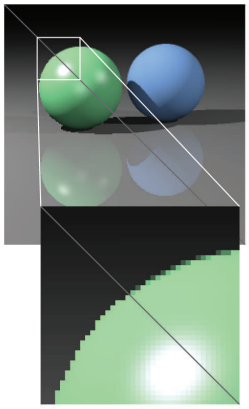


Figure 13.8. A simple scene rendered with one sample per pixel (lower left half) and nine samples per pixel (upper right half).

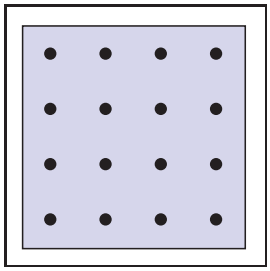


Figure 13.9. Sixteen regular samples for a single pixel.

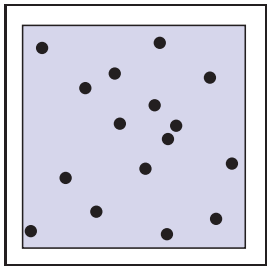


Figure 13.10. Sixteen random samples for a single pixel.

13.4.1 Antialiasing

Recall that a simple way to antialias an image is to compute the average color for the area of the pixel rather than the color at the center point. In ray tracing, our computational primitive is to compute the color at a point on the screen. If we average many of these points across the pixel, we are approximating the true average. If the screen coordinates bounding the pixel are $[i, i+1] \times [j, j+1]$, then we can replace the loop:

for each pixel (i, j) **do**
  $c_{ij} =$ ray-color$(i + 0.5, j + 0.5)$

with code that samples on a regular $n \times n$ grid of samples within each pixel:

for each pixel (i, j) **do**
  $c = 0$
  for $p = 0$ to $n - 1$ **do**
    for $q = 0$ to $n - 1$ **do**
      $c = c +$ ray-color$(i + (p + 0.5)/n, j + (q + 0.5)/n)$
  $c_{ij} = c/n^2$

This is usually called *regular sampling*. The 16 sample locations in a pixel for $n = 4$ are shown in Figure 13.9. Note that this produces the same answer as rendering a traditional ray-traced image with one sample per pixel at $n_x n$ by $n_y n$ resolution and then averaging blocks of n by n pixels to get a n_x by n_y image.

One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as moiré patterns can arise. These artifacts can be turned into noise by taking samples in a random pattern within each pixel as shown in Figure 13.10. This is usually called *random sampling* and involves just a small change to the code:

for each pixel (i, j) **do**
  $c = 0$
  for $p = 1$ to n^2 **do**
    $c = c +$ ray-color$(i + \xi, j + \xi)$
  $c_{ij} = c/n^2$

Here ξ is a call that returns a uniform random number in the range $[0, 1)$. Unfortunately, the noise can be quite objectionable unless many samples are taken. A compromise is to make a hybrid strategy that randomly perturbs a regular grid:

for each pixel (i, j) **do**
  $c = 0$
  for $p = 0$ to $n - 1$ **do**

for $q = 0$ to $n - 1$ **do**
$c = c +$ ray-color$(i + (p + \xi)/n, j + (q + \xi)/n)$
$c_{ij} = c/n^2$

That method is usually called *jittering* or *stratified sampling* (Figure 13.11).

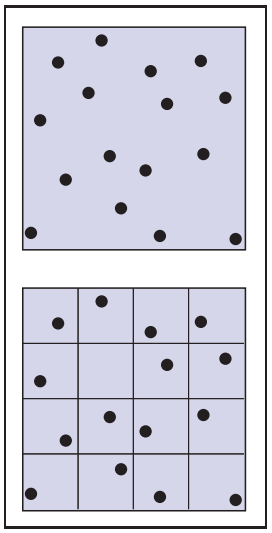


Figure 13.11. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

13.4.2 Soft Shadows

The reason shadows are hard to handle in standard ray tracing is that lights are infinitesimal points or directions and are thus either visible or invisible. In real life, lights have nonzero area and can thus be partially visible. This idea is shown in 2D in Figure 13.12. The region where the light is entirely invisible is called the *umbra*. The partially visible region is called the *penumbra*. There is not a commonly used term for the region not in shadow, but it is sometimes called the *anti-umbra*.

The key to implementing soft shadows is to somehow account for the light being an area rather than a point. An easy way to do this is to approximate the light with a distributed set of N point lights each with one Nth of the intensity of the base light. This concept is illustrated at the left of Figure 13.13 where nine lights are used. You can do this in a standard ray tracer, and it is a common trick to get soft shadows in an off-the-shelf renderer. There are two potential problems with this technique. First, typically dozens of point lights are needed to achieve visually smooth results, which slows down the program a great deal. The second problem is that the shadows have sharp transitions inside the penumbra.

Distribution ray tracing introduces a small change in the shadowing code. Instead of representing the area light at a discrete number of point sources, we represent it as an infinite number and choose one at random for each viewing ray. This amounts to choosing a random point on the light for any surface point being lit as is shown at the right of Figure 13.13.

If the light is a parallelogram specified by a corner point $\mathbf{c}$ and two edge vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 13.14), then choosing a random point $\mathbf{r}$ is straightforward:

$$\mathbf{r} = \mathbf{c} + \xi_1 \mathbf{a} + \xi_2 \mathbf{b},$$

where ξ_1 and ξ_2 are uniform random numbers in the range $[0, 1)$.

We then send a shadow ray to this point as shown at the right in Figure 13.13. Note that the direction of this ray is not unit length, which may require some modification to your basic ray tracer depending upon its assumptions.

We would really like to jitter points on the light. However, it can be dangerous to implement this without some thought. We would not want to always have the

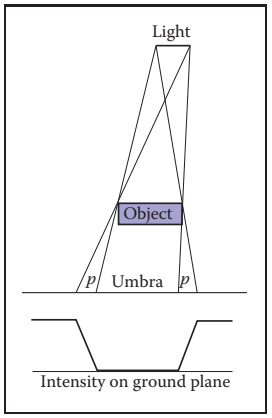


Figure 13.12. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the "penumbra" denoted by p in the figure.

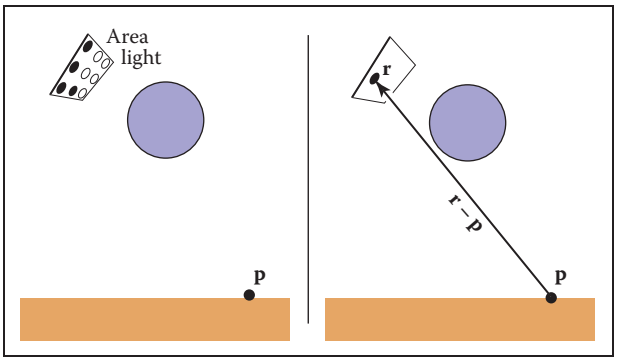


Figure 13.13. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

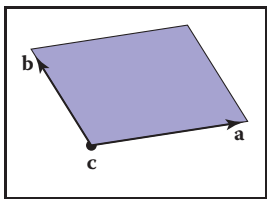


Figure 13.14. The geometry of a parallelogram light specified by a corner point and two edge vectors.

ray in the upper left-hand corner of the pixel generate a shadow ray to the upper left-hand corner of the light. Instead we would like to scramble the samples, such that the pixel samples and the light samples are each themselves jittered, but so that there is no correlation between pixel samples and light samples. A good way to accomplish this is to generate two distinct sets of n^2 jittered samples and pass samples into the light source routine:

for each pixel (i, j) **do**
  $c = 0$
  generate $N = n^2$ jittered 2D points and store in array r[ ]
  generate $N = n^2$ jittered 2D points and store in array s[ ]
  shuffle the points in array s[ ]
  for $p = 0$ to $N - 1$ **do**
    $c = c +$ ray-color$(i +$ r[p].x(), $j +$ r[p].y(), s[p]$)$
  $c_{ij} = c/N$

This shuffle routine eliminates any coherence between arrays r and s. The shadow routine will just use the 2D random point stored in $s[p]$ rather than calling the random number generator. A shuffle routine for an array indexed from 0 to $N - 1$ is:

for $i = N - 1$ downto 1 **do**
  choose random integer j between 0 and i inclusive
  swap array elements i and j

13.4.3 Depth of Field

The soft focus effects seen in most photos can be simulated by collecting light at a nonzero size "lens" rather than at a point. This is called *depth of field*. The

lens collects light from a cone of directions that has its apex at a distance where everything is in focus (Figure 13.15). We can place the "window" we are sampling on the plane where everything is in focus (rather than at the $z = n$ plane as we did previously) and the lens at the eye. The distance to the plane where everything is in focus we call the *focus plane*, and the distance to it is set by the user, just as the distance to the focus plane in a real camera is set by the user or range finder.

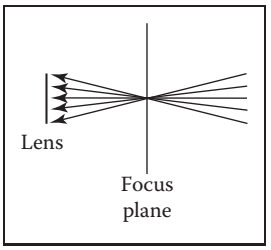


Figure 13.15. The lens averages over a cone of directions that hit the pixel location being sampled.

To be most faithful to a real camera, we should make the lens a disk. However, we will get very similar effects with a square lens (Figure 13.16). So we choose the side-length of the lens and take random samples on it. The origin of the view rays will be these perturbed positions rather than the eye position. Again, a shuffling routine is used to prevent correlation with the pixel sample positions. An example using 25 samples per pixel and a large disk lens is shown in Figure 13.17.

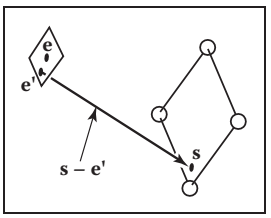


Figure 13.16. To create depth-of-field effects, the eye is randomly selected from a square region.

13.4.4 Glossy Reflection

Some surfaces, such as brushed metal, are somewhere between an ideal mirror and a diffuse surface. Some discernible image is visible in the reflection, but it is blurred. We can simulate this by randomly perturbing ideal specular reflection rays as shown in Figure 13.18.

Only two details need to be worked out: how to choose the vector $\mathbf{r}'$ and what to do when the resulting perturbed ray is below the surface from which the ray is

Figure 13.17. An example of depth of field. The caustic in the shadow of the wine glass is computed using particle tracing as described in Chapter 23.

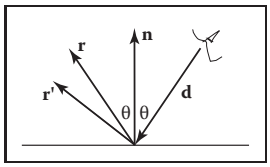


Figure 13.18. The reflection ray is perturbed to a random vector $\mathbf{r}'$.

reflected. The latter detail is usually settled by returning a zero color when the ray is below the surface.

To choose $\mathbf{r}'$, we again sample a random square. This square is perpendicular to $\mathbf{r}$ and has width a which controls the degree of blur. We can set up the square's orientation by creating an orthonormal basis with $\mathbf{w} = \mathbf{r}$ using the techniques in Section 2.4.6. Then, we create a random point in the 2D square with side length a centered at the origin. If we have 2D sample points $(\xi, \xi') \in [0, 1]^2$, then the analogous point on the desired square is

$$u = -\frac{a}{2} + \xi a,$$

$$v = -\frac{a}{2} + \xi' a.$$

Because the square over which we will perturb is parallel to both the $\mathbf{u}$ and $\mathbf{v}$ vectors, the ray $\mathbf{r}'$ is just

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

Note that $\mathbf{r}'$ is not necessarily a unit vector and should be normalized if your code requires that for ray directions.

13.4.5 Motion Blur

We can add a blurred appearance to objects as shown in Figure 13.19. This is called *motion blur* and is the result of the image being formed over a nonzero

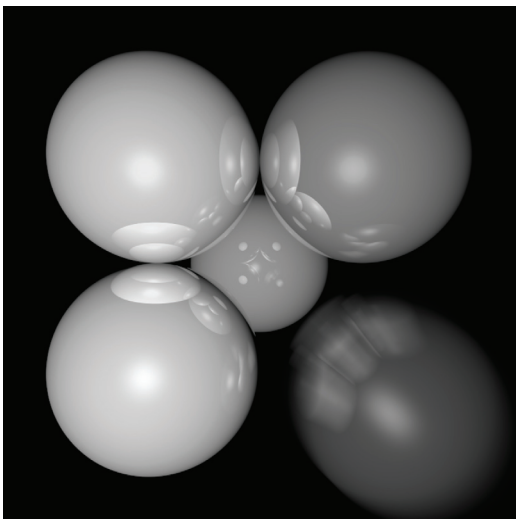


Figure 13.19. The bottom right sphere is in motion, and a blurred appearance results. *Image courtesy Chad Barb.*

span of time. In a real camera, the aperture is open for some time interval during which objects move. We can simulate the open aperture by setting a time variable ranging from T_0 to T_1. For each viewing ray we choose a random time,

$$T = T_0 + \xi(T_1 - T_0).$$

We may also need to create some objects to move with time. For example, we might have a moving sphere whose center travels from $\mathbf{c}_0$ to $\mathbf{c}_1$ during the interval. Given T, we could compute the actual center and do a ray–intersection with that sphere. Because each ray is sent at a different time, each will encounter the sphere at a different position, and the final appearance will be blurred. Note that the bounding box for the moving sphere should bound its entire path so an efficiency structure can be built for the whole time interval (Glassner, 1988).

Notes

There are many, many other advanced methods that can be implemented in the ray-tracing framework. Some resources for further information are Glassner's *An Introduction to Ray Tracing* and *Principles of Digital Image Synthesis*, Shirley's *Realistic Ray Tracing*, and Pharr and Humphreys's *Physically Based Rendering: From Theory to Implementation*.

Frequently Asked Questions

• What is the best ray-intersection efficiency structure?

The most popular structures are binary space partitioning trees (BSP trees), uniform subdivision grids, and bounding volume hierarchies. Most people who use BSP trees make the splitting planes axis-aligned, and such trees are usually called k-d trees. There is no clear-cut answer for which is best, but all are much, much better than brute-force search in practice. If I were to implement only one, it would be the bounding volume hierarchy because of its simplicity and robustness.

• Why do people use bounding boxes rather than spheres or ellipsoids?

Sometimes spheres or ellipsoids are better. However, many models have polygonal elements that are tightly bounded by boxes, but they would be difficult to tightly bind with an ellipsoid.