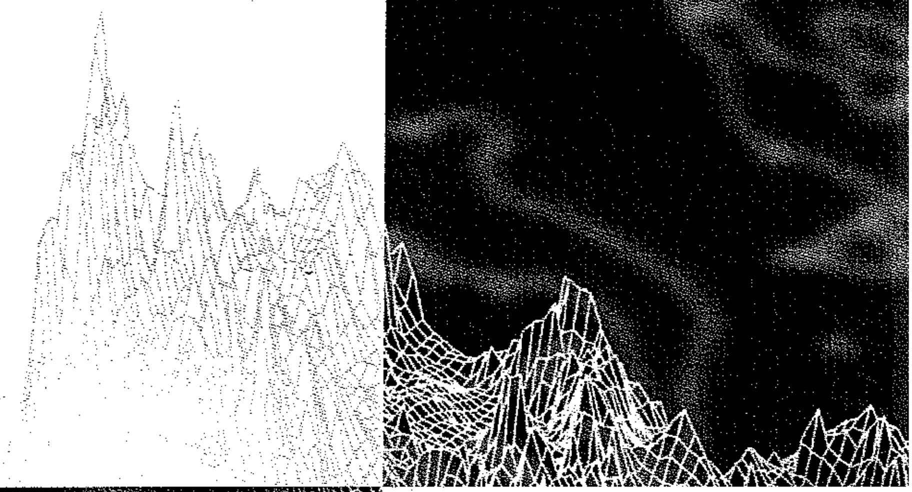
M. de Berg
M. van Kreveld
M. Overmars
O. Schwarzkopf



Computational Geometry

Algorithms and Applications

Second Edition

Springer

12 Binary Space Partitions

The Painter's Algorithm

These days pilots no longer have their first flying experience in the air, but on the ground in a flight simulator. This is cheaper for the air company, safer for the pilot, and better for the environment. Only after spending many hours in the simulator are pilots allowed to operate the control stick of a real airplane. Flight simulators must perform many different tasks to make the pilot forget that she is sitting in a simulator. An important task is visualization: pilots must be able

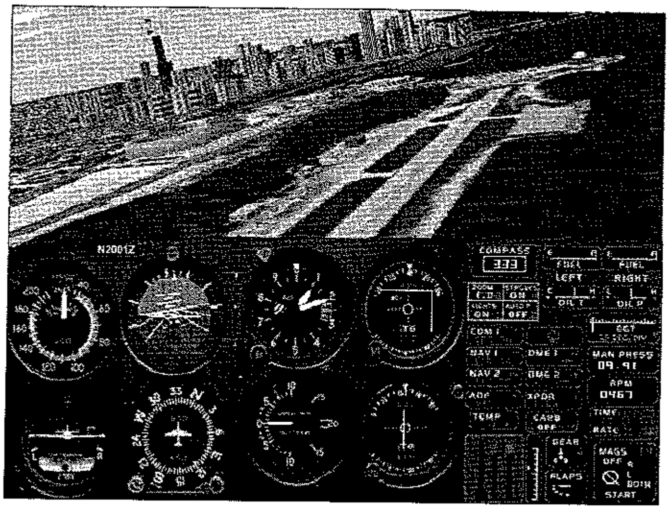


Figure 12.1
The Microsoft flight simulator for Windows 95

to see the landscape above which they are flying, or the runway on which they are landing. This involves both modeling landscapes and rendering the models. To render a scene we must determine for each pixel on the screen the object that is visible at that pixel; this is called *hidden surface removal*. We must also perform shading calculations, that is, we must compute the intensity of the light that the visible object emits in the direction of the view point. The latter task is very time-consuming if highly realistic images are desired: we must compute how much light reaches the object—either directly from light sources or indirectly via reflections on other objects—and consider the interaction of the light with the surface of the object to see how much of it is reflected in the direction of the view point. In flight simulators rendering must be performed in real-time, so there is no time for accurate shading calculations. Therefore a fast and simple shading technique is employed and hidden surface removal becomes an important factor in the rendering time.

The z-*buffer algorithm* is a very simple method for hidden surface removal. This method works as follows. First, the scene is transformed such that the viewing direction is the positive z-direction. Then the objects in the scene are scan-converted in arbitrary order. Scan-converting an object amounts to determining which pixels it covers in the projection; these are the pixels where the object is potentially visible. The algorithm maintains information about the already processed objects in two buffers: a frame buffer and a z-buffer. The frame buffer stores for each pixel the intensity of the currently visible object, that is, the object that is visible among those already processed. The z-buffer stores for each pixel the z-coordinate of the currently visible object. (More precisely, it stores the z-coordinate of the point on the object that is visible at the pixel.) Now suppose that we select a pixel when scan-converting an object. If the z-coordinate of the object at that pixel is smaller than the z-coordinate stored in the z-buffer, then the new object lies in front of the currently visible object. So we write the intensity of the new object to the frame buffer, and its z-coordinate to the z-buffer. If the z-coordinate of the object at that pixel is larger than the z-coordinate stored in the z-buffer, then the new object is not visible, and the frame buffer and z-buffer remain unchanged. The z-buffer algorithm is easily implemented in hardware and quite fast in practice. Hence, this is the most popular hidden surface removal method. Nevertheless, the algorithm has

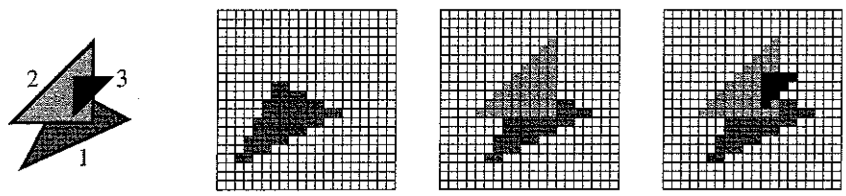
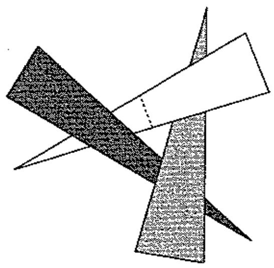


Figure 12.2
The painter's algorithm in action

some disadvantages: a large amount of extra storage is needed for the z-buffer, and an extra test on the z-coordinate is required for every pixel covered by an object. The *painter's algorithm* avoids these extra costs by first sorting the objects according to their distance to the view point. Then the objects are scan-converted in this so-called *depth order*, starting with the object farthest from the view point. When an object is scan-converted we do not need to perform any test on its z-coordinate, we always write its intensity to the frame buffer. Entries in the frame buffer that have been filled before are simply overwritten. Figure 12.2 illustrates the algorithm on a scene consisting of three triangles. On the left, the triangles are shown with numbers corresponding to the order in which they are scan-converted. The images after the first, second, and third triangle have been scan-converted are shown as well. This approach is correct because we scan-convert the objects in back-to-front order: for each pixel the last object written to the corresponding entry in the frame buffer will be the one closest to the viewpoint, resulting in a correct view of the scene. The process resembles the way painters work when they put layers of paint on top of each other, hence the name of the algorithm.

To apply this method successfully we must be able to sort the objects quickly. Unfortunately this is not so easy. Even worse, a depth order may not always exist: the in-front-of relation among the objects can contain cycles. When such a *cyclic overlap* occurs, no ordering will produce a correct view of this scene. In this case we must break the cycles by splitting one or more of the objects, and hope that a depth order exists for the pieces that result from the splitting. When there is a cycle of three triangles, for instance, we can always split one of them into a triangular piece and a quadrilateral piece, such that a correct displaying order exists for the resulting set of four objects. Computing which objects to split, where to split them, and then sorting the object fragments is an expensive process. Because the order depends on the position of the view point, we must recompute the order every time the view point moves. If we want to use the painter's algorithm in a real-time environment such as flight simulation, we should preprocess the scene such that a correct displaying order can be found quickly for any view point. An elegant data structure that makes this possible is the binary space partition tree, or BSP tree for short.



12.1 The Definition of BSP Trees

To get a feeling for what a BSP tree is, take a look at Figure 12.3. This figure shows a binary space partition (BSP) for a set of objects in the plane, together with the tree that corresponds to the BSP. As you can see, the binary space partition is obtained by recursively splitting the plane with a line: first we split the entire plane with ℓ_1, then we split the half-plane above ℓ_1 with ℓ_2 and the half-plane below ℓ_1 with ℓ_3, and so on. The splitting lines not only partition the plane, they may also cut objects into fragments. The splitting continues until there is only one fragment left in the interior of each region. This process is

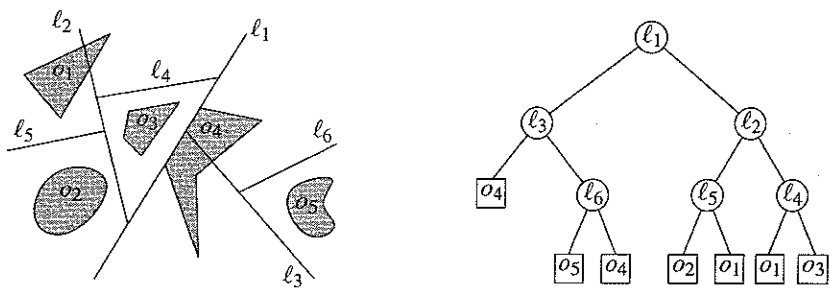


Figure 12.3
A binary space partition and the corresponding tree

naturally modeled as a binary tree. Each leaf of this tree corresponds to a face of the final subdivision; the object fragment that lies in the face is stored at the leaf. Each internal node corresponds to a splitting line; this line is stored at the node. When there are 1-dimensional objects (line segments) in the scene then objects could be contained in a splitting line; in that case the corresponding internal node stores these objects in a list.

For a hyperplane $h : a_1x_1 + a_2x_2 + \cdots + a_dx_d + a_{d+1} = 0$, we let h^+ be the open positive half-space bounded by h and we let h^- be the open negative half-space:

$$h^+ := \{(x_1, x_2, \ldots, x_d) : a_1x_1 + a_2x_2 + \cdots + a_dx_d + a_{d+1} > 0\}$$

and

$$h^- := \{(x_1, x_2, \ldots, x_d) : a_1x_1 + a_2x_2 + \cdots + a_dx_d + a_{d+1} < 0\}.$$

A binary space partition tree, or BSP tree, for a set S of objects in d-dimensional space is now defined as a binary tree $\mathcal{T}$ with the following properties:

- If $\mathrm{card}(S) \leqslant 1$ then $\mathcal{T}$ is a leaf; the object fragment in S (if it exists) is stored explicitly at this leaf. If the leaf is denoted by ν, then the (possibly empty) set stored at the leaf is denoted by $S(\nu)$.
- If $\mathrm{card}(S) > 1$ then the root ν of $\mathcal{T}$ stores a hyperplane h_ν, together with the set $S(\nu)$ of objects that are fully contained in h_ν. The left child of ν is the root of a BSP tree $\mathcal{T}^-$ for the set $S^- := \{h_\nu^- \cap s : s \in S\}$, and the right child of ν is the root of a BSP tree $\mathcal{T}^+$ for the set $S^+ := \{h_\nu^+ \cap s : s \in S\}$.

The *size* of a BSP tree is the total size of the sets $S(\nu)$ over all nodes ν of the BSP tree. In other words, the size of a BSP tree is the total number of object fragments that are generated. If the BSP does not contain useless splitting lines—lines that split off an empty subspace—then the number of nodes of the tree is at most linear in the size of the BSP tree. Strictly speaking, the size of the BSP tree does not say anything about the amount of storage needed to store it, because it says nothing about the amount of storage needed for a single fragment. Nevertheless, the size of a BSP tree as we defined it is a good measure to compare the quality of different BSP trees for a given set of objects.

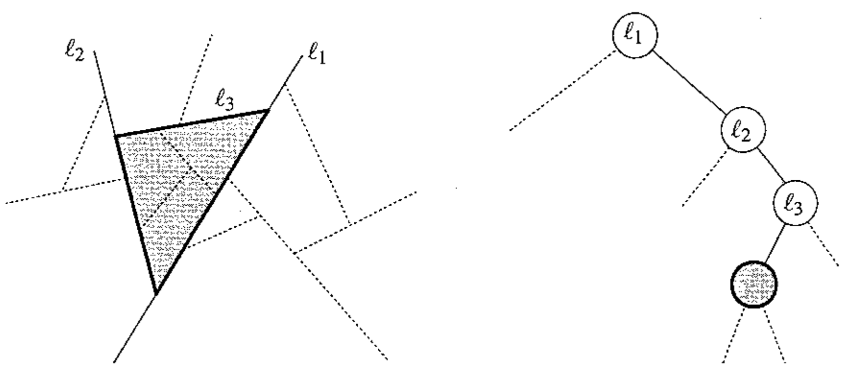
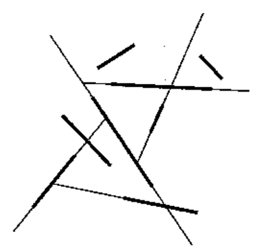


Figure 12.4
The correspondence between nodes and regions

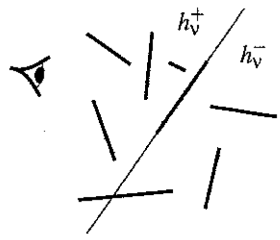
The leaves in a BSP tree represent the faces in the subdivision that the BSP induces. More generally, we can identify a convex region with each node ν in a BSP tree $\mathcal{T}$: this region is the intersection of the half-spaces $h_\mu^\diamond$, where μ is an ancestor of ν and $\diamond = -$ when ν is in the left subtree of μ, and $\diamond = +$ when it is in the right subtree. The region corresponding to the root of $\mathcal{T}$ is the whole space. Figure 12.4 illustrates this: the grey node corresponds to the grey region $\ell_1^+ \cap \ell_2^+ \cap \ell_3^-$.

The splitting hyperplanes used in a BSP can be arbitrary. For computational purposes, however, it can be convenient to restrict the set of allowable splitting hyperplanes. A usual restriction is the following. Suppose we want to construct a BSP for a set of line segments in the plane. An obvious set of candidates for the splitting lines is the set of extensions of the input segments. A BSP that only uses such splitting lines is called an *auto-partition*. For a set of planar polygons in 3-space, an auto-partition is a BSP that only uses planes through the input polygons as splitting planes. It seems that the restriction to auto-partitions is a severe one. But, although auto-partitions cannot always produce minimum-size BSP trees, we shall see that they can produce reasonably small ones.



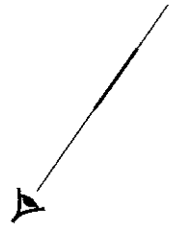
12.2 BSP Trees and the Painter's Algorithm

Suppose we have built a BSP tree $\mathcal{T}$ on a set S of objects in 3-dimensional space. How can we use $\mathcal{T}$ to get the depth order we need to display the set S with the painter's algorithm? Let p_{view} be the view point and suppose that p_{view} lies above the splitting plane stored at the root of $\mathcal{T}$. Then clearly none of the objects below the splitting plane can obscure any of the objects above it. Hence, we can safely display all the objects (more precisely, object fragments) in the subtree $\mathcal{T}^-$ before displaying those in $\mathcal{T}^+$. The order for the object fragments in the two subtrees $\mathcal{T}^+$ and $\mathcal{T}^-$ is obtained recursively in the same way. This is summarized in the following algorithm.



Algorithm PAINTERSALGORITHM($\mathcal{T}, p_{\text{view}}$)

```
Let ν be the root of T.
if ν is a leaf
   then Scan-convert the object fragments in S(ν).
   else if p_view ∈ h_ν^+
           then PAINTERSALGORITHM(T^-, p_view)
                Scan-convert the object fragments in S(ν).
                PAINTERSALGORITHM(T^+, p_view)
           else if p_view ∈ h_ν^-
                   then PAINTERSALGORITHM(T^+, p_view)
                        Scan-convert the object fragments in S(ν).
                        PAINTERSALGORITHM(T^-, p_view)
                   else (* p_view ∈ h_ν *)
                        PAINTERSALGORITHM(T^+, p_view)
                        PAINTERSALGORITHM(T^-, p_view)
```

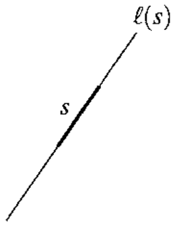


Note that we do not draw the polygons in $S(\nu)$ when p_{view} lies on the splitting plane h_ν, because polygons are flat 2-dimensional objects and therefore not visible from points that lie in the plane containing them.

The efficiency of this algorithm—indeed, of any algorithm that uses BSP trees—depends largely on the size of the BSP tree. So we must choose the splitting planes in such a way that fragmentation of the objects is kept to a minimum. Before we can develop splitting strategies that produce small BSP trees, we must decide on which types of objects we allow. We became interested in BSP trees because we needed a fast way of doing hidden surface removal for flight simulators. Because speed is our main concern, we should keep the type of objects in the scene simple: we should not use curved surfaces, but represent everything in a polyhedral model. We assume that the facets of the polyhedra have been triangulated. So we want to construct a BSP tree of small size for a given set of triangles in 3-dimensional space.

12.3 Constructing a BSP Tree

When you want to solve a 3-dimensional problem, it is usually not a bad idea to gain some insight by first studying the planar version of the problem. This is also what we do in this section.



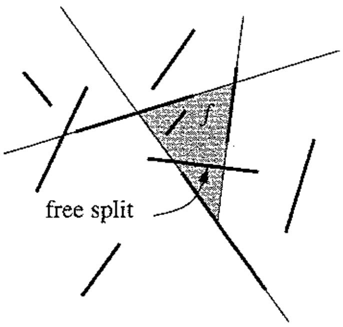
Let S be a set of n non-intersecting line segments in the plane. We will restrict our attention to auto-partitions, that is, we only consider lines containing one of the segments in S as candidate splitting lines. The following recursive algorithm for constructing a BSP immediately suggests itself. Let $\ell(s)$ denote the line that contains a segment s.

Algorithm 2DBSP(S)
Input. A set $S = \{s_1, s_2, \dots, s_n\}$ of segments.
Output. A BSP tree for S.
1. **if** card(S) $\leqslant 1$
2. **then** Create a tree $\mathcal{T}$ consisting of a single leaf node, where the set S is stored explicitly.
3. **return** $\mathcal{T}$
4. **else** (* Use $\ell(s_1)$ as the splitting line. *)
5. $S^+ \leftarrow \{s \cap \ell(s_1)^+ : s \in S\}$; $\mathcal{T}^+ \leftarrow$2DBSP(S^+)
6. $S^- \leftarrow \{s \cap \ell(s_1)^- : s \in S\}$; $\mathcal{T}^- \leftarrow$2DBSP(S^-)
7. Create a BSP tree $\mathcal{T}$ with root node ν, left subtree $\mathcal{T}^-$, right subtree $\mathcal{T}^+$, and with $S(\nu) = \{s \in S : s \subset \ell(s_1)\}$.
8. **return** $\mathcal{T}$

The algorithm clearly constructs a BSP tree for the set S. But is it a small one? Perhaps we should spend a little more effort in choosing the right segment to do the splitting, instead of blindly taking the first segment, s_1. One approach that comes to mind is to take the segment $s \in S$ such that $\ell(s)$ cuts as few segments as possible. But this is too greedy: there are configurations of segments where this approach doesn't work well. Furthermore, finding this segment would be time consuming. What else can we do? Perhaps you already guessed: as in previous chapters where we had to make a difficult choice, we simply make a random choice. That is to say, we use a random segment to do the splitting. As we shall see later, the resulting BSP is expected to be fairly small.

To implement this, we put the segments in random order before we start the construction:

Algorithm 2DRANDOMBSP(S)
1. Generate a random permutation $S' = s_1, \dots, s_n$ of the set S.
2. $\mathcal{T} \leftarrow$2DBSP(S')
3. **return** $\mathcal{T}$



Before we analyze this randomized algorithm, we note that one simple optimization is possible. Suppose that we have chosen the first few partition lines. These lines induce a subdivision of the plane whose faces correspond to nodes in the BSP tree that we are constructing. Consider one such face f. There can be segments that cross f completely. Selecting one of these crossing segments to split f will not cause any fragmentation of other segments inside f, while the segment itself can be excluded from further consideration. It would be foolish not to take advantage of such *free splits*. So our improved strategy is to make free splits whenever possible, and to use random splits otherwise. To implement this optimization, we must be able to tell whether a segment is a free split. To this end we maintain two boolean variables with each segment, which indicate whether the left and right endpoint lie on one of the already added splitting lines. When both variables become true, then the segment is a free split.

We now analyze the performance of algorithm 2DRANDOMBSP. To keep it simple, we will analyze the version without free splits. (In fact, free splits do not make a difference asympotically.)

We start by analyzing the size of the BSP tree or, in other words, the number of fragments that are generated. Of course, this number depends heavily on the particular permutation generated in line 1: some permutations may give small BSP trees, while others give very large ones. As an example, consider the

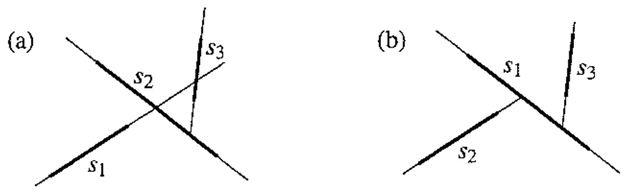


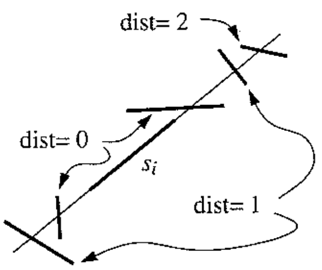
Figure 12.5
Different orders give different BSPs

collection of three segments depicted in Figure 12.5. If the segments are treated as illustrated in part (a) of the figure, then five fragments result. A different order, however, gives only three fragments, as shown in part (b). Because the size of the BSP varies with the permutation that is used, we will analyze the *expected* size of the BSP tree, that is, the average size over all $n!$ permutations.

Lemma 12.1 *The expected number of fragments generated by the algorithm* 2DRANDOMBSP *is* $O(n \log n)$.

Proof. Let s_i be a fixed segment in S. We shall analyze the expected number of other segments that are cut when $\ell(s_i)$ is added by the algorithm as the next splitting line.

In Figure 12.5 we can see that whether or not a segment s_j is cut when $\ell(s_i)$ is added—assuming it can be cut at all by $\ell(s_i)$—depends on segments that are



also cut by $\ell(s_i)$ and are 'in between' s_i and s_j. In particular, when the line through such a segment is used before $\ell(s_i)$, then it shields s_j from s_i. This is what happened in Figure 12.5(b): the segment s_1 shielded s_3 from s_2. These considerations lead us to define the distance of a segment with respect to the fixed segment s_i:

$$\mathrm{dist}_{s_i}(s_j) = \begin{cases} \text{the number of segments intersecting } \ell(s_i) \text{ in between } s_i \text{ and } s_j & \text{if } \ell(s_i) \text{ intersects } s_j \\ +\infty & \text{otherwise} \end{cases}$$

For any finite distance, there are at most two segments at that distance, one on either side of s_i.

Let $k := \mathrm{dist}_{s_i}(s_j)$, and let $s_{j_1}, s_{j_2}, \ldots, s_{j_k}$ be the segments in between s_i and s_j. What is the probability that $\ell(s_i)$ cuts s_j when added as a splitting line? For this to happen, s_i must come before s_j in the random ordering and, moreover, it must come before any of the segments in between s_i and s_j, which shield s_j from s_i. In other words, of the set $\{i, j, j_1, \ldots, j_k\}$ of indices, i must be the smallest one. Because the order of the segments is random, this implies

$$\Pr[\ell(s_i) \text{ cuts } s_j] \leqslant \frac{1}{\mathrm{dist}_{s_i}(s_j) + 2}.$$

Notice that there can be segments that are not cut by $\ell(s_i)$ but whose *extension* shields s_j. This explains why the expression above is not an equality.

We can now bound the expected total number of cuts generated by s_i:

$$\begin{aligned} \mathrm{E}[\text{number of cuts generated by } s_i] &\leqslant \sum_{j \neq i} \frac{1}{\mathrm{dist}_{s_i}(s_j) + 2} \\ &\leqslant 2 \sum_{k=0}^{n-2} \frac{1}{k+2} \\ &\leqslant 2 \ln n. \end{aligned}$$

By linearity of expectation, we can conclude that the expected total number of cuts generated by all segments is at most $2n \ln n$. Since we start with n segments, the expected total number of fragments is bounded by $n + 2n \ln n$. ⊟

We have shown that the expected size of the BSP that is generated by 2DRANDOMBSP is $n + 2n \ln n$. As a consequence, we have proven that a BSP of size $n + 2n \ln n$ *exists* for any set of n segments. Furthermore, at least half of all permutations lead to a BSP of size $n + 4n \ln n$. We can use this to find a BSP of that size: After running 2DRANDOMBSP we test the size of the tree, and if it exceeds that bound, we simply start the algorithm again with a fresh random permutation. The expected number of trials is two.

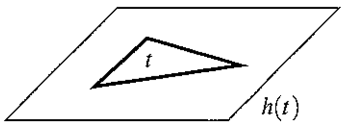
We have analyzed the size of the BSP that is produced by 2DRANDOMBSP. What about the running time? Again, this depends on the random permutation that is used, so we look at the expected running time. Computing the random permutation takes linear time. If we ignore the time for the recursive calls, then the time taken by algorithm 2DBSP is linear in the number of fragments in S. This number is never larger than n—in fact, it gets smaller with each recursive call. Finally, the number of recursive calls is obviously bounded by the total number of generated fragments, which is $O(n \log n)$. Hence, the total construction time is $O(n^2 \log n)$, and we get the following result.

Theorem 12.2 *A BSP of size $O(n \log n)$ can be computed in expected time $O(n^2 \log n)$.*

Although the expected size of the BSP that is constructed by 2DRANDOMBSP is fairly good, the running time of the algorithm is somewhat disappointing. In many applications this is not so important, because the construction is done off-line. Moreover, the construction time is only quadratic when the BSP is very unbalanced, which is rather unlikely to occur in practice. Nevertheless, from a theoretical point of view the construction time is disappointing. Using an approach based on segment trees—see Chapter 10—this can be improved: one can construct a BSP of size $O(n \log n)$ in $O(n \log n)$ time with a deterministic algorithm. This approach does not give an auto-partition, however, and in practice it produces BSPs that are slightly larger.

A natural question is whether it is also possible to improve the size of the BSP generated by 2DRANDOMBSP: is there an $O(n)$ size BSP for any set of segments in the plane, or are there perhaps sets for which any BSP must have size $\Omega(n \log n)$? The answer to this question is currently unknown.

The algorithm we described for the planar case immediately generalizes to 3-dimensional space. Let S be a set of n non-intersecting triangles in $\mathbb{R}^3$. Again we restrict ourselves to auto-partitions, that is, we only use partition planes that contain a triangle of S. For a triangle t we denote the plane containing it by $h(t)$.



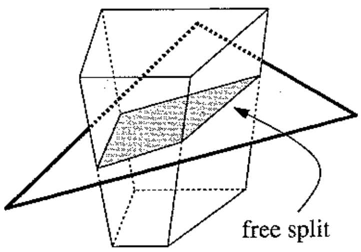
Algorithm 3DBSP(S)
Input. A set $S = \{t_1, t_2, \ldots, t_n\}$ of triangles in $\mathbb{R}^3$.
Output. A BSP tree for S.

if card(S) $\leqslant 1$
  then Create a tree $\mathcal{T}$ consisting of a single leaf node, where the set S is stored explicitly.
    return $\mathcal{T}$
  else (* Use $h(t_1)$ as the splitting plane. *)
    $S^+ \leftarrow \{t \cap h(t_1)^+ : t \in S\}$; $\mathcal{T}^+ \leftarrow$ 3DBSP(S^+)
    $S^- \leftarrow \{t \cap h(t_1)^- : t \in S\}$; $\mathcal{T}^- \leftarrow$ 3DBSP(S^-)
    Create a BSP tree $\mathcal{T}$ with root node ν, left subtree $\mathcal{T}^-$, right subtree $\mathcal{T}^+$, and with $S(\nu) = \{t \in T : t \subset h(t_1)\}$.
    return $\mathcal{T}$

The size of the resulting BSP again depends on the order of the triangles; some orders give more fragments than others. As in the planar case, we can try to get a good expected size by first putting the triangles in a random order.

This usually gives a good result in practice. However, it is not known how to analyze the expected behavior of this algorithm theoretically. Therefore we will analyze a variant of the algorithm in the next section, although the algorithm described above is probably superior in practice.

12.4* The Size of BSP Trees in 3-Space



The randomized algorithm for constructing a BSP tree in 3-space that we analyze in this section is almost the same as the improved algorithm described above: it treats the triangles in random order, and it makes free splits whenever possible. A free split now occurs when a triangle of S splits a cell into two disconnected subcells. The only difference is that when we use some plane $h(t)$ as a splitting plane, we use it in all cells intersected by that plane, not just in the cells that are intersected by t. (And therefore a simple recursive implementation is no longer possible.) There is one exception to the rule that we split all cells with $h(t)$: when the split is completely useless for a cell, because all the triangles in that cell lie completely to one side of it, then we do not split it.

Figure 12.6 illustrates this on a 2-dimensional example. In part (a) of the figure, the subdivision is shown that is generated by the algorithm of the previous section after treating segments s_1, s_2, and s_3 (in that order). In part (b) the subdivision is shown as generated by the modified algorithm. Note that the modified algorithm uses $\ell(s_2)$ as a splitting line in the subspace below $\ell(s_1)$, and that $\ell(s_3)$ is used as a splitting line in the subspace to the right of $\ell(s_2)$. The line $\ell(s_3)$ is not used in the subspace between $\ell(s_1)$ and $\ell(s_2)$, however, because it is useless there.

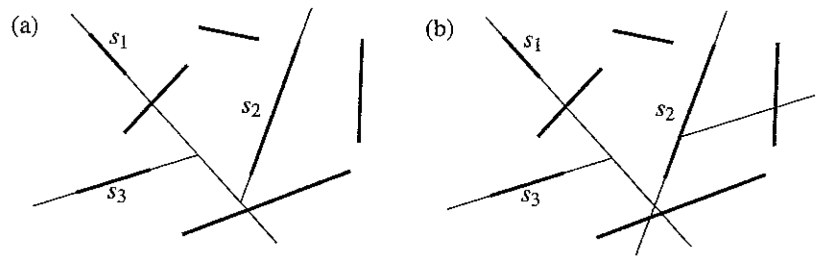


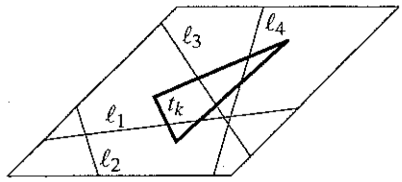
Figure 12.6
The original and the modified algorithm

The modified algorithm can be summarized as follows. Working out the details is left as an exercise.

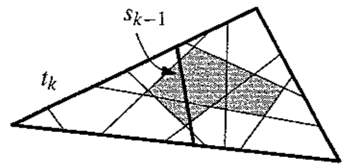
Algorithm 3DRANDOMBSP2(S)
Input. A set $S = \{t_1, t_2, \dots, t_n\}$ of triangles in $\mathbb{R}^3$.
Output. A BSP tree for S.
1. Generate a random permutation $t_1, \dots, t_n$ of the set S.
2. **for** $i \leftarrow 1$ **to** n
3.   **do** Use $h(t_i)$ to split every cell where the split is useful.
4.   Make all possible free splits.

The next lemma analyzes the expected number of fragments generated by the algorithm.

Lemma 12.3 *The expected number of object fragments generated by algorithm* 3DRANDOMBSP2 *over all $n!$ possible permutations is $O(n^2)$.*



Proof. We shall prove a bound on the expected number of fragments into which a fixed triangle $t_k \in S$ is cut. For a triangle t_i with $i < k$ we define $\ell_i := h(t_i) \cap h(t_k)$. The set $L := \{\ell_1, \dots, \ell_{k-1}\}$ is a set of at most $k-1$ lines lying in the plane $h(t_k)$. Some of these lines intersect t_k, others miss t_k. For a line ℓ_i that intersects t_k we define $s_i := \ell_i \cap t_k$. Let I be the set of all such intersections s_i. Due to free splits the number of fragments into which t_k is cut is in general *not* simply the number of faces in the arrangement that I induces on t_k. To understand this, consider the moment that t_{k-1} is treated. Assume that ℓ_{k-1} intersects t_k; otherwise t_{k-1} does not cause any fragmentation on t_k. The segment s_{k-1} can intersect several of the faces of the arrangement on t_k induced by $I \setminus \{s_k\}$. If, however, such a face f is not incident to the one of the edges of t_k—we call f an interior face—then a free split already has been made through this part of t_k. In other words, $h(t_{k-1})$ only causes cuts in exterior faces, that is, faces that are incident to one of the three edges of t_k. Hence, the number of splits on t_k caused by $h(t_{k-1})$ equals the number of edges that s_{k-1} contributes to exterior faces of the arrangement on t_k induced by I. (In the analysis that follows, it is important that the collection of exterior faces is independent of the order in which $t_1, \dots, t_{k-1}$ have been treated. This is not the case for the algorithm in the previous section, which is the reason for the modification.) What is the expected number of such edges? To answer this question we first bound the total number of edges of the exterior faces.



In Chapter 8 we defined the *zone* of a line ℓ in an arrangement of lines in the plane as the set of faces of the arrangement intersected by ℓ. You may recall that for an arrangement of m lines the complexity of the zone is $O(m)$. Now let e_1, e_2, and e_3 be the edges of t_k and let $\ell(e_i)$ be the line through e_i, for $i = 1,2,3$. The edges that we are interested in must be in the zone of either $\ell(e_1)$, $\ell(e_2)$, or $\ell(e_3)$ in the arrangement induced by the set L on the plane $h(t_k)$. Hence, the total number of edges of exterior faces is $O(k)$.

If the total number of edges of exterior faces is $O(k)$, then the average number of edges lying on a segment s_i is $O(1)$. Because $t_1, \dots, t_n$ is a random permutation, so is $t_1, \dots, t_{k-1}$. Hence, the expected number of edges on segment s_{k-1} is constant, and therefore the expected number of extra fragments on t_k caused by $h(t_{k-1})$ is $O(1)$. The same argument shows that the expected number of fragmentations on t_k generated by each of the splitting planes $h(t_1)$ through $h(t_{k-2})$ is constant. This implies that the expected number of fragments into which t_k is cut is $O(k)$. The total number of fragments is therefore

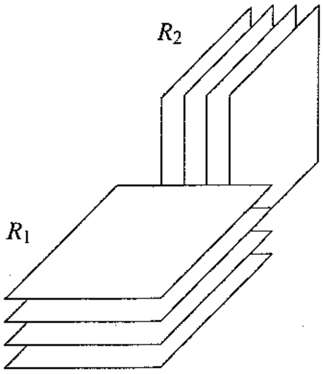
$$O(\sum_{k=1}^{n} k) = O(n^2).$$

$\square$

The quadratic bound on the expected size of the partitioning generated by 3DRANDOMBSP immediately proves that a BSP tree of quadratic size exists.

You may be a bit disappointed by the bound that we have achieved. A quadratic size BSP tree is not what you are hoping for when you have a set of 10,000 triangles. The following theorem tells us that we cannot hope to prove anything better if we restrict ourselves to auto-partitions.

Lemma 12.4 *There are sets of n non-intersecting triangles in 3-space for which any auto-partition has size $\Omega(n^2)$.*



Proof. Consider a collection of rectangles consisting of a set R_1 of rectangles parallel to the xy-plane and a set R_2 of rectangles parallel to the yz-plane, as illustrated in the margin. (The example also works with a set of triangles, but with rectangles it is easier to visualize.) Let $n_1 := \text{card}(R_1)$, let $n_2 := \text{card}(R_2)$, and let $G(n_1,n_2)$ be the minimum size of an auto-partition for such a configuration. We claim that $G(n_1,n_2) = (n_1+1)(n_2+1)-1$. The proof is by induction on n_1+n_2. The claim is obviously true for $G(1,0)$ and $G(0,1)$, so now consider the case where $n_1+n_2 > 1$. Without loss of generality, assume that the auto-partition chooses a rectangle r from the set R_1. The plane through r will split all the rectangles in R_2. Moreover, the configurations in the two subscenes that must be treated recursively have exactly the same form as the initial configuration. If m denotes the number of rectangles of R_1 lying above r, then we have

$$\begin{aligned} G(n_1,n_2) &= 1+G(m,n_2)+G(n_1-m-1,n_2) \\ &= 1+((m+1)(n_2+1)-1)+((n_1-m)(n_2+1)-1) \\ &= (n_1+1)(n_2+1)-1. \end{aligned}$$

$\square$

So perhaps we should not restrict ourselves to auto-partitions. In the lower bound in the proof of Lemma 12.4 the restriction to auto-partitions is definitely a bad idea: we have shown that such a partition necessarily has quadratic size,

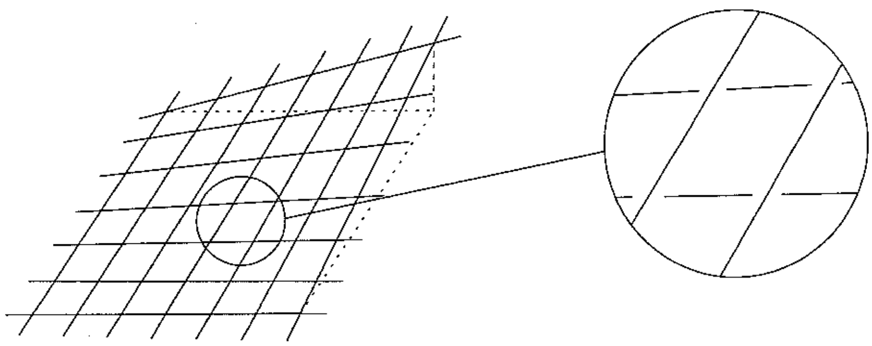


Figure 12.7
The general lower bound construction

whereas we can easily get a linear BSP if we first separate the set R_1 from the set R_2 with a plane parallel to the xz-plane. But even unrestricted partitions fail to give a small BSP for the configuration of Figure 12.7. This configuration is obtained as follows. We start by taking a grid in the plane made up of $n/2$

lines parallel to the x-axis and $n/2$ lines parallel to the y-axis. (Instead of the lines we could also take very long and skinny triangles.) We skew these lines a little to get the configuration of Figure 12.7; the lines now lie on a so-called hyperbolic paraboloid. Finally we move the lines parallel to the y-axis slightly upward so that the lines no longer intersect. What we get is the set of lines

$$\{y = i, z = ix : 1 \leqslant i \leqslant n/2\} \cup \{x = i, z = iy + \varepsilon : 1 \leqslant i \leqslant n/2\},$$

where ε is a small positive constant. If ε is sufficiently small then any BSP must cut at least one of the four lines that bound a grid cell in the immediate neighborhood of that cell. The formal proof of this fact is elementary, but tedious and not very instructive. The idea is to show that the lines are skewed in such a way that no plane fits simultaneously through the four "openings" at its corners. Since there is a quadratic number of grid cells, this will result in $\Theta(n^2)$ fragments.

Theorem 12.5 *For any set of n non-intersecting triangles in $\mathbb{R}^3$ a BSP tree of size $O(n^2)$ exists. Moreover, there are configurations for which the size of any BSP is $\Omega(n^2)$.*

The quadratic lower bound on the size of BSP trees might give you the idea that they are useless in practice. Fortunately this is not the case. The configurations that yield the lower bound are quite artificial. In many practical situations BSP trees perform just fine.

12.5 Notes and Comments

BSP trees are popular in many application areas, in particular in computer graphics. The application mentioned in this chapter is to perform hidden surface removal with the painter's algorithm [157]. Other application include shadow generation [102], set operations on polyhedra [260, 326], and visibility preprocessing for interactive walkthroughs [325]. They have also been used for cell decomposition methods in motion planning [24].

The study of BSP trees from a theoretical point of view was initiated by Paterson and Yao [283]; the results of this chapter come from their paper. They also proved bounds on BSPs in higher dimensions: any set of $(d-1)$-dimensional simplices in $\mathbb{R}^d$, with $d \geqslant 3$, admits a BSP of size $O(n^{d-1})$.

Paterson and Yao also studied the special case of line segments in the plane that are all either horizontal or vertical. They showed that in such a case a linear BSP is possible. The same result was achieved by D'Amore and Franciosa [116]. Paterson and Yao generalized the results to orthogonal objects in higher dimensions [284]. For instance, they proved that any set of orthogonal rectangles in $\mathbb{R}^3$ admits a BSP of size $O(n\sqrt{n})$, and that this bound is tight in the worst case.

The discrepancy between the quadratic lower bound on the worst-case size of a BSP in 3-dimensional space and their size in practice led de Berg et al. [39]

to study BSPs for scenes with special properties. In particular they showed that any set of *fat objects* in the plane—which means that long and skinny objects are forbidden—admits a BSP of linear size. A set of line segments where the ratio between the length of the longest and the shortest one is bounded admits a linear size BSP as well. These results strengthen the belief that any set of segments in the plane should admit a linear size BSP, but the best known worst-case bound is still the $O(n \log n)$ bound of Paterson and Yao. The result on fat objects was extended to higher dimensions and to a larger class of objects by de Berg [37].

Finally, we remark that two other well-known structures, kd-trees and quad trees, are in fact special cases of BSP trees, where only orthogonal splitting planes are used. Kd-trees were discussed extensively in Chapter 5 and quad trees will be discussed in Chapter 14.

12.6 Exercises

12.1 Prove that PAINTERSALGORITHM is correct. That is, prove that if (some part of) an object A is scan-converted before (some part of) object B is scan-converted, then A cannot lie in front of B.

12.2 Let S be a set of m polygons in the plane with n vertices in total. Let $\mathcal{T}$ be a BSP tree for S of size k. Prove that the total complexity of the fragments generated by the BSP is $O(n+k)$.

12.3 Give an example of a set of line segments in the plane where the greedy method of constructing an auto-partition (where the splitting line $\ell(s)$ is taken that induces the least number of cuts) results in a BSP of quadratic size.

12.4 Give an example of a set S of n non-intersecting line segments in the plane for which a BSP tree of size n exists, whereas any auto-partition of S has size at least $\lfloor 4n/3 \rfloor$.

12.5 Give an example of a set S of n disjoint line segments in the plane such that any auto-partition for S has depth $\Omega(n)$.

12.6 We have shown that the expected size of the partitioning produced by 2DRANDOMBSP is $O(n \log n)$. What is the worst-case size?

12.7 Suppose we apply 2DRANDOMBSP to a set of *intersecting* line segments in the plane. Can you say anything about the expected size of the resulting BSP tree?

12.8 In 3DRANDOMBSP2, it is not described how to find the cells that must be split when a splitting plane is added, nor is it described how to perform the split efficiently. Work out the details for this step, and analyze the running time of your algorithm.

12.9 Give a deterministic divide-and-conquer algorithm that constructs a BSP tree of size $O(n \log n)$ for a set of n line segments in the plane. *Hint:* Use as many free splits as possible and use vertical splitting lines otherwise.

12.10 Let C be a set of n disjoint unit discs—discs of radius 1—in the plane. Show that there is a BSP of size $O(n)$ for C. *Hint:* Start by using a suitable collection of vertical lines of the form $x = 2i$ for some integer i.

12.11 BSP trees can be used for a variety of tasks. Suppose we have a BSP on the edges of a planar subdivision.

a. Give an algorithm that uses the BSP tree to perform point location on the subdivision. What is the worst-case query time?
b. Give an algorithm that uses the BSP tree to report all the faces of the subdivision intersected by a query segment. What is the worst-case query time?
c. Give an algorithm that uses the BSP tree to report all the faces of the subdivision intersected by an axis-parallel query rectangle. What is the worst-case query time?

12.12 In Chapter 5 kd-trees were introduced. Kd-trees can also store segments instead of points, in which case they are in fact a special type of BSP tree, where the splitting lines for nodes at even depth in the tree are horizontal and the splitting lines at odd levels are vertical.

a. Discuss the advantages and/or disadvantages of BSP trees over kd-trees.
b. For any set of two non-intersecting line segments in the plane there exists a BSP tree of size 2. Prove that there is no constant c such that for any set of two non-intersecting line segments there exists a kd-tree of size at most c.

12.13* Prove or disprove: For any set of n non-intersecting line segments in the plane, there exists a BSP tree of size $O(n)$. (This problem is still open and probably difficult.)