Linear Size Binary Space Partitions for Uncluttered Scenes

Mark de Berg*

Department of Computer Science, Utrecht University,
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract

We describe a new and simple method for constructing binary space partitions in arbitrary dimensions. We also introduce the concept of *uncluttered* scenes, which are scenes with a certain property that we suspect many realistic scenes exhibit, and we show that our method constructs a BSP of size $O(n)$ for an uncluttered scene consisting of n objects. The construction time is $O(n \log n)$. Because any set of disjoint fat objects is uncluttered, our result implies an efficient method to construct a linear size BSP for fat objects.

We use our BSP to develop a data structure for point location in uncluttered scenes. The query time of our structure is $O(\log n)$, and the amount of storage is $O(n)$. This result can in turn be used to perform range queries with not-too-small ranges in scenes consisting of disjoint fat objects or, more generally, in so-called low-density scenes.

1 Introduction

Many geometric problems can be solved more easily if a decomposition of the space of interest into smaller subspaces, or cells, is given. Therefore decompositions of two-, three-, or higher-dimensional scenes play an important role in areas like computer graphics, geographic information systems, and robotics.

There is a variety of schemes available to construct decompositions. Quadtrees and octrees, and *kd*-trees are among the most popular ones [19, 20]. Another popular decomposition scheme is the *binary space partition*, or *BSP*. In this scheme the space is split into two subspaces with a hyperplane, so in $\mathbb{R}^2$ it is split with a line and in $\mathbb{R}^3$ with a plane. These two subspaces are again split with a hyperplane, and so on. The splitting process continues recursively until the subspaces are intersected by only one of the objects in the scene. (We assume that the objects in the scene don't intersect each other, otherwise we cannot require that each terminal subspace contain only one object.) Figure 1 shows a BSP of a two-dimensional scene; the line labelled ℓ is the first splitting line. Observe that the objects in the scene can be fragmented by the splitting process.

A natural way to model the splitting process is with a binary tree. The root of this tree stores the first splitting hyperplane, its right child stores the splitting hyperplane of the right subspace and its left child stores the splitting hyperplane of the left subspace, and so on—see

*Supported by the Dutch Organisation for Scientific Research (N.W.O.) and by ESPRIT Basic Research Action No. 7141 (project ALCOM II: *Algorithms and Complexity*)

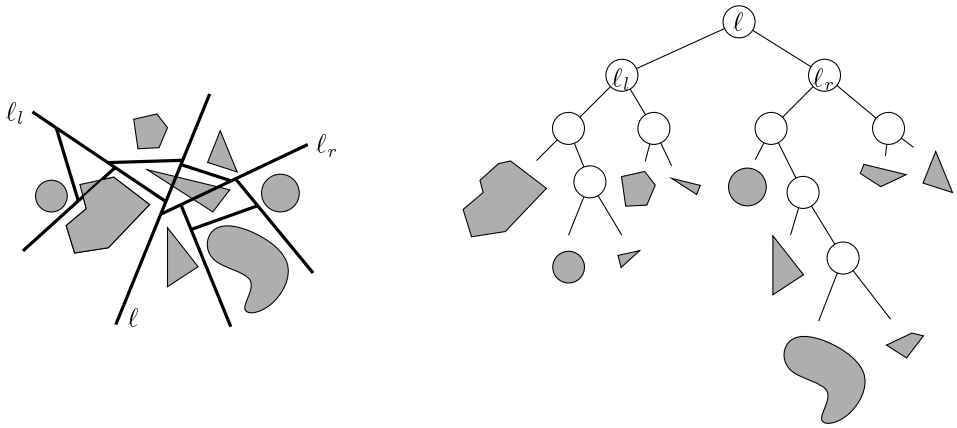

Figure 1: A BSP in the plane, and the corresponding tree.

Figure 1. (If the splitting hyperplane fully contains one or more objects, which must then be $(d-1)$-dimensional, it contains a list of these objects.) A leaf of the tree corresponds to a cell in the final decomposition; it stores the (fragment of) the object that intersects the cell. Such a tree is called a *binary space partition tree*, or *BSP tree*.

Binary space partitions are used for many purposes. For example, they are used for hidden surface removal with the painter's algorithm [10], for shadow generation [8], for set operations on polyhedra [14, 24], for visibility preprocessing for interactive walkthroughs [23], and for cell decomposition methods in motion planning [3].

The efficiency of algorithms based on BSPs depends crucially on the *size* of the BSP, that is, on the number of cells of the decomposition. Notice that this number is exactly the number of leaves in the corresponding BSP tree, which is related to the total number of fragments created by the splitting process. Hence, when constructing a BSP of a given scene, one should choose the splitting hyperplanes carefully, so that the fragmentation of the objects is kept small. In two-dimensional space it is always possible to keep the fragmentation reasonably small: Paterson and Yao[17] proved that any set of polygons in the plane with n edges in total admits a BSP of size $O(n \log n)$ and that any set of axis-parallel polygons admits a linear size BSP. It is still open whether it is possible to construct a linear size BSP for any set of polygons in the plane. In three-dimensional space—the setting most relevant to computer graphics—the situation is less rosy: the method of Paterson and Yao is only guaranteed to produce a BSP of size $O(n^2)$. (For axis-parallel polyhedra one can obtain a BSP of size $O(n\sqrt{n})$ [18].) They also gave an example of a three-dimensional scene such that *any* BSP must have quadratic size, which shows that their method is optimal in the worst case. A quadratic size BSP is, of course, useless in most practical applications. Nevertheless, BSPs usually perform fine in practice. Apparently realistic scenes have some property that makes it possible to construct efficient BSPs for them. Indeed, the example proving the $\Omega(n^2)$ lower bound on the size of BSPs is a quite artificial construction, which uses long and thin triangles in a grid-like pattern.

The discrepancy between theory and practice lead de Berg et al. [5] to study BSPs for scenes consisting of *fat objects*. Fat objects are objects that do not have long and skinny

parts; a formal definition is given in Section 2.3. Recently, fat objects have attracted a lot of attention in computational geometry [1, 9, 28, 13, 15, 26, 27]. De Berg et al. proved that scenes of fat objects always admit a BSP of linear size. Their algorithm for constructing a BSP runs in $O(n \log n \log\log n)$ time, where n is the number of objects. Unfortunately, their method only works in the plane, so it is not very useful in computer graphics applications. Moreover, it is rather complicated. We propose a new method for constructing BSPs. Our method yields a linear size BSP for collections of fat objects in arbitrary dimensions, and it is quite simple. The running time of the construction algorithm is $O(n \log n)$.

The method not only works for fat objects, it can be proved to produce a linear size BSP for a more general class of scenes, namely scenes that are *uncluttered*. A κ-cluttered scene is a scene with the following property:[1] any cube that is intersected by more than κ objects must contain a vertex of a bounding box of one or more of these objects. A scene is called uncluttered if it is κ-cluttered for a small constant κ. Although this is a rather technical condition, we believe it is usually satisfied in practice. To obtain the result for fat objects, we show that any set of fat objects is uncluttered. (In a recent paper by de Berg et al. [6] the relation between various of these so-called realistic input models is studied extensively.)

BSP trees are often used to perform point location queries (report the objects in a scene that contain a query point) and range searching queries (report the objects in a scene intersecting a query range). Because the depth of the BSP tree usually is not guaranteed to be small, this approach leads to solutions with a high query time. We show that it is possible to build an extra search structure on top of our BSP tree, which guarantees that point location queries can be performed efficiently: if the scene is uncluttered then the query time is $O(\log n)$ and the amount of storage for the data structure is $O(n)$. This improves and generalizes a result of Overmars [15], who showed that point location queries in a set of n disjoint fat objects can be done in $O(\log^{d-1} n)$ time with a structure using $O(n \log^{d-1} n)$ storage. If the scene consists of disjoint fat objects (or, more generally, is a so-called low-density scene) then our point location data structure can be used to perform range searching with ranges that are not too small compared to the smallest object in S (see Section 3 for details). The query time for range searching is then $O(\log n)$—the assumptions imply that the number of reported objects is $O(1)$, so no extra term is needed for this—and the amount of storage is still $O(n)$. This improves results by Overmars and van der Stappen [16] and by Schwarzkopf and Vleugels [22], who obtain $O(\log^{d-1} n)$ query time with a structure using $O(n \log^{d-1} n)$ storage.

2 The BSP construction

Let S be a set of n non-intersecting objects in $\mathbb{R}^d$, where $d \geqslant 2$. In Section 2.1 we describe our strategy for constructing a BSP for S. How to implement this strategy efficiently is described in Section 2.2, and the analysis of the size of the resulting BSP will be given in Section 2.3. Although the method works in arbitrary dimensions we shall mostly use three-dimensional terminology from now on.

[1] In a preliminary version of this paper [4] and in Vleugels's thesis [30] the term "bounding-box fitness" was used to express this condition.

2.1 The Partitioning Strategy

Before we start we need a few definitions. The *bounding box* of an object o is the smallest axis-aligned box that contains the object. Let $V = V(S)$ denote the set of vertices of the bounding boxes of the objects in S. For a set S in $\mathbb{R}^3$, the set V contains at most $8n$ points; in $\mathbb{R}^d$, it contains at most $2^d n$ points.

Our algorithm for constructing the BSP for S works in two stages. In the first stage we construct an intermediate BSP using axis-parallel planes only. This stage is guided by the set V of bounding-box vertices. The cells of the BSP that results from the first stage do not contain points from V in their interior. They can still be intersected by a number of objects, however. The remaining objects in each cell are separated in the second stage of the algorithm, which uses splitting planes that are not necessarily axis-parallel.

During the first stage we try to ensure that the cells we create are cubes, rather than arbitrary boxes. The reason for this is that a cell which is a thin box can easily be intersected by many objects without containing many bounding-box vertices; for such a cell the bounding-box vertices don't provide sufficient information to control the fragmentation of the objects. Most objects that intersect a cube, however, will have a vertex of their bounding box inside it. Indeed, an object that intersects a cube but whose bounding box does not have a vertex inside a cube must be relatively big with respect to the cube—its diameter must be at least the edge length of the cube—and it is unlikely that many big objects intersect the cube without starting to intersect each other. (Such objects would have to be very long and thin. Indeed we will show later that this cannot happen for fat objects—see Section 2.3 for details.) Hence, we can use the bounding-box vertices to control the fragmentation. Observe that it is possible for a small object to intersect a cube without having one of its own vertices inside the cube. This is the reason that we cannot use the vertices of the objects themselves to control the fragmentation.

The first stage. To construct the intermediate BSP we proceed as follows. In a generic partitioning step we have cell C, which is a cube, and a non-empty subset $V_C \subset V$ that contains all points from V lying in the interior of C. Initially, C will be a minimal enclosing cube of the set V, and V_C will contain all vertices from V that do not lie on the boundary of C.

We define an *octree split* to be a split of C into 2^d equal-sized sub-cubes. Thus an octree split is performed by taking the planes that are the perpendicular bisectors of the edges of C. We call an octree split *useless* if all the points in V_C are in the interior of the same sub-cube; otherwise it is called *useful*. The cube C is partitioned according to the following rules, which are illustrated in Figure 2 for the two-dimensional case.

1. If an octree split of C is useful, it is performed.

2. Otherwise, let $C_1, \ldots, C_{2^d}$ be the sub-cubes that would result from an octree split of C. Suppose that all points of V_C lie in C_j, and let v be the vertex of C that is also a vertex of C_j. Let C'_j be the smallest cube with v as one of its vertices that contains all points from V_C in its closure. Intuitively, C'_j is obtained by shrinking C_j, while keeping v as one of its vertices, until a point from V_C is hit. Now C is split using planes through the facets of C'_j; the order in which these planes are taken is arbitrary. We call such a split a *kd-split*, because, like in a *kd*-tree, we split successively on each of the coordinates.

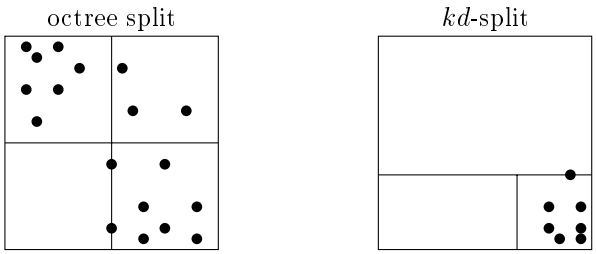

Figure 2: Splitting a cube.

The split is not exactly the same as in a kd-tree, however: there one splits the set of points into two equal halves, whereas we have an unbalanced split which guarantees the subspace containing the points to be a cube.

The splitting process is repeated recursively for the resulting subcells that have points in their interior. All subcells resulting from an octree split are cubes, but a kd-split can produce cells that are not cubes. The only cell on which we have to recurse after performing a kd-split, however, is a cube.

We now analyze the size of the intermediate decomposition.

Lemma 2.1 *The first stage of the algorithm results in an intermediate BSP consisting of $O(n)$ cells that are boxes and do not contain a vertex from V in their interior.*

Proof: The construction of the partitioning proceeds recursively until each cell is empty, and only axis-parallel splitting planes are used, so the second part of the lemma is clearly true. To see that there are $O(n)$ cells we note that any split increases the number of cells by a constant only. More precisely, an octree split results in $2^d - 1$ extra cells, and a kd-split results in d extra cells. Furthermore, when a cell C is split either one or more points from the current subset V_C are on the splitting planes, or V_C is partitioned into two or more subsets. The first case can occur at most $|V|$ times, and the second case at most $|V| - 1$ times. Hence, the total number of cells is at most $(2|V| - 1)(2^d - 1) + 1 = O(n)$. □

The second stage. The second stage of the decomposition partitions the cells of the intermediate decomposition further, so that each cell in the final BSP is intersected by only one object. This is done in the following standard way. For a cell C of the intermediate decomposition, let S_C be the set of object fragments inside C. If the objects are polygonal, then C is partitioned further by taking planes through the facets of the objects in S_C in an arbitrary order, until there is only one object left in each cell. If the objects are curved but convex, we use planes separating pairs of objects to further partition C.

2.2 An Efficient Algorithm to Construct the BSP

So far we have only specified which splitting planes to use to construct the BSP. Next we discuss how to find these planes efficiently.

The first stage. Let C be a cell that is split during the first stage of the algorithm, and let V_C be the set of bounding-box vertices in its interior. There are two tasks: we have to decide whether to apply an octree split or a kd-split, and we have to perform the split. The latter task involves distributing the bounding-box vertices over the subcells that we get. If these tasks are done in a brute-force manner, then the construction time can be high, even though the BSP that is produced may be small. The reason for this is that the splits need not be balanced. In fact, a kd-split is always unbalanced. So if we spend linear time on each split and the splits are very unbalanced, then the running time $T(n)$ would satisfy a recurrence like $T(n) = O(n) + T(n-1)$, leading to a quadratic running time. Vaidya [25] and Callahan and Kosaraju [7] describe techniques for building an octree-like structure, where this problem is solved. Both techniques are directly applicable in our situation and lead to a contruction time of $O(n \log n)$.

The second stage. To carry out the second stage of the algorithm, we need to know for each cell C in the intermediate decomposition (the decomposition resulting from the first stage) the set S_C of object fragments lying in it. In the plane a simple plane-sweep algorithm can compute the sets S_C efficiently, but in three and higher dimensions we need a different approach. We can maintain the objects intersecting a cell during the recursive calls, but this will take too much time for unbalanced splits. To get around this problem we could try to use a technique similar to the technique of Vaidya [25] or Callahan and Kosaraju [7], like in the first stage, but there are some difficulties with the approach when we deal with object fragments. Therefore we take a different approach: we preprocess the intermediate BSP for the following type of range searching queries: given a constant-complexity query range, report all cells in the BSP intersected by Q. This range searching problem is discussed in detail in Section 3; we prove in Lemma 3.4 that the k cells intersecting a query range can be reported in $O(k \log n)$ time, after $O(n \log n)$ preprocessing. In order to compute the sets S_C we query with each object in S to find the cells it intersects, and add the object to the set S_C of every intersected cell C. If we denote the number of cells intersecting an object o by k_o, then the total time for all queries is

$$\sum_{o \in S} O(k_o \log n) = O(\sum_C |S_C| \log n).$$

This leads to the following result.

Lemma 2.2 *Let S be a set of n objects in $\mathbb{R}^d$, each of constant complexity, and let $\mathcal{S}$ be the intermediate BSP resulting from the first stage of the BSP construction algorithm. The sets S_C of object fragments for each cell C of $\mathcal{S}$ can be computed in $O(\sum_{C \in \mathcal{S}} |S_C| \log n)$ time in total.*

2.3 The Analysis

Uncluttered scenes. We now analyze the method of the previous section under the assumption that the set S of objects is uncluttered, which is defined as follows.

Definition 2.3 *Let κ be a positive integer. A set of objects in $\mathbb{R}^d$ is called a* κ-cluttered scene *if any cube whose interior does not containing a vertex of one of the bounding boxes of the objects in S is intersected by at most κ objects in S.*

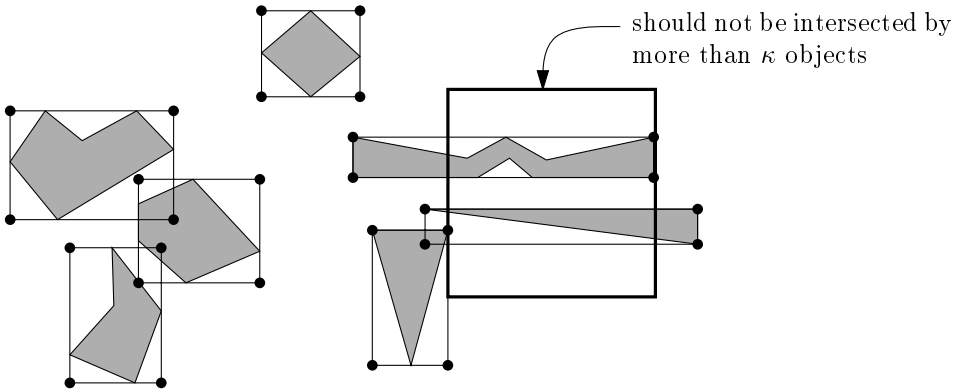

Figure 3: An uncluttered scene.

Figure 3 illustrates this definition. If we don't want to specify the exact *clutter factor* κ but assume it is a small constant—as we do in the remainder of this section—we will simply say that the scene is *uncluttered*.

It is essential that the definition only speaks of *cubes* without bounding-box vertices in their interior. If the condition would speak of arbitrarily shaped boxes it would be a lot stronger, but it would also be highly unrealistic. We believe that in many applications scenes are indeed uncluttered, despite the rather technical flavor of the condition.

The following lemma implies that our BSP strategy performs well for uncluttered scenes.

Lemma 2.4 *If the set S forms an uncluttered scene, then any cell in the intermediate decomposition is intersected by $O(1)$ objects from S.*

Proof: Let C be a cell of the intermediate decomposition. By construction, C does not contain a vertex of any bounding box in its interior. Hence, if C is a cube then the lemma follows from the unclutteredness condition. Now suppose that C is not a cube. Then C was created as one of the empty cells when a kd-split was performed. In this case C can be covered by a constant number of cubes that are contained in the union of all empty cells (that is, the L-shape which is the complement of the cell containing all points) created at this step. This

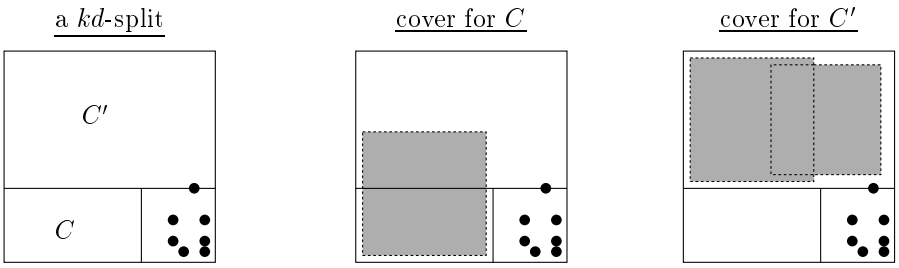

Figure 4: The cells resulting from a kd-split, and the ways to cover them by cubes.

is illustrated for the planar case in Figure 4; for clarity, the squares that cover the cells C and C' are shown slightly smaller than they actually are. Because the cubes that cover C are

contained in the union of the empty cells forming the L-shape, they contain no bounding-box vertices in their interior, so they are intersected by $O(1)$ objects. Hence, C is intersected by $O(1)$ objects as well. □

So after the first stage of the algorithm we have an intermediate BSP of linear size such that each cell is intersected by $O(1)$ objects. By Lemma 2.2 this implies that the construction time for the first phase will be $O(n \log n)$. Because the second stage is performed on a constant number of objects inside each cell, it does not increase the asymptotic size or preprocessing time of the BSP (provided that the objects are polygonal and of constant complexity, or are convex.)

Theorem 2.5 *Let S be a polygonal scene in $\mathbb{R}^d$ consisting of n non-intersecting polygonal objects of constant complexity, which is uncluttered. Then there exists a linear size binary space partition for the objects in S. This binary space partition can be constructed in $O(n \log n)$ time.*

Remark. The only place where we need that the objects are polygonal is in the second stage of the algorithm, where we separate the objects using planes through the facets of the objects. If the objects are curved we need a different strategy for this. If there is such a strategy—more precisely, if any pair of objects can be separated with a constant number of planes—then our method still works. For example, a linear size BSP exists for any uncluttered scene consisting of convex curved objects.

Fat objects. We now prove that fatness implies unclutteredness. Intuitively, an object is called fat if it does not contain any long and skinny parts. Van der Stappen [26] gives an extensive treatment of fatness in the context of motion planning. Fatness can be defined formally in various ways, which are basically all equivalent. We follow the Van der Stappen's definition.

Definition 2.6 *Let $0 \leqslant \alpha \leqslant 1$ be a constant. An object o in $\mathbb{R}^d$ is α-fat if, for any sphere σ whose center lies in o and whose boundary intersects o, the following holds*

$$\mathit{Volume}(o \cap \sigma) \geqslant \alpha\, \mathit{Volume}(\sigma).$$

(If the objects in S are convex, then the definition is equivalent to the following, simpler definition: an object is fat if its volume is at least β times the volume of its minimal enclosing d-dimensional sphere for some constant β.)

If one doesn't want to specify the exact value of α but assumes α is a fixed, not-too-small, positive constant, then an α-fat object is simply called fat. Van der Stappen [26] proves that any set of disjoint fat objects has low density, which is defined as follows. Let $\rho(\sigma)$ denote the radius of the hypersphere σ, and let mes(o) denote the minimal enclosing hypersphere of the object o.

Definition 2.7 *Let $\lambda > 0$ be a constant. A set S is a λ-low-density scene if any hypersphere σ intersects at most λ objects $o \in S$ for which $\rho(\mathrm{mes(o)}) \geqslant \rho(\sigma)$.*

Next we prove that a low-density scene is uncluttered.

Lemma 2.8 *A λ-low-density scene is $(\lfloor\sqrt{d}\rfloor^d \lambda)$-cluttered.*

Proof: Let C be a cube that does not have bounding-box vertices of objects in S in its interior. Consider an object $o \in S$ that intersects C. Because the bounding-box vertices of o do not lie in the interior of C, the diameter of o, denoted diam(o), must be at least the edge length of C. Hence,

$$\rho(\text{mes(o)}) \geqslant \frac{\text{diam(o)}}{2} \geqslant \frac{\text{(edge length of } C)}{2} = \frac{(2/\sqrt{\text{d}})\rho(\text{mes(C)})}{2} = \frac{\rho(\text{mes(C)})}{\sqrt{\text{d}}}.$$

Now cover C by $\lfloor\sqrt{d}\rfloor^d$ smaller cubes of side length $1/\lfloor\sqrt{d}\rfloor$ times the side length of C. Each subcube C' is only intersected by objects o such that $\rho(\text{mes(o)}) \geqslant \rho(\text{mes(C)})/\sqrt{\text{d}} \geqslant \rho(\text{mes(C}'))$. Because S has λ-low-density, this means that C' is intersected by at most λ objects, which proves that C is intersected by at most $\lfloor\sqrt{d}\rfloor^d\lambda$ objects. □

Because of the result of Van der Stappen [26] that a scene consisting of disjoint fat objects has low density, this lemma implies that such a scene is uncluttered. (Formulated more precisely: Let α be a fixed positive constant, and consider the family of all sets of α-fat objects. Then there is a constant κ, depending only on α, such that any set from the family is κ-cluttered.)

Although fatness implies the uncluttered ness, the reverse is certainly not true. Consider as an example an architectural model. The ceilings in such a model are very thin in the z-direction, so they are not fat. A similar observation can be made for the walls. If, however, the rooms have a reasonable shape (not extremely long and narrow) then an architectural model will be uncluttered.

The following result immediately follows from Theorem 2.5 and Lemma 2.8.

Corollary 2.9 *Let S be a polygonal scene in $\mathbb{R}^d$ consisting of n non-intersecting fat objects of constant complexity. Then there exists a linear size binary space partition for the objects in S. This binary space partition can be constructed in $O(n \log n)$ time.*

3 Applications to Point Location and Range Searching

3.1 Point location in uncluttered scenes

The point location problem we consider is as follows. We are given a scene consisting of n constant-complexity objects in $\mathbb{R}^d$. We want to preprocess the scene into a data structure such that, given a query point q, we can efficiently determine which object (if any) in the scene contains q. If the objects are not disjoint, then q may lie in more than one object and all containing objects should be reported.

Define a *rectilinear binary space partition* to be a binary space partition that only uses splitting planes that are orthogonal to one of the coordinate-axis. A rectilinear BSP tree is a BSP tree whose underlying binary space partition is rectilinear. Recall that the intermediate BSP resulting from the first stage of the algorithm of Section 2.1 is rectilinear.

Schwarz et al. [21] show that a rectilinear BSP can be preprocessed in linear time such that one can do point location in logarithmic time. To perform point location in an arbitrary scene S we proceed as follows. We use the first stage of the BSP construction algorithm of Section 2.1 to construct a rectilinear BSP on S. With each cell of the resulting BSP we associate a list of all objects in S intersecting that cell. To perform a point location on S we simply do point location on the rectilinear BSP and check all objects in the list associated

with the cell containing the query point. It follows from Lemmas 2.1 and 2.4 that this gives a good performance for uncluttered scenes.

Theorem 3.1 *Let S be a set of n constant-complexity objects forming an uncluttered scene. Then the set S can be preprocessed in $O(n \log n)$ time into a data structure that uses $O(n)$ storage, such that the objects containing a query point can be reported in $O(\log n)$ time.*

3.2 Range searching in low-density scenes

The range searching problem on a set S of objects is defined as follows: given a query range Q, report all objects from S that are intersected by Q. (Point location is a special case of range searching, where the query range is a point.) The theorem above has an immediate application to range searching with small ranges in low-density scenes. Recall that $\rho(\sigma)$ denotes the radius of the sphere σ, and that $mes(o)$ denotes the minimal enclosing hypersphere of the object o.

Corollary 3.2 *Let S be a set of objects in $\mathbb{R}^d$ forming a low-density scene, and let $\rho_{\min} = \min_{o \in S} \rho(mes(o))$. Then the set S can be preprocessed in $O(n \log n)$ time into a data structure that uses $O(n)$ storage, such that range queries with constant-complexity ranges Q for which $\rho(mes(Q)) \leqslant \rho_{\min}$ can be answered in $O(\log n)$ time.*

Proof: Schwarzkopf and Vleugels [22] have shown that range searching queries on S can be answered by performing a number of point location queries on a set S^*, which is obtained by taking the Minkowski sums of the objects in S with a hypercube of edge length $\rho_{\min}$. If S is a low-density scene and the query range is small, then the number of point locations is $O(1)$ and the number of objects reported per query is $O(1)$. Van der Stappen et al. [29] have shown that if S has low density, then S^* has low density as well. Since low density implies uncluttered ness (Lemma 2.8) it now follows from Theorem 3.1 that the point location queries on S^* can be performed in $O(\log n)$ time, from which the corollary follows. □

This result improves results by Overmars and Van der Stappen [16] and Schwarzkopf and Vleugels [22] by a factor of $O(\log^{d-1} n)$ in storage and of $O(\log^{d-2} n)$ in query time.

3.3 Range searching in rectilinear BSPs

There is one issue that still needs attention: we have to show that we can find for each object in the set S which cells it intersects in the rectilinear BSP resulting from the first stage of the BSP construction. This is necessary for the second stage of the BSP construction, and for the point location structure. In other words, we have to perform a range query on the rectilinear BSP. Although the rectilinear BSP can be shown to have low density, we cannot use Corollary 3.2 to do the range searching, because the ranges we are dealing with (the objects in S) are not necessarily small compared to the smallest cell in the BSP. Hence, we have to preprocess the rectilinear BSP in a different manner for range searching.

Our structure to report which cells of the rectilinear BSP are intersected by a query range has two components. The first is the structure of Schwarz et al. [21] to do point location in the BSP subdivision. The second component of the structure is the BSP tree itself, preprocessed for lowest-common-ancestor queries. Such queries can be answered in $O(1)$ time after linear preprocessing [12]. (In our application $O(\log m)$ query time would also be sufficient.)

The query algorithm works as follows: First we determine the highest node in the BSP tree whose splitting plane cuts Q. (More precisely, we should restrict our attention to the

part of the splitting plane lying inside the subspace associated with the node of the splitting plane.) This is the first node where the search path of Q splits. However, we do not find this node by walking down the BSP tree, but in a way described later. After we have found this node, we split Q into two pieces with the splitting plane. Then we perform a range query with both pieces separately, as if they were new query ranges. Notice that the pieces we query with are always the intersection of the original range Q with some number of half-spaces. Since the BSP is rectilinear, this means that the pieces are the intersection of Q with some box. Hence, the pieces have constant complexity. The recursion stops when we have a piece not cut by any splitting plane. Such a piece is contained in one cell of the BSP, which we can find by performing a point location query with an arbitrary point in the piece. The collection of cells found in the point locations is exactly the collection of cells intersecting the range Q.

The crucial step in the query algorithm is the computation of the highest node in the BSP tree whose splitting plane cuts the query range. This is done as follows. First, for each coordinate we compute a point of Q with maximum value for that coordinate and a point of Q with minimum value for that coordinate. In the plane, for instance, we compute a leftmost and a rightmost point, and a topmost and a bottommost point. Let q_1, q_2 be one of the d pairs of points obtained. We perform a point location query with both points. Let ν_1 and ν_2 be the two leaves whose associated cells contain q_1 and q_2, respectively. Next, we compute the lowest common ancestor of ν_1 and ν_2 in the BSP tree. Figure 5 illustrates this in the planar case with the pair of extreme points in the x-direction. The nodes ν_1 and ν_2 as well as their lowest common ancestor are shown in dark grey. Of all the d lowest common ancestors

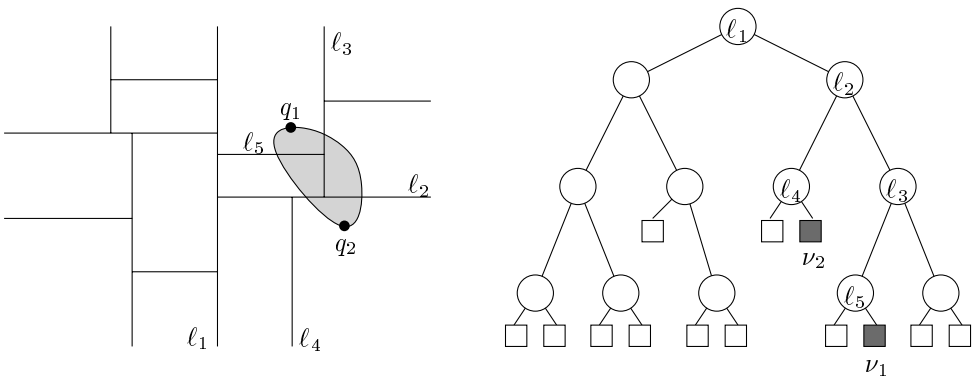

Figure 5: Finding the highest splitting plane cutting a range via lowest common ancestors.

computed in this manner, we take the highest node, and we claim that this is the node we seek.

Claim 3.3 *The highest of the d lowest common ancestors that we have computed is the highest node in the BSP tree whose splitting plane cuts Q.*

Proof: Let ν be the highest lowest common ancestor, and let q_1, q_2 be the pair of points for which it was found. Since q_1 and q_2 lie in different BSP subtrees of ν, they must lie on different sides of the splitting plane of ν, so the splitting plane must cut Q.

Conversely, suppose that the splitting plane of some node μ that is higher than ν cuts Q. Then the points of Q that are extreme in the direction orthogonal to the splitting plane

cannot lie in the same BSP subtree of μ. Hence, the query with this pair would have reported a node that is at least as high as μ, contradicting the fact that ν is the highest lowest common ancestor. □

This leads to the following result.

Lemma 3.4 *Let $\mathcal{S}$ be a subdivision of $\mathbb{R}^d$ that is induced by a rectilinear BSP tree $\mathcal{T}$ with m nodes. It is possible to preprocess the BSP in $O(m)$ time into a data structure that uses $O(m)$ storage, such that the k cells of $\mathcal{S}$ intersected by a constant-complexity query range Q can be reported in $O(k \log m)$ time.*

Proof: The bounds on the preprocessing time and the storage follow immediately from the fact that the point location structure of Schwarz et al. [21] uses linear preprocessing, and the fact that preprocessing a binary tree for lowest common ancestor queries takes linear time [12].

It remains to prove the bound on the query time. There are three steps in querying with some (piece of) a query range: we have to compute extreme points in d different directions, we have to perform point location with these points, and we have to perform d lowest common ancestor queries in the BSP tree and then select the highest node thus found. After these three steps, we may have to split the range, and perform two new queries with the resulting pieces. The first and third step take only constant time, and the second takes $O(\log m)$ time. So the query time follows if we can show that the total number of pieces that we query with is $O(k)$.

This can be seen as follows. Consider the recursion tree of the query process. The root of this tree corresponds to the query with the original range Q, its two children to the queries with the two pieces into which Q is split, and so on. The leaves of the tree are the pieces that fall completely inside a cell. Hence, the number of leaves is exactly equal to the number of reported cells. Because the recursion tree is a binary tree, its number of inner nodes is one less than its number of leaves, so the total number of queries is $2k - 1$. It remains to observe that two leaves in the recursion tree cannot report the same cell of the subdivision, because at some stage of the recursive process there was a node where the paths in the BSP tree to (the BSP leaves of) these cells split. □

4 Concluding Remarks

We presented a new and simple method for constructing BSPs. For uncluttered scenes—and we believe many realistic scenes are uncluttered—the method produces a linear size BSP in $O(n \log n)$ time. We also proved that any scene consisting of disjoint fat objects in $\mathbb{R}^d$ is uncluttered, which thus implies that any set of disjoint fat objects admits a linear size BSP.

By adding an extra structure on top of the BSP tree, it is possible to use it to perform efficient point location queries in uncluttered scenes. The point location structure can in turn be used to perform efficient range queries with small ranges in low-density scenes.

One direction of future research that suggests itself is an experimental one, namely to implement and compare the various existing decomposition schemes (BSP methods, octrees, kd-trees, and so on). It would also be interesting to experimentally verify our assumption that in many applications the scenes are uncluttered. More precisely, one would like to compute

the clutter factors of various typical architectural models, say, and see whether the clutter factor is indeed a fairly small.

A theoretical problem that still remains wide open is whether the conjecture of Paterson and Yao [17] (any set of line segments in the plane admits a linear size BSP) is true.

Acknowledgment

With David Kirkpatrick I discussed the preprocessing of the BSP extensively, and he was the one who discovered that Vaidya's result can be used for this. He also gave many valuable comments concerning the presentation of the paper.

References

[1] Pankaj K. Agarwal, M. J. Katz, and M. Sharir. Computing depth orders and related problems. In *Proc. 4th Scand. Workshop Algorithm Theory*, pages 1–12, 1994.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.

[3] C. Ballieux. Motion planning using binary space partitions. Technical Report Inf/src/93-25, Utrecht University, 1993.

[4] M. de Berg. Linear size binary space partitions for fat objects. In *Proc. 3rd Annual European Symposium on Algorithms (ESA '95)*, volume 979 of *Lecture Notes in Computer Science*, pages 252–263. Springer-Verlag, 1995.

[5] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. In *Proc. 4th Scand. Workshop Algorithm Theory*, volume 824 of *Lecture Notes Comput. Sci.*, pages 61–72. Springer-Verlag, 1994.

[6] M. de Berg, M.J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 294–303, 1997.

[7] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.

[8] N. Chin and S. Feiner. Near real time shadow generation using bsp trees. In *Proc. SIGGRAPH'89*, pages 99–106, 1989.

[9] A. Efrat, M. Sharir, and G. Rote. On the union of fat wedges and separating a collection of segments by a line. *Comput. Geom. Theory Appl.*, 3:277–288, 1994.

[10] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[11] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[12] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[13] J. Matoušek, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. *SIAM J. Comput.*, 23:154–169, 1994.

[14] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990. Proc. SIGGRAPH '90.

[15] M. H. Overmars. Point location in fat subdivisions. *Inform. Process. Lett.*, 44:261–265, 1992.

[16] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. In J. van Leeuwen, editor, *Algorithms – ESA'94*, volume 855 of *Lecture Notes Comput. Sci.*, pages 240–253, Berlin, 1994. Springer-Verlag.

[17] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[18] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[19] H. Samet. *Applications of Spatial Data Structures.* Addison Wesley, Reading, MA, 1990.

[20] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[21] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[22] Otfried Schwarzkopf and Jules Vleugels. Range searching in low-density environments. *Inform. Process. Lett.*, 60:121–127, 1996.

[23] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Comput. Graph.*, 25(4):61–69, July 1991. Proc. SIGGRAPH '91.

[24] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.*, 21(4):153–162, 1987. Proc. SIGGRAPH '87.

[25] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.

[26] A. F. van der Stappen. *Motion Planning amidst Fat Obstacles.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1994.

[27] A. F. van der Stappen and M. H. Overmars. Motion planning amidst fat obstacles. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 31–40, 1994.

[28] Marc van Kreveld. On fat partitioning, fat covering, and the union size of polygons. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes Comput. Sci.*, pages 452–463. Springer-Verlag, 1993.

[29] A.F. vander Stappen, M.H. Overmars, M. de Berg, and J. Vleugels. Motion planning in environments with low obstacle density. Technical Report UU-CS-1997-19, Utrecht University, 1997.

[30] J. Vleugels. *On Fatness and Fitness: Realistic Input Models for Geometric Algorithms.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1997.