Practical Techniques for Constructing Binary Space Partitions for Orthogonal Rectangles

Pankaj K. Agarwal[*] T. M. Murali[†] Jeffrey Scott Vitter[‡]
Box 90129, Duke University
Durham, NC 27708-0129
Email: {pankaj,tmax,jsv}@cs.duke.edu

Abstract

We present the first systematic comparison of the performance of algorithms that construct Binary Space Partitions for orthogonal rectangles in $\mathbb{R}^3$. We compare known algorithms with our implementation of a recent algorithm of Agarwal et al. [1]. We show via an empirical study that their algorithm constructs BSPs of near-linear size in practice and performs better than most of the other algorithms in the literature.

[*] Support was provided by National Science Foundation research grant CCR–93–01259, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corp, and by a grant from the U.S.-Israeli Binational Science Foundation.

[†] This author is affiliated with Brown University. Support was provided in part by National Science Foundation research grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013.

[‡] Support was provided in part by National Science Foundation research grant CCR–9522047, by Army Research Office grant DAAH04–93–G–0076, and by Army Research Office MURI grant DAAH04–96–1–0013.

1 Introduction

How to render a set of opaque or partially transparent objects in $\mathbb{R}^3$ quickly and in a visually realistic way is a fundamental problem in computer graphics [12, 24]. A central component of this problem is *hidden-surface removal*: given a set of objects, a viewpoint, and an image plane, compute the scene visible from the viewpoint on the image plane. Because of its importance, the hidden-surface removal problem has been studied extensively in both the computer graphics and the computational geometry communities [11, 12]. One of the conceptually simplest solutions to this problem is the so-called "painter's algorithm," which draws the objects to be displayed on the screen in a back-to-front order (in which no object is occluded by any object earlier in the order). In general, it is not possible to find a back-to-front order from a given viewpoint for an arbitrary set of objects. The *binary space partition* (BSP) [13, 23] is a data structure that fragments the objects and ensures that a back-to-front order from *any* viewpoint can be determined for the fragments. BSPs have proven to be versatile, with applications in many other problems—global illumination [5], shadow generation [7, 8, 9], visibility problems [3, 25], solid geometry [19, 20, 26], geometric data repair [16], ray tracing [18], robotics [4], and approximation algorithms for network design [15] and surface simplification [2].

Informally, a BSP $\mathcal{B}$ for a set of polygons in $\mathbb{R}^3$ is a binary tree. Each node v of $\mathcal{B}$ is associated with a convex region $\mathcal{R}_v$. The regions associated with the children of v are obtained by splitting $\mathcal{R}_v$ with a plane. The regions associated with the leaves of the tree form a convex decomposition of space, and the interior of the region associated with a leaf does not intersect any polygon. The polygonal faces of the decomposition intersect the polygons and divide them into fragments; these fragments are stored at appropriate nodes of the BSP.

The efficiency of most BSP-based algorithms depends on the size and/or the depth of the BSP (we formally define the size of a BSP later). Therefore, several heuristics to construct BSPs of small size and depth have been developed [3, 13, 25, 26]. Most of these heuristics construct a BSP in a top-down fashion: at each node, they compute a cutting plane based on some locally optimal criteria. However, they may construct a BSP of size $\Omega(n^3)$ for some instances of n polygons.

The first algorithm with a non-trivial provable bound on the size of a BSP was developed by Paterson and Yao. They show that a BSP of size $O(n^2)$ can be constructed for n disjoint triangles in $\mathbb{R}^3$, which is optimal in the worst case [21]. But in graphics-related applications, many common environments like buildings are composed largely of orthogonal rectangles, and non-orthogonal objects are approximated by their orthogonal bounding boxes [12]. Paterson and Yao [22] show that a BSP of size $O(n\sqrt{n})$ can be constructed for n non-intersecting, orthogonal rectangles in $\mathbb{R}^3$. This bound is also optimal in the worst case. If all but m of the rectangles have aspect ratio bounded by a constant, Agarwal et al. [1] describe an algorithm that constructs a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$. A related result of de Berg shows that a BSP of linear size can be constructed for fat polyhedra in $\mathbb{R}^d$ [10].

We have implemented the recent algorithm of Agarwal et al. [1] to study its performance on "real" data sets. While implementing their algorithm, we found that no systematic comparison of various existing algorithms to construct BSPs has been performed. Therefore, we implemented many other algorithms described in the literature. Our paper makes two main contributions. First, we show that the algorithm of Agarwal et al. is indeed practical: it constructs a BSP of near-linear size on real data sets. This algorithm performs better than not only Paterson and Yao's algorithm [22] but also most heuristics described in the literature [3, 13, 26]. The only algorithm that performs better than the Agarwal et al. algorithm on some data sets is Teller's algorithm [25]; even in these cases, the algorithm of Agarwal et al. has certain advantages (see Section 5). The second result of this paper is a methodical study of the empirical performance of a variety of known algorithms. Instead of just implementing them as they appear in the literature, we have modified them to improve their performance. We have implemented a representative sample of the BSP algorithms proposed in the literature. Naturally, many variants are possible to the algorithms we picked. We expect the performance of these other techniques to be similar to the ones we have implemented. For example, Naylor has proposed a new technique that uses estimates of the costs incurred when the BSP is used to answer standard queries to control the construction of the BSP [17]. While his idea is new, the measure functions he uses to choose cutting planes are very similar to the ones we discuss in this paper (see Section 4). Cassen et al. [6] use genetic algorithms to construct BSPs. We have not compared our algorithms to theirs since they report that their algorithm takes hours to run even for moderately sized data sets.

To compare the different algorithms, we measure the size of the BSP each algorithm constructs and the time spent in answering various queries. The size measures the storage needed for the BSP and the time taken to compute a back-to-front order using the BSP. We use queries that are typically made in many BSP-based algorithms [3, 14, 25]:

1. *point location*: determine the leaf of the BSP that contains a query point. Point location is the basis of many algorithms for answering other queries.
2. *ray shooting*: determine the first rectangle intersected by a query ray. Ray shooting is a very useful query in visibility problems, since it can be used to determine the object visible along a given direction.
3. *line stabbing*: compute the number of leaves of the BSP intersected by a query line. This query models the process of tracking the viewpoint in walk-through applications [3, 25].

It is worthwhile to point out the problem of robustness that bedevils geometric computing in general does not affect us. Since we are dealing with orthogonal rectangles, all calculations (coordinate comparisons, rectangle intersections, etc.) can be carried out using only the coordinates of the vertices of the input rectangles. A related issue is that our algorithms *must* be able to handle degeneracy. In fact, the algorithms should take advantage of coplanar rectangles to reduce the size of the BSP and should not resort to standard perturbation techniques to remove degeneracies.

The rest of the paper is organized as follows: Section 2 gives some definitions and notation, most of which are borrowed from Agarwal et al. [1]. We describe the algorithm of Agarwal et al. in Section 3 and the other algorithms and heuristics that we have implemented in Section 4. In Section 5, we present the results of our experiments.

2 Geometric Preliminaries

A *binary space partition* $\mathcal{B}$ for a set S of pairwise-disjoint rectangles in $\mathbb{R}^3$ is a tree defined as follows: Each node v in $\mathcal{B}$ represents a box (rectangular parallelepiped) $\mathcal{R}_v$ and a set of rectangles $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The box associated with the root is $\mathbb{R}^3$ itself. If S_v is empty, then node v is a leaf of $\mathcal{B}$. Otherwise, we partition $\mathcal{R}_v$ into two boxes by an orthogonal *cutting plane* H_v. At v, we store $\{s \cap H_v \mid s \in S_v\}$, the subset of rectangles in S_v that lie in H_v. If we let H_v^+ be the positive halfspace and H_v^- the negative halfspace bounded by H_v, the boxes associated with the left and right children of v are $\mathcal{R}_v \cap H_v^-$ and $\mathcal{R}_v \cap H_v^+$, respectively. The left subtree of v is a BSP for the set of rectangles $S_v^- = \{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of rectangles $S_v^+ = \{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of *interior* nodes in $\mathcal{B}$ and the total number of rectangles stored at all the nodes in $\mathcal{B}$.[1]

We will often focus on a box B and construct a BSP for the rectangles intersecting it. Given a set of rectangles R, let $R_B = \{s \cap B \mid s \in R\}$ be the set of rectangles obtained by clipping the rectangles in R within B. We say that a rectangle in S_B is *free* if none of its edges lies in the interior of B; otherwise it is *non-free.* A *free cut* is a cutting plane that does not cross any rectangle in S and that either divides S into two non-empty sets or contains a rectangle in S. Note that the plane containing a free rectangle is a free cut. Free cuts play a critical role in preventing excessive fragmentation of the rectangles in S.

Although a BSP is a tree, we will often discuss just how to partition the box represented by a node into two boxes. We will not explicitly detail the associated construction of the actual tree itself, since the construction is straightforward once we specify the cutting plane.

3 The Agarwal et al. Algorithm

In this section, we describe a variant of the algorithm of Agarwal et al. [1] that we have implemented. In our implementation, we have modified their algorithm slightly in order to improve its performance. We call this algorithm Rounds. Proofs of all claims we make in this section can be found in [1].

A box B in $\mathbb{R}^3$ has six faces—*top, bottom, front, back, right,* and *left.* We assume, without loss of generality, that the back, bottom, left corner of B is the origin (i.e., the back face of B lies on the yz-plane). We say that a rectangle r in S_B is *long* with respect to a box B if none of the vertices of r lie in the interior of B. Otherwise, r is said to be *short.* See Figure 1. We can partition long rectangles into three classes: a rectangle s that is long with respect to B belongs to the *top class* if two parallel edges of s are contained in the top and bottom faces of B. We similarly define the *front* and *right* classes. A long rectangle belongs to at least one of these three classes; a non-free rectangle belongs to a unique class. See Figure 1 for examples of rectangles belonging to different classes. Note that each class can have two sets of mutually orthogonal rectangles. For a set of points P, let P_B be the subset of P lying in the interior of B.

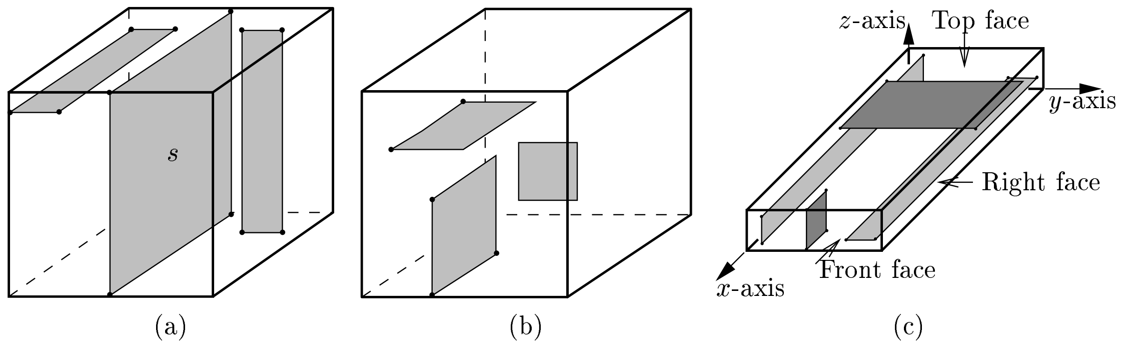
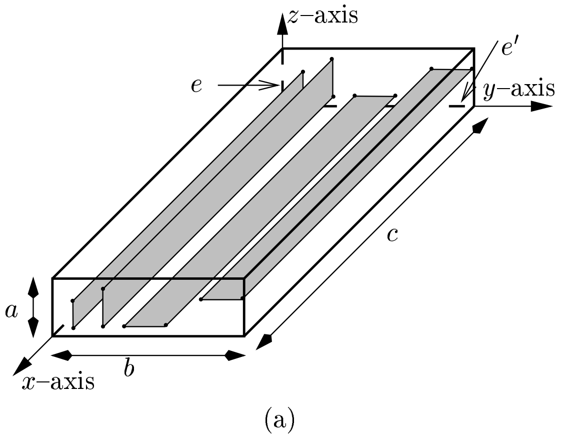


Figure 1: (a) Long rectangles, (b) short rectangles and (c) different classes of rectangles; heavy dots indicate the vertices of these rectangles that lie on the boundary of the box. Rectangle s is a free rectangle.

[1]Paterson and Yao define the size of $\mathcal{B}$ to be the number of nodes in $\mathcal{B}$ [21]. Our definition is more realistic since it measures the storage needed for $\mathcal{B}$ more accurately, especially if many rectangles in S are coplanar. We do not store the leaves explicitly since all the information about a leaf is captured by its parent and the cutting plane at the parent.



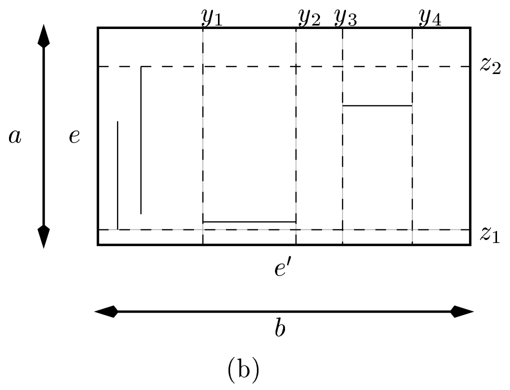


Figure 2: (a) Rectangles in S_B belonging to the sets R and T. (b) The back face of box B; dashed lines are intersections of the back face with the α-cuts.

The algorithm proceeds in rounds. At the beginning of the ith round, where $i > 0$, the algorithm has a top subtree $\mathcal{B}_i$ of the BSP for S. Let Q_i be the set of boxes associated with the leaves of $\mathcal{B}_i$ containing at least one rectangle. The initial tree $\mathcal{B}_1$ consists of one node and Q_1 consists of one box that contains all the input rectangles. The algorithm maintains the invariant that for each box $B \in Q_i$, all long rectangles in S_B are non-free, i.e., if a rectangle r intersects the interior of B then r's boundary also intersects the interior of B. If Q_i is empty, we are done. Otherwise, in the ith round, for each box $B \in Q_i$, we construct a top subtree $\mathcal{T}_B$ of the BSP for the set S_B and attach it to the corresponding leaf of $\mathcal{B}_i$. This gives us the new top subtree $\mathcal{B}_{i+1}$. Thus, it suffices to describe how to build the tree $\mathcal{T}_B$ on a box B during a round.

Let $F_B \subseteq S_B$ be the set of rectangles that are long with respect to B. Set $f = |F_B|$, and let k be the number of vertices of rectangles in S_B that lie in the interior of B (note that each such vertex is a vertex of an original rectangle in S). By assumption, all rectangles in F_B are non-free. We choose a parameter a, which remains fixed throughout the round (we discuss later what values we use for a). In a round, we partition B using a sequence of cuts in two stages, the separating stage and the dividing stage. The separating stage divides B into a set of boxes $\mathcal{C}$ such that for each box $C \in \mathcal{C}$, F_C contains only two classes of rectangles. The dividing stage further refines each such box C until a new round is to be started in the resulting boxes. We now describe each stage in detail.

Separating Stage: Assume without loss of generality that the longest edge of B is parallel to the x-axis. The rectangles in F_B that belong to the front class can be partitioned into two subsets: the set R of rectangles that are vertical (and parallel to the right face of B) and the set T of rectangles that are horizontal (and parallel to the top face of B). See Figure 2(a). Let e be the edge of B that lies on the z-axis and e' be the edge of B that lies on the y-axis. The intersection of each rectangle in R with the back face of B is a segment parallel to the z-axis. Let $\bar{r}$ denote the projection of this segment onto the z-axis, and let $\bar{R} = \{\bar{r} \mid r \in R\}$. Let $z_1 < z_2 < \cdots < z_{k-1}$ be the endpoints of intervals in $\bar{R}$ that lie in the interior of e but not in the interior of any interval of $\bar{R}$. Similarly, for each rectangle t in the set T, we define $\bar{t}$ to be the projection of t onto the y-axis, and $\bar{T} = \{\bar{t} \mid t \in T\}$. Let $y_1 < y_2 < \cdots < y_{l-1}$ be the endpoints of intervals in $\bar{T}$ that lie in the interior of e' but not in the interior of any interval of $\bar{T}$.

We divide B into kl boxes by drawing the planes $z = z_i$ for $1 \leq i < k$ and the planes $y = y_j$ for $1 \leq j < l$. See Figure 2(b). This decomposition of B into kl boxes can easily be constructed in a tree-like fashion by performing $(k-1)(l-1)$ cuts. We refer to these cuts as *α-cuts.* Let $\mathcal{C}$ be the set of boxes into which B is partitioned in this manner. Agarwal et al. show that for each box $C \in \mathcal{C}$, F_C contains only two classes of rectangles [1]. They also show that if the rectangles in F_C have aspect ratios bounded by a constant, then we make only a constant number of α-cuts.

Dividing Stage: We refine each box C in $\mathcal{C}$ by applying cuts as described below. We can show that if all rectangles in S_C are long, the dividing stage constructs a BSP of near-linear size inside C. Let V_C be the set of vertices of rectangles in S_C that lie in the interior of C. Recall that F_C is the set of rectangles in F_B that are clipped within C. We recursively invoke the dividing stage until $|F_C| + 2a|V_C| < (f + ak)/a$ and S_C does not contain any free rectangles.

1. If C has any free rectangle, we use the free cut containing that rectangle to split C into two boxes.
2. If the rectangles in F_C belong to two classes, we make
   (i) either one cut that partitions C into two boxes C_1 and C_2 so that $|F_{C_i}| + a|V_{C_i}| \leq 2(|F_C| + a|V_C|)/3$, for $i = 1, 2$ (if there are many such cuts, we choose the one that intersects the smallest number of rectangles), or
   (ii) at most two parallel cuts that divide C into three boxes C_1, C_2, and C_3 with an $i \leq 3$ such that $|F_{C_i}| + a|V_{C_i}| \geq (|F_C| + a|V_C|)/3$ and such that all rectangles in F_{C_i} belong to the same class (these two cuts are unique).
3. If F_C has only one class of rectangles, let g be the face of C that contains exactly one of the edges of each rectangle in F_C. Let P be the set of those vertices of the rectangles in F_C that lie in the interior of g. We use a plane that is orthogonal to g to partition C into two boxes C_1 and C_2 so that $|P \cap C_i| + a|V_{C_i}| \leq 2(|P| + a|V_C|)/3$, for $i = 1, 2$. If there are many such planes, we use the plane that intersects the smallest number of rectangles in S_C.

Implementation issues:

1. The performance of algorithm Rounds depends on the value of the parameter a. In our current implementation, if $k = 0$ (all rectangles in S_B are long), we set $a = f$; this setting ensures that the BSP inside B is constructed completely in one round. Otherwise we use $a = 2^{\sqrt{\log(f+k)}}$, the value Agarwal et al. use to analyze their algorithm [1]. We are currently investigating how other values of a change the performance of the algorithm.
2. We have implemented many variants of Rounds. For example, we can change the conditions for stopping the dividing stage as follows: let D be a box obtained by applying the dividing stage at a box $C \in \mathcal{C}$. Recall that F_C has rectangles belonging only to two classes. We stop the dividing stage at D if no rectangles in F_C belong to class m but m is the largest class in S_D. The intuition behind this change is that since the cuts made in the dividing stage do not intersect long rectangles belonging to the other two classes, it is "safe" to continue executing the dividing stage until the size of class m becomes "large." In another variant, if a box B contains $m > 0$ vertices in its interior, we split B into two boxes each with at most $2m/3$ vertices. If there are many planes that split B in this manner, we use the plane that intersects the smallest number of rectangles in S_B. If $m = 0$, we execute one round at B to

complete the construction of the BSP inside B. This algorithm is inspired by an algorithm that Agarwal et al. describe that constructs BSPs of size $O(n^{4/3})$ [1]. In this paper, we do not report on these variants since their performance is not better than that of Rounds.

4 Other Algorithms

In this section we discuss our implementation of some heuristics available in the literature for constructing BSPs. It should be noted that some of the heuristics discussed below were originally developed to construct BSPs for arbitrarily-oriented polygons in $\mathbb{R}^3$. All the algorithms work on the same basic principle: examine all the planes containing the rectangles in S_B and determine how "good" each plane is. Split B using the "best" plane and recurse. Our implementation refines the original descriptions of these heuristics in two respects: (i) At a node B, we first check whether S_B (recall that S_B is the set of rectangles in S clipped within B) contains a free rectangle; if it does, we apply the free cut containing that rectangle.[2] (ii) If there is more than one "best" plane, we choose the medial plane.[3] To complete the description of each heuristic, it suffices to describe how the heuristic measures how "good" a candidate plane is.

For a plane π, let f_π denote the number of rectangles in S_B intersected by π, f_π^+ the number of rectangles in S_B completely lying in the positive halfspace defined by π, and f_π^- the number of rectangles in S_B lying completely in the negative halfspace defined by π. We also define the *occlusion factor* α_π to be the ratio of the total area of the rectangles in S_B lying in π to the area of π (when π is clipped within B), the *balance* β_π to be the ratio $\min\{f_\pi^+, f_\pi^-\}/\max\{f_\pi^+, f_\pi^-\}$ between the number of polygons that lie completely in each halfspace defined by π, and σ_π to be the *split factor* of π, which is the fraction of rectangles that π intersects, i.e., $\sigma_\pi = f_\pi/|S_B|$. We now discuss how each algorithm measures how good a plane is.

ThibaultNaylor: Thibault and Naylor [26] present three different heuristics (w is a positive weight that can be changed to tune the performance of the heuristics):

1. Pick a plane the minimizes the function $|f_\pi^+ - f_\pi^-| + wf_\pi$. This measure tries to balance the number of rectangles on each side of π so that the depth of the BSP is small and also tries to minimize the number of rectangles intersected by π.
2. Maximize the measure $f_\pi^+ \cdot f_\pi^- - wf_\pi$. This measure is very similar to the previous one, except that much more weight is given to constructing a balanced BSP.
3. Maximize the function $f_\pi^+ - wf_\pi$. Thibault and Naylor state in their paper that this measure is motivated by applications in Constructive Solid Geometry (see their paper for more details). This heuristic performed very poorly in our experiments. We will not discuss it further.

In our experiments, we use $w = 8$, as suggested by Thibault and Naylor [26].

Airey: In his thesis, Airey [3] proposes a measure function that is a linear combination of a plane's occlusion factor, its balance, and its split factor:

$$0.5\alpha_\pi + 0.3\beta_\pi + 0.2\sigma_\pi.$$

[2]Only Paterson and Yao's algorithm [22] originally incorporated the notion of free cuts.

[3]Only Teller's algorithm [25] picked the medial plane; the other algorithms do not specify how to deal with multiple "best" planes.

Teller: Let $0 \leq \tau \leq 1$ be a real number. Teller [25] chooses the plane with the maximum occlusion factor α_π, provided $\alpha_\pi \geq \tau$. If there is no such plane, he chooses the plane with the minimum value of f_π. The intuition behind this algorithm is that planes that are "well-covered" are unlikely to intersect many rectangles. Further, if the data set contains many coplanar rectangles, planes containing such rectangles are likely to be used as cutting planes near the root of the BSP, thus constructing a BSP with a small number of nodes. We use the value $\tau = 0.5$ in our implementation. If S has many coplanar polygons, Teller's algorithm is likely to construct a BSP with a small number of nodes. However, queries like ray shooting might be costly since such queries involve processing the (large number of) rectangles stored with each bisecting plane.

PatersonYao: We have implemented a refined version of the algorithm of Paterson and Yao. For a box B, let s_x (resp., s_y, s_z) denote the number of edges of the rectangles in S_B that lie in the interior of B and are parallel to the x-axis (resp., y-axis, z-axis). We define the measure of B to be $\mu(B) = s_x s_y s_z$. We make a cut that is perpendicular to the smallest set of segments and divides B into two boxes, each with measure at most $\mu(B)/4$. (Paterson and Yao prove that given any axis, we can find such a cut perpendicular to that axis [22].) We can show that this algorithm also produces BSPs of size $O(n\sqrt{n})$ for n rectangles, just like Paterson and Yao's original algorithm [22].

5 Experimental Results

We have implemented the above algorithms and run them on the following data sets containing orthogonal rectangles:[4]

1. the `Fifth` floor of Soda Hall containing 1677 rectangles,
2. the `Entire` Soda Hall model with 8690 rectangles,
3. the Orange United Methodist `Church` Fellowship Hall with 29988 rectangles,
4. the Sitterson Hall `Lobby` with 12207 rectangles, and
5. `Sitterson` Hall containing 6002 rectangles.

We obtained the first two data sets from the Department of Computer Science, University of California at Berkeley and the last three from the Walkthrough Project, Department of Computer Science, University of North Carolina at Chapel Hill.

We present three sets of results. For each set, we first discuss the experimental set-up and then present the performance of our algorithms.

5.1 Size of the BSP

Recall that we have defined the size of a BSP to be the sum of the interior nodes in the BSP and the total number of rectangles stored at all the nodes of the BSP. The table below displays the size of the BSP and the total number of times the rectangles are fragmented by the cuts made by the BSP. Note that the size of a BSP is the sum of the number of interior nodes in the BSP, the number of input rectangles, and the number of fragments created by the BSP.

[4]We discarded all non-orthogonal polygons from these data sets. The number of such polygons was very small.

Number of Fragments						**Size of the BSP**				
`Fifth`	`Entire`	`Church`	`Lobby`	`Sitt.`	`Datasets`	`Fifth`	`Entire`	`Church`	`Lobby`	`Sitt.`
1677	8690	29988	12207	6002	#rectangles	1677	8690	29988	12207	6002
113	741	838	475	312	Rounds	2744	14707	45427	22225	9060
301	1458	873	514	153	Teller	2931	14950	33518	13911	7340
449	5545	12517	9642	6428	PatersonYao	3310	22468	56868	30712	20600
675	7001	5494	5350	8307	Airey	3585	24683	41270	21753	19841
1868	10580	13797	3441	1324	ThibaultNaylor1	6092	32929	65313	25051	10836
262	2859	6905	1760	1601	ThibaultNaylor2	3235	20089	58175	23159	12192

Examining this table, we note that, in general, the number of fragments and size of the BSP scale well with the size of the data set. For the Soda Hall data sets (`Fifth` and `Entire`), algorithm Rounds creates the smallest number of fragments and constructs the smallest BSP. For the other three sets, algorithm Teller performs best both in terms of number of fragments and size. However, there are some peculiarities in the table. For example, for the `Church` data set, Rounds creates a smaller number of fragments than Teller but constructs a larger BSP. We believe that this difference is explained by the number of distinct planes that support the input rectangles. For example, the 29998 rectangles in the `Church` model lie in a total of only 859 distinct planes. Since Teller makes cuts based on how much of a plane's area is covered by rectangles, it is reasonable to expect that the algorithm will "place" a lot of rectangles in cuts made close to the root of the BSP, thus leading to a BSP with a small number of nodes. We further examined the issue of how well the performance of the algorithms scaled with the size of the data by running the algorithms on increasingly larger subsets of the data sets. In Figure 3, we display the results of this experiment for the entire Soda Hall and the `Church` models. We have omitted graphs for the other data sets for lack of space.

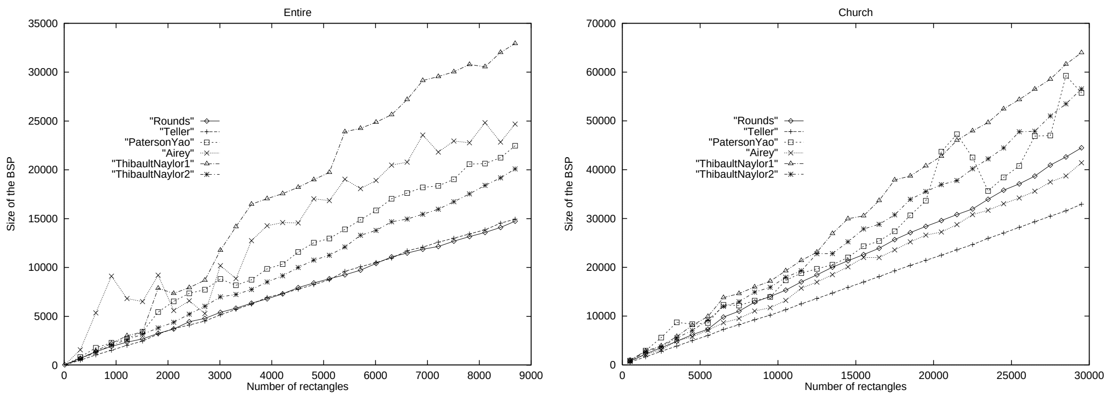


Figure 3: Graphs displaying BSP size vs. #rectangles in S.

The time taken to construct the BSPs also scaled well with the size of the data sets. Rounds took 11 seconds to construct a BSP for the `Fifth` floor of Soda Hall and about 4.5 minutes for the church data set. Typically, PatersonYao took about 15% less time than Rounds while the heuristics (Airey, ThibaultNaylor, and Teller) took 2-4 times as much time as Rounds to construct a BSP.

5.2 Point Location

In the point location query, we are given a set of random points and are asked to locate the leaf of the BSP that contains each point. We create the queries by generating random points that lie in the box B containing all the rectangles in S. We answer such queries by traversing the path from the root of the BSP that leads to the leaf that contains the query point.

We present a summary of the results for point location since they were similar for most of the algorithms. These results were highly correlated to the depth of the trees. For example, Rounds constructed BSPs of average depth between 11 and 16. The average cost of locating a random point ranged between 10 and 15. ThibaultNaylor1 constructed BSPs about twice as deep as ThibaultNaylor2, bearing out our intuition.

5.3 Ray Shooting

Given a ray ρ oriented in a random direction, we are required to determine the first rectangle in S that is intersected by ρ or report that there is no such rectangle. We first locate the leaf v containing the origin of ρ. Then we trace ρ through the leaves of the BSP as follows: we determine the point p where ρ intersects the boundary of v. If p lies inside a rectangle in S (such a rectangle must be stored with the bisecting plane of an ancestor of v or lie on the boundary of B), we report the rectangle as the answer to the query. Otherwise, we locate the other node w whose boundary p lies on by answering a point location query. If there is no such node, then p lies on the boundary of B and ρ does not intersect any rectangle. Otherwise, we continue tracing ρ at w. There are two components to the cost of answering the query with ρ: the number of nodes visited and the number of rectangles checked. We report the two factors separately below. The actual cost of a ray shooting query is a linear combination of these two components; its exact form depends on the implementation.

Ray shooting costs					
#nodes visited					
	`Fifth`	`Entire`	`Church`	`Lobby`	`Sitt.`
Rounds	44.71	12.57	326.40	89.68	55.92
Teller	17.19	13.74	96.64	13.04	37.30
PatersonYao	40.06	11.85	531.47	49.83	83.12
Airey	24.02	13.31	170.26	10.59	129.99
ThibaultNaylor1	44.10	31.34	256.81	102.61	69.14
ThibaultNaylor2	44.56	14.20	298.54	78.40	59.85
#rectangles checked					
Rounds	5.78	3.03	49.60	2.02	19.24
Teller	12.08	11.02	4828.28	20.47	44.18
PatersonYao	4.05	5.71	5461.23	84.24	114.09
Airey	5.51	4.10	4757.91	11.94	27.55
ThibaultNaylor1	4.60	14.63	20.77	2.19	38.56
ThibaultNaylor2	5.14	7.59	28.54	2.67	7.33

There is an interesting tradeoff between these two costs. This tradeoff is most sharply noticeable for the `Church` data set. Notice that the average number of nodes visited to answer ray shooting queries in the BSP constructed by Teller is about a third the number visited in the BSP built by Rounds but the number of rectangles checked in the Teller BSP is about 10 times higher! This apparent discrepancy actually ties in with our earlier conclusion that Teller is able to construct a

BSP with a small number of nodes for the `Church` model because the rectangles in this model lie on a small number of distinct planes. As a result, the ray shooting queries do not visit too many nodes. However, whenever a point is checked to see if it lies in an input rectangle, a large number of rectangles are checked; this is because each plane contains a large number of rectangles. This cost can be brought down by using an efficient data structure for range searching among rectangles. However, this change will increase the size of the BSP itself. Determining the right combination needs further investigation.

5.4 Line Stabbing

In the line stabbing query, we are asked to determine the number of leaves of the BSP intersected by a given line. To answer a line stabbing query with a line ℓ, we clip ℓ within the box containing all the rectangles in S to obtain a segment. We determine the leaf v of the BSP that contains one endpoint p of the segment by making a point location query with p. Then we trace ℓ through the leaves of the BSP as follows: suppose p' is the other point where ℓ intersects v's box. We find the other leaf v' on whose boundary p' lies; v' is the next leaf intersected by ℓ. The total cost of this procedure is the number of nodes visited by all the point location queries. We are currently performing experiments that use different kinds of stabbing lines to query a BSP with.

6 Conclusions

Our comparison indicates that Rounds and Teller are the best algorithms for constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. Teller is best for applications like painter's algorithm in which the entire BSP is traversed. On the other hand, for queries such as ray shooting, it might be advisable to use Rounds since the size of the BSP constructed by this algorithm is not much more than Teller but the query costs are better. It is possible that a better solution might be a combination of both algorithms. For example, one possibility is that we take rectangle areas into account while making cuts in the dividing stage of algorithms Rounds.

Note that we can change the performance of Teller, Airey and ThibaultNaylor by changing the internal weights used by these algorithms. It is possible to use a technique like simulated annealing to determine the best set of weights. However, it is likely that the time taken by such procedures will be prohibitive.

Clearly, there is a tradeoff between the amount of time spent on constructing the BSP and the size of the resulting BSP. Our experience suggests that Rounds and Teller are likely to be fast in terms of execution and will also build compact BSPs.

Acknowledgments We would like to thank Seth Teller for providing us with the Soda Hall data set created at the Department of Computer Science, University of California at Berkeley. We would also like to thank the Walkthrough Project, Department of Computer Science, University of North Carolina at Chapel Hill for providing us with the data sets for Sitterson Hall, the Orange United Methodist Church Fellowship Hall, and the Sitterson Hall Lobby.

References

[1] P. K. Agarwal, E. F. Grove, T. M. Murali, and J. S. Vitter, Binary space partitions for fat rectangles, *Proc. of the 37th Annual IEEE Symp. on the Found. of Comp. Sci.*, 1996, pp. 482–491.

[2] P. K. Agarwal and S. Suri, Surface approximation and geometric partitions, *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, 1994, pp. 24–33.

[3] J. M. Airey, *Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations*, Ph.D. Thesis, University of North Carolina, Chapel Hill, 1990.

[4] C. Ballieux, Motion planning using binary space partitions, Tech. Rep. inf/src/93-25, Utrecht University, 1993.

[5] A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. Thesis, University of Texas, Austin, December 1991.

[6] T. Cassen, K. R. Subramanian, and Z. Michalewicz, Near-optimal construction of partitioning trees by evolutionary techniques, *Proc. Graphics Interface '95*, 1995, pp. 263–271.

[7] N. Chin, *Near Real-Time Object-Precision Shadow Generation Using BSP Trees*, M.S. Thesis, University of Columbia, 1990. Available as Technical Report CUCS-068-90.

[8] N. Chin and S. Feiner, Near real-time shadow generation using BSP trees, *Proc. SIGGRAPH '89*, *Comput. Graph.*, Vol. 23, ACM SIGGRAPH, 1989, pp. 99–106.

[9] N. Chin and S. Feiner, Fast object-precision shadow generation for areal light sources using BSP trees, *Computer Graphics (1992 Symp. on Interactive 3D Graphics)* Vol. 25, 1992, pp. 21–30.

[10] M. de Berg, Linear size binary space partitions for fat objects, *Proceedings of the Third European Symposium on Algorithms (ESA)*, LNCS 979, Springer-Verlag, September 1995, pp. 252–263.

[11] S. E. Dorward, A survey of object-space hidden surface removal, *Internat. J. Comput. Geom. Appl.*, 4 (1994), 325–362.

[12] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

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

[14] T. A. Funkhouser, RING: A client-server system for multi-user virtual environments, *1995 Symp. on Interactive 3D Graphics* ACM SIGGRAPH, 1995, pp. 85–92.

[15] C. Mata and J. S. Mitchell, Approximation algorithms for geometric tour and network design problems, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 360–369.

[16] T. M. Murali and T. A. Funkhouser, Consistent solid and boundary representations from arbitrary polygonal data, *Proc. of the 1997 Symp. on Interactive 3D Graphics (to appear)*, 1997.

[17] B. Naylor, Constructing good partitioning trees, *Proc. Graphics Interface '93*, 1993, pp. 181–191.

[18] B. Naylor and W. Thibault, Application of BSP trees to ray-tracing and CSG evaluation, Technical Report GIT-ICS 86/03, Georgia Institute of Tech., School of Information and Computer Science, 1986.

[19] B. F. Naylor, SCULPT: an interactive solid modeling tool, *Proc. Graphics Interface '90*, May 1990, pp. 138–148.

[20] B. F. Naylor, J. Amanatides, and W. C. Thibault, Merging BSP trees yields polyhedral set operations, *Proc. SIGGRAPH '90*, *Computer Graphics*, Vol. 24, ACM SIGGRAPH, 1990, pp. 115–124.

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

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

[23] R. A. Shumacker, R. Brand, M. Gilliland, and W. Sharp, Study for applying computer-generated images to visual simulation, Report AFHRL-TR-69-14, U.S. Air Force Human Resources Lab., 1969.

[24] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *ACM Comput. Surv.*, 6 (1974), 1–55.

[25] S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. Thesis, Department of Computer Science, University of California, Berkeley, 1992.

[26] W. C. Thibault and B. F. Naylor, Set operations on polyhedra using binary space partitioning trees, *Proc. SIGGRAPH '87, Computer Graphics*, Vol. 21, ACM SIGGRAPH, 1987, pp. 153–162.