Constructing Binary Space Partitions for Orthogonal Rectangles in Practice*

T. M. Murali** Pankaj K. Agarwal*** Jeffrey Scott Vitter†

Center for Geometric Computing
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708-0129
Email: {tmax,pankaj,jsv}@cs.duke.edu
WWW: http://www.cs.duke.edu/~{tmax,pankaj,jsv}

Abstract. In this paper, we develop a simple technique for constructing a Binary Space Partition (BSP) for a set of orthogonal rectangles in $\mathbb{R}^3$. Our algorithm has the novel feature that it tunes its performance to the geometric properties of the rectangles, e.g., their aspect ratios.
We have implemented our algorithm and tested its performance on real data sets. We have also systematically compared the performance of our algorithm with that of other techniques presented in the literature. Our studies show that our algorithm constructs BSPs of near-linear size and small height in practice, has fast running times, and answers queries efficiently. It is a method of choice for constructing BSPs for orthogonal rectangles.

1 Introduction

The Binary Space Partition (BSP) is a hierarchical partitioning of space that was originally proposed by Schumacker et al. [19] and was further refined by Fuchs et al. [10]. The BSP has been widely used in several areas, including computer graphics (global illumination [4], shadow generation [6, 7], visibility determination [3, 21], and ray tracing [15]), solid modeling [16, 22], geometric data repair [12], network design [11], and surface simplification [2]. The BSP has

* A preliminary version of this paper appeared as a communication in the *Proceedings of the 13th Annual ACM Symposium on Computational Geometry, 1997*, pages 382–384.

** 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–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.

† 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.

been successful since it serves both as a model for an object (or a set of objects) and as a data structure for querying the object.

Before proceeding further, we give a definition of the BSP. A *binary space partition* $\mathcal{B}$ for a set S of pairwise-disjoint triangles in $\mathbb{R}^3$ is a tree defined as follows: Each node v in $\mathcal{B}$ represents a convex polytope $\mathcal{R}_v$ and a set of triangles $S_v = \{s \cap \mathcal{R}_v \mid s \in S\}$ that intersect $\mathcal{R}_v$. The polytope 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 convex polytopes by a *cutting plane* H_v. At v, we store the equation of H_v and the subset of triangles in S_v that lie in H_v. If we let H_v^- be the negative halfspace and H_v^+ be the positive halfspace bounded by H_v, the polytopes 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 triangles $\{s \cap H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of triangles $\{s \cap H_v^+ \mid s \in S_v\}$. The size of $\mathcal{B}$ is the sum of the number of internal nodes in $\mathcal{B}$ and the total number of triangles stored at all the nodes in $\mathcal{B}$.[1]

The efficiency of most BSP-based algorithms depends on the size and/or the height of the BSP. Therefore, several techniques to construct BSPs of small size and height have been developed [3, 10, 21, 22]. These techniques may construct a BSP of size $\Omega(n^3)$ for some instances of n triangles. The first algorithms with non-trivial provable bounds on the size of a BSP were developed by Paterson and Yao. They show that a BSP of size $\Theta(n^2)$ can be constructed for n disjoint triangles in $\mathbb{R}^3$ [17] and that a BSP of size $\Theta(n\sqrt{n})$ can be constructed for n non-intersecting, orthogonal rectangles in $\mathbb{R}^3$ [18]. Agarwal et al. [1] consider the problem of constructing BSPs for fat rectangles. A rectangle is said to be *fat* if its aspect ratio is at most α, for some constant $\alpha \geq 1$; otherwise, it is said to be *thin*. If m rectangles are thin and the rest are fat, they present 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 $\mathbb{R}^d$ [8].

In this paper, we consider the problem of constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. In many applications, common environments like buildings are composed largely of orthogonal rectangles. Further, it is a common practice (for example, in the BRL-CAD solid modeling system [13, 20]) to approximate non-orthogonal objects by their orthogonal bounding boxes, since such approximations are simple, easy to manipulate, and often serve as very faithful representations of the original objects [9].

Our paper makes two important contributions. First, we develop and implement a simple technique for constructing a BSP for orthogonal rectangles in $\mathbb{R}^3$. Our algorithm has the useful property that it tunes its performance to the geometric structure present in the input, e.g., the aspect ratios of the input rectangles. While Agarwal et al. [1] use similar ideas, our algorithm is considerably simpler than theirs and is much more easy to implement. Moreover, our

[1] For each internal node v, we store the description of the polytope $\mathcal{R}_v$. However, if v is a leaf, we do not store $\mathcal{R}_v$, since it is completely defined by $\mathcal{R}_w$ and the cutting plane h_w, where w is the parent of v. Hence, we do not include the number of leaves while counting the size of $\mathcal{B}$.

algorithm is "local" in the sense that in order to determine the cutting plane for a node v, it examines only the rectangles intersecting $\mathcal{R}_v$. On the other hand, the algorithm of Agarwal et al. is more "global" in nature: to determine how to partition a node v, it uses splitting planes computed at ancestors of v in the BSP. Other "local" algorithms presented in the literature [3, 21, 22] can be easily incorporated into the framework of our algorithm but not into the Agarwal et al. algorithm. We also show that a slightly modified version of our algorithm constructs a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$ for a set of $n-m$ fat and n thin orthogonal rectangles in $\mathbb{R}^3$, achieving the same bound as the algorithm of Agarwal et al. [1].

We have implemented our algorithm to study its performance on "real" data sets. Our experiments show that our algorithm is practical: it constructs a BSP of near-linear size on real data sets (the size varies between 1.5 and 1.8 times the number of input rectangles).

The second contribution of our paper is a methodical study of the empirical performance of a variety of known algorithms for constructing BSPs. Our experiments show that our algorithm performs better than not only theoretical algorithms like that of Paterson and Yao [18] but also most other techniques described in the literature [3, 10, 22]. The only algorithm that performs better than our algorithm on some data sets is Teller's algorithm [21]; even in these cases, our algorithm has certain advantages in terms of the trade-off between the size of the BSP and query times (see Section 4).

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. We use queries that are typically made in many BSP-based algorithms: *point location* (determine the leaf of the BSP that contains a query point) and *ray shooting* (determine the first rectangle intersected by a query ray).

2 Our Algorithm

In this section, we describe our algorithm New for constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. We first give some definitions, most of which are borrowed from Agarwal et al. [1].

We will often focus on a box B and construct a BSP for the rectangles intersecting it. We use S_B to denote the set $\{s \cap B \mid s \in S\}$ of rectangles obtained by clipping the rectangles in S 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.

A box B in $\mathbb{R}^3$ has six faces—*top, bottom, front, back, right,* and *left*. 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*. 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. Finally, for a set of points P, let P_B be the subset of P lying in the interior of B.

Our algorithm is recursive. At each step, we construct a BSP for the set S_B of rectangles intersecting a box B by partitioning B into two boxes B_1 and B_2 using an orthogonal plane, and recursively constructing a BSP for the sets of rectangles S_{B_1} and S_{B_2}. We start by applying the algorithm to a box that contains all the rectangles in S. We use F_B to denote the set of long rectangles in S_B. We define the measure $\mu(B)$ of B to be the quantity $|F_B| + 2k_B$, where k_B is the number of vertices of rectangles in S_B that lie in the interior of B. We say that a class in F_B is *large* if the number of rectangles in that class is at least $\mu(B)/6$. We split B using one of the following steps:

1. If S_B contains free rectangles, we use the free cut containing the median free rectangle to split B into two boxes. Note that such a free cut is not partitioned by any further cuts in B.
2. If all three classes in F_B are large, we split B as follows: Assume without loss of generality that among all the faces of B, the back face of B has the smallest area. Each long rectangle in the front class has an edge that is contained in the back face of B. We can find an orthogonal line ℓ in the back face that passes through an endpoint of one of these edges and does not intersect the interior of any other edge.[2] We split B using a plane that contains ℓ and is perpendicular to the back face of B.
   This step is most useful when all rectangles in F_B are fat (recall that a rectangle is fat if its aspect ratio is bounded by a constant α). In such a case, we can prove that there are $O(\alpha)$ candidates for line ℓ and show that this step is used to split only $O(\alpha)$ boxes that B is recursively partitioned into before one of Steps 3, 4 or 5 is invoked. Thus, we "separate" the long rectangles into distinct boxes without increasing the total number of rectangles by more than a constant factor.
3. If only two classes in F_B are large, we make one cut that does not intersect any rectangle in the two large classes and partitions B into two boxes B_1 and B_2 such that
   (i) either $\mu(B_i) \leq 2\mu(B)/3$, for $i = 1, 2$, or
   (ii) there is an $i \in \{1, 2\}$ such that $\mu(B_i) \geq \mu(B)/3$.
4. If only one class in F_B is large, let g be the face of B that contains exactly one of the edges of each rectangle in B. We use a plane that is orthogonal to g to partition B into two boxes B_1 and B_2 so that after all free cuts in B_1 and B_2 are applied (by repeated invocation of Step 1), each of the resulting boxes has measure at most $2\mu(B)/3$.
5. If no class in F_B is large, we split B into two boxes B_1 and B_2 such that $\mu(B_i) \leq 2\mu(B)/3$, for $i = 1, 2$.

[2] If there is no such line ℓ, we can prove that F_B contains at most two classes of rectangles [1].

The intuition behind Steps 3–5 is that when B contains more short rectangles than long rectangles, we partition B into boxes that contain roughly half the number of short rectangles as B (but possibly as many long rectangles as B).

In Steps 2–5, if there are many planes that satisfy the conditions on the cuts, we use the plane that intersects the smallest number of rectangles in S_B. We recursively apply the above steps to the sub-boxes created by partitioning B. Due to lack of space, we defer an explanation of how we compute these cuts to the full version of the paper.

Remark: If all n rectangles in S are fat, we can prove that there are $O(\alpha)$ candidate lines to consider in Step 2. If we modify Step 2 to partition B using *all* the planes defined by these lines, we can prove that our algorithm constructs a BSP of size $n2^{O(\sqrt{\log n})}$. Furthermore, when m of the rectangles in S are thin, we can further modify our algorithm to construct a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$. The analysis is similar to that of Agarwal et al. [1]. We believe that the size of the BSP constructed by the simpler algorithm New is also $n\sqrt{m}2^{O(\sqrt{\log n})}$. However, we have been unable to prove this claim so far. The experiments we describe in Section 4 show that algorithm New constructs BSPs of linear size in practice.

3 Other Algorithms

In this section, we discuss our implementation of some other techniques presented in the literature for constructing BSPs. Note that some of the algorithms 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: To determine which plane to split a box B with, they examine each plane that supports a rectangle in S_B (recall that S_B is the set of rectangles in S clipped within B) and determine how "good" that plane is. They split B using the "best" plane and recurse. Our implementation refines the original descriptions of these algorithms in two respects: (i) At a node B, we first check whether S_B contains a free rectangle; if it does, we apply the free cut containing that rectangle.[3] (ii) If there is more than one "best" plane, we choose the medial plane.[4] To complete the description of each technique, it suffices to describe how it 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 define the *occlusion factor* α_π to be the

[3] Only Paterson and Yao's algorithm [18] originally incorporated the notion of free cuts.

[4] Only Teller's algorithm [21] picked the medial plane; the other algorithms do not specify how to deal with multiple "best" planes.

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: We discuss two of the three heuristics that Thibault and Naylor [22] present (the third performed poorly in our experiments). Below, 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 height of the BSP is small and also tries to minimize the number of rectangles intersected by π.
2. Maximize the measure $f_\pi^+ f_\pi^- - wf_\pi$. This measure is very similar to the previous one, except that it gives more weight to constructing a balanced BSP.

In our experiments, we use $w = 8$, as suggested by Thibault and Naylor [22].

Airey: Airey [3] proposes a measure function that is a linear combination of a plane's occlusion factor, its balance, and its split factor: $0.5\alpha_\pi + 0.3\beta_\pi + 0.2\sigma_\pi$.

Teller: Let $0 \leq \tau \leq 1$ be a real number. Teller [21] 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. We use the value $\tau = 0.5$ in our implementation, as suggested by Teller. The intuition behind this algorithm is that planes that are "well-covered" are unlikely to intersect many rectangles and that data sets made up of orthogonal rectangles are likely to contain many coplanar rectangles.

PatersonYao: We have implemented a refined version of the algorithm of Paterson and Yao [18]. 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 family of edges 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.) We can show that this algorithm also constructs produces BSPs of size $O(n\sqrt{n})$ for n rectangles, just like Paterson and Yao's original algorithm [18].

Rounds: We briefly describe the algorithm of Agarwal et al. [1]. Their algorithm proceeds in rounds. Each round partitions a box B using a sequence of cuts in two stages, the separating stage and the dividing stage. The separating stage partitions B into a set of boxes $\mathcal{C}$ such that for each box $C \in \mathcal{C}$, F_C contains only two classes of long rectangles. To effect this partition, they use cuts similar to the cut we make in Step 2 of algorithm New. In the dividing stage, they refine each box $C \in \mathcal{C}$ using cuts similar to those made in Steps 3 and 4 of algorithm New until the "weight" of each resulting box is less than the "weight" of B by a certain factor. A new round is executed recursively in each of these boxes. See Agarwal et al. [1] for more details. Below, we refer to their technique as algorithm Rounds.

Our implementations of these algorithms are efficient in terms of running time because we exploit the fact that we are constructing BSPs for orthogonal rectangles. After an initial sort, we determine the cut at any node in time linear in the number of the rectangles intersecting that node. If we were processing arbitrarily-oriented objects, computing a cut can take time quadratic in the number of objects.

We now briefly mention some other known techniques that we have not implemented, since we expect them to have performance similar to the algorithms we have implemented. Naylor has proposed a technique that controls the construction of the BSP by using estimates of the costs incurred when the BSP is used to answer standard queries [14]. While his idea is different from standard techniques used to construct BSPs, the measure functions he uses to choose cutting planes are very similar to the ones used in the algorithms we have implemented. Cassen et al. [5] 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. Note that de Berg's algorithm for constructing BSPs for fat polyhedra [8] cannot be used to solve our problem since rectangles in $\mathbb{R}^3$ are not fat in his model.

4 Experimental Results

We have implemented the above algorithms and run them on the following data sets containing orthogonal rectangles:[5]

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. `Sitt`erson Hall containing 6002 rectangles.

We present three sets of results. For each set, we first discuss the experimental set-up and then present the performance of our algorithms. These experiments were run on a Sun SPARCstation 5 running SunOS 5.5.1 with 64MB of RAM.

4.1 Size of the BSP

Recall that we have defined the size of a BSP to be the sum of the number of interior nodes in the BSP and the total number of rectangles stored at all the nodes of the BSP. The total number of rectangles stored in the BSP is the sum of the number of input rectangles and the number of fragments created by the

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

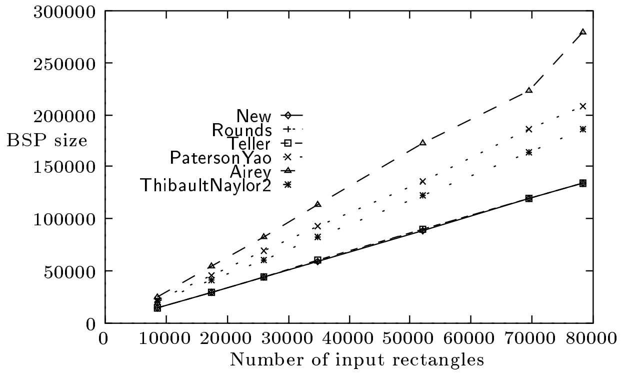
cutting planes in the BSP. The table below displays the size of the BSP and the total number of times the rectangles are fragmented by the cuts made by the BSP.

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
89	**660**	881	681	332	New	**2715**	**14470**	45528	22226	8983
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

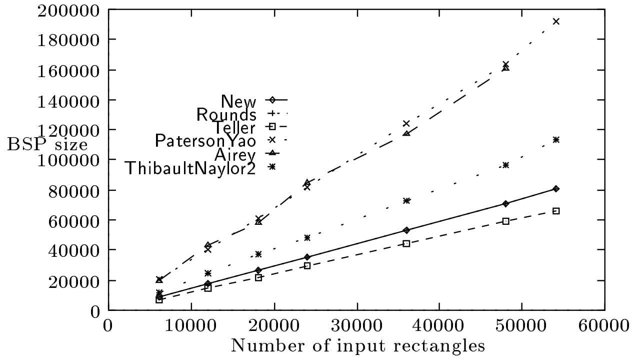
Examining this table, we note that, in general, the number of fragments and size of the BSP scale well with the size of the data set. For the Soda Hall data sets (`Fifth` and `Entire`), algorithm New creates the smallest number of fragments and constructs the smallest BSP. For the other three sets, algorithm Teller performs best in terms of BSP size. However, there are some peculiarities in the table. For example, for the `Church` data set, algorithm Rounds creates a smaller number of fragments than algorithm Teller but constructs a larger BSP. We believe that this difference is explained by the fact that the 29998 rectangles in the `Church` model lie in a total of only 859 distinct planes. Since algorithm 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 data sets that we "created" by making translated and rotated copies of the original data sets. In Figure 1, we display the results of this experiment for the `Entire` Soda Hall and the `Sitterson` models. We have omitted graphs for the other data sets due to lack of space. In these graphs, we do not display the curve for algorithm ThibaultNaylor1 since its performance is always worse than the performance of algorithm ThibaultNaylor2. The graphs show that the size of the BSP constructed by most algorithms increases linearly with the size of the data. The performance of algorithms New, Rounds, and Teller is nearly identical for the `Entire` data set. However, algorithm Teller constructs a smaller BSP than algorithm New for the `Sitterson` data set. For this data set, note the performance of algorithm New is nearly identical to the performance of algorithm Rounds.

The time taken to construct the BSPs also scaled well with the size of the data sets. Algorithm New 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, algorithm PatersonYao took about 15% less time than algorithm New while the other algorithms (Airey, ThibaultNaylor, and Teller) took 2–4 times as much time



(a) `Entire`: Algorithms New, Rounds, and Teller have nearly identical graphs.



(b) `Sitterson`: Algorithms New and Rounds have nearly identical graphs.

Fig. 1. Graphs displaying BSP size vs. the number of input rectangles.

as algorithm New to construct a BSP. While the difference in time is negligible for small data sets, it can be considerable for large data sets. For example, for the data set obtained by placing 9 copies of the `Sitterson` model in a 3×3 array, algorithm New took 13 minutes to construct a BSP while algorithm Teller took 51 minutes.

4.2 Point Location

In the point location query, given a point, we want to locate the leaf of the BSP that contains the point. We answer a query by traversing the path from the root of the BSP that leads to the leaf that contains the query point. The cost of a query is the number of nodes in this path. In our experiments, we create the queries by generating 1000 random points from a uniform distribution over the box B containing all the rectangles in S.

Due to lack of space, we present a summary of the results for point location, concentrating on algorithms New, Rounds, and Teller These results were highly correlated to the height of the trees. Algorithms New and Rounds constructed BSPs of average height between 11 and 16 with the standard deviation of the height ranging from 2 to 2.5. The average cost of locating a point ranged between 10 and 15. The average height of the BSP constructed by algorithm Teller ranged between 15 and 20 with standard deviation ranging from 4 to 5, while the average cost of point location ranged between 7 and 15.

4.3 Ray Shooting

Given a ray ρ, we want to determine the first rectangle in S that is intersected by ρ or report that there is no such rectangle. To answer such a query, we trace ρ through the leaves of the BSP that ρ intersects. At each such leaf v, we check whether the first point where ρ intersects the boundary of v is contained in 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). If so, we output the rectangle and stop the query. Otherwise, we continue tracing ρ. 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. In our experiment, we constructed 1000 rays for each data set by generating 1000 random (origin, direction) pairs, where the origin was picked from a uniform distribution over B and the direction was chosen from a uniform distribution over the sphere of directions.

#nodes visited						#rects. checked				
Fifth	Entire	Church	Lobby	Sitt.		Fifth	Entire	Church	Lobby	Sitt.
43.7	10.8	311.5	86.8	56.0	New	5.6	3.7	48.3	2.6	19.8
44.7	12.5	326.4	89.6	55.9	Rounds	5.7	3.0	49.6	2.0	19.2
17.1	13.7	96.6	13.0	37.3	Teller	12.0	11.0	4828.2	20.4	44.1
40.0	11.8	531.4	49.8	83.1	PatersonYao	4.0	5.7	5461.2	84.2	114.0
24.0	13.3	170.2	10.5	129.9	Airey	5.5	4.1	4757.9	11.9	27.5
44.1	31.3	256.8	102.6	69.1	ThibaultNaylor1	4.6	14.6	20.7	2.1	38.5
44.5	14.2	298.5	78.4	59.8	ThibaultNaylor2	5.1	7.5	28.5	2.6	7.3

There is an interesting tradeoff between these two costs, which 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 algorithm Teller is about a third the number visited in the BSP built by algorithm New but the number of rectangles checked for algorithm Teller is about 10 times higher! This apparent discrepancy actually ties in with our earlier conclusion that algorithm Teller is able to construct a BSP with a small number of nodes for the Church model because the rectangles in this model lie in a small number of distinct planes. As a result, we do not visit too many nodes during a ray shooting query. However, when we check whether the intersection of a ray with a node is contained in a rectangle in S, we process a large number of rectangles since each cutting plane contains a large number of rectangles. This cost can be brought down by using an efficient data structure for point location among rectangles. However, this change will increase the size of the BSP itself. Determining the right combination needs further investigation.

5 Conclusions

Our comparison indicates that algorithms New, Rounds and Teller construct the smallest BSPs for orthogonal rectangles in $\mathbb{R}^3$. Algorithms New and Rounds run 2-4 times faster and construct BSPs with smaller and more uniform height than algorithm Teller.

Algorithm Teller is best for applications like painter's algorithm [9] in which the entire BSP is traversed. On the other hand, for queries such as ray shooting, it might be advisable to use algorithm New or Rounds since they build BSPs whose sizes are not much more than algorithm Teller's BSPs but have better height and query costs. Note that we can prove that algorithms New and Rounds construct BSPs whose height is logarithmic in the number of rectangles in S [1]. Such guarantees are crucial to extending these BSP-construction algorithms to scenarios when the input rectangles move or are inserted into and deleted from the BSP.

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 while algorithm Teller constructs the smallest BSPs, algorithms New and Rounds are likely to be fast in terms of execution, will build compact BSPs that answer queries efficiently, and can be efficiently extended to dynamic environments.

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. 37th Annu. IEEE Sympos. Found. Comput. Sci.*, October 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, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1990.

4. A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. Thesis, Dept. of Computer Sciences, University of Texas, Austin, 1991.
5. 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.
6. 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.
7. N. Chin and S. Feiner, Fast object-precision shadow generation for areal light sources using BSP trees, *Proc. 1992 Sympos. Interactive 3D Graphics*, 1992, pp. 21–30.
8. M. de Berg, Linear size binary space partitions for fat objects, *Proc. 3rd Annu. European Sympos. Algorithms*, *Lecture Notes Comput. Sci.*, Vol. 979, Springer-Verlag, 1995, pp. 252–263.
9. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.
10. H. Fuchs, Z. M. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, *Proc. SIGGRAPH 80*, *Comput. Graph.*, Vol. 14, ACM SIGGRAPH, 1980, pp. 124–133.
11. C. Mata and J. S. B. Mitchell, Approximation algorithms for geometric tour and network design problems, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 360–369.
12. T. M. Murali and T. A. Funkhouser, Consistent solid and boundary representations from arbitrary polygonal data, *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997, pp. 155–162.
13. M. J. Muus, Understanding the preparation and analysis of solid models, in: *Techniques for Computer Graphics* (D. F. Rogers and R. A. Earnshaw, eds.), Springer-Verlag, 1987.
14. B. Naylor, Constructing good partitioning trees, *Proc. Graphics Interface '93*, 1993, pp. 181–191.
15. 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, February 1986.
16. B. F. Naylor, J. Amanatides, and W. C. Thibault, Merging BSP trees yields polyhedral set operations, *Proc. SIGGRAPH 90*, *Comput. Graph.*, Vol. 24, ACM SIGGRAPH, 1990, pp. 115–124.
17. 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.
18. M. S. Paterson and F. F. Yao, Optimal binary space partitions for orthogonal objects, *J. Algorithms*, 13 (1992), 99–113.
19. R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, Study for applying computer-generated images to visual simulation, Tech. Rep. AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, 1969.
20. P. J. Tanenbaum. Applications of computational geometry in army research and development. Invited talk, Second CGC Workshop on Computational Geometry, 1997.
21. S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. Thesis, Dept. of Computer Science, University of California, Berkeley, 1992.
22. W. C. Thibault and B. F. Naylor, Set operations on polyhedra using binary space partitioning trees, *Proc. SIGGRAPH 87*, *Comput. Graph.*, Vol. 21, ACM SIGGRAPH, 1987, pp. 153–162.