Improved Binary Space Partition Merging [⋆]

Mikola Lysenko [a,*] Roshan D'Souza [b] Ching-Kuan Shene [a]

[a] *Michigan Technological University, Department of Computer Science, Houghton, MI 49931 US*

[b] *Michigan Technological University, Department of Mechanical Engineering, Houghton, MI 49931 US*

Abstract

This paper presents a new method for evaluating boolean set operations between Binary Space Partition (BSP) trees. Our algorithm has many desirable features including both numerical robustness and $O(n)$ output sensitive time complexity, while simultaneously admitting a straightforward implementation. To achieve these properties, we present two key algorithmic improvements. The first is a method for eliminating null regions within a BSP tree using linear programming. This replaces previous techniques based on polygon cutting and tree splitting. The second is an improved method for compressing BSP trees based on a similar approach within binary decision diagrams. The performance of the new method is analyzed both theoretically and experimentally. Given the importance of boolean set operations, our algorithms can be directly applied to many problems in graphics, CAD and computational geometry.

Key words: Constructive Solid Geometry, Binary Space Partition, Linear Programming, Tree Merging

1 Introduction

Boolean operations are important in a wide variety of computer aided geometric design problems, including range searching, collision detection, motion

[⋆] Expanded version of a poster presented at SIGGRAPH 2007 (San Diego, CA. August 2007)

[*] Corresponding author.

Email addresses: mikolalysenko@gmail.com (Mikola Lysenko), rmdsouza@mtu.edu (Roshan D'Souza), shene@mtu.edu (Ching-Kuan Shene).

planning and visibility. Yet despite their ubiquity, simple algorithms for evaluating boolean operators are largely unknown. Current approaches are dogged by poor performance, numerical instability and labyrinthine complexity, which forces programmers to resort to expensive commercial packages. In this paper, we give a simple method for computing boolean set operations using Binary Space Partition (BSP) trees. A key improvement within this algorithm is the use of a linear programming feasibility test which removes the need to perform difficult tree partitioning and polygon cutting used in current BSP tree merging methods[1]. Compared to existing BSP tree algorithms, our new approach is substantially simpler, more efficient and robust. We also derive a method for reducing the amount of memory consumed by a labeled leaf BSP tree using a collapsing scheme derived from binary decision diagrams[2].

2 Previous Work

Rossignac gives a good overview of current solid modeling techniques and applications[3]. For this paper, we briefly summarize two general approaches to evaluating boolean operations: Boundary Representations (BREPs) and Constructive Solid Geometry (CSG). BREP methods directly operate on meshes and easily interface with standard file formats and display systems. Laidlaw et al.[4] gave the first BREP algorithm for boolean operations, which Hubbard[5] later improved for triangulated meshes. Recently, Smith and Dodgson gave a BREP intersection algorithm with provable conditions for topological robustness[6]. For NURBs surfaces, Krishnan and Manocha discovered an optimal output sensitive algorithm[7]. Though BREP methods are the most popular category of boolean algorithms, they require complex case-by-case analysis and are difficult to implement. Achieving reasonable robustness and performance requires the use of multiple supporting data structures. Our algorithm eliminates these dependencies by exploiting the implicit spatial indexing within a BSP tree. Like Krishnan and Manocha, we use linear programming to narrow intersection searches, however we go one step further in that the same pruning operation is used throughout the entire BSP tree, thereby reducing the number of special cases.

CSG methods are dual to BREPs in that they represent objects using their interior instead of their boundary[8,9]. Under a sufficiently broad interpretation, this includes voxels, implicit surfaces and spatial indexing trees. CSG defers evaluation of the boolean expression until the last possible moment, such as the time of ray intersection. A similar effect can be achieved with polygon rasterization and z-buffer clipping, though performance scales badly with expression size[10,11]. Recently, Rossignac et al. used modern graphics processing units to dramatically improve the performance of this approach through the use of B-Lists[12]. Common in many BREP and CSG schemes is the use of spatial indexing data structures for accelerating various geometric queries. Samet and Tamminen first made this connection explicit by using quad-trees to compute boolean operations directly[13]. Along the same line of thought, Thibault and Naylor introduced an incremental BSP tree construction to compute boolean operations between arbitrary polyhedra[14], which they later improved using tree merging[1]. Though BSP tree merging is conceptually straightforward, it requires a difficult tree partitioning subroutine to remove null trees. Tree partitioning implicitly uses both null-object detection[15] and active zones[16]. Our algorithm eliminates tree partitioning by making null-object detection explicit.

As both formats have their advantages, it is often necessary to convert between CSG and BREP formulations. For implicit surfaces and voxels, many techniques are known[17–19]. In their original paper on BSP tree merging, Naylor et al. used a dual representation scheme by indexing the polygons within the BSP tree itself and incrementally clipping the boundary while merging[1]. Vanecek gives a more thorough development of this approach, calling the scheme a BREP-index[20]. Comba and Naylor further improved on this result using the topological BSP (t-BSP) tree formulation, which allows for regularized operations and consistent topology[21]. The opposite conversion from BREP to CSG is most relevant for the construction of spatial indexing data structures, and in general it is much more difficult. In the specific case of BSP trees, BREP to CSG conversion is known as BSP construction. Ogayar et al. observe that while querying BSP trees is extremely fast, construction time and memory costs are major problems[22]. Tóth gives a current survey of results on BSP tree construction[23]. Paoluzzi et al. recently proposed a novel method for reducing the cost of BSP tree construction and related operations using progressive construction[24]. Our algorithm is both orthogonal and complementary to these aforementioned techniques, as we only consider the specific problem of merging BSP trees.

3 Binary Space Partition Trees

BSP trees come in several distinct varieties: node storing[25], leaf-storing[26] and solid[14]. Fuchs et al. first introduced node storing BSP trees to induce a back-to-front ordering on polygons for visible surface determination[25]. Leaf-storing BSP trees generalize planar spatial indexing trees and are used to speed up search problems such as collision detection[27], ray tracing[26] and range finding. For this paper, we use the term BSP tree to mean a solid BSP tree[14], which represents a polyhedral subset of $\Re^d$. The leaves of the tree are labeled as filled or empty spaces, while the nodes are formed by recursively clipping two trees against a partition and then joining them together. As a result, each filled leaf represents a region formed by the intersection of its ancestors' partitions. In this paper, we recursively define a solid BSP tree in terms of its set-theoretic expansion[28,29,24]:

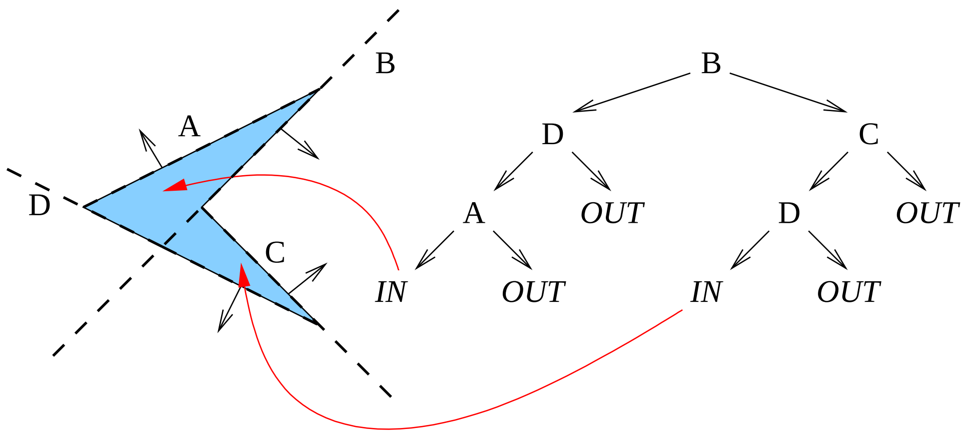

Fig. 1. An example of a binary space partition. On the left, the quadrilateral, ABCD, is a subset of $\Re^2$. A BSP representing this set is shown in the right. The regions represented by the filled nodes are indicated by the arcs.

Definition. A **BSP tree** is written as either **IN**, **OUT** or (B_h, B^+, B^-), which are defined as,

$$\begin{aligned} \mathbf{IN} &\equiv \Re^d \\ \mathbf{OUT} &\equiv \emptyset \\ (B_h, B^+, B^-) &\equiv (B^+ \cap B_h) \cup (B^- \cap B_h^C) \end{aligned}$$

where B_h^C denotes the complement of the set B_h with respect to $\Re^d$, $B_h \subset \Re^d$, and B^+ and B^- are also BSP trees. The BSP trees **IN** and **OUT** are known as **filled** and **empty leaves** respectively. The remaining case, (B_h, B^+, B^-), is a **node**. B_h is called the **partition** of the node, and is a planar halfspace [1]

An instance of a BSP tree is shown in Fig. 1. The quadrilateral on the left is defined by the BSP tree on the right. The bounds of the polygon are formed by the regions $\{A, B, C, D\}$. Diagrammatically, the nodes of the BSP tree are labeled by their corresponding half space, with arrows pointing to their subtrees. Leaves in Fig. 1 are labeled using **IN** for filled and **OUT** for empty. In string form, the BSP tree can be written as,

$$(B, (D, (A, \mathbf{IN}, \mathbf{OUT}), \mathbf{OUT}), (C, (D, \mathbf{IN}, \mathbf{OUT}), \mathbf{OUT})).$$

In set theoretic notation, this BSP tree is equivalent to:

$$(B \cap D \cap A) \cup (B^C \cap C \cap D).$$

Note that $(B \cap D \cap A)$, corresponds to the left **IN** leaf in Fig. 1, while $(B^C \cap C \cap D)$ corresponds to the right **IN** leaf. The geometric interpretation of these sets are shown by the arcs in Fig. 1 going from the leaf nodes of the BSP to

[1] Note: B_h is the region $\{\mathbf{x} | \mathbf{x} \cdot \hat{\mathbf{n}} \leq 0\}$, not the plane $\{\mathbf{x} | \mathbf{x} \cdot \hat{\mathbf{n}} = 0\}$.

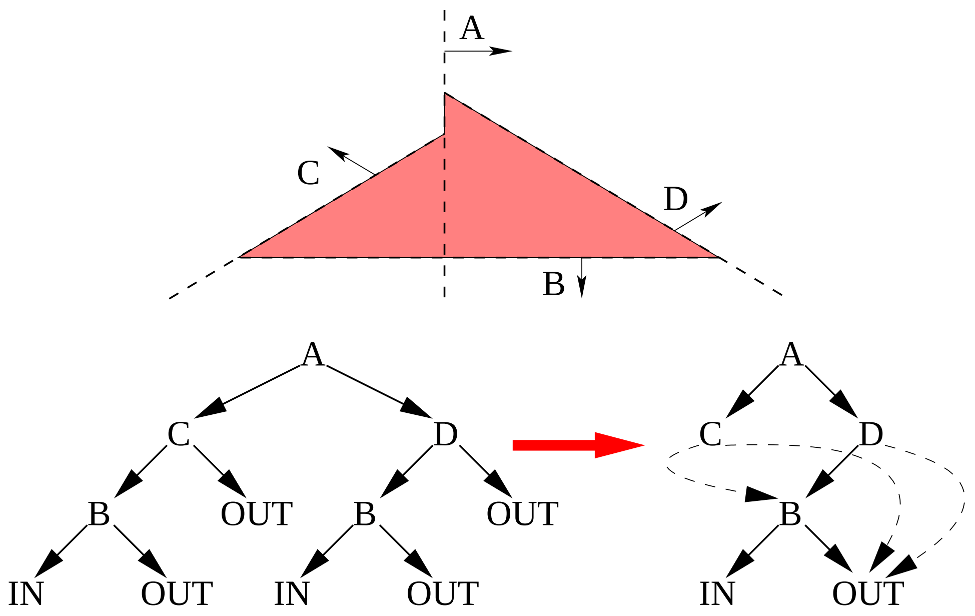

Fig. 2. The result of performing tree collapse. Once again, the quadrilateral ABCD is represented by the BSP tree on the left, as in Fig. 1. The left BSP tree is collapsed using Alg. 1 into the reduced tree on the right. In the compressed tree, redundant subtrees have been replaced with pointers indicated by the dashed lines.

the corresponding regions in the quadrilateral. In general, the filled leaves of a BSP tree form a set of disjoint regions whose union is the original set.

4 Tree Collapsing

Though it is possible to use the simple BSP tree definition given above, without space compression, the trees may grow very large after repeated operations. Ideally, we would like to replace BSP trees representing the same set with a pointer along the lines of Fig. 2, as is done in binary decision diagrams[2]. In the case of Fig. 2, the size of BSP tree is reduced from 5 nodes to 4. Given an ordering on the partitions (such as a lexicographic sorting), it is possible to directly compare equality by enumerating and testing each region within the two trees, however the cost of tree collapse in this case is $O(n^2 log(n))$ on the size the BSP tree. Rather than solve the more difficult general problem, we propose a simpler test based on tree isomorphism. Let $H = \{h_i | h_i \subset \Re^d\}$ be the set of partitions indexed by $i \geq 1$. Define id as an indexing function on the set of all BSP trees such that any two BSP trees, $A = (A_h, A^+, A^-)$ and $B = (B_h, B^+, B^-)$, $id[A] = id[B]$ if

$$A_h = B_h,\ id[A^-] = id[B^-] \text{ and } id[A^+] = id[B^+]. \qquad (1)$$

The case where A and B are leaves is handled by reflexivity. Eqn. 1 enables us to compress the BSP tree by replacing chunks of the tree with pointers into subtrees. However, Eqn. 1 is not sufficient to remove subtrees altogether. In order to achieve this, we add an additional constraint,

$$id[(h_i, B, B)] = id[B], \tag{2}$$

where h_i is a partition and B is a BSP. In [1], a condition similar to Eqn. 2 is given for reducing the size of the BSP trees which considers only the case where B is a leaf. However, because we also consider Eqn. 1, our improved collapsing method can collapse entire trees as well. To calculate *id*, we traverse the BSP tree from the bottom up, as in [2]. To avoid multiple tests, we mark the visited nodes using an hash map, *visit*. The keys in *visit* are BSP trees which are assigned a hash, $hash[B]$, consisting of a 3-tuple of integers,

$$hash[\mathbf{IN}] = (0, 1, 0) \tag{3}$$
$$hash[\mathbf{OUT}] = (0, 2, 0) \tag{4}$$
$$hash[(h_i, B^+, B^-)] = (i, id[B^+], id[B^-]), \tag{5}$$

where $hash[(h_i, B^+, B^-)]$ is the hash of the BSP node (h_i, B^+, B^-), with partition h_i indexed by i and subtrees B^+ and B^-. The base cases for the hash map are determined by Eqn. 3 and Eqn. 4. Using a similar argument to [2], *hash* will never result in a collision between two BSP trees which do not represent the same set. Initially, *visit* is empty and *id* is not set for any nodes. In order to compute *id*, we define a counter *count* which is initialized to 0. As we visit new BSP trees, we set their *id* to the current value of *count* and then increment *count*. The final procedure is summarized in Alg. 1.

Algorithm 1 Collapses the BSP tree, B.

```
procedure collapse[$B$]
 1: if $id[B]$ is set then
 2:     return $B$
 3: end if
 4: if $B = (B_h, B^+, B^-)$ then
 5:     $B \leftarrow (B_h,$ collapse$[B^+]$, collapse$[B^-])$
 6:     if $id[B^+] = id[B^-]$ then
 7:         return $B^+$
 8:     end if
 9: end if
10: if $hash[B] \in visit$ then
11:     return $visit[hash[B]]$
12: end if
13: $id[B] \leftarrow count$
14: $count \leftarrow count + 1$
15: $visit[hash[B]] \leftarrow B$
16: return $B$
```

The values of *count*, *visit* and *id* should be maintained between subsequent calls to Alg. 1. In this manner it is possible to amortize the cost of Alg. 1 during other BSP tree processing algorithms to an optimal constant $O(1)$. If a BSP tree has already been collapsed, then recursion terminates. Another practical concern is finding a consistent indexing on the set of partitions. In our implementation, we used the pointer to the partition, h_i, as its index, i. To achieve reasonable efficiency, some care must be used to recycle the references between planes when constructing the BSP tree.

Strictly speaking, tree collapsing is not necessary for correctness. It does not asymptotically affect the cost of merging, tree construction or element testing. Also, if the BSP trees being merged are simultaneously performing the role of a spatial indexing structure (as is the case with node or leaf storing BSPs), tree collapse does not provide any benefit due to the fact that the objects contained in each sub-tree are distinct. However, in the case of labeled leaf BSP trees, Alg. 1 can drastically reduce the amount of memory consumed by the BSP tree, which in turn improves overall application performance. Though our tree collapse does not guarantee optimal compression, in practice the savings are worthwhile. At the very minimum, the size of the BSP tree is reduced by half due to the compression of leaf nodes. It is also worth noting that tree collapse does not use any geometric information beyond lexicographic ordering.

5 Tree Merging

We define each of the boolean operators (ie. complement, union, intersection and subtraction) in terms of binary functions from $op : \{\mathbf{IN}, \mathbf{OUT}\} \times \{\mathbf{IN}, \mathbf{OUT}\} \rightarrow \{\mathbf{IN}, \mathbf{OUT}\}$. In the case of complement, $A^C = A \oplus IN$, with $\oplus$ being the exclusive-or operator. Given leaves $A, B \in \{\mathbf{IN}, \mathbf{OUT}\}$, we denote the exchange of *op*'s arguments as op^T:

$$A\ op^T\ B = B\ op\ A. \tag{6}$$

A boolean operator can be evaluated over a BSP tree using tree merging. Tree merging recursively inserts one BSP into another until the problem is reduced to an operation between leaf nodes. This process is illustrated in Fig. 3. In this example, the triangle XYZ is intersected with the quadrilateral ABCD, giving the region CDXY. Set union may be performed by switching the roles of **IN** and **OUT**.

While this process does indeed produce correct set operations, it is not very efficient, since the final BSP contains many unnecessary nodes. In Fig. 3, the subtree BDAXYZ is empty and could therefore be removed from the final tree. This can be shown by observing that the region $B \cap D \cap A \cap X = \emptyset$. Using this fact, it is possible to drastically reduce the size of the final BSP. Let T_1 be

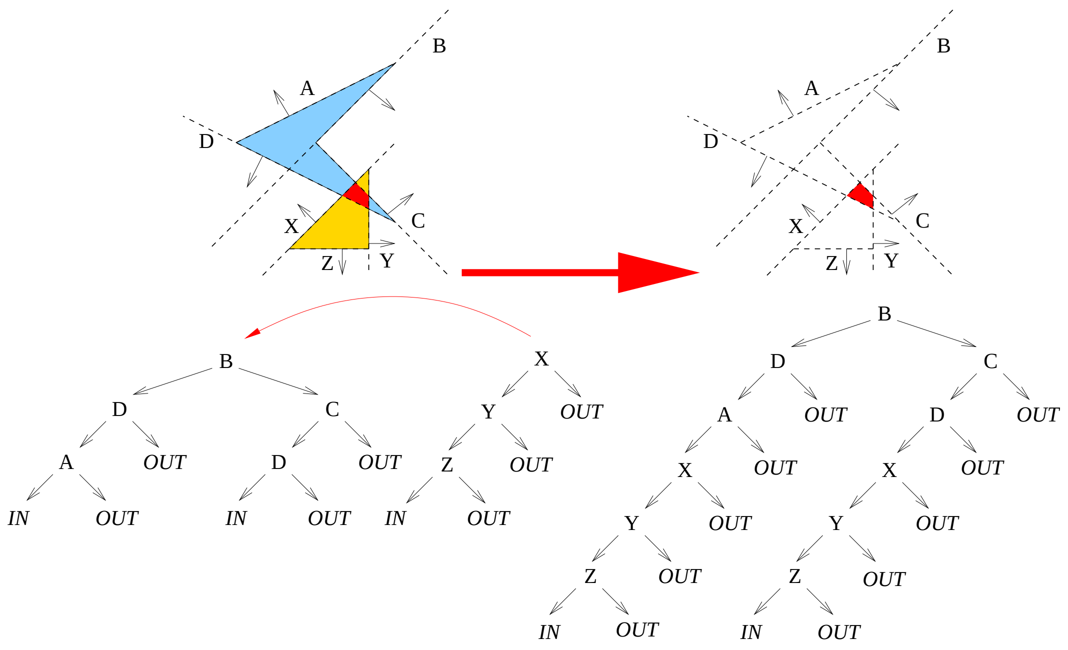

Fig. 3. Tree merging illustrated: On the left, the BSP representing the triangle XYZ is merged with the quadrilateral ABCD to compute their intersection. The corresponding BSPs are shown in the trees below. The result of naïvely merging the trees is shown on the right. Alg. 1 has not been applied to any of the trees.

the subtree YZ and T_2 be the right subtree CDXYZ. Then, the BSP in Fig. 3 may be written as

$$(B, (D, (A, (X, T_1, \mathbf{OUT}), \mathbf{OUT}), \mathbf{OUT}), T_2).$$

Converting this BSP tree into set notation and rearranging terms gives,

$$(B \cap D \cap A \cap X \cap T_1) \cup (B^C \cap T_2) = (B^C \cap T_2). \tag{7}$$

The manipulation in Eqn. 7 factors out the empty region BDAX and T_1 from the rest of the set. Since BDAX is empty, it is possible to remove not only that region, but also T_1. To convert back into a BSP, observe that

$$B^C \cap T_2 = (B \cap \emptyset) \cup (B^C \cap T_2),$$

which is by definition equivalent to the simplified BSP tree $(B, \mathbf{OUT}, T_2)$. In this instance, the total size of the final tree was reduced by half. Detecting empty regions, such as BDAX, requires the use of geometric information from the partitioning sets. Fig. 4 depicts the content of BDAX visually. For the case of planar halfspaces, the problem of detecting empty regions is precisely *linear programming feasibility testing*[30,31]. Since all lower subtrees must be bounded by BDAX, they cannot contain any points, and therefore they should

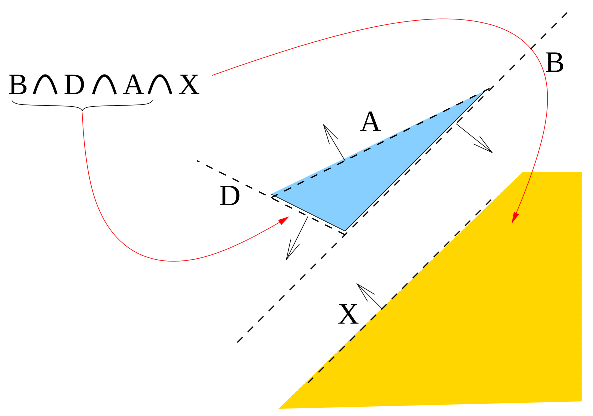

Fig. 4. The region formed by the intersection of hyperplanes BDAX contains no points, since the subregions $B \cap D \cap A$ and X are disjoint.

be dropped from the BSP tree to improve efficiency. A similar observation applies to the final exterior subtree bounded by Z. Eliminating these extraneous regions early in the merging process reduces computational cost by avoiding unnecessary tree traversals. In the case of Fig. 3, merging could be terminated upon reaching the subregion BDAX to avoid visiting nodes Y and Z. The use of linear programming is the main advantage of our merge algorithm over Naylor et al. Naylor's algorithm uses a costly tree partitioning procedure to cull extraneous nodes. Naylor's tree partitioning is based on a difficult polygon clipping test to perform vertex enumeration, which is both slow and numerically unstable. Our algorithm eliminates these issues by making the test for infeasible regions explicit with linear programming, which replaces tree partitioning. To test feasibility, the bounding region formed by the intersection of all ancestor nodes' planes must be stored during traversal. Depending on the linear programming algorithm, intermediate results from this calculation should be cached to ensure optimal performance.

Though linear programming is often considered expensive, in geometric computations it is possible to achieve good performance by exploiting the fact that the number of dimensions, d, is small, $d \leq 3$, with respect to the number of constraints, n. For these cases, solvers such as Seidel's algorithm[32] are ideal. Not only is Seidel's algorithm simple, but it has a running time in $O(d!n)$. There exist even faster algorithms, such as BasisLP, which runs in $O(d^2 n + d^{\sqrt{dlog(d)}} log(n))$[33].

To further reduce the size, we also apply a tree collapsing procedure as described in Section 4 from the bottom up while merging. Our final merge algorithm is as follows: Given two BSP trees, A and B, and a binary boolean operator op, the objective is to compute a new BSP, S, such that $S = A \, op \, B$. To solve for A op B when A and B are not leaves, we recursively insert B into A and solve the problem over the subtrees. The basic structure of this procedure is similar to the example in Fig. 3. To eliminate extraneous regions, as in Fig. 4, we must keep track of the external region as we merge the two trees. To accomplish this, assume that S is bounded by the non-empty region, R, formed by the intersection of its ancestors' partitions. From this, we may rewrite our objective S as

$$S = (A \; op \; B) \cap R. \tag{8}$$

We use a stack to represent R while merging, exploiting the traversal order of the insertion. We use a recursive procedure to solve Eqn. 8. The base case occurs when A and B are both leaves, and is handled by the definition of *op*. If B is a node and A is a leaf, we may swap the order of A and B using Eqn. 6. Therefore, all that remains is the case where $A = (A_h, A^+, A^-)$. Expanding Eqn. 8 yields

$$\begin{aligned}
&(A \; op \; B) \cap R \\
&= (((A^+ \cap A_h) \cup (A^- \cap A_h^C)) \; op \; B) \cap R, \\
&= ((A^+ \; op \; B) \cap A_h \cap R) \cup ((A^- \; op \; B) \cap A_h^C \cap R).
\end{aligned}$$

To simplify, let T^+, T^- be BSP trees such that,

$$\begin{aligned}
T^+ &= (A^+ \; op \; B) \cap (A_h \cap R), \\
T^- &= (A^- \; op \; B) \cap (A_h^C \cap R), \\
S &= T^+ \cup T^-.
\end{aligned}$$

Since T^+ and T^- are partitioned by A_h, $S = (A_h, T^+, T^-)$. This summarizes the content of the tree merging algorithm: split A into A^- and A^+ and recursively apply *op* B. After this is done, recombine the subtrees across the partition A_h. To eliminate extraneous trees, there remain two special cases which must still be handled. If $A_h \cap R = \emptyset$, then $T^+ = \emptyset$ and $S = T^-$. Likewise, if $A_h^C \cap R = \emptyset$, then $S = T^+$ using a symmetric argument. Using this trick, we avoid visiting extraneous BSP trees as we did in our motivating example. To test if $A_h \cap R = \emptyset$, we can use linear programming feasibility if our partitions are planar halfspaces.

There remains one final issue. Given a leaf B, it may occur that for any set X, $X \; op \; B = C$, where C is also a leaf. Under these conditions, B dominates the result, regardless of X. An example of this is the situation where $op = \cap$ and $B = \mathbf{OUT}$, so $X \cap \mathbf{OUT} = \mathbf{OUT}$ for any X. In this case, there is no reason to continue merging since the result will always be C. This can be considered as a base case, and used to terminate recursion when

$$\mathbf{IN} \; op \; B = \mathbf{OUT} \; op \; B. \tag{9}$$

Combining these ideas with Alg. 1 yields Alg. 2.

Algorithm 2 Evaluates a binary operation, *op*, between two binary space partitions, A and B within the region represented by the stack of partitions, R.

```
procedure merge[$A, B, op, R$]
 1: if $R = \emptyset$ then
 2:     return NULL
 3: else if $A$ and $B$ are leaves then
 4:     return collapse[$A\ op\ B$]
 5: else if $A$ is a leaf or heuristic_swap[A,B] then
 6:     swap[$A, B$]
 7:     $op \leftarrow op^T$
 8: end if
 9: if $\mathbf{IN}\ op\ B = \mathbf{OUT}\ op\ B$ then
10:     return collapse[$\mathbf{IN}\ op\ B$]
11: end if
12: push[$A_h, R$]
13: $T^+ \leftarrow$ merge[$A^+, B, op, R$]
14: pop[$R$]
15: push[$A_h^C, R$]
16: $T^- \leftarrow$ merge[$A^-, B, op, R$]
17: pop[$R$]
18: if $T^+ =$ NULL then
19:     return $T^-$
20: else if $T^- =$ NULL then
21:     return $T^+$
22: end if
23: return collapse[$(A_h, T^+, T^-)$]
```

Linear programming is used to handle extraneous subtree elimination in Line 1. This single test replaces the entire splitting procedure in Naylor's algorithm along with its polygon clipping. Since R is an intersection of planar half-spaces, testing whether $R = \emptyset$ is equivalent to linear programming feasibility. When a tree is marked as empty, this is propagated upward by the NULL result, which is used to prune the resulting tree in Lines 18-22. Because A_h partitions $\Re^d$, it is impossible for both T^+ and T^- to be NULL, so NULLs are propagated no more than one level up.

In Line 3, the base case where A and B are leaves is handled. For the sake of simplicity, the algorithm always inserts B into A. Because we change the order of the BSP trees in Line 6, we must exchange the order of operands in *op*. This is done by transposing *op* in Line 7 according to Eqn. 6. The choice of which tree to insert into is arbitrary, and determined by the helper function heuristic_swap. One practical strategy is to always merge into the shortest tree in an attempt to reach the termination condition early. Doing so tends to decrease the size of the final tree, which improves performance, though such a strategy may miss possible early out opportunities. The other base case occurs in Line 9 which handles the termination condition described in Eqn. 9.

The recursion in Alg. 2 occurs in Lines 13 and 16. The node A is split into A^- and A^+, and the Alg. 2 is invoked on both. In order to add the partitions A_h and A_h^C to R, we push them onto the stack in Lines 12 and 15 before calling merge. After merge completes, A_h and A_h^C are popped off the stack in Lines 14 and 17. Finally, we use tree collapsing to save memory in Lines 4, 10 and 23.

It should also be noted that due to the linear programming test used in Line 1, Alg. 2 also solves the non-convex linear programming problem over the set $A\,op\,B$. This is due to the fact that feasibility testing for linear programming is as hard as solving the linear programming problem. This feature is not unique to our algorithm, as Naylor's merge algorithm can be modified to solve linear programming as well. Because Naylor's tree partitioning traverses every vertex of each convex leaf cell, evaluating the objective function incurs an overhead of no more than $O(d)$ per vertex, where d is the dimension of the space $\Re^d$. Taking the maximum of all values over the vertices is an additional $O(1)$ per vertex and solves the problem with no added cost. However, unlike Naylor's algorithm, our method uses linear programming instead of polygon splitting to avoid traversing every vertex of each node inside the BSP tree. Moreover, our linear programming method admits a straightforward generalization to higher dimensional spaces, unlike polygon clipping.

6 Tree Merging Time Complexity

Proving useful time complexity bounds for BSP tree algorithms is difficult. As an illustrative example, consider a set which divides $\Re^2$ into n horizontal bars. Intersecting this set with another set consisting of n vertical bars gives a checkerboard BSP of n^2 boxes. As a consequence, our BSP merging procedure must run in at least $\Omega(n^2)$ worst case. On the opposite extreme, suppose the horizontal BSP has an additional 2^n nodes placed such that they are disjoint from the vertical bars. Intersecting this new BSP gives the same result as before and runs in the same time. Yet, the exponential size of the input dominates the quadratic size of the result, giving the mistaken impression that the algorithm may run in $O(1)$.

This suggests reformulating the problem in terms of an output sensitive complexity measure. Though output sensitive complexity gives no information about Alg. 2's performance relative to non-BSP methods, it does give a direct comparison to Naylor's tree merging algorithm, which was previously the fastest known method for solving boolean set operations between BSP trees. This comparison is valid because both methods generate identical uncollapsed BSP trees, given that they use the same insertion heuristic.

Consider the BSP tree produced by merging two BSP trees without using Alg. 1 to collapse the result. Let n be the number of nodes within this BSP tree, and let h be its height. Alg. 2 must run in at least $\Omega(n)$, since it constructs this result incrementally. After this BSP tree is constructed, adding tree collapse has no effect on the running time of Alg. 2, since it visits every node in the tree no more than once.

Alg. 2 is called no more than twice as often as the size of the final tree. This is because in Lines 13 and 16, it is impossible for both T^+ and T^- to be NULL due to the fact that any partition of a non-empty region, R, must contain at least one non-empty subregion. In this case, since there are only two components to the partition, either T^+ or T^- must be a BSP. Therefore, the total number of calls of merge is also in $\Theta(n)$.

The depth of recursive merge calls never exceeds the height of the uncollapsed BSP, so the number of constraints in the enclosing region, $|R|$, must be within $O(h)$. Therefore, the amount of time consumed by Line 1 is in $O(LP(h, d))$, where $LP(k, d)$ is the cost of solving the linear programming feasibility problem with k constraints in d dimensions. Combining these facts, the total time for executing Alg. 2 is in $O(nLP(h, d))$.

In the best case, $h \in \Omega(log(n))$, giving a running time in $O(nLP(log(n), d))$, although h may be as large as n giving a worst case time of $O(nLP(n, d))$. Empirical evidence suggests that the first value is more accurate for relatively balanced BSP trees. Using a linear time LP algorithm, the total cost becomes $O(nh)$. Incremental linear programming algorithms further improve this bound by using the fact that each successive bound is added one-at-a-time as they are pushed onto the stack. In this case, the cost for testing linear programming feasibility is reduced to a constant, and so the algorithm runs in an optimal $O(n)$ time, indicating that the linear programming test is completely amortized by the cost of BSP traversal.

In Naylor's algorithm, feasibility is solved through polytope clipping. For each partition, they project an enormous polytope onto it and then clip it against R. Doing this requires h cuts, and each cut requires d flops times the number of vertices in the polygon. Since the number of vertices is in this polygon is approximately bounded by $O(h^{\lfloor \frac{d-1}{2} \rfloor})$[34], the total time for evaluating feasibility is within $O(dh^{1+\lfloor \frac{d-1}{2} \rfloor})$. This places the running time for Naylor's algorithm within $O(ndh^{1+\lfloor \frac{d-1}{2} \rfloor})$, which is slower than $O(nLP(h, d))$ given a suitable choice of $LP(h, d)$. From a geometric perspective, Naylor's polytope clipping is isomorphic to linear programming via naïve vertex enumeration. By recasting the problem in terms of linear programming, we directly check for null regions while visiting the minimal number of vertices.

7 Tree Merging Robustness

Numerical robustness is important in all computations involving real numbers. This is especially the case within geometric problems, which rely on the consistency of calculations in order to produce correct results. Many schemes have been proposed to deal with these standard problems, such as introducing random perturbations to avoid degenerate cases or using exact arithmetic to prevent truncation[35].

In Alg. 1 and Alg. 2, the only numerical operation is the linear programming feasibility test, $R = \emptyset$, on Line 1 of Alg. 2. Therefore, the robustness of Alg. 2 is determined by the robustness of the linear programming method. Because there exist methods for solving linear programming *exactly* using finite precision rational arithmetic[36,37], one could validly claim that Alg. 2 is also exact. While these exact algorithms are asymptotically no slower than their floating point counter-parts for a fixed precision, in practice their overhead is substantial. Therefore, for interactive applications it may be desirable to sacrifice exactness for the sake of performance.

To understand the impact of robustness on our algorithm, we consider the consequences of getting an incorrect result from the linear programming feasibility test. There are only two possible ways the linear programming feasibility test can fail,

1. R is incorrectly marked as feasible.
2. R is incorrectly marked as infeasible.

Case 1 is not particularly harmful – the final BSP will be somewhat larger and merging will be slowed down accordingly, but the set represented by the final BSP tree will remain unchanged. Case 2 is pathological – removing non-empty nodes changes the represented set. While improved accuracy will reduce both sorts of errors, it is possible to reduce the number of errors in case 2, at the expense of creating more errors in case 1 with little change in the underlying arithmetic. We do this by offsetting each plane along the normal by an arbitrary epsilon value. More general strategies, such as randomly perturbing the normals of the planes to avoid degeneracies, are also used to improve results with low overhead. Taken together, these constitute a far simpler and faster method for evaluating robust CSG operations.

Compared to Naylor et al.'s algorithm, this represents a substantial improvement in robustness. In their work, the vertices of each polygon were evaluated eagerly at the time of clipping, which leads to a rapid loss of precision and inconsistent results from tree splitting. To solve this issue, they proposed using an epsilon term in each vertex test to add some slack to the computation. Today, exact methods for polygon-plane clipping are known[38], which solve this problem by storing the d-tuple of planes associated with each vertex instead

Table 1
A table of size reductions due to Alg. 1. The size of the uncollapsed tree is listed in the second column, while the collapsed tree is given in the third. The amount of memory which was reduced is shown in the fourth column, with larger values indicating more savings.

Example	No Collapse	Collapsed	% Reduced
Sphere (Unbalanced)	2000	2000	0.0 %
Sphere (Balanced)	10035	9695	3.4 %
Mushroom	1189	1095	6.0 %
Pig	8045	7422	7.6 %
Beethoven	18367	16579	9.6 %
Bunny	128780	120750	6.2 %
Bunny ∩ Knot	46195	32887	28.8 %
Pig ∪ Mushroom	11907	8242	30.8 %
(Pig ∪ Mushroom) ∩ Beethoven	22064	11504	47.8 %

Table 2
Performance benchmarks of Alg. 2 vs. Naylor et al.'s merge. Times do not include BSP construction or surface clipping. All sizes are given in terms of uncollapsed BSP trees. In each benchmark the algorithms were run 100 times, and the average execution time was recorded. All measurements were taken on an AMD Athlon 64 3500+ processor with 1GB of RAM.

Benchmark	Result Size	Result Height	Naylor Time (s)	Alg. 2 Time (s)
Horizontal Planes ∩ Vertical Planes	6561	160	0.56 s	0.02 s
Beethoven ∩ Sphere (Unbalanced)	16393	175	5.80 s	0.67 s
Beethoven ∩ Sphere (Balanced)	34269	51	1.33 s	0.50 s
Pig ∪ Mushroom	26051	61	1.00 s	0.12 s
Bunny ∩ Knot	119883	52	15.79 s	0.83 s
Beethoven ∩ Goblet	21159	44	1.54 s	0.12 s

of the evaluated d-tuple of floats. Likewise, a similar technique is applicable in vertex-based linear programming schemes such as Seidel's algorithm. However, linear programming still maintains an advantage in that even the non-exact solution can be made arbitrarily robust with a direct performance trade off. Because of the traversal pattern of Naylor et al.'s algorithm, it is not easy to realize analogous behavior in handling degeneracies.

8 Results

In our implementation, boundaries were maintained by incrementally clipping each surface against the other BSP tree as in Naylor et al.[1]. For BREP to BSP conversion we used the standard method of selecting a partition, then recursively clipping polygons[14]. For choosing a partition, we tested two different heuristics. In our first heuristic, we chose planes from the polygons which split the minimum number of facets. This heuristic tends to favor small trees,

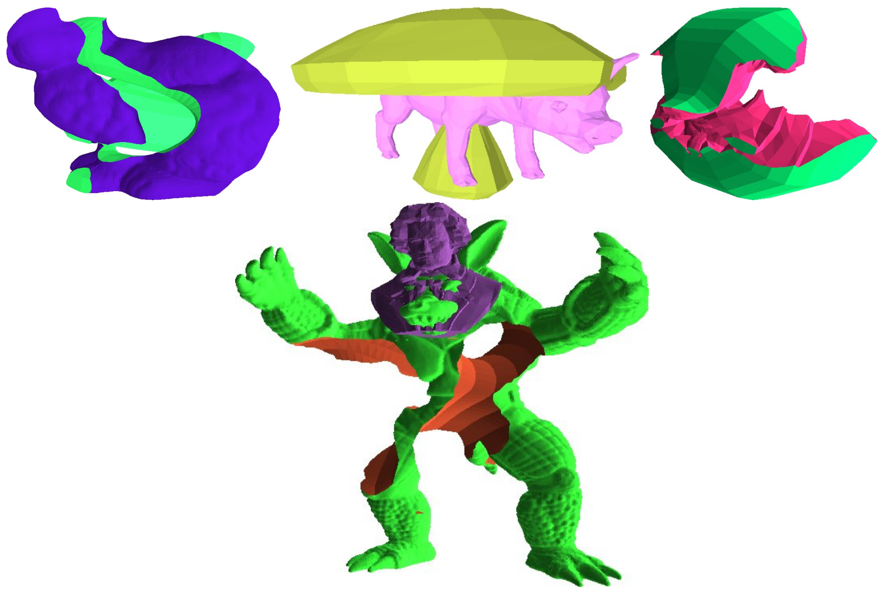

Fig. 5. Some shapes produced by Alg. 2. From left to right: Bunny ∩ Knot, Pig ∪ Mushroom, Beethoven ∩ Goblet, (Armadillo ∪ Beethoven) ∩ Knot

but they are usually not very balanced. Our second heuristic favored balanced trees by introducing a few levels of randomly chosen partitions that cut the polygon set in half. Once the number of polygons was below a fixed threshold, we switched to the first heuristic to finish construction. Even though our second heuristic outperformed the first in every example, we have listed the first in our benchmarks for discussion purposes.

To evaluate the performance of Alg. 1, we tested it on several different shapes using both of our heuristics. The results of the compression are shown in Tab. 1. A convex polyhedron with a completely unbalanced BSP tree is the worst case for Alg. 1 because the initial tree is already optimally compressed. For more typical BSP trees, the average savings given by Alg. 1 are on the order of 3-5%. These values are somewhat low because we don't identify any symmetries in the initial polygonal data. If we labeled all coplanar partitions with the same index, it might be possible to increase this percentage by a nominal amount. The real benefit of Alg. 1 is apparent when used in conjunction with Alg. 2, where Alg. 1 removes on average about 25% more nodes than the worst case. Repeatedly applying Alg. 2 leads to even greater savings, as shown in Tab. 1.

We benchmarked Alg. 2 against Naylor's algorithm, currently the best known method for merging labeled leaf BSP trees. To simplify the benchmark, we only used offsetting and perturbation for robustness. The results of our benchmark are shown in Tab. 2. While timing, we only measured the cost of merge and collapse. Surface clipping, BSP construction and display were not counted. Some of the shapes produced during our test are shown in Fig. 8. One of the most pathological cases for Naylor's algorithm is the case where a number of horizontal bars are intersected against a number of vertical bars. In this particular instance our algorithm is over 95% faster, due to the advantages of our linear programming technique. Since the difference in performance between Alg. 2 and [1] are proportional to the height of the resulting BSP tree, larger and deeper trees exhibit a greater benefit from our approach compared to smaller balanced trees, a fact which is reflected in the data. An interesting example of this are the two different Beethoven $\cap$ Sphere benchmarks, which illustrate the speed up due to linear programming. In the balanced case, our algorithm does not achieve as large a speedup due to the fact that the resulting BSP contains a large number of short trees which quickly terminate. It is also worth noting that for finely tessellated objects, Naylor et al.'s algorithm is prone to failure due to degenerate polygons and numerical instability, which our linear programming method handles robustly.

9 Conclusion

In this paper, we gave an efficient, original method for compressing the size of a BSP tree using a simple bottom-up tree traversal. Additionally, we found a simple, fast and general algorithm for evaluating polyhedral set operations. With an appropriate linear programming algorithm, this technique can be used to evaluate CSG expressions on meshes. Altogether, these contributions generalize and extend a number of fundamental mechanisms in spatial indexing tree structures, and are applicable to a wide range of problems in computational geometry and computer graphics.

Although Alg. 2 is efficient, the times presented in Tab. 2 can be improved with better BSP construction heuristics. Neither of our tree building strategies create many opportunities for rejecting non-intersecting objects, and thus they require many additional tree insertions per boolean operator. A better constructed BSP tree would focus on constructing bounds around objects in order to provide early-out opportunities to quickly terminate traversal. This would in turn reduce the size of the resulting BSP tree and improve performance. Though optimal construction algorithms are known for some limited cases, the general problem remains unresolved. Likewise, recovering a BREP from a BSP tree is also complicated, though easier than the inverse problem. Our surface clipping method is acceptable for a benchmark, but in a real world application a better method would be necessary. Nonetheless, these issues are not unique to Alg. 2, as all existing BSP tree algorithms must deal with them in some way or another, and it should not diminish the importance of our contribution. These issues may prove to be interesting topics for future research.

Acknowledgments

Thanks to Gilbert Bernstein, who caught several errors in earlier drafts of this paper. Also thanks to Nick Smolinske, Ryan Richards and John Earnest. All meshes used in this paper are from the Brown Mesh Set, courtesy of Morgan McGuire and Nick Musurca. This work was partially funded by the NSF through the grant CMMI 0729280. Any opinions, findings and conclusions are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] B. Naylor, J. Amanatides, W. Thibault, Merging bsp trees yields polyhedral set operations, in: SIGGRAPH 90, AT&T Bell Laboratories, ACM Press, New York, NY, USA, 1990, pp. 115–124.

[2] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comp. 35 (8) (1986) 677–691.

[3] J. Rossignac, Solid and physical modeling, Wiley Encyclopedia of Electrical and Electronics Engineering.

[4] D. H. Laidlaw, W. B. Trumbore, J. F. Hughes, Constructive solid geometry for polyhedral objects, in: SIGGRAPH 86, Brown University, ACM Press, New York, NY, USA, 1986, pp. 161–170.

[5] P. M. Hubbard, Constructive solid geometry for triangulated polyhedra, Tech. Rep. CS-90-07, Brown University (January 1990).

[6] J. Smith, N. Dodgson, A topologically robust algorithm for boolean operations on polyhedral shapes using approximate arithmetic, Computer-Aided Design 39 (2) (2007) 149–163.

[7] S. Krishnan, D. Manocha, An efficient surface intersection algorithm based on lower-dimensional formulation, ACM Trans. Graph. 16 (1) (1997) 74–106.

[8] R. B. Tilove, Set membership classification: A unified approach to geometric intersection problems, IEEE Trans. Comput. 29 (10) (1980) 874–883.

[9] A. G. Requicha, Representations for rigid solids: Theory, methods, and systems, ACM Comput. Surv. 12 (4) (1980) 437–464.

[10] J. Goldfeather, J. P. M. Hultquist, H. Fuchs, Fast constructive-solid geometry display in the pixel-powers graphics system, in: SIGGRAPH 86, University of North Carolina at Chapel Hill, ACM Press, New York, NY, USA, 1986, pp. 107–116.

[11] N. Stewart, G. Leach, S. John, An improved z-buffer csg rendering algorithm, in: SIGGRAPH 98, RMIT University, Melbourne, Australia, ACM Press, New York, NY, USA, 1998, pp. 25–30.

[12] J. Hable, J. Rossignac, Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes, in: SIGGRAPH 05, Georgia Institute of Technology, ACM Press, New York, NY, USA, 2005, pp. 1024–1031.

[13] H. Samet, M. Tamminen, Bintrees, csg trees, and time, in: SIGGRAPH 85, University of Maryland, ACM Press, New York, NY, USA, 1985, pp. 121–130.

[14] W. C. Thibault, B. F. Naylor, Set operations on polyhedra using binary space partitioning trees, in: SIGGRAPH 87, Georgia Institute of Technology, ACM Press, New York, NY, USA, 1987, pp. 153–162.

[15] R. B. Tilove, A null-object detection algorithm for constructive solid geometry, Commun. ACM 27 (7) (1984) 684–694.

[16] J. R. Rossignac, H. B. Voelcker, Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms, ACM Trans. Graph. 8 (1) (1989) 51–87.

[17] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3d surface construction algorithm, in: SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, Vol. 21, ACM Press, New York, NY, USA, 1987, pp. 163–169.

[18] T. Ju, F. Losasso, S. Schaefer, J. Warren, Dual contouring of hermite data, in: Siggraph 2002, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2002, pp. 339–346.

[19] S. Forstmann, J. Ohya, Visualization of large iso-surfaces based on nested clip-boxes, in: SIGGRAPH '05: ACM SIGGRAPH 2005 Posters, ACM, New York, NY, USA, 2005, p. 126.

[20] G. V. Jr., BRep-Index: A Multidimensional Space Partitioning Tree, Symposium on Solid Modeling Foundations and CAD/CAM Applications (1991) 35–44.

[21] J. L. D. Comba, B. F. Naylor, Conversion of Binary Space Partitioning Trees to Boundary Representation, Springer Verlag, 1996, Ch. Geometric Modeling: Theory and Practice: Chapter II - Representations, pp. 286–301, iSBN 3-540-61883-X.

[22] C. J. Ogayar, R. J. Segura, F. R. Feito, Point in solid strategies, Computers & Graphics 29 (4) (2005) 616–624.

[23] C. D. Tóth, Binary space partitions: recent developments, Combinatorial and Computational Geometry 52 (2005) 525–552.

[24] A. Paoluzzi, V. Pascucci, G. Scorzelli, Progressive dimension-independent boolean operations, in: G. Elber, N. Patrikalakis, P. Brunet (Eds.), ACM Symp. on Solid Modeling and Applications, 2004, pp. 203–212, sM 04.

[25] H. Fuchs, Z. M. Kedem, B. F. Naylor, On visible surface generation by a priori tree structures, in: SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1980, pp. 124–133.

[26] W. C. Thibault, Application of binary space partitioning trees to geometric modeling and ray-tracing, Ph.D. thesis, Georgia Institute of Technology, director-Bruce F. Naylor (1987).

[27] S. Melax, Dynamic plane shifting BSP traversal, in: Graphics Interface 2000, 2000, pp. 213–220.

[28] S. F. Buchele, Three-dimensional binary space partitioning tree and constructive solid geometry tree construction from algebraic boundary representations, Ph.D. thesis, The University of Texas at Austin, supervisor-Alan K. Cline and Supervisor-Donald S. Fussell (1999).

[29] S. F. Buchele, A. C. Roles, Binary space partitioning tree and constructive solid geometry representations for objects bounded by curved surfaces, in: CCCG, 2001, pp. 49–52.

[30] N. Karmarkar, A new polynomial-time algorithm for linear programming, Combinatorica 4 (4) (1984) 373–395.

[31] P. K. Agarwal, M. Sharir, Efficient algorithms for geometric optimization, ACM Comput. Surv. 30 (4) (1998) 412–458.

[32] R. Seidel, Linear programming and convex hulls made easy, in: Symp. on Computational Geometry, University of California at Berkeley, ACM Press, New York, NY, USA, 1990, pp. 211–215.

[33] K. L. Clarkson, Las vegas algorithms for linear and integer programming when the dimension is small, Journal of the ACM 42 (2) (1995) 488–499.

[34] P. K. Agarwal, Intersection and Decomposition Algorithms for Planar Arrangements, Cambridge University Press, 1991.

[35] C. M. Hoffmann, Geometric and solid modeling: an introduction, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[36] B. Gärtner, Exact arithmetic at low cost - a case study in linear programming, Computational Geometry: Theory and Applications 13 (2) (1999) 121–139.

[37] D. L. Applegate, W. Cook, S. Dash, D. G. Espinoza, Exact solutions to linear programming problems, Operations Research Letters 35 (6) (2007) 693–699.

[38] J. R. Shewchuk, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete & Computational Geometry 18 (3) (1997) 305–363.

[39] M. Lysenko, Realtime constructive solid geometry, in: SIGGRAPH 07: Posters, ACM, New York, NY, USA, 2007, p. 132.

[40] M. Sharir, E. Welzl, A combinatorial bound for linear programming and related problems, in: Symp. on Theoretical Aspects of Computer Science, Springer-Verlag, London, UK, 1992, pp. 569–579.