An Improved Ray Shooting Method for Constructive Solid Geometry Models via Tree Contraction*

Michael T. Goodrich†

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
E-mail: goodrich.cs.jhu.edu

Abstract

In the Constructive Solid Geometry (CSG) representation a geometric object is described as the hierarchical combination of a number of primitive shapes using the operations *union*, *intersection*, *subtraction*, and *exclusive-union*. This hierarchical description defines an expression tree, T, called the *CSG tree*, with leaves associated with primitive shapes, internal nodes associated with operations, and whose "value" is the geometric object. Evaluation of CSG trees is an important computation that arises in many rendering and analysis problems for geometric models, with ray shooting (also known as "ray casting") being one of the most important. Given any CSG tree T, which may be unbalanced, we show how to convert T into a functionally-equivalent tree, D, that is balanced. We demonstrate the utility of this conversion by showing how it can be used to improve the worst-case running time for ray shooting against a CSG model from $O(n^2)$ to $O(n \log n)$, which is optimal.

Keywords: Boolean algebra, constructive solid geometry, CSG tree, geometric data structures, ray casting, ray shooting, solid modeling, tree contraction

*This research was announced in preliminary form in "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, 1990, 118–128.

†This research supported by the NSF and DARPA under Grant CCR-8908092, and by the NSF under Grants CCR-8810568, CCR-9003299, and IRI-9116843.

Figure 1: A CSG Representation. We use $+$ to denote *union* and $*$ to denote *intersection*. Other possible operations are subtraction ($-$) and exclusive-union ($\oplus$).

1 Introduction

Allowing a user to define, manipulate, and analyze realistic geometric objects easily and efficiently is an important goal of geometric modeling. One representation for geometric objects that has received much attention of late, relative to this goal, is the *Constructive Solid Geometry* (CSG) representation (e.g., see [46] or [34]). An object is represented in this model by a set of primitive objects that are hierarchically combined according to various set operations, such as *intersection*, *union*, *subtraction*, and *exclusive-union* (i.e., all points in one or the other, but not both). (See Figure 1.) This hierarchical description defines an expression tree, T called the *CSG tree*, with leaves associated with primitive shapes, internal nodes associated with operations, and whose "value" is the geometric object.

One of the principle advantages of the CSG representation is that it allows the user to define objects in a "bottom-up" fashion, and coincides naturally with the way many people visualize the object being defined. Unfortunately, many natural ways in which to define CSG trees give rise to unbalanced trees, whereas balanced CSG trees would often be most desirable from a computational standpoint. Indeed, there are a number of fundamental computations involving CSG representations that benefit from being given a balanced tree.

Perhaps the most important such computation is *line classification* [55, 56]: determining the intersections of a line with an object defined by a CSG representation, for it is the primitive computation used in many CSG rendering algorithms [26, 28, 34, 38, 46, 48].

1.1 Our Results

In this paper we show how to convert any CSG tree T into a functionally-equivalent balanced tree D of roughly the same size as T, and we show how this leads to a method for line classification in a CSG model that runs in $O(n \log n)$ time, which is optimal and improves the previous $O(n^2)$ bound, due to Tilove [55, 56] and Getto [28], where n is the total number of edges that determine all the primitives stored at the leaves of T. Our method can be implemented using simple operations, such as sorting and table look-ups, and does not require any sophisticated data structures. Moreover, one can apply many of the previous heuristic techniques for speeding up CSG ray shooting to our methods, such as "bounding boxes" [28, 29, 47, 49, 50, 51] and face cutting [37], although none of these heuristics would improve the worst case running time. As an aid to practitioners, we also enumerate some additional heuristics that one can apply to make our method run faster in practice. Thus, we believe our method should run very fast in practice, compared to previous methods. Indeed, justification for this claim has been given recently by Facello [25] in some benchmarking tests.

The main paradigm we exploit for our algorithms is the *tree contraction* technique [1, 7, 36, 45, 40] from parallel algorithm design. The particular approach we follow is that proposed by Miller and Reif [40], where one applies two procedures, "rake" and "compress", to evaluate an m-leaf arithmetic expression tree in parallel [1, 36, 40]. This results in a method that runs in $O(\log m)$ parallel steps, even if the tree is quite imbalanced. We use this tree contraction paradigm to convert any given m-leaf CSG tree T into a *functionally-equivalent* m-leaf expression tree D. Letting $h(D)$ denote the height[1] of D, we show that construction guarantees that $h(D)$ is $O(\log m)$ and that the size of D is $O(m)$, even if T is quite unbalanced and contains exclusive-union nodes. Moreover, even though this is an asymptotic bound, our method guarantees that $h(D) \leq h(T)$. We call D the *dwarf CSG tree*.

1.2 Related Results

Our research is directed at the application of computational geometry techniques to problems in computer graphics. This area has become a very rich research area of late, and the reader interested in an overview of this previous work is referred to the excellent papers by Dobkin [16] and Yao [58]. A good example of such work is the recent work of a number of researchers, including Dobkin and Kirkpatrick [18, 20, 19] and others [2, 4, 8, 10, 11, 15, 30, 32, 33, 43, 52] directed at maintaining a collection of objects in $\Re^d$ so as to quickly answer ray shooting queries. Such data structures could be used, for example, in ray-tracing of a 3-d solid model, assuming one is given a polygonal description of all the faces of the model (which, of course, one does not have in the CSG representation). Interestingly, another example application of computational geometry techniques to computer graphics has been for the problem of constructing a concise CSG representation of an object given its boundary

[1] Recall that the *height* of a binary tree T is the length of a longest root-to-leaf path in T (so that the height of single-node tree is 0).

description (e.g., see Dobkin *et al.* [17] or Patterson and Yao [42]), which can be viewed as an "inverse" problem to the one that we address here.

More specific to this paper, however, our research is directed at applying a technique from parallel algorithm design to a sequential problem in solid modeling. Our application of this paradigm is therefore somewhat reminiscent of a technique due to Megiddo [39], which is also an application of techniques from parallel algorithm design to improve the time bounds for sequential problems. Incidentally, variants of Megiddo's technique have proven very effective of late at improving the running times of several geometric optimization problems. Some examples include the ray shooting method of Agarwal and Matoušek [4], the "biggest stick" method of Agarwal, Sharir, and Toledo [5], the "largest polygon" algorithm of Toledo [57], the ham-sandwich cut algorithm of Cole [13], the distance-selection method of Agarwal *et al.* [3], and the closest-pair algorithms of Chazelle *et al.* [9].

One of the nice side effects of using a technique from parallel algorithm design for a sequential algorithm, is that it makes the parallelization of that method much more likely. Indeed, this is exactly what has occurred, in that subsequent to the initial announcement of this work, Goodrich, Ghouse, and Bright [31] have shown how our tree-contraction method can be applied to ray shooting of CSG models, and even to more complicated classification problems, such as boundary evaluation. Besides this, the most notable previous parallel effort has been the work of Ellis *et al.* [24] on a special-purpose machine, which they call the RayCasting Engine, for line classification of 3-dimensional CSG objects. Their machine has over 250,000 special-purpose processors dedicated to CSG classification. Incidentally, our method could also be implemented in hardware in a fashion similar to that used in the RayCasting Engine, which would lead to a pipelined parallel scheme for CSG ray shooting that would achieve a latency[2] of $O(\log n)$, whereas the RayCasting Engine may have a latency of $\Theta(n)$ if the input CSG tree is unbalanced.

The specific problem of classifying a line against a CSG model was first studied by Tilove [55, 56], who gave a method from which one can derive an $O(n^2)$-time method. This bound was achieved explicitly more recently by Getto [28], who also gave some heuristics that should help this classification algorithm run faster in practice. Since classification of lines and other geometric objects (such as the CSG model's boundary) is such a fundamental problem, a considerable amount of work (e.g., see [24, 28, 29, 37, 41, 47, 48, 49, 54]) has been directed at methods for improving these running times. None of these methods run faster than $O(n^2)$ in the worst case, however—they are all based upon heuristics that should work well in practice but do not improve the worst-case running time.

Some methods have running times that are *intersection sensitive*, meaning that their time bounds depend on I, the number of intersection points determined by the m primitives. Examples include the 2-D boundary evaluation methods of Ottmann *et al.* [41], which runs in $O((n+I)(m+\log n))$ time, and Tawfik [54], which runs in $O(n \log n + I)$ time. If I is much smaller than in the worst case (i.e., if $I << \binom{n}{2}$), then these methods will run in $o(n^2)$ time, but it is also easy to construct examples where they run in $\Omega(n^2)$ time, as well.

Some other methods are based upon a heuristic that exploits spatial locality, that is, the

[2]Recall that the latency of a pipeline is the time between the input of a set of values and the output value for this set.

geometric relationships that the primitives have with one another because they are embedded in a Euclidean space. The usual way that this locality is exploited is to partition the space to which the CSG solid belongs into simple cells, such as boxes, so as to reduce the number of computations that must be performed. The main idea is to restrict comparisons to primitives that are geometrically "near" the object being classified [28, 29, 47, 49, 50, 51]. Typically, this is implemented using the well-known "bounding box" technique, where one bounds a primitive or group of primitives by a box and first performs geometric comparisons with this box before attempting comparisons with any of the primitives it contains (for these comparisons may be unnecessary). This approach works well if most of the bounding boxes are disjoint, but will actually increase the running time if most of them overlap.

Yet another kind of locality that some methods have tried to exploit is structural locality, that is, the structural relationships that the primitives have with one another because they define the leaves of the CSG tree T. The usual way that this locality is exploited is to restructure T into a functionally-equivalent tree T' that is "simpler" than T. The tree T' is *functionally-equivalent* to T if there is a clear mapping of the inputs of T to inputs of T' and the result of applying the operations of T to a set of input values (e.g., geometric primitives) is the same as produced by applying the operations of T' to the corresponding input values. A common way that this approach is implemented is to "push" all the negations in T to the leaves using DeMorgan's Laws and then define T' to be the sum-of-products expansion of the resulting logical expression (e.g., see Goldfeather et al. [29]), i.e., T' is a *disjunctive normal form*. Of course, if T is sufficiently complicated, this could end up exploding the size of the CSG tree *exponentially*. In fact, if T contains exclusive-union nodes (which are used in VLSI masking [41]), then simply pushing all the negations to the leaves of T could produce an exponential explosion in size. So, in order to be practical, this approach must be combined with some kind of a "tree-pruning" heuristic based on partially evaluating T while restructuring it (say using bounding boxes) [29]. Rossignac and Voelcker [48] show how to further exploit the structural locality in T by using a notion they call the *active zone* of a primitive p, which, intuitively, is the portion of the CSG solid that is affected by p. For any primitive p, they show how to decompose T (after all negations have been pushed to the leaves) into an expression, Z, describing p's active zone, and an expression, $S_\emptyset$, that does not depend upon p. Then they distribute p over a (pruned) disjunctive normalization of Z, and use this and $S_\emptyset$ to solve CSG classification problems relative to subsets of p. In fact, they also show how this can be done implicitly, without actually changing T. If Z has a simpler structure than T, this approach may achieve improvements over a simple-minded sum-of-products expansion of T. Of course, if T contains a lot of exclusive-union nodes, or if Z is complicated and cannot be significantly pruned, then the size-explosion of this expansion could still be considerable. Our tree-contraction method does not have any of these negative side effects.

The reader interested in more information on constructive solid geometry is referred to [26, 34, 38, 46, 47, 48] for some excellent discussions of this representation and related issues.

In the next section we give our method for restructuring the CSG tree T and we show how we can use this restructured tree to speed-up ray shooting in CSG models in Section 3. We describe how to deal with degeneracies in Section 4, and we conclude in Section 5.

2 CSG Tree Contraction

In this section we show how to convert a possibly unbalanced CSG tree T into a functionally-equivalent balanced tree D. To motivate our general approach, let us consider the most trivial classification problem, namely, *point classification*.

2.1 Motivating our Approach: Point Classification

In this classification problem one is given a point p and a CSG tree T, and asked to determine if p is inside or outside of R, the region defined by T. As Requicha reviews [46], it is easy to classify a point p in this way in $O(n)$ time. One begins by determining for each primitive region P whether p is inside or outside of P, which can be done in $O(n)$ time. Then, for each leaf v of T, which is associated with a primitive P, one labels v with a 0 or 1, depending on whether or not p is inside or outside of P (with "0" meaning "outside of P" and "1" meaning "inside of P"). One then evaluates T as a boolean tree, after performing the following transformations of the operations in T: *union*→**or**, *intersection*→**and**, *subtraction*→**minus**, and *exclusive-union*→**xor**. By a simple inductive argument, one can show that an evaluation of T determines whether or not p is inside R (with "0" meaning "outside of R" and "1" meaning "inside of R"). Since T has m leaves, this evaluation can easily be implemented in $O(m)$ time by a simple bottom-up procedure.

This view of T as a boolean tree seems to be the motivation for several of the previous classification methods, and it is the motivation for our methods. In our applications, however, this simple approach is not enough to achieve improved running times, for we wish to classify many different points, not just one. We desire a balanced CSG tree.

2.2 Shrinking T in $O(\log m)$ Rounds

So, suppose we are given an m-leaf boolean expression tree T, such that each leaf of T stores a boolean value (0 or 1) and each internal node is labeled with an operation from the set {**or**, **and**, **minus**, **xor**}. Our method for converting T might at first seem strange, but by the end of this section we hope that the usefulness of this method will become apparent.

Our method is based on converting T into D in a series of *rounds*, where we begin with $T_0 = T$ and we construct each T_{i+1} in round i by removing nodes from T_i, and updating some associated auxiliary structures, where $i \in \{0, 1, 2, \ldots, l\}$ for some l. In any round i we are allowed to visit each node in T_i at most a constant number of times. Each node v in T_i stores a pointer to a small tree D_v, which produces a value for v. When we visit a node v in T_i we may leave v unchanged or we may remove v and combine D_v with the small tree associated with one of v's neighbors in T_i. We are not allowed to use any small tree that was not defined in a previous round. Our goal is to reduce T_0 to a single node r in as few rounds as possible, so as to define a single tree D_r, which will be functionally equivalent to T.

Our method is based on a modification of the tree contraction technique from parallel algorithm design [7, 35, 40, 45] for evaluating an arithmetic expression tree whose operations come from the set $\{+, \times\}$. In particular, we use a sequential implementation of the rake-

and-compress algorithm of Miller and Reif [40]. Let $T_0 = T$ (as above), so that each internal node v in T_0 is labeled with one of the following operations (we follow each operation by our notation for it): **or** $(c + d)$, **and** $(c * d)$, **minus** $(c - d)$, or **xor** $(c \oplus d)$, where c and d are the values stored in v's children. For each node v in T_0, we define D_v to be the null tree if v is an internal node, and, if v is a leaf, then we define D_v to be a single node storing the Boolean value associated with v.

Each round consists of two steps, a "rake" step and a "compress" step. In the *rake* step we remove each leaf v in the tree and combine its dwarf tree with its parent's dwarf tree (which may be null). We say that an internal node v is *improper* if v is not the root and it has only one child. A node is *proper* if it not a leaf nor an *improper* internal node. In the *compress* step we then remove each improper node v that has an improper child and is an odd distance from its nearest proper ancestor. We then set the parent pointer for v's child to point to v's parent. Both operations also involve some update and combination operations to be performed on the D_v trees. Let us, therefore, explain the implementations of these two operations.

2.2.1 The Rake Step

In the rake step we remove all the leaves in T_i. So, let v be a leaf. There are three cases to how we update the D_v's.

Case 1: v's sibling, w, is also a leaf (assume, without loss of generality, that w is a right child). In this case we may inductively assume that the dwarf trees D_v and D_w are available and they define the (single bit) values associated with v and w, respectively. If we let z denote v's parent, then we can define D_z by creating a new node r to be the root of D_z, giving r the roots of D_v and D_w as its children, and defining the operation at r to be the operation, **op**, that was stored at z. That is, if b_v is the bit returned by D_v and b_w is the bit returned by D_w, then the function associated with r is $E(\mathbf{op}, b_v, b_w) = b_v$ **op** b_w, which evaluates to a single bit. We call E the *evaluation* function. (See Figure 2a.)

Case 2: v's sibling w is not a leaf. In this case we still remove v, but, following the approach of Brent [7], we *symbolically* represent the effect v has on z's value, where z is v's parent. Specifically, we associate two bits, a and b, with z. These bits are defined so that if we let x represent the value that will be eventually sent to z from its remaining child w, then $ax + b\bar{x}$ represents the value that would eventually be at z. Incidentally, this formulation seems to be new, as the previous methods for tree-contraction require two operations that form a commutative semi-ring [7, 35, 40, 45], not the four Boolean operations that we consider. Figure 3 shows how to set the ab labels for z, depending on the bit b associated with v and the operation **op** at z. This defines our *rake* function $R(\mathbf{op}, b)$, which evaluates to an ab pair (i.e., 2 bits). Note that this can be implemented as a simple table look-up in a table that can be represented with only 20 bits (a look-up that could, in fact, be implemented in hardware). For example, if z is an **and** node, and v's value is 1, then R returns the pair $(a, b) = (1, 0)$. We therefore define D_z by creating a new node r to be D_z root, giving r the root of D_v as its only child (recall that the old D_z was a null tree). We label r with the function $R(\mathbf{op}, b)$, indicating that r's two-bit value is determined by **op** and the bit b returned from D_v. (See Figure 2b.)

Figure 2: **The tree contraction operations**. The operations for the simple Case 1 rake rake are illustrated in (a), the operations for the Case 2 rake are illustrated in (b), and the operations for the simple Case 3 rake are illustrated in (c). The operations for a compress are illustrated in (d).

	z's operation				
v's value	**or**	**and**	**minus**$_1$	**minus**$_2$	**xor**
0	$(1,0)$	$(0,0)$	$(0,0)$	$(1,0)$	$(1,0)$
1	$(1,1)$	$(1,0)$	$(0,1)$	$(0,0)$	$(0,1)$

Figure 3: **An interior rake operation**. The table defines the rake function R, giving the (a,b) values for z when raking one of z's children, v. We use **minus**$_1$ (resp., **minus**$_2$) to denote the case when z's operation is **minus** and v is z's left (resp., right) child.

	a_2b_2			
a_1b_1	00	01	10	11
00	00	00	00	00
01	11	10	01	00
10	00	01	10	11
11	11	11	11	11

Figure 4: **The Compress look-up table**. The (a, b) values for compressing a node labeled (a_1, b_1) with its child labeled (a_2, b_2). This table defines the compress function C. Note: this table could also be used to implement the substitution function S, for $S(ab, x) = C(ab, xx)$.

Case 3. v has no sibling. This case is a consequence of our removing v's sibling in a previous round. Thus, we may assume inductively that v's parent z has a non-null dwarf tree D_z whose root is labeled with a rake function. Fortunately, this is a simple case, for the value associated with z can be determined by substituting the single bit x returned by v (as leaves always return single bits) into the equation $ax + b\bar{x}$, where a and b are the two bits returned by the rake function associated with the root of D_z. This gives us the true value for z, and defines our *substitution* function, $S(ab, x) = ax + b\overline{x}$, which evaluates to a single bit. Thus, to define the new D_z we create a new node r to be D_z's root, giving r the old root of D_z and the root of D_v as its only children. We label r with the function $S(ab, x)$, where ab is the pair returned by r's left child and x is the bit returned by r's right child. (See Figure 2c.)

2.2.2 The Compress Step

Let T_i' be the tree resulting from the rake step (we do not maintain the old tree, T_i). Recall that in the compress step we remove each improper internal node v in T_i' that has an improper child and is an odd distance from its nearest proper ancestor, and set the parent pointer of v's child to point to v's parent. This will give us the tree T_{i+1}. First, of course, we must identify the improper nodes in T_i' that must be compressed, but this is easily done by a simple tree-traversal (e.g., see [14]). While we are performing this traversal it is an easy matter to note which improper internal nodes have an improper child and are an odd distance from their nearest proper ancestor.

When we compress an improper internal node v we may assume inductively that v has a dwarf tree D_v that produces an a_1b_1 pair corresponding to the equation $a_1x + b_1\bar{x}$ for v. Moreover, since v's child, w, is also improper, we may assume that w has a dwarf tree D_w that produces an a_2b_2 pair corresponding to the equation $a_2x + b_2\bar{x}$ for w. But the free variable, x, in v's equation denotes the value returned from w. Thus, in cutting out v, if we want to return the correct value to v's parent, then we must substitute w's equation for x in v's equation and put the result into a form suitable to become w's new equation. Specifically,

Figure 5: **A CSG tree (a) and its corresponding dwarf CSG tree (b).** The figure gives the dwarf tree for the tree of (a) with the leaves given in the same left-to-right order.

if v's equation is $a_1x + b_1\bar{x}$, and w's equation is $a_2x + b_2\bar{x}$, then w's new equation is

$$a_1(a_2x + b_2\bar{x}) + b_1\overline{(a_2x + b_2\bar{x})}. \tag{1}$$

By DeMorgan's Laws and the fact that boolean + and $\cdot$ form a commutative semi-ring, we can re-write (1) as

$$(a_1a_2 + b_1\bar{a}_2)x + (a_1b_2 + b_1\bar{b}_2)\bar{x} + b_1\bar{a}_2\bar{b}_2. \tag{2}$$

This is not exactly the form we would like, but we can simplify it. Specifically, if the term $b_1\bar{a}_2\bar{b}_2$ is 1, then the terms $(a_1a_2 + b_1\bar{a}_2)$ and $(a_1b_2 + b_1\bar{b}_2)$ are also 1. Thus, this additive term is redundant. So we can in fact re-write (2) simply as

$$(a_1a_2 + b_1\bar{a}_2)x + (a_1b_2 + b_1\bar{b}_2)\bar{x}. \tag{3}$$

This defines our *compress* function $C(a_1b_1, a_2b_2)$, which evaluates to an ab pair (i.e., 2 bits), such that $a := a_1a_2 + b_1\bar{a}_2$ and $b := a_1b_2 + b_1\bar{b}_2$. Note that the above derivation also implies that the ab representation is closed under the compress operation. We can compute the value of C by a simple look-up in the table of Figure 4 (which could be implemented in hardware). Moreover, this table can be completely represented with only 32 bits, which, on most computers, can fit in a single word! We define the new dwarf tree D_w for w, then, by creating a new node r as the root, giving r the root of D_v and the old root of D_w as its children, and labeling r with the function $C(a_1b_1, a_2b_2)$, where a_1b_1 is the bit pair returned by r's left child and a_2b_2 is the bit pair returned by r's right child. (See Figure 2d.)

We note in passing that our implementation of this compress step is simpler than that of a straightforward simulation of the Miller and Reif parallel implementation, since our procedure is strictly sequential. We illustrate the conversion of an unbalanced CSG tree to a balanced dwarf CSG tree in Figure 5.

2.2.3 The resulting Dwarf CSG Tree

When the rake-and-compress procedure completes, and we have reduced T to a single node, r, which was originally the root of T, we let D denote the tree D_r. This is the dwarf

expression tree. Clearly, D is a binary tree, such that, given any collection of boolean values stored at the leaves of T, if one inputs these values to the corresponding leaves of D, and one evaluates D in a bottom-up fashion using the table look-up procedures described above, then the resulting value will be the same as the value of T given these inputs. In this way, the tree D has the same power as T in describing the value of a boolean expression, but, as we show in the proof of the following theorem, D has height $O(\log m)$ and $O(m)$ size.

Theorem 2.1: *Given an m-leaf boolean expression tree T, whose operations come from the set* {**or**, **and**, **minus**, **xor**}, *one can convert T to a functionally-equivalent m-leaf tree D, such that D has $O(m)$ nodes and height $O(\log m)$.*

Proof: By a simple inductive argument based upon the construction above it is easy to show that D is functionally-equivalent to T. We have only to show that D is a compact representation of T's functionality.

First, note that the height of D is bounded by the number of rounds in the rake-and-compress computation, since we increment the height of any D_v by at most $O(1)$ in any round. And we may bound the number of rounds by deriving a bound on the number of nodes of T_i we remove in any round i. We claim that at least one fourth of the nodes are eliminated in any round. To prove this claim consider any four nodes in T_i with consecutive in-order numbers. If these four nodes do not form a connected subtree in T_i, then one of them is a leaf; hence, one will be removed. On the other hand, if these four nodes form a connected subtree in T_i, then they form a 4-node chain and it is either the case that one of them is a leaf (which will be removed in the rake step) or three of them have only one child (hence one of these three will be removed in the compress step). Therefore, a fourth of the nodes in T_i are removed in round i, as we claimed; hence, the total number of rounds is $O(\log m)$. This, of course, implies that D has height $O(\log m)$.

As for the size of D, note that, in the worst case, we could add a new node to D for every node of T_i that survives to be in T_{i+1}. This implies a worst-case bound on $s(m)$, the size of D, that can be characterized by the recurrence relation $s(m) \leq s(\frac{3}{4}m) + m$. This implies that $s(m)$ is $O(m)$ (and it could be exactly m in some cases). □

Thus, we have a worst-case optimal bound on the size and height of the tree D. Moreover, the constant factors "hiding behind the big-Ohs" are quite small. In fact, we can show the following.

Theorem 2.2: *Given a Boolean expression tree T, as above, the above construction produces a functionally-equivalent tree D such that $h(D) \leq h(T)$.*

Proof: Our proof is by induction on i, the iteration number in our rake-and-compress procedure. For each node $v \in T_{i+1}$, let $\delta_i(v)$ denote the height of D_v after iteration i, and let $\tau_i(v)$ denote the height of v in T_{i+1}. Initially, $\delta_0(v) = 0$ for all $v \in T$ and $\tau_0(v)$ denotes the height of v in $T = T_0$. Also, let $h_i(v)$ denote the maximum length in T of all v-to-u paths where u is a leaf in D_v after iteration i. Intuitively, $h_i(v)$ is the height v would have if its subtree were only the v-to-u paths in T where u is a leaf in D_v. Finally, to handle

the intermediate stage, let $\delta'_i(v)$ denote the height of D_v after the rake step of iteration i, let $\tau'_i(v)$ denote the height of v in T'_i, and let $h'_i(v)$ be analogously defined. Our induction hypothesis is that for $i \geq 0$,

$$\delta_i(v) + \tau_i(v) \leq h_i(v) + \tau_{i-1}(v) - 1,$$

for each $v \in T_{i+1}$.

Before we prove this hypothesis, let us observe why it establishes the theorem. Note that in the very last iteration, l, the tree T is reduced to a single node, the root r of T. Moreover, the last operation performed on T_l must be a rake; hence, r had height 1 after the previous iteration, $l-1$ (i.e., in T_l). Thus, assuming our induction hypothesis is correct, we have

$$h(D) = \delta_l(r) = \delta_l(r) + \tau_l(r) \leq h_l(r) + \tau_{l-1}(r) - 1 = h_l(r) = h(T).$$

Let us therefore prove our induction hypothesis.

The base case ($i = 0$). Let us consider the first rake step, case by case. As an intermediate step to proving our induction hypothesis, we show that after the rake step, for each $v \in T'_i$, we have

$$\delta'_i(v) + \tau'_i(v) \leq h'_i(v) + \tau_{i-1}(v) - 1, \tag{4}$$

r1. Suppose both of v's children u and w were raked in the rake step of iteration i (note that there are no nodes with only one child at this point). Then $\delta'_i(v) + \tau'_i(v) = 1 + 0 = h'_i(v) + \tau_{i-1}(v) - 1$, since $h'_i(v) = 1$ and $\tau_{i-1}(v) = 1$.

r2. Suppose exactly one of v's two children was raked in the rake step of iteration i. Let u denote the raked child. Then $\delta'_i(v) = 1$ and $h'_i(v) = 1$. Moreover, since the lowest internal node in the subtree rooted at v must have both its children (leaves) raked, $\tau'_i(v) \leq \tau_{i-1}(v) - 1$. Thus,

$$\delta'_i(v) + \tau'_i(v) \leq 1 + \tau_{i-1}(v) - 1 = h'_i(v) + \tau_{i-1}(v) - 1.$$

r3. Suppose no child of v was raked in iteration i. Then $\delta'_i(v) = 0$ and $h'_i(v) = 0$. As in the previous case, we have that the lowest internal node in the subtree rooted at v must have both its children (leaves) raked; hence, $\tau'_i(v) \leq \tau_{i-1}(v) - 1$. Thus, $\delta'_i(v) + \tau'_i(v) \leq h'_i(v) + \tau_{i-1}(v) - 1$.

Next, let us consider the compress step in iteration i.

c1. Suppose v's (only) child u is "spliced out" in the compress step of iteration i. Then $\delta_i(v) = \max\{\delta'_i(v), \delta'_i(u)\} + 1$ and $\tau_i(v) \leq \tau'_i(v) - 1$ (since we splice out u). There are therefore two subcases:

c1.1. Suppose $\delta_i(v) = \delta'_i(v) + 1$. Then $h_i(v) = h'_i(v)$. Combining this with Equation (4), we get that

$$\begin{aligned} \delta_i(v) + \tau_i(v) &\leq \delta'_i(v) + 1 + \tau'_i(v) - 1 \\ &\leq h'_i(v) + \tau_{i-1}(v) - 1 - \tau'_i(v) + \tau'_i(v) - 1 \\ &< h_i(v) + \tau_{i-1}(v) - 1, \end{aligned}$$

which establishes the claim for case c1.1.

c1.2. Suppose $\delta_i(v) = \delta'_i(u) + 1$. Then $h_i(v) = h'_i(u) + 1$ and $\tau_{i-1}(u) \leq \tau_{i-1}(v) - 1$. Moreover, since u is v's only child in T'_i, $\tau'_i(u) = \tau'_i(v) - 1$. Combining this with Equation (4), we get that

$$\begin{aligned}
\delta_i(v) + \tau_i(v) &\leq \delta'_i(u) + 1 + \tau'_i(v) - 1 \\
&\leq h'_i(u) + \tau_{i-1}(u) - 1 - \tau'_i(u) + 1 + \tau'_i(v) - 1 \\
&\leq h_i(v) + \tau_{i-1}(v) - 2 - \tau'_i(u) + 1 + \tau'_i(u) \\
&= h_i(v) + \tau_{i-1}(v) - 1,
\end{aligned}$$

which establishes the claim for case c1.2.

c2. Suppose no child of v is spliced out in the compress step of iteration i. Then $\delta_i(v) = \delta'_i(v)$ and $\tau_i(v) \leq \tau'_i(v)$; hence, the claim follows immediately from Equation (4).

This establishes the base case.

So let us now consider the induction step (when $i \geq 1$). Assume that our claim is true after iteration $i-1$; let us consider iteration i, beginning with the rake step. As in our base case, we will show that Equation (4) holds for i as an intermediate step.

r1. Suppose all of v's children were raked in the rake step of iteration i. Let us further restrict our attention to the case when v has two children, u and w, since the case when v has only one leaf child is similar (actually, it is easier). Then $\delta'_i(v) = \max\{\delta_{i-1}(u), \delta_{i-1}(w)\} + 1$ and $\tau'_i(v) = 0$ and $\tau_{i-1}(v) = 1$. Moreover, since $i \geq 1$, u and w had height 1 in T_{i-1}, i.e., $\tau_{i-2}(u) = \tau_{i-2}(w) = 1$. By induction, then,

$$\delta_{i-1}(u) = \delta_{i-1}(v) + \tau_{i-1} \leq h_{i-1}(u) + \tau_{i-2}(u) - 1 = h_{i-1}(u),$$

and a similar inequality holds for w. Since $h'_i(v) = \max\{h_{i-1}(u), h_{i-1}(w)\} + 1$, this implies that

$$\begin{aligned}
\delta'_i(v) + \tau'_i(v) &\leq \max\{\delta_{i-1}(u), \delta_{i-1}(w)\} + 1 + \tau_{i-1}(v) - 1 \\
&\leq \max\{h_{i-1}(u), h_{i-1}(w)\} + 1 + \tau_{i-1}(v) - 1 \\
&= h'_i(v) + \tau_{i-1}(v) - 1.
\end{aligned}$$

r2. Suppose exactly one of v's two children was raked in the rake step of iteration i. Let u denote the raked child. Then $\delta'_i(v) = \delta_{i-1}(u) + 1$ and $h'_i(v) = h_{i-1}(u) + 1$. Moreover, since the lowest internal node in the subtree rooted at v must have both its children (leaves) raked, $\tau'_i(v) \leq \tau_{i-1}(v) - 1$. Moreover, since $i \geq 1$, u had height 1 in T_{i-1}, i.e., $\tau_{i-2}(u) = 1$. By induction, then, $\delta_{i-1}(u) \leq h_{i-1}(u)$, as in case r1. This, in turn, implies that

$$\delta'_i(v) + \tau'_i(v) \leq \delta_{i-1}(u) + 1 + \tau_{i-1}(v) - 1 \leq h'_i(v) + \tau_{i-1}(v) - 1.$$

r3. Suppose no child of v was raked in iteration i. Then $\delta'_i(v) = 0$ and $h'_i(v) = 0$. As in the previous case, we have that the lowest internal node in the subtree rooted at v must have both its children (leaves) raked; hence, $\tau'_i(v) \leq \tau_{i-1}(v) - 1$. Thus, $\delta'_i(v) + \tau'_i(v) \leq h'_i(v) + \tau_{i-1}(v) - 1$.

Having established Equation (4) for the general rake step, we can derive our claim for the compress step using the same arguments used to establish the claim for this step in the base case, since they only depended upon Equation (4). Thus, $\delta_i(v) + \tau_i(v) \leq h_i(v) + \tau_{i-1}(v) - 1$. As mentioned above, this establishes the theorem. □

So, to sum up, it never hurts to construct the dwarf CSG tree, and it often pays considerably to do so. In the section that follows we show how to apply the dwarf CSG tree to solve two important classification problems.

3 Line Classification and Ray Shooting

The first application we give for the dwarf CSG tree is for line classification, that is, computing all the intersections of a line and a CSG object. We then address an important special case of this classification problem, where one is simply interested in computing the first intersection point of a ray with a CSG object, which is the *ray shooting* problem. Our method works for both 2- and 3-dimensional objects, and runs in $O(n \log n)$ time, improving the previous methods, due to Tilove [55, 56] and Getto [28], by almost a linear factor. We describe our method assuming a 2-dimensional representation, and then show how to extend it to 3-dimensions. Furthermore, we describe it first assuming the primitives are in general position, even though this will often not be the case in practice, and we then show how to allow for "degenerate" configurations in Section 4.

3.1 2-Dimensional Line Classification

Suppose we are given a line L and a CSG tree T, describing a 2-dimensional region R, and wish to classify L against R. Specifically, we wish to identify all those intervals along L where L intersects the interior of R. For the sake of simplicity, let us assume that no two primitives share a common vertex and that the intersection of any two primitive edges is either null or a single point that is not the vertex of any primitive (we show how to handle these cases in Section 4). As always, we let n denote the total number of primitive edges, and let m denote the number of polygonal primitives. Our method is based on the approach of Roth [49] of "walking" down L, and is as follows.

Step 1. We begin our algorithm by using the method of the previous section to construct a dwarf CSG tree D for T, where we interpret T as a boolean tree using the natural operation transformations mentioned in Section 2.1. This takes $O(m)$ time.

Step 2. In this step we compute the intersection points of L with the boundaries edges of each of the primitive regions. With each such point v on L, we store pointers to the leaves in D corresponding to primitives that we would either enter or exit in crossing v from left to right (assuming a small neighborhood around v). This step takes $O(n)$ time.

Step 3. In this step we sort the intersection points as they occur along L. This can easily be done in $O(n \log n)$ time given the information computed in Step 2. We initialize each leaf of D corresponding to a primitive object to its (default) 0 or 1 value based on whether or not a point p on L to the left of any of the intersection points is inside or outside

Figure 6: **The traversal for line classification.** When we enter the interior of a polygon P we update the path in D from the leaf, v, associated with P, to the root.

the primitive. If all the primitives are bounded, then all these leaves should be labeled 0, since p must necessarily be outside all of the primitives. We then evaluate D by the obvious bottom-up evaluation procedure, storing the value of each internal node at that node. Note that some values are ab pairs. This step takes $O(n \log n + m)$ time.

Step 4. In this last step of our classification procedure, we "walk" down L, labeling each edge as being either inside or outside the region. Each time in our traversal that we enter or exit a primitive region P we update the leaf node corresponding to P to reflect this, i.e., setting the leaf's value to 0 if we are leaving P or setting it to 1 if we are entering P. This is not enough, however, for we also must update all the nodes in D whose value changes as a result of this change. But since D has $O(\log m)$ depth, this can be accomplished in $O(\log m)$ time (i.e., we need only visit the $O(\log m)$ ancestors of P's associated leaf node). (See Figure 6.) For any intersection vertex v on L such that the edge on its left is labeled differently from the edge on its right, we label v as being a "boundary" point. (See Figure 6.) When we update D for the last (semi-infinite) edge on L, then we will have correctly labeled each segment on L (this follows by a simple induction argument).

Thus, we have the following theorem:

Theorem 3.1: *Suppose one is given a planar CSG object R, described by an m-leaf CSG tree T whose leaves are associated with primitives that are simple polygons (which could contain holes) and whose internal nodes are labeled with operations that come from the set {union, intersection, subtraction, exclusive-union}. Then one can determine all the intersections of a line L with R in $O(n \log n)$ time, where n is the total number of edges describing the polygonal primitives, and this running time is optimal in the algebraic computation tree model.*

Proof: We have already established the upper bound. The lower bound is established by a simple reduction from the *set disjointness* problem, which has $\Omega(n \log n)$-time lower bound in the algebraic computation tree model [6, 53]. We leave the details to the reader. □

3.2 Extending Our Method to 3-Dimensional Line Classification

If we take a closer look at our method it is easy to see how to extend this approach to higher dimensional line classification. In particular, we need only change our implementation of Step 2, the computation of the intersections of L with the primitive regions. If each region has an $O(1)$ storage description, then this step is still a simple computation that can easily be done in $O(m)$ time. Otherwise, for each face f we must compute the intersection of L with f. If we assume the primitive objects, and our ray are in general position, then we can assume that L and each f lie in different planes and their intersection is either empty or is a single point, which we can easily determine in $O(\log n)$ time, after an $O(n)$-time preprocessing step, using standard computational geometry triangulation and point-location methods (e.g., see [22, 44]). Thus, we can find all the intersections of L with the CSG primitives in $O(n + m \log n)$ time in the general case, and in $O(m) = O(n)$ time if all the objects have an $O(1)$ storage description. Since the rest of the algorithm is exactly as in the 2-dimensional case, this gives us the following theorem:

Theorem 3.2: *Suppose one is given a 3-dimensional CSG object R, described by an m-leaf CSG tree T whose leaves are associated with primitives that are simple polyhedra (which could contain holes) and whose internal nodes are labeled with operations that come from the set {union, intersection, subtraction, exclusive-union}. One can determine all the intersections of a line L with R in $O(n \log n)$ time, where n is the total number of edges describing the polyhedral primitives.*

3.3 Stream-Lining Our Method for Ray Shooting

For the case of ray-shooting a CSG solid R whose primitives all have an $O(1)$ storage description (which is usually the case), we can in fact do slightly better than using the method described above. For, in this application, we do not wish to completely classify a line L with R, but, instead, we wish to find the first intersection of a ray $\vec{r}$ with R. In this case, we still convert the CSG tree T into a dwarf CSG tree D, and compute all the intersections of $\vec{r}$ with the primitives associated with the leaves of T, as before. We then initialize the values for

the leaves and internal nodes of D using the starting point of $\vec{r}$. This all can easily be done in $O(m)$ time. We then traverse the ray $\vec{r}$, moving through the list of intersection points as they occur along $\vec{r}$ until we find an intersection point p on the boundary of R. The point p is the first point we traverse such that the value associated with the root of D changes. If we implement the traversal along $\vec{r}$ using a heap (as in the heapsort procedure [14]) to give us the intersection points as they occur along $\vec{r}$, then this takes $O(\omega \log m)$ time, where ω is the number of (spurious) intersections $\vec{r}$ has with primitives of the CSG tree before it hits the boundary of R. So the total time to trace $\vec{r}$, then, is $O(m + \omega \log m)$. Since it seems unlikely that a ray $\vec{r}$ would intersect more than $O(m/\log m)$ primitive faces before striking the boundary of R, this procedure should run in linear time in practice.

One could also imagine a situation where one would like to perform a collection of ray shooting queries on a particular CSG model. In such a case it may be more beneficial to build a ray shooting data structure [2, 4, 8, 10, 11, 15, 18, 20, 19, 30, 32, 33, 43, 52] for the boundary edges (and faces) of the primitives and apply the above scheme for each ray in turn. We leave the details to the interested reader.

In addition, there are a couple of simple heuristics that one can apply to our method, which should achieve additional improvements in practice. The first is to only traverse up D, for re-computing internal node values that change as a result of traversing an intersection point p, as long as the new values are different than the old ones. One can terminate the traversal as soon as one encounters a new ab value, for some node v, that is the same the old ab value at v—for none of the values stored in the ancestors of v can change.

The second heuristic one can apply is the well-known *bounding box* or *octree* heuristics (see [50, 51]) to avoid computing intersections between L and faces that are "far away" from L. This does not improve the worst-case performance of the method, but should improve the running time in practice. In fact, Facello [25] performed a series of benchmark tests of our method (optimized with these two heuristics) versus the ray-shooting procedure used in the CSG software package BRL-CAD developed at the Ballistic Research Laboratory of the U.S. Army Aberdeen Proving Grounds [21], which is an implementation of the method of Laidlaw, Trumbore, and Hughes [37] (and is also optimized with these two heuristics). Facello's results showed that, for "random" models of 50 or more primitives, our method out performed the method of BRL-CAD. In fact, even for objects described with only 500 primitives our method achieved a 50% speed-up over the BRL-CAD method. (See [25] for details.)

Thus, we have established efficient bounds for line classification and ray shooting in 2-d and 3-d CSG models. Our algorithm descriptions to this point have assumed the objects are in general position, however, which will often not be the case.

4 Dealing with "Degeneracies"

Many CSG modeling systems will commonly produce a CSG model made up of a collection of primitives that are not in general position. For example, it is quite common for different primitives to share vertices, edges, and/or faces. Powerful and general computational geometry techniques, such as the *simulation of simplicity* technique of Edelsbrunner and

Mücke [23] and the *symbolic perturbation* technique of Yap [59], have been developed for designing computational geometry algorithms assuming there are no degeneracies. Intuitively, these methods apply an "infinitesimal" perturbation to the input so that all the objects are put into general position. Thus, the algorithm designer applying one of these techniques need not worry about degeneracies at all.

Unfortunately, these techniques are not applicable in the context of Constructive Solid Geometry. The reason for this is that the CSG operations of intersection, union, subtraction, and exclusive-union are defined so that they will result in a *solid* object (or the empty set). Formally, these operations are defined so that the result of applying an operation to two objects is the closure of the result of applying that operation to the interiors of the two objects. Such operations are said to be *regularized* [26, 28, 34, 38, 46, 48]. Degeneracies cannot be resolved using an infinitesimal perturbation scheme, since such a perturbation could create spurious portions of the CSG object. For example, the regularized intersection of two objects with disjoint interiors and intersecting boundaries is empty, whereas a standard intersection operation applied to an infinitesimal perturbation of these two objects could be non-empty. Clearly, if the primitives are in general position, then the regularized operations are equivalent to the standard operations, however, so we did not concern ourselves with this issue in our algorithm descriptions above.

If the objects are not in general position, then we must modify our algorithms slightly to take this into consideration. Fortunately, it is fairly easy to make Steps 2 and 3 robust using infinitesimal perturbation [23, 59], since they deal with the computation of the ordered intersections along the line L. Indeed, in the 2-d case we still assume that L intersects no vertices of the arrangement [22] of the primitive boundary edges, and in the 3-d case we still assume that L intersects no vertices or edges of the arrangement of the primitive boundary faces. In Step 4, the "walk" down L, we must now allow for two intervals to share a common endpoint, and, as mentioned above, we shouldn't use an infinitesimal perturbation scheme to deal with such a case. Nevertheless, we can easily handle this case by performing, in any order, all the updates required by an interval endpoint (i.e., the updates determined by the primitives we are entering and leaving as we cross that endpoint) and *then* checking the value of the root. If the value at the root is the same on both sides of the endpoint, then we assume we have not crossed the boundary of the CSG model. By checking the value at the root only when we are guaranteed to be entirely inside or outside each primitive we can be assured that the value is correct.

5 Conclusion

We have shown how to apply the tree contraction paradigm from parallel algorithm design to improve the sequential time complexity of line classification and ray shooting problems for Constructive Solid Geometry (CSG) models. The tree contraction paradigm allowed us to improve the previous CSG line classification and ray shooting algorithms by near-linear factors by converting the CSG tree T to a functionally-equivalent balanced tree D that is roughly the same size as T. All of our methods can be implemented with simple-to-program procedures, and still allow one to apply any of the previous heuristics that exploit spatial

locality.

One natural question to ask is whether our methods can be applied to improve the complexity of the CSG boundary evaluation problem, where one wishes to construct a representation of the boundary of the object described by T. In joint work with Ghouse and Bright, we have shown how our approach can be extended to perform 2-d boundary evaluation in parallel [31] using work that is $O(n^2 \log n)$ in the worst case. As for the sequential complexity, the best known method is that of Tawfik [54], which runs in $O(n^2)$ time in the worst case. It seems very unlikely that one will be able to perform CSG boundary evaluation in $o(n^2)$ time in the worst case, however, FOR it is easy to show that this problem belongs to the class of so-called "n^2-hard" problems in computational geometry [27] (we leave the details of this reduction to the interested reader).

There are a number of other possible directions for future research, however. One of the most interesting could be the search for new heuristics for further speeding up CSG evaluation procedures that use the dwarf CSG tree. We have mentioned some possible heuristics in this paper, but there are probably many more, especially if one were to examine the algebraic structure of the dwarf CSG more closely (as Rossignac and Voelcker [48] do for the standard CSG tree).

Finally, our methods assume one is interested in performing classification tasks against a static CSG tree representation. Thus, a natural question is whether one can still perform classification efficiently in an environment where the user may dynamically update the CSG representation (say, by inserting and deleting primitives and operations). Cohen and Tamassia [12] give an elegant method for dynamically maintaining a CSG representation relative to a fixed point in space, but we know of no efficient general methods for dynamic CSG tree maintenance that have good worst-case behavior.

Acknowledgements

We would like to thank Richard Beigel, John K. Johnstone, S. Rao Kosaraju, Mark Overmars, and Jaroslaw Rossignac for helpful conversations relating to topics in this paper.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, "A simple parallel tree contraction algorithm," *J. Algorithms*, **10**, 287–302, 1989.

[2] P. K. Agarwal, "Ray shooting and other applications of spanning trees and low stabbing number," in *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, 315–325, 1989.

[3] P. K. Agarwal, B. Aronov, M. Sharir, and S. Suri, "Selecting distances in the plane," in *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, 321–331, 1990.

[4] P. K. Agarwal and J. Matoušek, "Ray shooting and parametric search," in *Proc. 24th Annu. ACM Sympos. Theory Comput.*, 517–526, 1992.

[5] P. K. Agarwal, M. Sharir, and S. Toledo, "Applications of parametric searching in geometric optimization," in *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, 72–82, 1992.

[6] M. Ben-Or, "Lower bounds for algebraic computation trees," in *Proc. 15th Annu. ACM Sympos. Theory Comput.*, 80–86, 1983.

[7] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM*, **21**(2), 201–206, 1974.

[8] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink, "Ray shooting in polygons using geodesic triangulations," in *Proc. 18th Internat. Colloq. Automata Lang. Program.*, Springer-Verlag, 661–673, 1991.

[9] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, "Diameter, width, closest line pair, and parametric searching," in *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, 120–129, 1992.

[10] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, "Lines in space: combinatorics, algorithms, and applications," in *Proc. 21st Annu. ACM Sympos. Theory Comput.*, 382–393, 1989.

[11] S. W. Cheng and R. Janardan, "Space-efficient ray shooting and intersection searching: algorithms, dynamization and applications," in *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, 7–16, 1991.

[12] R. F. Cohen and R. Tamassia, "Dynamic expression trees and their applications," in *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, 52–61, 1991.

[13] R. Cole, "Slowing down sorting networks to obtain faster sorting algorithms," *J. ACM*, **34**, 200–208, 1987.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, NY, 1990.

[15] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, "Efficient ray shooting and hidden surface removal," in *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, 21–30, 1991.

[16] D. Dobkin, "Computational geometry and computer graphics," *Proceedings of the IEEE*, **80**(9), 1400–1411, 1992.

[17] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink, "An efficient algorithm for finding the CSG representation of a simple polygon," *Proc. SIGGRAPH '88, Computer Graphics*, **22**(4), 31–40, 1988.

[18] D. P. Dobkin and D. G. Kirkpatrick, "Fast detection of polyhedral intersection," *Theoret. Comput. Sci.*, **27**, 241–253, 1983.

[19] D. P. Dobkin and D. G. Kirkpatrick, "A linear algorithm for determining the separation of convex polyhedra," *J. Algorithms*, **6**, 381–392, 1985.

[20] D. P. Dobkin and D. G. Kirkpatrick, "Determining the separation of preprocessed polyhedra – a unified approach," in *Proc. 17th Internat. Colloq. Automata Lang. Program.*, *Lecture Notes in Computer Science*, vol. 443, Springer-Verlag, 400–413, 1990.

[21] P. C. Dykstra and M. J. Muuss, "The BRL-CAD package: An overview," in *Proc. BRL-CAD Symposium '89*, Aberdeen Proving Ground, Maryland, 1–9, 1989.

[22] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[23] H. Edelsbrunner and E. P. Mücke, "Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms," *ACM Trans. Graph.*, **9**, 66–104, 1990.

[24] J. L. Ellis, G. Kedem, T. C. Lyerly, D. G. Thielman, R. J. Marisa, J. P. Menon, and H. B. Voelcker, "The ray casting engine and ray representations: a technical summary," *Internat. J. Comput. Geom. Appl.*, **1**(4), 347–380, 1991.

[25] M. A. Facello, "Improved ray tracing methods for CSG modeling systems," Report JHU-90/05, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1990.

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

[27] A. Gajentaan and M. H. Overmars, "n^2-hard problems in computational geometry," Report RUU-CS-93-15, Dept. Comput. Sci., Utrecht Univ., Utrecht, The Netherlands, 1993.

[28] P. Getto, "Fast ray tracing of unevaluated constructive solid geometry models," in *New Advances in Computer Graphics: Proc. of Computer Graphics International '89*, R. A. Earnshaw and B. Wyvel, editors, Springer-Verlag, 563–578, 1989.

[29] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, "Near real-time CSG rendering using tree normalization and geometric pruning," *IEEE Computer Graphics and Applications*, **9**(3), 20–28, 1989.

[30] M. Goodrich and R. Tamassia, "Dynamic ray shooting and shortest paths via balanced geodesic triangulations," in *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, 318–327, 1993.

[31] M. T. Goodrich, M. Ghouse, and J. Bright, "Generalized sweep methods for parallel computational geometry," in *Proc. 2nd ACM Sympos. Parallel Algorithms Architect.*, 280–289, 1990.

[32] L. Guibas, M. Overmars, and M. Sharir, "Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques," in *Proc. 1st Scand. Workshop Algorithm Theory*, *Lecture Notes in Computer Science*, vol. 318, Springer-Verlag, 64–73, 1988.

[33] L. Guibas, M. Overmars, and M. Sharir, "Ray shooting, implicit point location, and related queries in arrangements of segments," Report 433, Dept. Comput. Sci., New York Univ., New York, NY, March 1989.

[34] F. W. Jansen, "Depth-order point classification techniques for CSG display algorithms," *ACM Trans. on Graphics*, **10**(1), 40–70, 1991.

[35] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, editor, Elsevier/The MIT Press, Amsterdam, 869–941, 1990.

[36] S. R. Kosaraju and A. L. Delcher, "Optimal parallel evaluation of tree-structured computations by raking," in *Proc. AWOC 88*, *Lecture Notes in Computer Science*, vol. 319, Springer-Verlag, 101–110, 1988.

[37] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes, "Constructive solid geometry for polyhedral objects," *Comput. Graph.*, **20**(4), 161–170, 1986.

[38] M. Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, 1988.

[39] N. Megiddo, "Applying parallel computation algorithms in the design of serial algorithms," *J. ACM*, **30**, 852–865, 1983.

[40] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *Proc. 26th IEEE Symp. on Foundations of Computer Science*, Aberdeen Proving Ground, Maryland, 478–489, 1985.

[41] T. Ottmann, P. Widmayer, and D. Wood, "A fast algorithm for Boolean mask operations," *Comput. Vision Graph. Image Process.*, **30**, 249–268, 1985.

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

[43] M. Pellegrini, "Stabbing and ray shooting in 3-dimensional space," in *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, 177–186, 1990.

[44] F. P. Preparata and M. I. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, 1985.

[45] M. Reid-Miller, G. L. Miller, and F. Modugno, "List ranking and parallel tree contraction," in *Synthesis of Parallel Algorithms*, J. H. Reif, editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 115–194, 1993.

[46] A. A. G. Requicha, "Representations of rigid solids: theory, methods, and systems," *ACM Comput. Surv.*, **12**, 437–464, 1980.

[47] A. A. G. Requicha and H. B. Voelcker, "Boolean operations in solid modelling: boundary evaluation and merging algorithms," *Proc. of the IEEE*, **73**(1), 1985.

[48] J. R. Rossignac and H. B. Voelcker, "Active zones in constructive solid geometry for redundancy and interference detection," Report RC 11991 (#53914), Auto. Res. Sci., IBM T. J. Watson Res. Center, Yorktown Heights, NY, 1986.

[49] S. D. Roth, "Ray casting for modeling solids," *Computer Graphics and Image Processing*, **18**, 109–144, 1982.

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

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

[52] O. Schwarzkopf, "Ray shooting in convex polytopes," in *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, 286–295, 1992.

[53] J. M. Steele and A. C. Yao, "Lower bounds for algebraic decision trees," *J. Algorithms*, **3**, 1–8, 1982.

[54] M. S. Tawfik, "An efficient algorithm for CSG to B-Rep conversion," in *ACM SIGGRAPH Symp. on Solid Modeling Foundations and CAD/CAM Applications*, Austin, TX., 1991.

[55] R. B. Tilove, "Set membership classification: a unified approach to geometric intersection problems," *IEEE Trans. Comput.*, **C-29**, 874–883, 1980.

[56] R. B. Tilove, "A null-object detection algorithm for constructive solid geometry," *Commun. ACM*, **27**, 684–694, 1984.

[57] S. Toledo, "Extremal point containment problems," in *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, 176–185, 1991.

[58] F. F. Yao, "Computational geometry," in *Algorithms in Complexity*, R. A. Earnshaw and B. Wyvel, editors, Elsevier, Amsterdam, 345–490, 1990.

[59] C. K. Yap, "A geometric consistency theorem for a symbolic perturbation scheme," *J. Comput. Syst. Sci.*, **40**, 2–18, 1990.