Speed-ups in Constructive Solid Geometry

David Eppstein

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 92-87

August 25, 1992

Abstract

We convert constructive solid geometry input to explicit representations of polygons, polyhedra, or more generally d-dimensional polyhedra, in time $O(n^d)$, improving a previous $O(n^d \log n)$ bound. We then show that any Boolean formula can be preprocessed in time $O(n \log n / \log\log n)$ so that the value of the formula can be maintained, as variables are changed one by one, in time $O(\log n / \log\log n)$ per change; this speeds up certain output-sensitive algorithms for constructive solid geometry.

1 Introduction

Computational geometry typically deals with explicit representations of geometric input (polygons, polyhedra, etc) as cell complexes of facets. However another representation is possible: a shape may be described *implicitly*, in terms of the method of its construction. Such a representation is the *Constructive Solid Geometry (CSG) formula*, in which a shape is built up from primitive shapes such as halfspaces and spheres, by means of Boolean combinations such as set union, set intersection, and complementation.

If we are given such an input, in order to apply many geometric algorithms, we must convert it to an explicit representation. This is the *CSG evaluation* problem, studied by several authers [8, 10, 11]. Many other problems, such as testing whether a given object is non-empty [11], can also be expressed as CSG evaluation. The dual problem of converting an explicit polygon or polyhedron to a CSG representation has also been studied [4, 9].

In general, n surfaces in dimension d can form $O(n^d)$ points of intersection, and a CSG formula involving n variables can thus give rise to polygons of complexity at most $O(n^d)$. This is tight: it is easy to form a "checkerboard" of $\Omega(n^d)$ disconnected cubes. By connecting the cubes by thin "wires" one can achieve a simply-connected figure with complexity $\Omega(n^d)$. Thus, if we are concerned with the worst case complexity of CSG evaluation, we cannot hope to do better than $O(n^d)$. The best known result, a bound of $O(n^d \log n)$ by Goodrich [8], comes close to this, but is off by a logarithmic factor. Our first result is that we can avoid this factor, and achieve an optimal worst-case time of $O(n^d)$ for CSG evaluation.

In practice, CSG input will not have complexity near that of the the worst case. If an input has small complexity, the lower bound of $\Omega(n^d)$ no longer holds. It makes sense, then, to ask for an algorithm for which the time complexity depends on the complexity of the output. If the output involved k facets the time might be something like $O(n \log n + k)$. No such algorithm is known. However, Goodrich [8] proposed an intermediate measure of complexity. Given a CSG formula in the plane, let α be the number of vertices formed by pairwise intersections between geometric primitives. The primitives must be triangles or similar bounded objects, since n halfplanes always determine $\Theta(n^2)$ intersections. Then $k \leq \alpha = O(n^2)$, so α is a weak measure of output complexity. Goodrich proposed an algorithm with time complexity $O((n + \alpha) \log n)$ for CSG evaluation; this matched the best known worst case complexity while allowing a speed-up for inputs composed of infrequently-intersecting primitives. Similarly, in R^3 let β be the number

of vertices formed by intersecting three primitives; then Goodrich gave an algorithm for CSG evaluation in time $O((n^2 + \beta) \log n)$.

Goodrich's algorithms are based on a data structure, the *dwarf CSG tree*, which enables him to change the value of a single variable in the CSG formula and update the formula value in $O(\log n)$ time. Our second result is a data structure which performs the same operations in $O(\log n / \log \log n)$ time. As a consequence, we can improve Goodrich's intersection-sensitive results. We solve the planar CSG evaluation in time $O(n \log n + \alpha \log n / \log \log n)$, and the R^3 problem in time $O(n^2 \log n + \beta \log n / \log \log n)$. We can add a second form of intersection sensitivity to further improve the last result. Let γ be the number of line segments formed by the intersection of two primitives; then we solve the 3d CSG problem in time $O(n^2 + \gamma \log n + \beta \log n / \log \log n)$.

2 Tree partitions

Our algorithms use a technique of partitioning trees into smaller subtrees, introduced by Frederickson [7]. We will apply this technique to trees representing CSG formulae.

Definition 1 (Frederickson [7]). *A restricted partition of order z with respect to a binary tree T is a partition of the vertices of V such that:*

1. *Each set in the partition contains at most z vertices.*
2. *Each set in the partition induces a connected subtree of T.*
3. *For each set S in the partition, if S contains more than one vertex, then there are at most two tree edges having one endpoint in S.*
4. *No two sets can be combined and still satisfy the other conditions.*

Such a partition can easily be found in linear time by merging sets until we get stuck. Restricted partitions have the important property that the number of sets in the partition must be small.

Lemma 1 (Frederickson [7]). *There are at most $6n/z$ sets in any restricted partition of order z with respect to a binary tree with n vertices.*

Proof: If we contract each set of the partition down to a single vertex, we form a smaller rooted tree with outdegree at most two. We classify the vertices of this contracted tree (i.e. the sets of the partition) into four types:

1. Vertices of outdegree zero (leaves of the contracted tree).

2. Vertices of outdegree one, with a child also of outdegree one.

3. Other vertices of outdegree one.

4. Vertices of outdegree two (corresponding to singleton sets).

Denote the number of vertices of type one by m. Since this is a binary tree, there are $m-1$ vertices of type four. Each vertex of type three has a child of type one or four, so the number of vertices of type three is at most $2m-1$.

If a vertex of type one has a type four parent, it must correspond to a set in the partition containing z vertices, since otherwise it could be merged with its parent. Similarly, if it has a type three parent, it and its parent together must have cardinality greater than z. So for each vertex of type one, we can find a set of vertices in the original binary tree of cardinality at least z; all these sets are disjoint from each other. Thus $m \leq n/z$ and the number of sets in types one, three, and four is at most $4n/z$.

We now consider the sets of type two. These can be grouped into chains of such sets, together with sets of type three at the bases of the chains. No two adjacent sets can contain fewer than z tree vertices (since otherwise the two sets could be merged). So if t denotes the number of sets of type two, we can find at least $t/2$ sets containing at least z vertices each, and t is at most $2n/z$.

Adding the at most $2n/z$ sets of type two with the at most $4n/z$ sets of other types gives our result. □

The same argument proves that there can be at most $5n/z$ non-singleton sets in the partition.

3 Subtree values

Suppose we are given a CSG formula f, with n variables representing basic geometric objects such as halfspaces which are to be combined by the formula into some more complicated object. We wish to construct an explicit representation for this object. Our algorithms for this task first manipulate the CSG formula as a formula of Boolean algebra, ignoring the geometric content of its variables. We split operations taking more than two arguments into an appropriate number of binary operations. This gives us a formula

with the same number of variables, in which all operations are binary. The formula defines a binary tree with $2n - 1$ vertices.

Our algorithms will involve restricted partitions of this tree. To simplify matters, we assume that the tree root forms a singleton set in this partition. Recall that such a partition involves sets of three types:

1. Sets connected by a single edge to the rest of the tree.
2. Sets connected by two edges, one upward to the root of the tree and one downward to a subtree.
3. Singleton sets with one upward edge and two downward edges.

Given a partition of the CSG formula tree, and given an assignment of values to the variables in the formula, we will extend the assignment to values defined on the sets in the partition. The value of each set will depend only on the variables corresponding to tree leaves that are part of the set. Just as the value of the formula can be determined from the values of its variables, we wish the value of the formula to be determined from the values of each set in the partition. Thus if we contract each partition set to a single node, the resulting contracted tree can still be interpreted as a formula involving the values of contracted node.

In the original CSG formula, the variables have Boolean values; that is, they are either true or false (geometrically, a point is either inside or outside the basic object corresponding to the variable). We can similarly assign a value to each partition set of the first type (a subtree connected by a single edge to the rest of the tree): each such set corresponds to a subformula of the boolean formula, and the value of the set can be found as the value of the subformula.

The second type of partition set has two edges, one up to the root of the tree and one down to a subtree. The possible values of the set will instead be *functions* of a single Boolean variable. There are four such values: the constant false function, the constant true function, the identity function, and the complement function. The subtree rooted at the upward edge corresponds to a subformula s of the CSG formula, which contains as a sub-sub-formula t the subtree rooted at the downward edge. Given an assignment to the variables of the partition set, we can compute for each possible value of t the resulting value of s. The value of the set is this function mapping values of t to values of s.

The final type of partition set is a singleton tree node, corresponding to a binary Boolean operation without any variables. We do not define values for such sets.

As an example, let f be $(a \wedge_1 b) \vee_1 (c \wedge_2 (d \vee_2 e))$. We have assigned numbers to the Boolean operations to distinguish between two operations of the same type. T is a tree with five leaves corresponding to the five variables, and four internal nodes corresponding to the four Boolean operations. Let the sets of the partition be $S_1 = \{a, \wedge_1, b\}$, $S_2 = \{\vee_1\}$, $S_3 = \{c, \wedge_2\}$, $S_4 = \{d, \vee_2\}$, and $S_5 = \{e\}$. S_1 is of the first type, so given values of a and b we can compute the value of S_1 as the Boolean value $a \wedge b$. S_2 is of the third type, and has no value. S_3 is of the second type, since it has an upward edge connecting it to S_2 and a downward edge connecting it to S_4. The subformulae s and t described above are $c \wedge (d \vee e)$ and $d \vee e$ respectively. If c is true, the value of s will be $d \vee e = t$. If c is false, the value of s will be false no matter what value t takes. Thus the value of S_3 is either the identity function (if c is true) or the constant false function (if c is false). Similarly, the value of S_4 is either the constant true function or the identity function, depending on the value of d. The value of S_5 is just that of e.

Let us verify that we can compute the value of f from the values of the partition sets. Suppose a is false, b is false, c is true, d is false, and e is false. Then S_1 has value false, S_3 and S_4 have the identity function as their values and S_5 has value false. To determine the value of the subformula corresponding to subtree $S_4 \cup S_5$, we apply the value of S_4 (identity) as a function to the value of S_5 (false). This subformula is thus false. Similarly, the value of subformula $S_3 \cup S_4 \cup S_5$ is also false. The value of S_1 is $a \wedge b = false$. The value of the entire formula is then computed by performing the Boolean operation at $S_2 = \vee_1$, resulting in a value of false.

The same technique computes the values of all subtrees rooted at the upward edges of the partition sets. We can also compute values of sets in coarser partitions: e.g. the set $S_3 \cup S_4$, if it were part of a partition, would be of the second type. Its value is a function which can be found by composing the values of S_3 and S_4. In the example above, the composition of two identity functions is again the identity function.

Lemma 2. *Let S be a set in a restricted partition of a CSG formula tree T, and let P be a restricted partition of the subtree induced by the nodes in S. Then the value of S can be determined by the values of each set in P, in time proportional to the number of sets in P.* □

As a special case, the value of the entire formula can be determined from the values of the sets of a restricted partition of the whole tree.

4 Worst-case optimal CSG

We are now ready to describe our algorithm for computing an explicit representation of a CSG polytope, in worst-case optimal time. For simplicity of exposition we start with the planar case, but our methods extend without difficulty to any higher dimension.

We are given a CSG formula f, involving n variables (halfplanes), and we wish to convert this into an explicit description of the polygon it represents. We split operations taking more than two arguments into an appropriate number of binary operations. This gives us a formula with the same number of variables, in which all operations are binary. The formula defines a binary tree with $2n-1$ vertices. We find a restricted partition of order $22n/\log_2 n$, with respect to this tree, in time $O(n)$.

By Lemma 1, we know that there are $O(\log n)$ sets in the restricted partition, of which at most $5/11 \log_2 n$ are non-singletons. The singleton sets correspond to solitary binary operations in our original formula. The non-singleton sets of type one, that is, the sets connected by a single edge to the rest of the tree, form subformulae of our original formula. If we know the value of each such subformula, we can deduce the value of the formula as a whole, without knowing how each subformula value is derived.

Any remaining non-singleton set S is connected to the rest of the tree by two edges. One such edge connects S to a portion of the computation tree not containing the tree root; this edge carries the value of a subformula f' into the portion of our main formula corresponding to S. The other edge connects S to the rest of the tree, and carries the value of the subformula formed by S together with f'. Thus the effect of S is to combine the value of f' with the variables in S itself, producing another Boolean value. This effect can be represented as a unary Boolean function, of which there are four possibilities: S can pass the value of f' unchanged, it can invert that value, or it can be one of two constant functions, producing the value true or false no matter what value f' takes.

Since each non-singleton set can have one of at most four possible effects, and since there are at most $5/11 \log_2 n$ such sets, there are at most $4^{5/11 \log_2 n} = n^{10/11}$ ways these effects can be combined to produce the total formula value. Each such combination can be represented with $O(\log n)$ bits,

which (in the standard PRAM model of computation) can fit in a single computer word. For each possible combination, we compute the total formula value in $O(\log n)$ time, giving a total time for this stage of preprocessing of $O(n^{10/11} \log n) = o(n)$.

The halfplanes corresponding to the variables correspond to a planar line arrangement of complexity $O(n^2)$. This arrangement can be constructed in time $O(n^2)$. Within a cell of the arrangement, the formula value is constant; that is, each cell is either entirely contained in the CSG polygon or it is entirely exterior to the polygon. Our goal is to determine the formula value within each cell; then the polygon itself will be the union of those cells within which the formula is true.

Each set of the restricted partition also gives us a set of $O(n/\log n)$ halfplanes, one for each variable appearing in the set. We can again construct an arrangement, of complexity $O((n/\log n)^2)$, in time proportional to the complexity. Thus the time to compute each such smaller arrangement, for each of the $O(\log n)$ partition sets, is $O(n^2/\log n)$.

Recall that each partition set can affect the entire formula in one of four ways. Within each cell of the smaller arrangement corresponding to the partition set, this effect is constant. We can compute this effect for each cell, by traversing the arrangement and updating the values in the portion of the formula involved in the partition set, using the algorithm of Goodrich [8], in time $O(\log n)$ per cell. Thus the total time to compute all these effects for all the partition sets is $O(n^2)$.

By combining these effects, we will compute the formula values in each cell of the main arrangement of all halfplanes. We therefore need to be able to relate locations in the main arrangement to locations in the smaller arrangements. Each line in the main arrangement corresponds to a halfplane in one set of the partition, and hence to a line in a single smaller arrangement. Any line is partitioned segments by the main arrangment, and into larger segments by the smaller arrangement containing it. We construct a data structure consisting of a pointer from each segment in the main arrangement to the segment containing it in the smaller arrangement. This can be done in time proportional to the complexity of the main arrangement, which is $O(n^2)$. With this structure, we can determine in constant time, whenever we move from cell to cell in the main arrangement, the corresponding movement in the smaller arrangement.

Finally, we traverse the cells of the main arrangement. We maintain, in a single $O(\log n)$ bit word, the effects of each of the $O(\log n)$ sets in the partition. These effects change only when we cross a line corresponding

to a variable in a given set, in which case we can look up the new cell in the corresponding smaller arrangement, and thus update the effect for that partition set, in constant time. We then look up the total formula value in the table we computed of the $O(n^{10/11})$ possible values, again in constant time. The time per cell is $O(1)$, so the total time is $O(n^2)$.

This completes the proof of the following result.

Theorem 1. *If we are given a planar CSG formula with n variables, each of which corresponds to a halfplane, we can construct an explicit description of the polygon represented by the formula, in time $O(n^2)$.* □

This is optimal, since it is not difficult to find formulae for which the corresponding polygons have complexity $\Theta(n^2)$.

The only points at which we used planarity were (1) the complexity analysis and time bounds for constructing arrangements of halfspaces, and (2) the data structure for relating smaller arrangements to larger arrangements.

The first point is answered by noting that in any dimension d, arrangements have complexity $O(n^d)$, and can be constructed in that time [6].

For the second point, each cell in a traversal of an arrangement is reached by crossing a $(d-1)$-face. We maintain a data structure mapping each $(d-1)$-face on a hyperplane of the main arrangement to the $(d-1)$-face containing it in the appropriate smaller arrangement. The data structure can be constructed in $O(n^d)$ time, and answers queries in constant time.

Thus the same techniques extend to any higher dimension. In any dimension, we can convert a CSG formula to an explicit description of the corresponding polytope, in time $O(n^d)$. As in the planar case, this is tight.

5 Dynamic maintenance of formula values

Goodrich's [8] algorithm for testing CSG emptiness (and for constructing an explicit representation of a CSG polygon) traverses the space in which the polygon is defined, maintaining the value of the CSG formula at each point in space. The traversal will at certain times cross the boundary between regions in which a particular variable is true, and in which it is false; at each such boundary the algorithm updates the value of the formula to determine whether the points across the boundary are in or out of the CSG polygon.

The most simple method for updating Boolean formulae is to maintain the value of subformulae, and update in constant time each such value that

depends on the changing variable. If we represent the formula as a binary tree, with leaves representing variables and nodes representing Boolean operations, such an update can be seen as following a path in the tree, from leaf to root, updating the values at each node in the path. The time for this computation is proportional to the path length, so the worst case time is simply the depth of the tree. However in general a formula may have depth proportional to its size, so this method might not be faster than recomputing the formula from scratch.

Goodrich improves this method using a *dwarf tree* with depth $O(\log n)$, constructed from the CSG formula using techniques originally developed in the context of parallel algorithms. A dwarf tree can be thought of as a formula involving values that are themselves Boolean functions. Using the method of dwarf trees, the value of the Boolean function can be updated in $O(\log n)$ time per variable change. A similar result can be obtained using a data structure for dynamic trees [3]; this has the further advantage (unneeded in our application) that certain changes to the structure of the formula can also be performed in logarithmic time.

Our approach is similar to that of Goodrich. As in Goodrich's approach, our algorithm can be thought of as constructing formulae defined over a non-Boolean domain (the values of sets in a restricted partition). We restructure the original CSG formula into a new tree with smaller depth. Each node corresponds to a subtree of the original CSG formula, and its children form a restricted partition of that subtree. Our approach differs in that our tree is not binary. We allow the tree to have non-constant degree: each internal node may have $\log^c n$ children, for some constant c. This allows us to form a tree in which the maximum depth is $O(\log n / \log\log n)$, leading to a savings in formula update time.

The restricted partition of any subtree of our formula itself forms a binary tree, in which the nodes are the sets of the partition, and the edges are the edges of T that connect two partition sets. If there are not many sets in the partition, there can not be many topologically different partition trees. More specifically, let $x = \log^c n$; then there are $2^{O(x)}$ different trees involving that many nodes, and for each such tree there are $2^{O(x)}$ ways that values can be assigned to the nodes of the tree. For each such assignment of values, we can compute the value of the entire tree, in time $O(x)$. If $c < 1$, we can precompute a table of all such results in time $o(n)$.

Then, given a restricted partition of our CSG formula, of order n/x, we can represent the values of the $O(x)$ partition sets using a description of $O(1)$ bits each; the entire description fits into a single machine word. We

can then look up the value of the formula in constant time by using that description as an index into our precomputed table.

Similarly, each set in the partition, involving $m \leq n/x$ nodes, forms a subtree of the original tree. If we form a restricted partition of order m/x, we can compute the set's value from the values in its partition, by composing functions in a bottom-up manner similar to that of Lemma 2. This value can be looked up in constant time from a representation of the partition values, using the same table precomputed above.

Repeating this construction gives us a partition of the formula into partition sets, each of which is further partitioned recursively until each set has at most x nodes in it. At each level of the recursion, the number of nodes in the sets shrinks by a factor of x, so the number of levels is $O(\log_x n) = O(\log n / \log\log n)$. Each level can be constructed in linear time, so the total time for constructing this structure is $O(n \log n / \log\log n)$.

If a variable changes in the formula, we need to recompute the values of the sets containing that node at each level of the recursive decomposition. Each such update takes constant time, so the time per update is $O(\log n / \log\log n)$.

We have proved the following theorem:

Theorem 2. *Given a Boolean formula f with n variables, we can in time $O(n \log n / \log\log n)$ construct a data structure of size $O(n)$, with which we can update the value of the formula in time $O(\log n / \log\log n)$ per variable change.* □

6 Output-sensitive CSG evaluation

Now suppose we have a formula in which the variables (geometric primitives) consist of shapes that do not intersect very often. For instance in the plane, the shapes might be polygons that are often disjoint from each other. As in [8], let α be the number of vertices formed by the intersection of two polygon boundaries. Then the lower bound of $\Omega(n^2)$ may no longer apply, because the output complexity is limited by α. Goodrich [8] was able to exploit this observation: if there are n polygonal primitives with m total facets, he gave an $O(m \log m + \alpha \log n)$ time algorithm for CSG evaluation. We can improve this, using the data structure of Theorem 2.

Goodrich's algorithm is as follows. We first form the arrangement of the m line segments bounding the polygonal primitives. This can be done in

time $O(m \log m + \alpha)$, using the output-sensitive line segment arrangement algorithm of Chazelle and Edelsbrunner [2]. Within a single cell, the CSG formula takes on a constant value. Goodrich computes these values by traversing the cells of the arrangement, updating the formula value each time he crosses a cell boundary using his *dwarf CSG tree representation* in time $O(\log n)$. Thus the time for this stage of the computation is $O(\alpha \log n)$. If we replace the dwarf tree used in this stage with the data structure of Theorem 2, we get the following improvement:

Theorem 3. *If we are given a planar CSG formula with n variables, each of which corresponds to a polygon, such that there are m polygon facets and α intersections of facets, we can construct an explicit description of the polygon represented by the formula, in time $O(m \log m + \alpha \log n / \log \log n)$.* □

Similarly, in R^3 suppose the primitives are convex polyhedra with a total of m facets, and β vertices formed by the intersection of three facets. In general, $\beta = O(mn^2)$ but β may be much smaller. We can compute the arrangement of all facets, by constructing separately the portion of the arrangement that lies in each facet. Thus the first stage of the CSG evaluation algorithm can be performed in $O(m^2 \log m + \beta)$. Goodrich performs the second stage in time $O(\beta \log n)$, resulting in a total bound of $O(m^2 \log m + \beta \log n)$.

We improve this bound in two ways. First, of course, we can replace Goodrich's dwarf trees by our data structure. Second, we can use a further form of intersection dependence. Let γ be the number of line segments formed by intersecting two facets of the input polyhedra; $\gamma = O(mn)$. We enumerate these segments by intersecting each pair of polyhedra, using Chazelle's linear time algorithm [1], in total time $O(mn)$. Then we can compute the arrangement of segments within each facet, using Chazelle's output-sensitive line segment arrangement algorithm as we did in the planar case. The time for constructing the facet arrangement is thus $O(mn + \gamma \log m + \beta)$, which can significantly improve the previous $O(m^2 \log m + \beta)$ when there are many more polyhedra than facets, or when γ is small. Thus we have our final result:

Theorem 4. *If we are given a three dimensional CSG formula with n variables, each of which corresponds to a convex polyhedron, such that there are m polyhedron facets, γ line segments formed by intersecting two facets, and β vertices formed by intersecting three facets, we can construct an explicit description of the polygon represented by the formula, in time $O(mn + \gamma \log m + \beta \log n / \log \log n)$.* □

References

[1] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. 30th IEEE Symp. Foundations of Computer Science (1989) 38–48.

[2] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM* 39 (1992) 1–54.

[3] R.F. Cohen and R. Tamassia. Dynamic expression trees and their applications. 2nd ACM/SIAM Symp. Discrete Algorithms (1991) 52–61.

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

[5] D.P. Dobkin and D.G. Kirkpatrick. Fast detection of polyhedral intersection. 9th Int. Colloq. Automata, Languages, and Programming, Springer-Verlag LNCS 140 (1982) 154–165.

[6] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, EATCS Monog. Theor. Comput. Sci., Springer-Verlag, Berlin, 1987.

[7] G.N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. 32nd IEEE Symp. Foundations of Computer Science (1991) 632–641.

[8] M.T. Goodrich. Applying parallel processing techniques to classification problems in constrictive solid geometry. 1st ACM/SIAM Symp. Discrete Algorithms (1990) 118–128.

[9] M.S. Paterson and F.F. Yao. Binary partitions with applications to hidden-surface removal and solid modelling. 5th ACM Symp. Computational Geometry (1989) 23–32.

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

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