The Visual Computer

Approximating CSG trees of moving objects *

Hanan Samet
and Markku Tamminen †

Computer Science Department, University of Maryland, College Park, MD 20742, USA

A discussion of the relationship between two solid representation schemes is presented: CSG trees and recursive spatial subdivision exemplified by the bintree, a generalization of the quadtree and octree. Detailed algorithms are developed and analyzed for evaluating CSG trees by bintree conversion. These techniques are shown to enable the addition of the time dimension and motion to the approximate analysis of CSG trees. This facilitates the solution of problems such as static and dynamic interference detection. A technique for projecting across any dimension is also shown. For "well-behaved" CSG trees the execution time of the conversion algorithm is directly related to the spatial complexity of the object represented by the CSG tree (i.e., as the resolution increases, it is asymptotically proportional to the number of bintree nodes and does not depend on the size or form of the CSG tree representation). The set of well-behaved CSG trees include all trees that define multidimensional polyhedra in a manner that does not give rise to tangential intersections at CSG tree nodes.

Key words: CSG – Solid modeling – Bintrees – Hierarchical data structures – Time – Motion – Interference detection

* This is an expanded version of a paper titled "Bintrees, CSG Trees, and Time" which appeared in *Proceedings of the SIGGRAPH '85 Conference*, San Francisco (July 1985), pp. 121–130. This work was supported in part by the National Science Foundation under Grants DCR-83-02118 and IRI-88-02457 and in part by the Finnish Academy
† Deceased on August 5, 1989

1 Introduction

Constructive solid geometry (CSG) uses trees (CSG trees) of building block primitives (parallelepipeds, spheres, cylinders, etc.), combined by geometric transformations and Boolean set operations as a representation of 3D solid objects (Requicha 1980; Voelcker and Requicha 1977). Each primitive solid can be decomposed into a subtree whose leaves are halfspaces, each described by an equation of the form:

$$f(x, y, z) \geq 0.$$

Substitution of this subtree for every occurrence of that primitive in the original CSG tree gives rise to an expanded tree having only halfspaces as leaves. In the present article we shall assume this simple halfspace formulation of CSG (see also Okino et al. 1973; Woodwark and Quinlan 1982). Clearly, the CSG approach can be used to describe objects of any dimensionality and many interesting solid modelers have been based on it (Requicha and Voelcker 1982, 1983).

A bintree represents discrete solid objects of arbitrary dimensionality (e.g., binary images in two dimensions) by a binary tree defining a recursive subdivision of space and recording which parts are empty (WHITE), and which are solid (BLACK). The bintree is a dimension-independent variant of the more familiar quadtree and octree representations. In Requicha's (1980) taxonomy these methods are classified as cell decompositions. Samet (1990a, b) gives a comprehensive survey and bibliography of quadtree- and octree-related methods. Mentions of arbitrary dimensionality are found in the literature (Hunter 1978; Jackins and Tanimoto 1983; Meager 1980; Yau and Srihari 1983); however, few concrete applications have been demonstrated.

Figures 1 and 2 are examples of solids described by CSG trees. The icosahedron of Fig. 1 is defined by the intersection of 20 linear halfspaces, while the carburetor of Fig. 2 is described by 55 linear and 19 quadratic halfspaces (Jansen and van Wijk 1984). The latter contains 2 spheres and 1 ellipsoid, the rest being cylindrical halfspaces. The figures have been generated by applying the methods discussed in this paper to hidden surface viewing (Koistinen 1985a). Figure 1a contains a shaded image in which voxels with more than one active halfspace (i.e., "edges") are BLACK. This demonstrates the conversion process. Figure 1b contains the corresponding bintree at resolution 32 with hidden lines removed.

The Visual Computer (1990) 6:182–209
© Springer-Verlag 1990

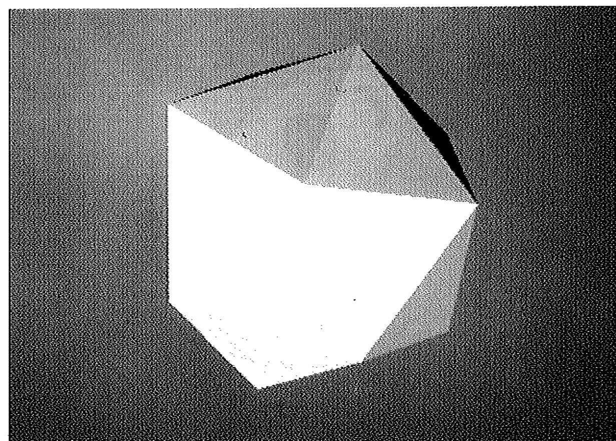
It has been known for some time that octree-like recursive subdivision can facilitate the evaluation of CSG trees; e.g., the analysis (Lee and Requicha 1982a, b; Wallis and Woodwark 1984) and display (Woodwark and Quinlan 1982) of solid objects modeled by them. See Cohen and Hickey 1979 for an earlier reference to a similar method for analyzing convex objects of arbitrary dimensionality. A hardware processor with such a capability is described by Meagher (1984). In this paper we continue this work by demonstrating the simplicity and efficiency of algorithms based on bintrees.

Time and motion are important elements of advanced solid modeling. In particular, given a moving object we may wish to determine whether it intersects a stationary object (*static interference detection*) or whether it intersects another moving object (*dynamic interference detection*). Even though it appears that the time dimension can be added to CSG trees in a conceptually simple fashion, rather little attention has been focussed on CSG trees in a dynamic situation (but see Cameron 1984). Perhaps this is due to the difficulty of evaluation in the now 4D space.

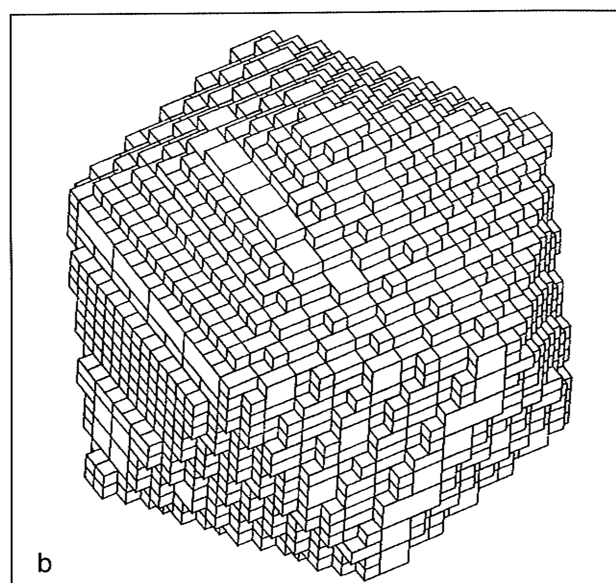
Static interference detection is discussed by Boyse (1979) but only boundary representations are considered. Tilove (1981, 1984) has provided a good analysis of the equivalent "NULL object detection" problem in the CSG setting. Our work seems to complement that of Tilove. We do not repeat his formal analysis of the "pruning" of CSG trees, but show in detail how a CSG tree can be efficiently pruned against an adaptable grid (i.e., bintree), even in the case of nonbounded halfspace primitives. We also show that for "well-behaved" CSG trees, the amount of work involved in pruning the CSG tree against all the cells of such an adaptable grid is asymptotically proportional to the number of cells and does not depend on the number of nodes in the CSG tree representation.

Although we have worked mainly with the boundary modeler GWB (Mantyla and Sulonen 1982) and found bintrees useful in this setting also, here we shall only discuss the CSG solid representation scheme. We find the simplicity of the algorithms relating the CSG and bintree representations striking when compared with algorithms for converting boundary representations to bintrees (Tamminen and Samet 1984).

In the rest of this paper we show how bintree conversion provides an efficient and dimension-independent tool for evaluating CSG trees. The time dimension is handled without extra conceptual overhead. Our emphasis is on CSG trees defined by linear halfspaces and on motion along a piecewise linear trajectory; however, the techniques are shown to extend to the general case. We present and analyze the evaluation (conversion) algorithm and compare 3D bintrees to octrees. We show that asymptotically, as resolution is increased, the amount of work involved in the conversion process is directly related to the spatial complexity of the object represented by the CSG tree. Thus, despite the added dimension, dynamic interference detection by bintree conversion is often efficient (because the object sought is the null object). Finally we



a



b

Fig. 1 a, b. Example of an icosahedron. **a** shaded image; **b** bintree at resolution 32

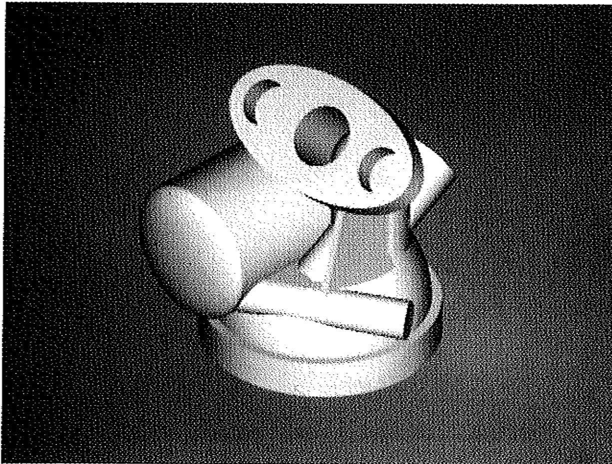


Fig. 2. Example of a carburetor. (From Jansen and van Wijk 1984)

present some experimental results obtained by using the discrete solid modeler described in (Tamminen et al. 1984).

2 Definitions

The quadtree of a 2D binary image is formed by a recursive quaternary partition of the image until homogeneous blocks (BLACK or WHITE) are reached. A binary image tree or *bintree* is formed analogously, except that at each level of recursion we only divide into two parts. The first partition is assumed to be in the x direction with partitions in the y and x directions alternating thereafter. Figure 3c shows an example of a bintree corresponding to a 2D binary image (Fig. 3a) consisting of two connected regions. We have assigned the internal nodes names A, B, C, D, and E and the leaves names 1, 2, 3, 4, 5, and 6. In the x partition we postulate the left subtree to correspond to the western (W) half of the image; in the y partition it corresponds to the southern (S) half.
In our algorithms we use a linear tree representation that is based on the preorder traversal of a bintree. The traversal yields a string over the alphabet "(", "B", "W", corresponding respectively to internal nodes, BLACK leaves, and WHITE leaves. This string is called a *DF-expression* (Kawaguchi and Endo 1980). For the image of Fig. 3 it becomes ((((BWWW(BW. See Kawaguchi and Endo (1980), Meagher (1980), Samet (1990b) and Samet and Tamminen (1985) for more details on such linear image tree representations.

Similar methods can be applied to 3D images (discrete solid models). An octree is defined as a recursive eight-way partition of a 3D image into octants until homogeneous blocks (SOLID corresponding to BLACK and EMPTY corresponding to WHITE) are reached (Hunter 1978; Jackins and Tanimoto 1980; Meagher 1980, 1982a, b; Srihari 1981). A 3D bintree is formed in an analogous manner except that at each level of recursion we only divide into two parts. Figure 4 is an example of a 3D image and its corresponding bintree. We have performed the partitions in the order x, y, and z. In the x partition we postulate the left subtree to correspond to the western (W) half of the image; in the y partition it corresponds to the southern (S) half. Let us speak similarly of the low (L) and high (H) halves of the z partition and let the left subtree correspond to the L half.
The preorder traversal of a 3D bintree yields a string (DF-expression) over the alphabet "(", "B", "W" corresponding, respectively, to internal nodes, SOLID leaves, and EMPTY leaves. For the image of Fig. 4 this representation becomes (B(B(BW. It should be clear that the same method (recursive subdivision along a cyclically varying coordinate axis) can be applied in a space of arbitrary dimensionality to obtain a d-dimensional bintree. Note that bintrees are size-independent; i.e., a given tree can define an object in a universe of any size. However, we usually portray each bintree as embedded in the d-dimensional unit cube. Let us say that the *resolution* of a bintree, say M, is the maximal number of units into which each side of the d-dimensional universe of a d-dimensional bintree can be divided. A cube of side length 1/M is called a *voxel*.
As already mentioned, we shall restrict our attention to CSG trees in the very simplest of settings, that of halfspaces defined by hyperplanes (linear halfspaces). We call such CSG trees *linear*. Note, however, that in direct analogy with the approximation of objects having curved surfaces by polyhedra (as so-called faceted modelers do) arbitrary CSG trees can be approximated by CSG trees whose leaves are linear halfspaces. This is done, for instance, in the system described by Woodwark and Quinlan (1982).
We first need a few definitions on homogeneous coordinates and linear geometric transformations. For more detail see Newman and Sproull (1979). In particular, we do not describe here how the transformation matrix and its inverse are formed

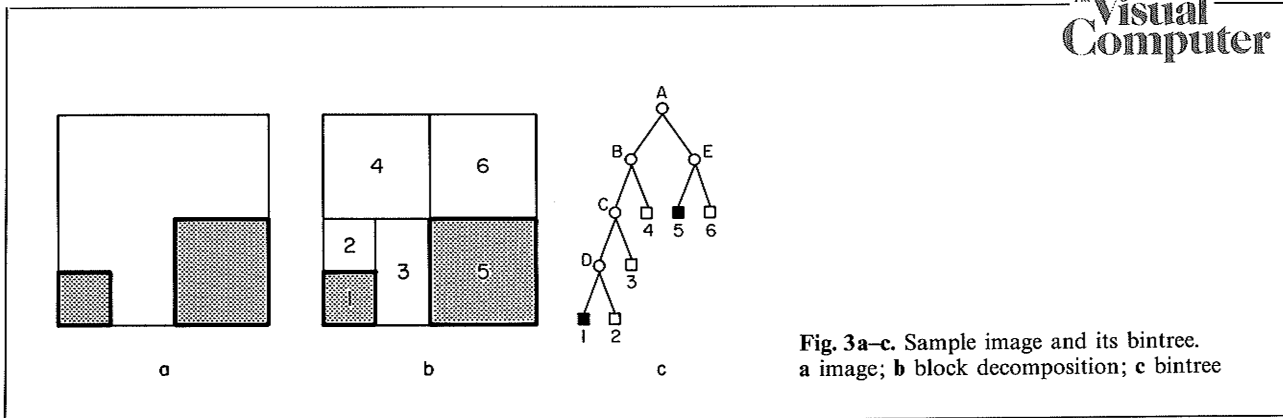


Fig. 3a–c. Sample image and its bintree. **a** image; **b** block decomposition; **c** bintree

to correspond to given components for scaling, translation and rotation.

A point x in a d-dimensional universe (d-space) is represented by a row vector of $d+1$ homogeneous coordinates with x_0 denoting the scale factor.

$$[x_0\, x_1 \ldots x_d]$$

We shall only consider the case with $x_0=1$. In the general case, the d-ordinary Euclidean coordinates are obtained by dividing $x_1, \ldots, x_d$ by x_0. Note that usually the scale factor is taken to be the last component of x. With our choice, the scale factor retains its original index when the time dimension is added.

We shall use xA to denote the multiplication of row vector x by matrix A and $a \cdot x$ to denote the dot product of vector a and vector x. Let $x'=xA$ denote a (linear) geometric transformation from coordinate system x to x' in d-space. It is defined by a $(d+1)\times(d+1)$ matrix A whose first column is $(1, 0, \ldots, 0)$.

A linear geometric transformation as we define it leaves the scale factor of homogeneous coordinates unchanged as 1. Two geometric transformations A and B can be composed by matrix multiplication. A transformation composed of scalings, rotations and translations can thus be represented by a single matrix A.

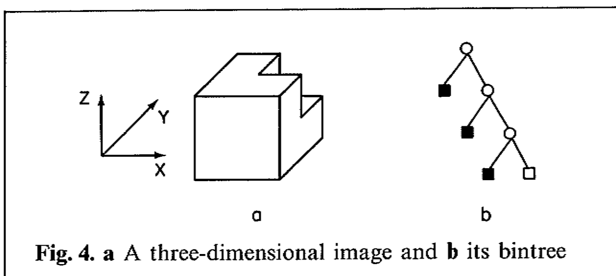


Fig. 4. a A three-dimensional image and **b** its bintree

A (linear) halfspace in d-space is defined by the following inequality on the $d+1$ homogeneous coordinates:

$$\sum_{i=0}^{d} a_i \cdot x_i \geq 0 \qquad (1)$$

The halfspace is represented by a column vector a. In vector notation (1) is written as $a \cdot x \geq 0$, with column vector a representing the halfspace. In the case of equality, (1) defines a hyperplane with a as its normal. For example, Fig. 5 shows the halfspace represented by $4x-2y-1\geq 0$. The point set satisfying this relation lies to the right of the line (partially shaded). Given a point x, the value of the left side of (1) at x is called the *value* of halfspace a at x. Element a_0 corresponds to the scale factor and is called the *constant* of a. For a description of geometric modeling with non-linear halfspaces see Okino et al. (1973) and Voelcker and Requicha (1977), and also Requicha's review (1980) and the systems and articles referenced by Requicha and Voelcker (1983). The geometric transformation A transforms halfspace a to halfspace a' as follows:

$$a' = A^{-1} a$$

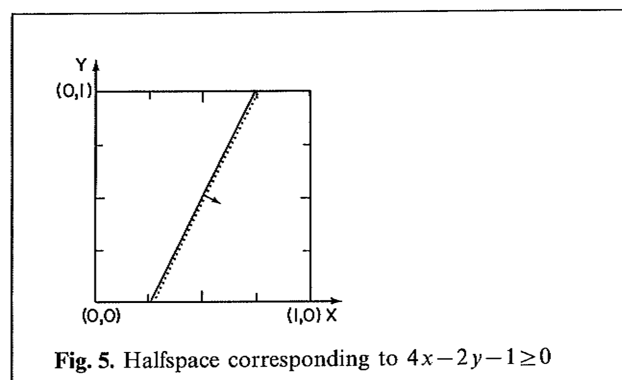


Fig. 5. Halfspace corresponding to $4x-2y-1\geq 0$

To understand this, we observe that given a point x and applying transformation A to it yields xA. The value of the transformed halfspace at xA yields $(A^{-1}a)\cdot xA = xA\cdot A^{-1}a = x\cdot a = a\cdot x$.

In this paper we define a data structure for CSG trees as follows. A CSG tree is a binary tree in which internal nodes[1] correspond to geometric transformations[2] and Boolean set operations while leaves correspond to halfspaces[3]. A node of a CSG tree is described by a record of type *csgnode* with six fields. The first two fields, LEFT and RIGHT, contain pointers to the node's left and right sons respectively. The TYP field indicates the node's type. There are five node types – UNION, INTERSECTION, BLACK, WHITE, and HALFSPACE. Types UNION and INTERSECTION correspond to the Boolean set operations. HALFSPACE, BLACK, and WHITE correspond to leaves. The field HSP contains an identifier for the halfspace. It is an index to a table, HS, of $d+1$ element coefficient vectors of the different halfspaces involved in the CSG tree. The remaining two fields, MIN and MAX, are used for auxiliary data in our algorithms. They record the minimum and maximum values, respectively, of a halfspace in a given bintree block. These fields are only used in conjunction with nodes of type HALFSPACE. Note that this representation is chosen for its simplicity in describing the algorithms. In an actual implementation, leaf and nonleaf nodes could be implemented as two different record types.

Our definition of a CSG tree allows for leaves that are completely BLACK or WHITE. This is required as an intermediate stage in the algorithm that prunes a CSG tree to a subuniverse corresponding to a bintree block[4]. In addition, in contrast to the conventional use of CSG, we only use the Boolean set operations UNION and INTERSECTION, as the effect of the third one, MINUS, can be achieved by substituting A UNION COMPLEMENT(B) everywhere for A MINUS B and applying De Morgan's laws to propagate the COMPLEMENT operations to the leaves of the tree. The regularized complement of a halfspace is obtained by changing the signs of all the coefficients (i.e., the direction of its normal). Regularization thus means that we consider the points sets $x\cdot a \geq 0$ and $x\cdot a > 0$ to be equivalent. Note that our universe is finite, as required by the bintree representation.

3 Converting CSG trees to bintrees

The conversion of a CSG tree to a bintree can proceed in two ways. One approach is to construct the bintree by converting the individual halfspaces in sequence and then performing the Boolean set operations on the results. This is a relatively inefficient process that does not take advantage of not processing areas that cannot possibly be in the final bintree. As an example of the use of this technique, consider the conversion of the triangle given in Fig. 6 whose CSG tree is given in Fig. 7. It is composed of the intersection of the three halfspaces $2x-1\geq 0$, $2y-1\geq 0$ and $-2x-2y+3\geq 0$ labeled A, B, and C respectively. Conversion starts with the unit square universe. Recall that we partition the x coordinate before the y coordinate. Processing the halfspaces A, B and C in sequence forces us to convert the segment of B between $x=0$ and $x=0.5$ whereas that region will be WHITE in the resulting bintree.

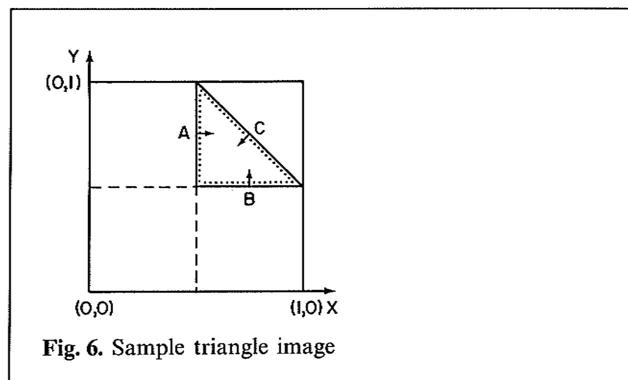


Fig. 6. Sample triangle image

[1] Unfortunately, we will need to use the terms *tree* and *node* in the context of both bintrees and CSG trees. However, in each case we will appropriately qualify their usage

[2] Our discussion assumes that the transformations have been propagated to the leaves. We also assume a bounded universe, for simplicity, in the form of the unit cube

[3] Actually, so-called regularized versions of the set operations must be used (Requicha 1980) in order to guarantee that the resulting objects correspond to our intuitive notion of solids. In such a case, the result of any set operation involving objects in d-space is always either null or has a nonzero d-dimensional measure. Regularized set operations form a Boolean algebra (Requicha 1980) so that, for instance, De Morgan's laws hold for them. However, we shall not repeat the qualification regularized in what follows

[4] Note that any CSG tree with one or more BLACK or WHITE leaves is equivalent to the whole universe, the empty set, or a CSG tree with only halfspaces as leaves

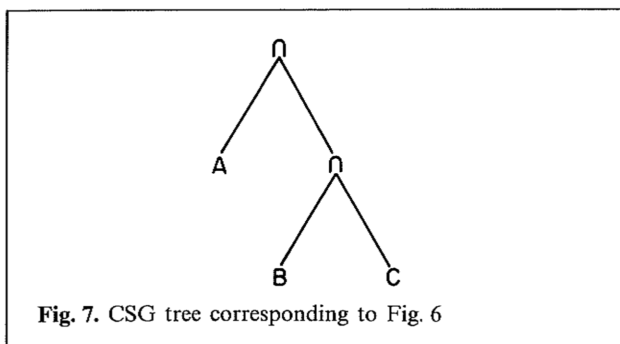


Fig. 7. CSG tree corresponding to Fig. 6

The second approach is one that traverses the universe in a depth-first manner and evaluates each successive subuniverse against the CSG tree. This enables pruning areas of no interest. Whenever the hyperplane of a halfspace, say H, passes through a subuniverse (i.e., a bintree node), say S, then we say that H is *active* in S (i.e., there exists a point in S such that $a \cdot x = 0$). An internal node of a CSG tree with only halfspaces as its leaves is said to be *active* in S if both of its sons are active in S. As an example of the use of these terms, let us consider the conversion of the triangle of Fig. 6. We start with the unit square universe. In this case halfspaces A, B, and C are all active and so is the CSG tree (in other words, it is not totally BLACK or WHITE). Thus, we first have to split the unit square into two halves (split along the x coordinate). Now, evaluating the CSG tree against the left half of the bintree, say L, we find that A is WHITE and thus not active. Therefore, L will have to be WHITE since A is combined with the rest of the CSG tree by an INTERSECTION node and the intersection of WHITE with anything is WHITE. In the rest of this section we focus on this approach. In our presentation we first demonstrate how to construct a bintree corresponding to a given halfspace. Next, we show how to convert a CSG tree to a bintree. In both of the constructions the resulting bintree is represented using a DF-expression.

3.1 Algorithm for a single halfspace

The construction of a bintree corresponding to a halfspace as given by (1) is achieved by traversing the universe in the DF-order and determining the range of (i.e., the interval of values obtained by) the left side of (1) in each subuniverse. In essence, what we are doing is intersecting the halfspace with a bintree block corresponding to a BLACK subuniverse. Each BLACK node in the bintree in which the halfspace is active is decomposed into two BLACK sons and the intersection process is recursively applied to them. The process stops when the halfspace is not active in a bintree node or if the bintree node corresponds to a voxel. In the version of the algorithm presented here, all voxels in which the halfspace is active are labeled according to the value of $a \cdot x$ at the centroid of the voxel, i.e., if $a \cdot x \geq 0$, then the voxel is labeled BLACK and WHITE otherwise.

Determining whether a halfspace is active in a bintree node is facilitated by keeping track of the minimum and maximum values of $a \cdot x$ for each bintree node. Whenever the maximum is ≤ 0 the bintree node is WHITE and whenever the minimum is ≥ 0 it is BLACK. Otherwise the halfspace is active and subdivision is required. Initially, for the unit cube, the minimum value of $a \cdot x$ is the constant of a plus the sum of all of the negative coefficients in a. The maximum value is the constant of a plus the sum of all the positive coefficients of a. For example, for Fig. 5, the initial minimum value is -3 and the initial maximum value is 3.

Whenever a bintree node is subdivided, either the maximum or minimum (never both at the same time) of $a \cdot x$ for each son node changes with respect to that of the father. Let the subdivision be performed on a hyperplane (e.g., a line in two dimensions) perpendicular to the axis corresponding to coordinate $i (1 \leq i \leq d)$ and let w_i be the width of the side along coordinate i of the block resulting from the subdivision. The amount of change is $\delta_i = a_i \cdot w_i$. For the left son, if $\delta_i > 0$, then δ_i is subtracted from the maximum; otherwise, δ_i is subtracted from the minimum. For the right son, if $\delta_i > 0$, then the minimum is incremented by δ_i; otherwise, δ_i is added to the maximum.

As an example, consider again the halfspace given by $4x - 2y - 1 \geq 0$ as shown in Fig. 5. Assume that the universe is the unit square. The maximum and minimum values 3 and -3 are attained respectively at (1,0) and at (0,1). Subdividing along the x axis yields two sons. The maximum value of $a \cdot x$ in the left son has decreased by 1/2 times the coefficient of the x coordinate (i.e., 2) to 1 and is attained at (0.5,0), while the minimum value remains the same. The minimum value of $a \cdot x$ in the right son has increased by 1/2 times the coefficient of the

x coordinate (i.e., 2) to -1 and is attained at (0.5,1), while the maximum value remains the same.

The conversion of a halfspace to a bintree is performed by procedures HALFSPACE_TO_BINTREE and HTRAVERSE. They are listed below using a variant of ALGOL. In these and all other procedures we shall use the following global constants: D is the dimensionality of the space and VOXEL_LEVEL is the level of the bintree corresponding to voxels. Procedure HALFSPACE_TO_BINTREE serves to initialize the traversal process by computing the minimum and maximum values of the halfspace in the whole universe (i.e., the d-dimensional unit cube). Procedure HTRAVERSE traverses the universe by recursively subdividing it corresponding to the depth-first traversal order of the resulting bintree. Subdivision stops upon encountering a bintree node at level VOXEL_LEVEL or when the block can be certified as WHITE or BLACK by comparing the minimum or maximum values of the halfspace to zero (note that some tolerance, say epsilon, should actually be used here instead of zero). At level VOXEL_LEVEL the value of a linear halfspace at the centroid of the corresponding geometric cell (i.e., bintree node) is obtained simply as the average of the minimum and maximum values of the halfspace at the node. The range of the left side of (1) is determined in each subuniverse by computing δ and as a result of the traversal each resulting bintree node either satisfies (1) completely or not at all (above voxel level), or contains points satisfying it (at voxel level). Procedure HTRAVERSE could avoid recalculating the δ's by precomputing them and storing them in an array indexed by level. In such a case, the traversal can be implemented with only one addition operation for each node whose subuniverse intersects the hyperplane defined by (1). Actually, in each internal node two addition operations must be performed to obtain two son nodes.

Recall that procedures HALFSPACE_TO_BINTREE and HTRAVERSE do not construct an explicit tree representation of the bintree. Instead, they output its nodes in an order corresponding to its depth-first traversal (i.e., a DF-expression representation). The sequence of nodes that is output is not minimal in the sense that collapsing may yet have to be performed (i.e., when two terminal brother nodes are BLACK). For example, Fig. 8a is the uncollapsed bintree with VOXEL_LEVEL $=5$ corresponding to the halfspace of Fig. 5. The nodes have been numbered in the order in which they were output. The result of the application of collapsing is shown in Fig. 8b where nodes 7 and 8 were merged into A and nodes 18 and 19 into B. Collapsing is only necessary when two brother nodes both become BLACK at voxel level. In an actual system in which these techniques are implemented, collapsing is performed on the DF expression by utilizing a buffer whose size is bounded by twice the maximal depth of the bintree (Tamminen et al. 1984). Notice that the bintree blocks at level VOXEL_LEVEL are rectangles instead of squares. In general, bintree blocks at level VOXEL_LEVEL are d-dimensional cubes only when VOXEL_LEVEL mod $d=0$. Otherwise, they correspond to d-dimensional rectangles the length of whose sides take on one of two possible values (e.g., 1/8 and 1/4 in Fig. 8).

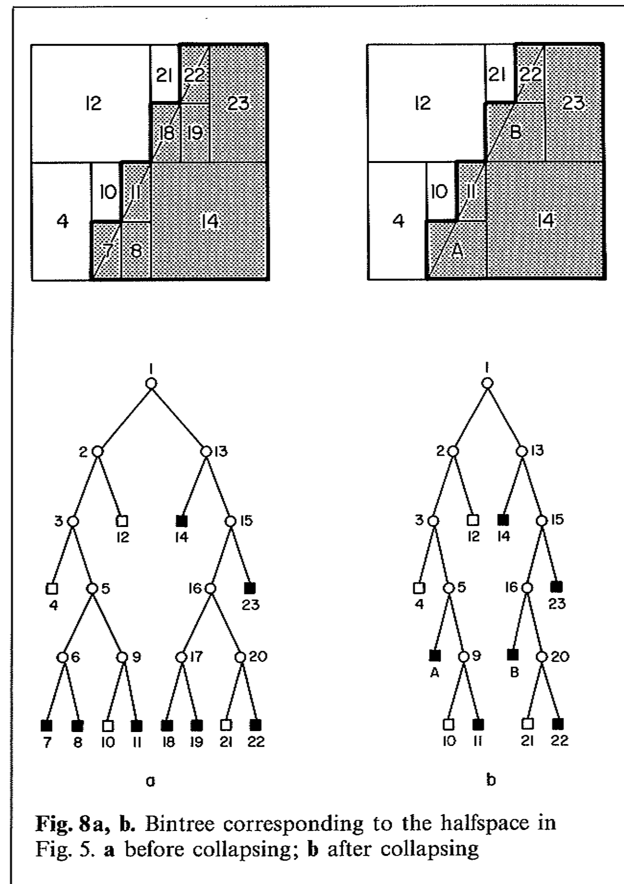


Fig. 8a, b. Bintree corresponding to the halfspace in Fig. 5. **a** before collapsing; **b** after collapsing

```
procedure HALFSPACE_TO_BINTREE(HS);
/* Convert the D-dimensional halfspace HS to a bintree.*/
begin
  global value real array HS[0:D];
  real MIN,MAX;
```

```
    integer I;
    /* Compute the minimum and maximum values of HS in
       the D-dimensional unit cube */
    MIN ← MAX ← HS[0];
    for I ← 1 step 1 until D do
      begin
        if HS[I] > 0.0 then MAX ← MAX + HS[I]
        else MIN ← MIN + HS[I];
      end;
    HTRAVERSE(0,1.0,MIN,MAX);
  end;
recursive procedure HTRAVERSE(LEV,W,MIN,MAX);
/* Convert the portion of the halfspace represented by HS that
   intersects the D-dimensional subuniverse of volume 2^-LEV
   whose smallest side has width W. MIN is the minimum value
   of the halfspace in the subuniverse and MAX is its maximum
   value. */
begin
  value integer LEV;
  value real W,MIN,MAX;
  integer I;
  real DELTA;
  if MAX ≤ 0.0 then output('WHITE')
  else if MIN ≥ 0.0 then output('BLACK')
  else if LEV = VOXEL_LEVEL then
    if((MAX + MIN)/2 ≥ 0.0) then output('BLACK')
    else output('WHITE')
  else
    begin /* The halfspace is active in the subuniverse (i.e., it
             intersects it) */
      I ← LEV mod D;
      if I = 0 then W ← W/2;
      DELTA ← HS[I + 1]*W;
      /* Note that DELTA only depends on the level */
      output('NON_LEAF');
      HTRAVERSE(LEV + 1,W,
                      if DELTA ≤ 0.0 then MIN-DELTA
                      else MIN,
                      if DELTA > 0.0 then MAX-DELTA
                      else MAX); /* Process the left son */
      HTRAVERSE(LEV + 1,W,
                      if DELTA > 0.0 then MIN + DELTA
                      else MIN,
                      if DELTA ≤ 0.0 then MAX + DELTA
                      else MAX); /* Process the right son */
    end;
end;
```

Our method of checking the halfspace inequality (1) at each bintree block involves just one addition and one multiplication operation by keeping track of minimum and maximum values of (1) in the parent block. Thus, there is no need to check the value of (1) separately at each of the 2^d vertices of each of the bintree blocks. In fact, the multiplication operation is not necessary if we precompute δ and store it in a table indexed by level.

Lee and Requicha (1982b) make use of a technique that checks the center of each block against two offset halfspaces. The offset halfspaces are determined so that if the center of a block of this size is contained within one of these halfspaces, then the whole block either satisfies the halfspace inequality, or fails it. This method requires two halfspace evaluations per block and is applicable to nonlinear halfspaces as well.

Meagher (1982b) discusses the conversion of halfspaces to octrees. His approach to halfspace evaluation is one which checks the value of (1) at two test vertices of a block. This is analogous to our use of the minimum and maximum of (1) in the parent block. Values at all vertices obtained from those of the father node by averaging (in the linear case) the value of (1) at two father vertices for each son vertex. For each son node, 2^d averaging operations are performed. This is clearly more efficient than recomputing (1) at each of the 2^d vertices. Note that this technique is also applicable to the nonlinear case.

The technique we use in the conversion of halfspaces to bintrees could also be applied to octrees in a straight-forward manner. We must compute the minimum and maximum values of (1) in each of the eight sons of the octree node, called P. These values are contained in 15 points corresponding to vertices of the sons of P. Figure 9 shows 10 of these points labeled in a manner that facilitates the subsequent discussion. The values of two of the vertices are the same as those of P (enclosed by a square in Fig. 9) and thus we only need to compute (1) at 13 points (8 of which are visible and enclosed by a circle in Fig. 9). Note that one of the 13 points corresponds to the vertex in the center of P's block and is shared by two of P's sons (the ones whose vertices are enclosed by a square in Fig. 9) in the sense that the value of (1) at it serves as the minimum value of one son and as the maximum value of the other son. Thus, we need 13 addition operations to compute the mini-

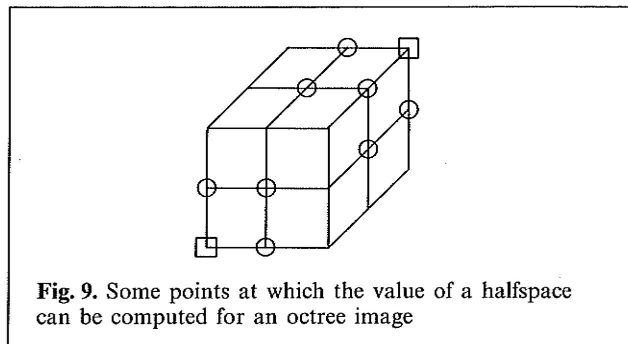


Fig. 9. Some points at which the value of a halfspace can be computed for an octree image

mum and maximum values at the sons of an octree node. Equivalently, we obtain 8 nodes with 13 addition operations. In contrast, with bintrees we obtain 2 nodes with 2 addition operations. Thus, if we were to split a bintree node three times to get the effect of an octree node, we would require $2+4+8=14$ addition operations to compute the minimum and maximum values of (1) at each of the eight sons. Of course, not all of the operations may be necessary because of possible merging. In fact, bintrees and octrees corresponding to a polyhedron have been observed to have an approximately equal amount of nodes (Tamminen et al. 1984), so that the bintree conversion turns out to be more economical beside being simpler to implement.

3.2 Algorithm for a CSG tree

A CSG tree is evaluated, i.e., converted, to a bintree by traversing the universe in depth-first order and evaluating each subuniverse against the CSG tree. Leaf nodes (halfspaces) are evaluated using the method described in Section 3.1 and their results are combined by *pruning* the CSG tree to the subuniverse. Pruning means that only that part of the CSG tree that is active within the subuniverse is retained (Tilove 1981, 1984; Woodwark and Quinlan 1984). Once pruning has reduced the CSG tree to a leaf node (i.e., a halfspace), the conversion procedure becomes identical to that described in the previous section for converting a halfspace to a bintree. An alternative way to conceptualize the conversion process is to note that it is equivalent to intersecting the CSG tree with a bintree corresponding to a BLACK universe. Each node in the bintree, say B, in which the CSG tree is active, is decomposed into two BLACK sons and they are in turn intersected with only that part of the CSG tree that is active in B. This process stops when the CSG tree is not active in a bintree node or if the bintree node corresponds to a voxel. All voxels in which a CSG tree is active are labeled by procedure CLASSIFY_VOXEL. At its simplest (as used in most experiments described in Sect. 6), it treats all such voxels as BLACK (or WHITE). At its most complex, CLASSIFY_VOXEL corresponds to Tilove's NULL object algorithm applied to the active CSG subtree at the voxel (Tilove 1984). A compromise that is often good and used in the algorithm below, is to have CLASSIFY_VOXEL evaluate the CSG tree at the center of the voxel to obtain a bintree that corresponds to the true object at a certain spatial resolution. Note that the value of each linear halfspace at the center of the voxel is the average of the MIN and MAX fields of the halfspace node. In such a case, CLASSIFY_VOXEL becomes a version of the pruning algorithm. Other implementors might prefer yet different choices for CLASSIFY_VOXEL, e.g., one applying Monte Carlo methods.

The conversion of a CSG tree to a bintree is performed by procedures CSG_TO_BINTREE, INIT_HALFSPACES, CSG_TRAVERSE, PRUNE, and HSPEVAL. They are listed below using a version of ALGOL. They make use of BLACK_CSG_NODE and WHITE_CSG_NODE, which are global pointers to BLACK and WHITE CSG tree nodes. Procedure CSG_TO_BINTREE serves to initialize the traversal process. First, it invokes procedure INIT_HALFSPACE to traverse the CSG tree to compute the minimum and maximum values of each halfspace in the whole universe (i.e., the unit cube). These values are stored in the MIN and MAX fields of the CSG tree node corresponding to each halfspace. Next, it calls on CSG_TRAVERSE to perform the actual conversion. Procedure CSG_TRAVERSE traverses the universe by recursively subdividing it corresponding to the depth-first traversal order of the resulting bintree. At each subdivision step, procedure PRUNE is called to attempt to reduce the size of the CSG tree that will be evaluated in the bintree block. PRUNE traverses the CSG tree in depth-first order and removes inactive CSG nodes with the aid of HSPEVAL that determines if a halfspace is active within a given bintree block. Assuming that T is CSG node, PRUNE applies the following four rules to the CSG tree.

1. BLACK UNION T = BLACK
2. WHITE UNION T = T
3. BLACK INTERSECTION T = T
4. WHITE INTERSECTION T = WHITE.

As an example, consider the triangle of Fig. 6 whose CSG tree is given in Fig. 7. Figure 10a is the corresponding bintree, with VOXEL_LEVEL = 6. The nodes have been numbered in the order in which they were output. Initially, the entire CSG tree (i.e., Fig. 7) is assumed to be active in the whole universe (i.e., the unit square). Node 1 is output as a NON-LEAF node and we process its left son next. First, we attempt to prune the

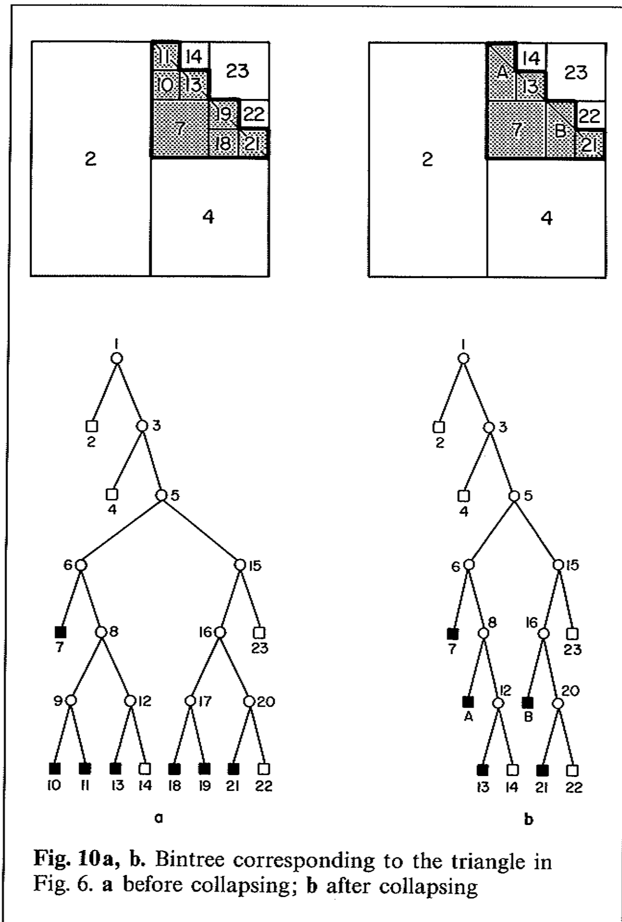


Fig. 10a, b. Bintree corresponding to the triangle in Fig. 6. **a** before collapsing; **b** after collapsing

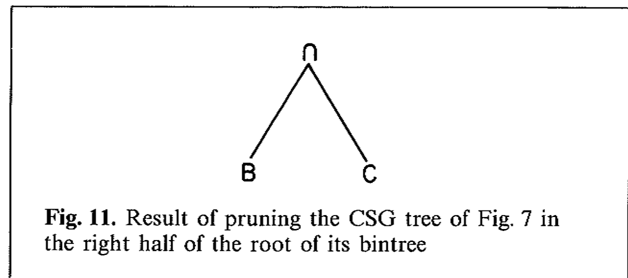


Fig. 11. Result of pruning the CSG tree of Fig. 7 in the right half of the root of its bintree

CSG tree with respect to the left half of the universe. Since halfspace A is inactive here (i.e., it is WHITE), we can apply pruning rule (4) and there is no need to further process the remainder of the CSG tree in Fig. 7. We output node 2 as WHITE and process the right son of node 1 next. Pruning the CSG tree results in halfspace A being inactive, but this time it is BLACK. Since both halfspaces B and C are active here, pruning rule (3) leaves us with the CSG tree given by Fig. 11. We now output node 3 as NON-LEAF and process its left son next. Pruning the CSG tree results in halfspace B being inactive (i.e., it is WHITE). Pruning rule (4) means that there is no need to further process the CSG tree of Fig. 11. We output node 4 as WHITE and process the right son of node 3 next. This time pruning the CSG tree results in halfspace B being inactive again, but now it is BLACK. Pruning rule (3) leaves us with just halfspace C. Node 5 is output as NON-LEAF and the conversion process is next applied to its two sons.

The remainder of the conversion is equivalent to that described in Sect. 3.1 for the conversion of a halfspace as the CSG tree has been reduced to one halfspace. The result is given in Fig. 10a. Once again we have a DF-expression representation of the bintree. In order to get the minimal DF-expression, we must perform collapsing (i.e., merge identically colored terminal nodes that are brothers). The result of the application of collapsing is shown in Fig. 10b where nodes 10 and 11 were merged into A and nodes 18 and 19 into B.

Closer examination of procedures CSG_TRAVERSE, PRUNE, and HSPEVAL reveals that there is much copying of CSG nodes. In particular, each time that CSG_TRAVERSE is invoked to evaluate a CSG tree in a bintree block, a copy is made of the pruned CSG tree. Nevertheless, the number of nodes that are copied is smaller than the number of evaluations of CSG nodes. Thus, the copying is not determinant to the complexity of the algorithm. Copying is necessary because we store the minimum and maximum values of the active halfspaces in the MIN and MAX fields of each leaf node of the CSG tree. Since these values change dynamically as the bintree is constructed and the CSG tree is evaluated, by keeping a copy we are able to take advantage of recursion to implicitly restore their previous values. Notice that whenever procedure CSG_TRAVERSE has completed processing a level of the bintree (i.e., both sons), the storage that was allocated for the CSG tree at that level and below is released. At worst, we must make VOXEL_LEVEL copies of the CSG tree before we can start to release and reclaim storage. This situation arises when all of the halfspaces are active in a single voxel. In practical cases the CSG tree gets pruned rather quickly at increasing levels of the bintree so that the total amount of

storage required is typically 2–3 times the size of the original CSG tree. See also our asymptotical analysis in Sect. 5.

```
procedure CSG_TO_BINTREE(P,N,HS);
/* Convert the D-dimensional CSG tree P to a bintree. HS con-
   tains N halfspaces. */
begin
  value pointer csgnode P;
  global value integer N;
  global value real array HS[1:N,0:D];
  INIT_HALFSPACES(P);
  CSG_TRAVERSE(P,0,1.0);
end;
recursive procedure INIT_HALFSPACES(P);
/* Compute the minimum and maximum values of each of the
   halfspaces in the CSG tree P of the D-dimensional unit uni-
   verse. */
begin
  value pointer csgnode P;
  integer I,J;
  if TYP(P)='HALFSPACE' then
    begin
      I←HSP(P);
      MIN(P)←MAX(P)←HS[I,0];
      for J←1 step 1 until D do
        begin
          if HS[I,J]>0.0 then MAX(P)←MAX(P)+HS[I,J]
          else MIN(P)←MIN(P)+HS[I,J];
        end;
    end
  else
    begin
      INIT_HALFSPACES(LEFT(P));
      INIT_HALFSPACES(RIGHT(P));
    end;
end;
recursive procedure CSG_TRAVERSE(P,LEV,W);
/* Convert the portion of the CSG tree P that overlaps the
   D-dimensional subuniverse of volume 2^-LEV whose smallest
   side has width W. The bintree is constructed by evaluating
   the CSG tree in the subuniverse. The evaluation process con-
   sists of pruning the nodes of the CSG tree that are outside
   of the subuniverse. A new copy of the relevant part of the
   CSG tree is created as each level is descended in the bintree.
   This storage is reclaimed once a subuniverse at a given level
   has been processed. */
begin
  value pointer csgnode P;
  value integer LEV;
  value real W;
  pointer csgnode FS; /* Pointer of stack of free nodes */
  if TYP(P)='BLACK' or TYP(P)='WHITE' then
    output (TYP(P))
  else if LEV=VOXEL_LEVEL then
    output(TYP(CLASSIFY_VOXEL(P)))
  else /* Subdivide and prune the CSG trees */
  begin
    FS←first_free(csgnode);
    /* Save pointer to free storage stack */
    output('NON-LEAF');
    if LEV mod D=0 then W←W/2;
    CSG_TRAVERSE(PRUNE(P,LEV+1,W,'LEFT'),
                      LEV+1,W);
    free(FS); /* Free storage allocated for CSG tree nodes start-
              ing at FS */
    CSG_TRAVERSE(PRUNE(P,LEV+1,W,'RIGHT'),
                      LEV+1,W);
    free(FS);
  end;
end;
recursive pointer csgnode procedure PRUNE(P,LEV,W,DIR);
/* Evaluate the portion of the CSG tree P that overlaps the
   D-dimensional subuniverse of volume 2^-LEV whose smallest
   side has width W. The subuniverse corresponds to the DIR
   (LEFT or RIGHT) subtree of its father bintree node. */
begin
  value pointer csgnode P;
  value integer LEV;
  value real W;
  value direction DIR;
  pointer csgnode T,Q; /* Auxiliary variables */
  pointer csgnode L,R; /* Auxiliary pointers to left and right
                          pruned subtrees */
  if TYP(P)='HALFSPACE' then
    return(HSPEVAL(P,LEV,W,DIR))
  else
    begin
      T←if TYP(P)='UNION' then BLACK_CSG_NODE
        else if TYP(P)='INTERSECTION' then
          WHITE_CSG_NODE
        else<error>; /* Enable the quick application of prun-
                        ing rules (1) and (4) */
      L←PRUNE(LEFT(P),LEV,W,DIR);
      if L=T then return(T)
      else
        begin
          R←PRUNE(RIGHT(P),LEV,W,DIR);
          if R=T then return(T)
          else if TYP(L)=OPPOSITE(TYP(T)) then
            return(R)
            /* OPPOSITE of BLACK is WHITE and vice
               versa */
          else if TYP(R)=OPPOSITE(TYP(T)) then
            return(L)
          else /* Evaluation has not eliminated one of P's
                  sons */
            begin
              Q←create(csgnode);
              TYP(Q)←TYP(P);
              LEFT(Q)←L;
              RIGHT(Q)←R;
              return(Q);
            end;
        end;
    end;
end;
recursive pointer csgnode procedure CLASSIFY_VOXEL(P);
/* Evaluate the portion of the CSG tree P that overlaps a voxel.
   Return BLACK_CSG_NODE if the centroid of the voxel
   satisfies P, and WHITE_CSG_NODE otherwise. Note that
   the maximum and minimum values of a halfspace at a given
   voxel have been computed at the time of evaluation of the
   voxel's father node. */
begin
  value pointer csgnode P;
  pointer csgnode T,Q; /* Auxiliary variables */
```

```
pointer csgnode L,R; /*Auxiliary pointers to left and right
  pruned subtrees*/
if TYP(P)='HALFSPACE' then /*Evaluate centroid.*/
  return(if((H_MAX(P)+H_MIN(P))/2≥0.0) then
           BLACK_CSG_NODE)
         else WHITE_CSG_NODE)
else
  begin
    T←if TYP(P)='UNION' then BLACK_CSG_NODE
        else if TYP(P)='INTERSECTION' then
          WHITE_CSG_NODE
        else<error>; /*Enable the quick application of
                 pruning rules (1) and (4)*/
    L←CLASSIFY_VOXEL(LEFT(P),LEV,W,DIR);
    if L=T then return(T)
    else
      begin
        R←CLASSIFY_VOXEL(RIGHT(P),LEV,W,
          DIR);
        if R=T then return(T)
        else if TYP(L)=OPPOSITE(TYP(T)) then
          return(R)
          /*OPPOSITE of BLACK is WHITE and vice
            versa*/
        else if TYP(R)=OPPOSITE(TYP(T)) then
          return(L)
        /*Note that evaluation always eliminates one of P's
          sons*/
        else<error>; /*Impossible case*/
      end;
  end;
end;
pointer csgnode procedure HSPEVAL(P,LEV,W,DIR);
/* Determine if the D-dimensional subuniverse of volume 2^-LEV
  and smallest side of width W intersects halfspace P or corre-
  sponds to a BLACK or WHITE region. The subuniverse
  is the DIR (LEFT or RIGHT) subtree of its father. If the
  halfspace intersects the subuniverse, then the subuniverse will
  have to be subdivided again and a new CSG tree node is
  allocated for the halfspace to record the new minimum and
  maximum values of the halfspace.*/
begin
  value pointer csgnode P;
  value integer LEV;
  value real W;
  value direction DIR;
  integer I,J;
  real DELTA;
  pointer csgnode Q;
  Q←create_and_copy(P);
  J←HSP(P);
  I←LEV mod D;
  DELTA←HS[J,I+1]*W;
  if DIR='LEFT' then
    begin
      if DELTA≤0.0 then MIN(Q)←MIN(Q)−DELTA
      else MAX(Q)←MAX(Q)−DELTA;
    end
  else
    begin
      if DELTA>0.0 then MIN(Q)←MIN(Q)+DELTA
      else MAX(Q)←MAX(Q)+DELTA;
    end;
  if MIN(Q)≥0.0 then return(BLACK_CSG_NODE)
  else if MAX(Q)≤0.0 then return(WHITE_CSG_NODE)
  else return(Q); /*The halfspace intersects the subuniverse*/
end;
```

4 Time and motion

Often a geometric representation, such as CSG, is not convenient for a desired computation. The solution that is frequently adopted is to transform the object into another representation, i.e., one in which the computation is simpler. In the previous section we saw how a CSG representation can be converted to a bintree. In this section we show how the time dimension can be added to a CSG representation so that motion can be analyzed using the algorithm of the previous section. Our approach is different from that of Meagher (1982b) and Weng and Ahuja (1987) that remain in the scope of pure octree modeling. In particular, Meagher computes the volume swept by an object's (modeled as an octree) motion along a curve that is specified by its chain code; whereas Weng and Ahuja (1987) are concerned with the translation and rotation of objects represented by octrees. We shall define motion by a time-dependent linear transformation $A(t)$ (i.e., trajectory). Our implemented algorithms are related to linear motion, i.e., the case that the trajectory is a piecewise linear function of t and can be decomposed into sequences of translations. However, we also show how similar methods can be applied to handle general motion. We conclude by showing how to implement a projection operation that eliminates the time dimension. This is not easy to perform directly in the CSG representation and thus we accomplish it by first approximating the (location, time) CSG tree by a (location, time) bintree, on which a projection can be performed. Projection is useful in computing swept areas.

4.1 Straight-line motion

Let T be a solid model described by a CSG tree and assume that it is defined in some model-specific coordinate system. We describe the motion of T in some common world coordinate system by a time-varying geometric transformation matrix $A(t)$. Each value of $A(t)$ is a matrix defining a rigid motion from the local coordinates of T to its position and orientation in world coordinates at time t.

Note that if our world coordinate system is the unit cube, then we may also have to include a scaling in $A(t)$. We call $A(t)$ the *trajectory* of T. Let $A(t)$ be piecewise linear, meaning that it can be broken down into a series of segments defined by time points $(t_0, t_1, \ldots)$ so that $A_{i+1} = A_i B_i$, where B_i is a transformation matrix corresponding to a translation describing the motion during that time segment. In the following we discuss the motion accomplished in one time segment in a more concrete setting.
Translating a halfspace, say given by (1), along a vector v gives rise to the translated halfspace

$$\sum_{i=0}^{d} a_i \cdot x_i - \sum_{i=0}^{d} a_i \cdot v_i \geq 0 \quad (2)$$

If point x satisfies (1), then the transformed (translated) point $x+v$ satisfies (2). For example, translating the halfspace $4x-2y-1 \geq 0$ given in Fig. 5 by the vector (0.5, 0.5) yields the halfspace $4x-2y-2 \geq 0$. In order to be dimensionally consistent with the d-dimensional unit cube, our discussion always assumes a unit time interval. Motion in a unit time interval, at a fixed speed defined by vector s with $s_0 = 0$, is described by a vector v such that for all i, $v_i = s_i \cdot t$. Thus, using v to translate halfspace (1) we find that at each instant, say t, it corresponds to the halfspace given by (3), below. Letting t vary, we obtain a linear halfspace with an additional variable t.

$$\sum_{i=0}^{d} a_i \cdot x_i - \left(\sum_{i=0}^{d} a_i \cdot s_i\right) \cdot t \geq 0 \quad (3)$$

When we have a CSG tree in motion, transformation (3) can be applied to each halfspace separately and the tree of Boolean set operations applied to the resulting $(d+1)$-dimensional halfspaces to define a set of points in (location, time) space satisfying the CSG tree.
For dynamic interference detection, we must determine whether the intersection of two (location, time) objects is empty, while for static interference detection, we must check whether two stationary objects intersect or whether a moving object intersects a stationary one. The intersection of two different (location, time) CSG trees, (each derived from a separate motion but with a common "time axis") is obtained by attaching them as sons to a newly created CSG node of type INTERSECTION. The actual evaluation of the intersection can be performed by applying the bintree conversion algorithm of Sect. 3.2 in the $(d+1)$-dimensional space with time included. For static interference detection there is no need to add time as an extra dimension if we can otherwise solve it for the swept area of the moving object. Note also that in the case of interference detection there is often little motivation for storing the entire resulting tree. Instead, a variable can be included in the tree taversal algorithm to indicate the minimal t value of a BLACK node encountered so far in the traversal. Any subtree whose minimum value of t is greater than this value need not be inspected.
Usually primary interest is not in motion along a straight vector but in more complicated trajectories. Assume that such trajectories can be approximated by a sequence of segments, each with motion corresponding to a linear translation at a fixed speed. For example, suppose that we wish to determine whether two motions, defined by piecewise linear trajectories A_{1i} and A_{2i} of objects T_1 and T_2, respectively, intersect in the unit time interval. Let the time intervals defining the n_1 and n_2 linear pieces of the trajectories be $(t_{10}, t_{11}, \ldots)$, respectively. Now, during the time interval $(0, \min(t_{10}, t_{20}))$ both motions are straight-line and their intersection can be determined as discussed above, i.e., by evaluating the CSG tree resulting from the addition of a node of type INTERSECTION with T_1 and T_2 as sons. This same procedure is applied to the remaining intervals (a maximum of $n_1 + n_2 - 1$ intervals), each preceded by an application of the appropriate transformations A_{ij} to the halfspaces of T_1 and T_2.

4.2 General motion

In the general case, a CSG tree can contain nonlinear halfspaces or the motion itself cannot be described as a series of translations. Instead, A_{i+1} is derived from A_i by multiplying it by, say, a rotation matrix with the rotation angle depending on t. In this case the trajectory is no longer a (piecewise) linear matrix function of t. Nevertheless, we can still use methods similar to the ones described in Sect. 4.1. In particular, each bintree node corresponds to a $(d+1)$-dimensional *interval* such that

$$\begin{gathered} x_0 \leq x < x_1 \\ y_0 \leq y < y_1 \\ \vdots \\ t_0 \leq t < t_1 \end{gathered}$$

Interval arithmetic is a method of evaluating functions $f(x, y, \ldots)$ in cases where the arguments are not exact values but intervals corresponding to the range of the true value. The result of the interval function corresponding to function f is also an interval, i.e., a range of values covering any values that f can obtain given as arguments any values in the argument intervals. It should be clear that interval arithmetic is appropriate for the CSG tree to bintree conversion process since for an arbitrary function the value of the corresponding interval function covers the function's possible values in the bintree node. If zero does not belong to this range, then the bintree block need not be subdivided.

For example, let us apply the above to determine the (location, time) bintree of a linear halfspace a subjected to general motion defined by the matrix function $A(t)$. Denoting intervals by capital letters, the interval function that must be evaluated at each node of the bintree is $F(X, T)=(A^{-1}(T)\,a)\cdot X$ where X is an interval in d-dimensional space and T is a time interval. Remember that at each time instant t, $A(t)$ is a linear transformation and the image of a halfspace, say a, is obtained by multiplying a by the inverse of $A(t)$. When $A(t)$ contains a rotation, the interval function will be a linear composition of sine and cosine functions with respect to T. For instance, in the 2D case of motion around the origin with angular speed α, the function $A^{-1}(T)$ would be given by the matrix

$$\begin{bmatrix} \cos(\alpha\cdot T) & \sin(\alpha\cdot T) & 0 \\ -\sin(\alpha\cdot T) & \cos(\alpha\cdot T) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In any sub-interval of T, where each component of the composite function is monotonic, interval arithmetic can be applied in such a way that tight bounds are provided for the resulting interval.

Interval arithmetic is easy to incorporate in our CSG tree to bintree conversion since the main change is only to recast procedure HSPEVAL in terms of interval evaluations. Procedure INTERVAL_HSPEVAL, given below, achieves this and a call to it can be substituted for the call to procedure HSPEVAL in procedure PRUNE of Sect. 3. Notice the use of INTERVAL_EVALUATE to determine the range of the function corresponding to the nonlinear halfspace. Its value is a pointer to a record of type *interval* with two fields MIN and MAX corresponding to an interval covering the function values in the node. Nevertheless, if procedure HSPEVAL is replaced by INTERVAL_HSPEVAL, CLASSIFY_VOXEL still has to correctly classify each voxel. CLASSIFY_VOXEL would ordinarily be implemented by computing the value of the active halfspaces at the centroid of the voxel by ordinary arithmetic.

```
pointer csgnode procedure INTERVAL_HSPEVAL(P,LEV,W,
                                                DIR);
/* Use interval arithmetic to determine if the D-dimensional
   subuniverse of volume 2^-LEV intersects nonlinear halfspace
   P or is BLACK or WHITE. W is its smallest side. The subuni-
   verse is the DIR subtree of its father.*/
begin
  value pointer csgnode P; /*A leaf of the CSG tree*/
  value integer LEV;
  value real W;
  value direction DIR;
  interval I;
  I ← INTERVAL_EVALUATE(P,LEV,W,DIR);
  return(if MAX(I) ≤ 0.0 WHITE_CSG_NODE
         else if MIN(I) ≥ 0.0 then BLACK_CSG_NODE
         else 'HALFSPACE');
end;
```

Interval arithmetic has been applied to the somewhat similar task of evaluating curved surfaces by recursive subdivision (Alander 1984; Mudur and Koparkar 1984). Nevertheless, this technique should be used with caution. In particular, interval arithmetic does not necessarily yield the minimal range covering the function's values given the domains of the arguments; instead, it may be a wider interval guaranteed to cover the function's values. This estimate may sometimes be very poor, and the poorer the substitute for the true range of function values, the more unnecessary subdivisions must be performed in the bintree conversion. This problem can usually be overcome by use of suitable transformations on functions. Alander et al. (1984) have built a practical system and shown that function transformations greatly enhance and simplify interval arithmetic algorithms for curved surfaces.

A conversion algorithm based on interval arithmetic evaluates an interval function once for each active halfspace at each node. Furthermore, an ordinary function evaluation has to be performed by CLASSIFY_VOXEL for each active halfspace at each voxel. Thus, the performance of the conversion algorithm depends on the precision with which interval arithmetic estimates the intervals of halfspace values within nodes. Note that whenever INTERVAL_EVALUATE returns an interval

containing 0 when the true interval does not contain 0, then superfluous subdivision will result.
Implementation experience has shown interval arithmetic extensions to be from 5 to 15 times slower than ordinary arithmetic (Clemmesen 1983; Cohen and Hickey 1979). The tightness of the range given by interval arithmetic depends completely on the characteristics of the function within the argument range (Moore 1979). The more that is known about the function, the better the estimates are that can be obtained. For instance, Mudur and Koparkar (1984) decompose functions into monotonic parts in order to be able to use the tighter intervals applicable in such a case.
A general treatment of the characteristics of interval arithmetic is outside the scope of the present paper. However, our experience has shown that from the standpoint of performance it is generally advisible to consider interval arithmetic more as a conceptual model than as an automatic computational device. Thus, efficient custom-tailored methods for obtaining precise range estimates should be developed for each class of halfspaces that is being used. In our case this has been done for quadratic halfspaces (Koistinen 1985). However, that work reveals that standard techniques for enhancing the efficiency of interval arithmetic, as presented by Ratschek and Rokne (1984), lead to computational methods that are identical to our specialized ones.

4.3 Projecting on time

Usually we are not interested in time as such. Often it merely serves as an auxiliary variable for describing motion. For example, the process of determining the swept area for static interference detection is equivalent to a transformation that eliminates the time dimension. In geometric terms it is a projection parallel to the t-axis. In general, we do not know how to perform such a projection directly in the CSG representation. Given a CSG tree having A OP B as its root, we cannot necessarily distribute the projection operation, i.e.,

PROJECTION(A OP B) $\neq$ PROJECTION(A) OP PROJECTION(B)

For example, suppose we are given two nonintersecting objects, as in Fig. 12, that are moving at identical speeds in the direction of the x-axis.

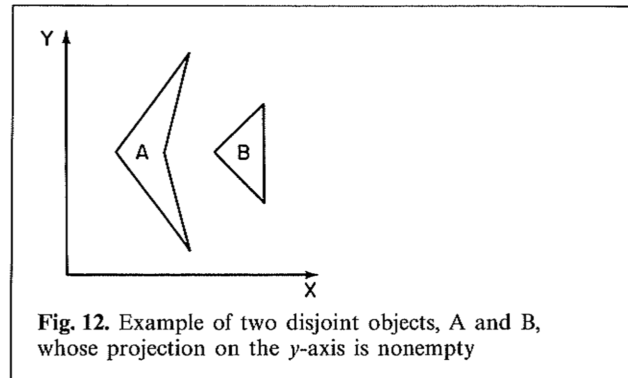


Fig. 12. Example of two disjoint objects, A and B, whose projection on the y-axis is nonempty

Clearly, their swept areas intersect, whereas the objects themselves do not intersect.
Fortunately, projection in the discrete bintree domain is simple. Approximate evaluation of a CSG tree involving a projection operation is a two-step process. We first generate the $(d+1)$-dimensional bintree and then project it to d dimensions to obtain an evaluation of the projected CSG tree as a d-dimensional bintree. Projection consists of eliminating one coordinate and keeping track of all occupied locations in the resulting d-dimensional space. In three dimensions, the projection algorithm is almost identical to that for viewing a 3D bintree in the direction of a coordinate axis (Tamminen et al. 1984). The only difference is that in viewing, some shading information must be recorded at each 2D pixel that is "covered", whereas the projection discussed here only records whether or not such a pixel is covered. In order to simplify our presentation, the subsequent discussion uses the terminology of projecting from two to one dimension and is illustrated by the projection of the 2D image of Fig. 3 on the y-axis (i.e., the elimination of the x coordinate).
Projection is performed by procedure PROJ given below. A bintree node is implemented as a record of type *node* with three fields, LEFT, RIGHT, and TYP. LEFT and RIGHT correspond to the left and right sons, respectively, of a node while TYP indicates a node's type, i.e., BLACK, WHITE, or NON-LEAF. PROJ is invoked with a pointer to the bintree corresponding to the object whose projection is desired and the name of the coordinate (i.e., dimension) that is being eliminated. The output bintree (i.e., corresponding to the projection) is initialized to the WHITE universe. Procedure PROJ traverses the input and output trees in tandem. Two input brother nodes may be either "side-

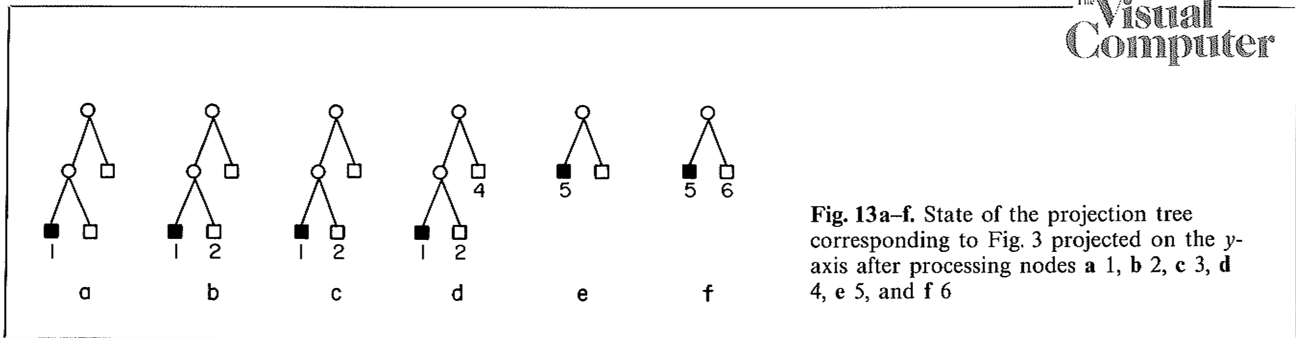


Fig. 13a–f. State of the projection tree corresponding to Fig. 3 projected on the y-axis after processing nodes **a** 1, **b** 2, **c** 3, **d** 4, **e** 5, and **f** 6

by-side" in the projected universe (in cases in which the division is not with respect to the axis of projection) or "on top" of each other. In the latter case the same subtree of the projection tree is processed against both brothers. For example, in Fig. 3 node pairs C and 4, 5 and 6, and 1 and 2 are side-by-side while node pairs B and E, and D and 3 are on top of each other. Note that while processing the projection tree we must create nodes as well as release them. The latter is necessary when an input node is BLACK and the corresponding node in the projection tree is not a leaf (e.g., when processing node 5 in Fig. 3). This situation may also arise when the union of two input brother nodes that are on top of each other in the projection tree is BLACK (e.g., suppose that node 3 in Fig. 3 is replaced by a NON-LEAF node having a WHITE left son and a BLACK right son). Whenever the projection tree is known to be BLACK, then the corresponding part of the input bintree need not be processed. For example, if node 3 in Fig. 3 is replaced by a NON-LEAF node, then once a node corresponding to node 1 has been created in the projection tree, there is no need to process the left son of node 3. Of course, a WHITE input bintree leaves the projection tree unchanged (e.g., when processing nodes 2, 3, 4, and 6 in Fig. 3). Figure 13 shows the state of the projection tree corresponding to Fig. 3 after processing nodes 1, 2, 3, 4, 5, and 6.

```
recursive procedure PROJ(IN,OUT,C,X);
/* Construct a (D-1)-dimensional bintree rooted at OUT cor-
   responding to the projection of the D-dimensional bintree
   rooted at IN such that coordinate X is eliminated. Initially,
   OUT corresponds to a single WHITE node. The node
   pointed at by IN partitions the universe along coordinate
   C.*/
begin
  value pointer node IN,OUT;
  value integer C,X;
  global integer D;
  if TYP(IN)='NON-LEAF' then
    begin
      if TYP(OUT)='BLACK' then
        return /* The output is totally covered */
      else if C NEQ X then /* Coordinate C is not eliminated */
        begin /* The two sons are "side-by-side" in the pro-
                 jected universe */
          if TYP(OUT)='WHITE' then
            begin /* Allocate two son nodes */
              LEFT(OUT)←create(node);
              RIGHT(OUT)←create(node);
              TYP(LEFT(OUT))←TYP(RIGHT(OUT))
                ←'WHITE';
              TYP(OUT)←'NON-LEAF';
            end;
          PROJ(LEFT(IN),LEFT(OUT),(C+1) mod D,X);
          PROJ(RIGHT(IN),RIGHT(OUT),(C+1) mod
               D,X);
          if TYP(LEFT(OUT))='BLACK' and
             TYP(RIGHT(OUT))='BLACK' then
            begin /* Merge left and right sons */
              TYP(OUT)←'BLACK';
              returntoavail(LEFT(OUT));
              returntoavail(RIGHT(OUT));
            end;
        end
      else /* Eliminate this coordinate by projecting on top
              of the brother */
        begin
          PROJ(LEFT(IN),OUT,(C+1) mod D,X);
          PROJ(RIGHT(IN),OUT,(C+1) mod D,X);
        end;
    end
  else if TYP(IN)='BLACK' then
    /* Output must be BLACK */
    begin
      if TYP(OUT)='NON-LEAF' then
        begin /* Merge left and right sons */
          returntoavail(LEFT(OUT));
          returntoavail(RIGHT(OUT));
        end;
      TYP(OUT)←'BLACK';
    end;
    /* Nothing needs to be done for a WHITE input node. */
end;
```

Procedure PROJ has an execution time that is proportional to the number of nodes in the input bintree. The algorithm as given assumes an explicit tree representation for both the input and projection bintrees. However, it can be easily modified to process a bintree represented by a DF-expression. In particular, the only change that is required is to skip the subtrees of a node in the input tree

when encountering a BLACK projection tree. The projection tree cannot be as easily handled by a DF-expression since parts of the tree must be retraversed and possibly modified as is the case when two sons in the input tree are on top of each other in the projection tree.

5 Analysis

A quick perusal of procedure CSG_TO_BINTREE, as given in Sect. 3.2, reveals that the amount of work performed in the conversion is proportional to the sum of the sizes of the CSG trees that are active at the bintree nodes (i.e., blocks) that are evaluated. This number can be quite large even though procedure CSG_TRAVERSE attempts to prune the CSG tree each time it descends to a deeper level in the tree. The fact that the unpruned part of the CSG tree is copied at each level of descent does not affect the time complexity, nevertheless some of the copying can be avoided by more careful programming as outlined in Sect. 3.2. However, in a typical case, as we descend in the bintree, many of the CSG tree nodes are no longer active, thereby reducing the number of CSG nodes that must be visited. In particular, in this section we will show that for "well-behaved" CSG trees, as the resolution increases, the number of CSG tree node evaluations per bintree block approaches unity. We first examine, heuristically, the average number of halfspaces that are active in a node of voxel size. Next, we prove an asymptotical result concerning the average number of active CSG tree nodes in an arbitrary bintree block.

We are not interested in the absolute worst-case value of the complexity. Very poor cases can be attained by constructing a complicated CSG tree that evaluates to the NULL object in such a way that the whole CSG tree is active in a large number of nodes. For example, consider the intersection of a halfspace with its complement. Instead, we shall focus on the "practical" efficiency of these algorithms. This emphasis is recognized as central in most of the references on the CSG solid representation scheme (e.g., Tilove 1984). In order to do so we refer to the results of Hunter (1978) and Meagher (1980) on the complexity of quadtrees and octrees corresponding to polygons and polyhedra respectively. In particular, the worst-case size (i.e., number of nodes) in the bintree of a polygon (polyhedron) is proportional to its perimeter (surface area) measured at the given resolution. This is also a lower bound for polygons (polyhedra) whose edges (faces) do not coincide with the boundaries of quadtree (octree) blocks.

Prior to presenting a more formal analysis, we say something about the number of halfspaces that are active at each node of voxel size in a bintree of a polyhedron. This discussion is heuristic in that we assume that we speak of voxels as if they were infinitely small. At each vertex, at least three halfspaces are active. Elsewhere, at each edge of the polyhedron, exactly two halfspaces are active. Elsewhere, at each face, only one halfspace is active. We can estimate the total number of active halfspaces by counting the number of voxels that intersect the edges, vertices, and faces of the polyhedron. Recalling the results of Hunter (1978) and Meagher (1980), we know that the total number of voxels intersecting faces is proportional to the surface area, while the total number of voxels that intersect edges is proportional to the sum of the edge lengths at the given resolution. The number of voxels containing vertices is always bounded by the number of vertices, irrespective of resolution. Assuming a resolution of M, the number of voxels with more than one active halfspace grows only linearly with M, while the total number of voxels grows with M^2. Thus, the average number of halfspaces active in a node of voxel size approaches one asymptotically in a CSG tree that corresponds to a polyhedron. A similar result will hold for polyhedron-like objects of arbitrary dimension. In the remainder of this section we shall prove more formally a similar result for CSG tree nodes active at bintree blocks. We shall also elaborate on the meaning of "well-behaved."

It should be clear that the amount of work necessary in performing the conversion is at least proportional to the number of nodes in the bintree. Thus, the goal of our analysis is to try to define a class of CSG trees for which the complexity of their evaluation is of the same order as the number of nodes in the bintree of the corresponding object. Generalizing Hunter's and Meagher's image complexity results for polygons and polyhedra to d dimensions leads to a complexity of $O(M^{d-1})$ bintree nodes for bintrees of resolution M. Attaining this bound is feasible if we can show that the proportion of bintree nodes in which more than one CSG tree node is active approaches zero as the resolution increases. Such CSG trees are said to be "well-

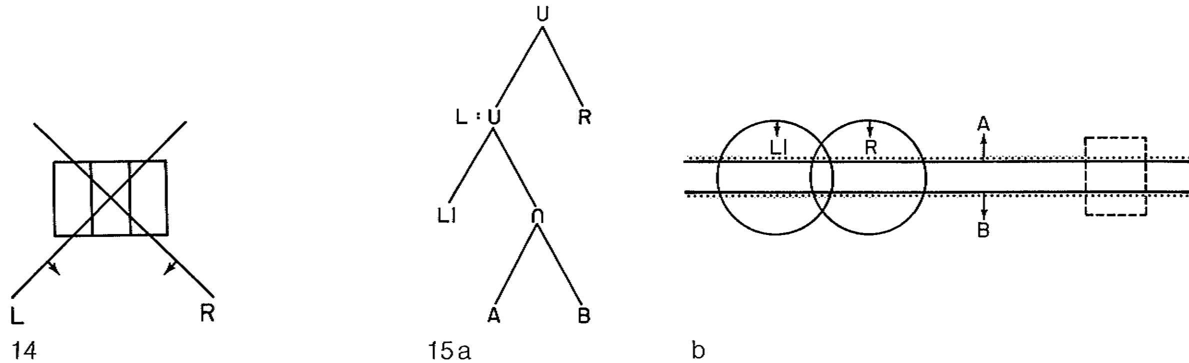


Fig. 14. Example of an intersection of two halfspaces, L and R

Fig. 15a, b. a A CSG tree and **b** its corresponding object

behaved", and this concept applies also to CSG trees with nonlinear halfspaces. This characteristic is determined solely by the way the objects defined by the pair of brother subtrees of the CSG tree intersect each other. In a well-behaved CSG tree the intersections are not allowed to be "tangential". In two dimensions, this means that the boundaries of the objects corresponding to brother subtrees intersect at only a finite number of points. In three dimensions, for polyhedra, the boundaries should not coincide, but are permitted to intersect along one-dimensional edges. In the general case, for d dimensions, the permitted intersection must similarly be at most $(d-2)$-dimensional (see below for more details). Note that at such bintree nodes (with more than one active CSG tree node) the number of active CSG tree nodes is bounded by the total number of CSG tree nodes. Thus, the average number of CSG tree nodes active at bintree blocks approaches one as the resolution is increased.

At this point let us examine more carefully what constitutes a "well-behaved" CSG tree. However, let us first make a few observations. By the nature of CSG, the objects with which we are dealing are multidimensional polyhedra. Let T, L, R, and U refer to CSG tree nodes such that U is some node in T having L and R as its left and right subtrees respectively. Being CSG tree nodes, T, L, R, and U also correspond to some geometric objects. In our discussion when we speak of boundaries of T, L, R, and U we are referring to the boundaries of the corresponding objects.

In the following we analyze the number of CSG tree node evaluations that are required to evaluate T at a given bintree level. We shall use the term block to denote nodes of the bintree at some arbitrary level k. $w(k)$ indicates the longest distance (i.e., diagonal) within a bintree block at level k and $w'(k)$ indicates the shortest side length of such a block. The CSG tree T is evaluated with respect to descendants of a bintree block B only if T does not evaluate to BLACK or WHITE with respect to B. The size of the tree to be evaluated with respect to the sons of B depends on the result of the pruning algorithm applied in B. The CSG nodes that remain in the tree pruned with respect to B are active in B. Let U be a CSG node whose sons L and R are leaves (halfspaces). Node U is active only in bintree blocks that intersect the boundary of the corresponding object. Now, due to the pruning that has been applied in our evaluation algorithm, we see that all three CSG tree nodes U, L, and R are active only in those bintrees blocks at level k that intersect portions of boundaries of L and R (termed *critical portions*) where the distance to the brother boundary (i.e., L for R and vice versa) is at most w. In all other bintree blocks at level k only one of CSG tree nodes L or R is active. For example, consider Fig. 14 that illustrates the 2D case and shows that U, L, and R are all active in three bintree blocks for the given value of w.

In the example above the subtrees of U were leaves. The following lemma shows that a similar correspondence between pruned trees and boundaries of objects corresponding to subtrees is valid even when the subtrees of U are not leaves. Note that the result is not trivial: a pruned CSG tree may be active in a bintree block even though the CSG

tree defines a NULL object in the region corresponding to the block. For example, consider the CSG tree given in Fig. 15a consisting of the two circles L1 and R and the halfspaces A and B as shown in Fig. 15b. The CSG tree of Fig. 15a is active in the bintree block represented by the dashed square in Fig. 15b even though the object defined by it does not extend so far.

Lemma 1. *Let T be a CSG tree and B a bintree block. If the CSG tree resulting from pruning T with respect to B contains more than one halfspace, then B intersects the boundaries of both the left subtree L and right subtree R of some node U of T.*

Proof. Assume that the lemma is not true. That is, there exists a block B, whose pruned tree TP contains at least two leaves (i.e., halfspaces) such that at no node U of T does B intersect the boundaries of L and R (i.e., sons of U). Let TP have d levels where the root is said to be at level 0. First, we will show that B must intersect the boundaries of the two sons (i.e., leaves) of some internal node of TP at level $d-1$. Assume that B does not intersect the halfspace boundaries of both sons at any node at this level. This means that at each such node at least one of the halfspaces will be pruned away, and the depth of the pruned tree is at most $d-1$, which contradicts the assumption that the depth is d. Therefore, B must intersect two leaves that are halfspaces and are brothers in TP. Let these halfspaces be H_1 and H_2. Now, the pruning algorithm operates so that H_1 is the result of pruning one subtree of TP with respect to B, and H_2 is the result of pruning its brother with respect to B. Clearly, H_1 and H_2 define the part of the boundaries of these subtree that lie within B. Thus we have proved the lemma by contradiction.

Given internal CSG tree node U with sons L and R, define $c(U,k)$ to be the number of bintree blocks at level k that intersect the boundaries of both L and R. When U is a CSG tree leaf node, $c(U,k)$ is zero. $c(U,k)$ must be bounded if we are to achieve our goal that the proportion of bintree nodes with more than one active CSG tree node approaches zero with increasing resolution. $c(U,k)$ is bounded only if the objects intersect in a well-behaved manner so that their critical boundary portion is situated around the intersection points of their boundaries and the "size" of the critical boundary portion goes to zero with w. For polyhedra this situation arises when there are no tangential intersections (i.e., intersections where L and R coincide along a portion of boundary with a non-zero $(d-1)$-dimensional measure). We show below in the 2D case (i.e., $d=2$), that $c(U,k)$ remains bounded as k increases by proving that the length of the critical portion divided by w remains bounded. Note that irrespective of whether the operation at a CSG node is UNION or INTERSECTION, the complexity of evaluating the node depends on how the objects corresponding to L and R intersect each other.

Theorem 1. *If L and R correspond to two polygons that are nowhere tangential to each other, then $c(U,k)$ remains bounded as k increases.*

Proof. Let i be the number of intersection points between the boundaries of L and R, δ be the minimum distance between nonintersecting edges of the boundaries, and α be the minimal angle (measured as an absolute value) between edges of L and R at the intersection points of the boundaries (e.g., Fig. 16). Let level k be such that $w(k)<\delta$. For such a k, only bintree blocks that intersect portions of boundaries of both L and R around intersection points can contribute to the value of $c(U,k)$. The remaining bintree blocks at level k evaluate to BLACK or WHITE, or merely intersect the boundary of L, or that of R by our choice of δ. Around each intersection point, the length of those portions of the boundary of L that are within $w(k)$ of R is bounded by $2\cdot w(k)/\sin(\alpha)$ since there are two such segments and each has a maximum length of $w(k)/\sin(\alpha)$. An upper bound on the number of bintree blocks that can be intersected by a line segment of length x is $2\cdot(2+x/w'(k))$. Therefore, the total number of bintree blocks at level k that are intersected is bounded by $i\cdot(4+2\cdot w(k)/(w'(k)$

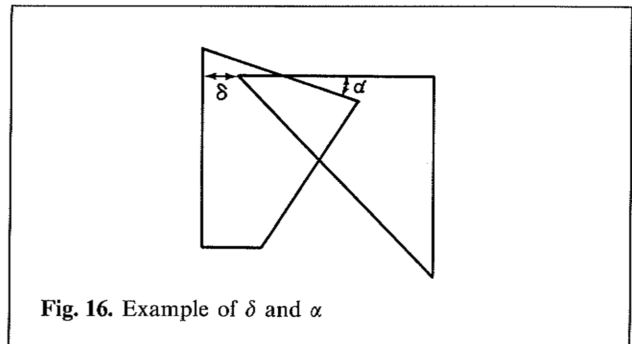


Fig. 16. Example of δ and α

$\cdot \sin(\alpha)$)), which is bounded by a constant as k increases.
Results similar to Theorem 1 can be obtained for $d=3$ and arbitrary dimensions. Let $c(U,k,d)$ denote the number of bintree blocks of level k that intersect the boundaries of both L and R, corresponding to d-dimensional polyhedra, sons of CSG tree node U. We say that a node U of a d-dimensional CSG tree is well-behaved if there exists a constant $K(U,d)$, such that $c(U,k,d) \leq 2^{(d-2)\cdot k} \cdot K(U,d)$. Now, let us place an interpretation on this bound. When $d=3$, the intersection of two halfspaces is well-behaved if the halfspaces intersect each other in a nontangential fashion (i.e., along a line and termed an *intersection edge*). More generally, if L and R define two polyhedra, U is well-behaved if the faces of these polyhedra do not coincide with each other, but intersect along 2D edges. In the case of L and R intersecting along such intersection edges, we know that the intersection edges, in turn, intersect $O(2^k)$ bintree blocks of level k. Using an analysis similar to that used in the proof of Theorem 1, we can show that in the neighborhood of each bintree block that intersects an intersection edge, there can exist only a bounded number (i.e., $O(1)$) of bintree blocks that intersect the boundaries of both L and R. Thus, by multiplying the two quantities $O(2^k)$ and $O(1)$, the number of blocks where both L and R can be active is $O(2^k)$.
At this point let us apply these bounds to d-dimensional polyhedra represented by CSG trees. A CSG tree is well-behaved if all of its nodes are well-behaved. From Lemma 1 we know that a CSG node, say T, is active in a bintree block, say B, only if B intersects the boundaries of both the left and right subtrees (say L and R) of some node U of T. Equivalently, if two or more halfspaces of T are active at some bintree block B, then B must intersect the boundaries of both L and R at some node U of T. Assuming a well-behaved CSG tree rooted at T with N nodes, let $C(T,k,d)$ be the maximum of $c(U,k,d)$ over all CSG tree nodes U. Therefore, the total number of bintree blocks at level k where more than one CSG tree node is active is at most $N \cdot C(T,k,d)$. Summing up over all k levels of the resulting bintree results in the total number of bintree nodes where more than one CSG tree node is active being bounded by $k \cdot R \cdot C(T,k,d)$, where R is some constant. We are now ready to prove our main result. In doing so we assume a d-dimensional polyhedron with irrational vertex coordinates so that the vertices do not coincide with bintree subdivision points. This guarantees that the bintree grows with increasing resolution.

Theorem 2. *Let T be a well-behaved CSG tree defining a d-dimensional polyhedron with irrational vertex coordinates and a nonzero $(d-1)$-dimensional boundary measure. The proportion of bintree nodes where more than one CSG tree node is active approaches zero asymptotically as the resolution increases.*

Proof. From Hunter (1978) and Meagher (1980), and extrapolating to d dimensions, we know that the number of nodes in the bintree of the object defined by T, say $B(T)$, with resolution $M=2^k$ is of the order of the $(d-1)$-dimensional boundary measure, which in turns is $O(2^{(d-1)\cdot k})$. Let R and $K(T,d)$ be constants. From Theorem 1 and its generalization to d dimensions we know that in a well-behaved CSG tree T the total number of bintree nodes in which more than one CSG tree node is active is $k \cdot R \cdot C(T,k,d)$, where $C(T,k,d) \leq 2^{(d-2)\cdot k} \cdot K(T,d)$. The ratio of this quantity to $B(T)$ is of the form $k/2^k$ that approaches zero asymptotically as k increases.
The above analysis and discussion have been restricted to polyhedra defined by linear halfspaces. In fact, Theorem 2 and the concept of a CSG tree being well-behaved are not limited to planar faced polyhedra and can be generalized to arbitrary d-dimensional objects including those whose corresponding CSG trees contain nonlinear halfspaces in the following manner. Let T be a CSG tree defining a d-dimensional object S. Moreover, let $B(S,k,d)$ denote the number of nodes in the bintree of S at resolution 2^k. Recall that the bintree of an object is defined as the bintree obtained by classifying voxels as BLACK if they are contained in the object or intersect its surface, and WHITE otherwise.
We say that S is d-dimensionally *nondegenerate* if there exists a constant $A>0$, such that $B(S,k,d)/2^{(d-1)\cdot k} > A$ for all k. Aside from not allowing the dimensionality of the object to be "too small", nondegeneracy implies that the object is not representable exactly by a finite bintree. In addition we say that T is *well-behaved* if at each node of T, locally, the number of bintree blocks at level k where both subtrees of T are active is $O(2^{(d-2)\cdot k})$. In other words, the objects defined by the two sub-

trees do not intersect "tangentially". With these definitions we can recast Theorem 2 as Theorem 2′ given below, which can be proved in a similar manner.

Theorem 2′. *Let T be a well-behaved CSG tree defining a d-dimensionally nondegenerate object. The proportion of bintree nodes where more than one CSG tree node is active approaches zero asymptotically as the resolution increases.*

The above results lead us to draw the following unexpected but practical conclusions about the performance of our algorithms for the conversion of CSG trees to bintrees when the CSG trees are well-behaved.

1. The "practical" complexity of CSG tree evaluation is $O(M^{d-1})$ as resolution M is increased.
2. The average number of active CSG tree nodes in a bintree block approaches one asymptotically as resolution is increased.
3. The computational complexity of converting a CSG tree approximation of a given object to a bintree is asymptotically independent of the number of halfspaces used in the approximation.

Result (3) means that the linear approximation of curved halfspaces can be computationally practical even though it leads to a great increase in the size of the CSG tree.
Our definition of well-behaved eliminates many practically relevant CSG trees from the analysis. This limitation is discussed further, and partially remedied, in Sect. 7. Of course, the above results are asymptotical and thus are directly relevant only when the number of halfspaces is not large in comparison to the resolution. Nevertheless, the next section shows that in such cases the observed behavior correlates well with our predictions.

6 Empirical results

In order to verify the analysis of Sect. 5 we conducted a number of experiments with polyhedron-like objects in several dimensions. Our experiments have been performed with versions of the algorithms of Sect. 3.2, as implemented in C in the system (Tamminen et al. 1984) and executed on a VAX 11/750 running version 4.2BSD of UNIX. In the experiments, unless indicated otherwise, a simplified version of procedure CLASSIFY_VOXEL was used that classified all GRAY voxels as BLACK. Note that the contribution to CPU time incurred by writing the packed DF-expresion into a file is quite noticeable.
At each bintree node a fixed amount of work is performed for each CSG tree node that is active in it. Thus, an implementation-independent measure of work is the total number of CSG tree nodes active at all of the bintree nodes. This is reported as the statistic "CSG evaluations" in Table 2. The statistic "Halfspace evaluations" forms part of it and denotes the number of halfspace value range computations performed. Note that the statistic "BIN nodes" gives the number of bintree node evaluations and not the size of the final bintree, in which brother leaves with identical color have been merged. This is the statistic that we need to compare with the theorems of the previous section since collapsing is an artifact of the way we choose to deal with nodes at the voxel level and does not enter into the analysis. Note that collapsing could reduce the number of nodes by at most one half (e.g., when all GRAY voxels are classified as BLACK). However, the theorems pertain to the conversion process and are still valid. From these values we can derive the average number of CSG tree nodes (or halfspaces) evaluated at each potential bintree node for the purpose of comparison with the theoretical analysis of Sect. 5. Note that the program that we instrumented used a linear CSG tree representation, which allows less pruning than the algorithm we have described in Sect. 3.2. Thus, the number of CSG node evaluations reported below is an upper bound on the true value obtained by the algorithm, although the number of halfspace evaluations is valid.
For our first experiment we approximated a circle with an 11-gon and formed its bintree at resolution 4096. The corresponding CSG tree with 10 INTERSECTION nodes is shown in Fig. 17. This approximation produced a bintree with 80828 nodes and required 87592 CSG tree node and 81283 halfspace evaluations. Of course, collapsing reduced the number of nodes by about one half. Thus, prior to collapsing, on the average, each bintree node contained less than 1.1 active CSG tree nodes, which correlates with our prediction. The CPU time required was 19.2 s, including about 6 s necessary to output the packed DF expression. The time reuired to perform the same task by a program specifically designed to convert convex polyhedra was 17.1 CPU s, so that the overhead of

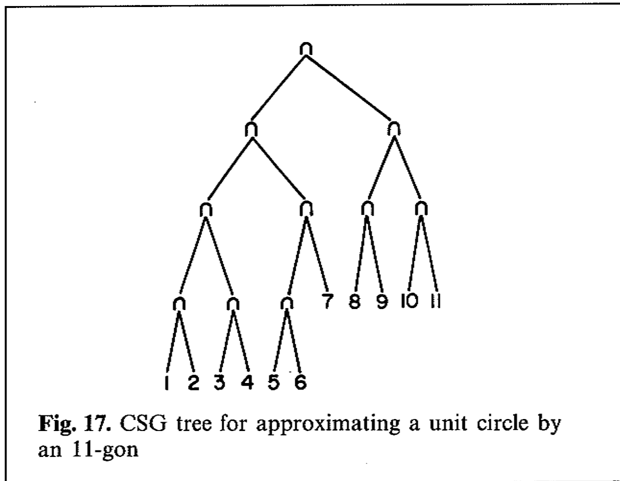


Fig. 17. CSG tree for approximating a unit circle by an 11-gon

the general CSG tree representation was not very large.

The second experiment demonstrates that the complexity of CSG tree evaluation is $O(M^{d-1})$ as resolution M is increased. For this experiment we tabulate in Table 1 the CPU conversion times for a series of approximations of a unit circle by 5, 11, and 19 halfspaces at various resolutions. Notice that the execution time doubles with resolution as predicted for $d=2$. The different approximations are not completely comparable as they represent different objects. Thus, the bintree at resolution 4096 contains 76294, 80628 and 81410 nodes for 5, 11 and 19 halfspaces, respectively.

The third experiment shows that the size of the 3D bintree of a polyhedron is proportional to the square of the resolution, and a 4D bintree is proportional to the third power of the resolution. The execution times for large values of resolution exhibited similar behavior. For this we modeled the motion of two identical square blocks situated at opposite corners of the unit square, moving towards each other, so that at time $t=1$ they overlap on an area of size 0.05 by 0.05. We also performed an identical 3D experiment (two moving boxes) resulting in a 4D bintree. The halfspaces (including the additional variable t) of the block in the lower left corner for the 2D case are given below. Note that, even though the objects are defined by halfspaces parallel to the coordinate axes, the computations in the program are performed as in the general case.

$x-0.30*t\geq 0$

$x-0.30*t\leq 0.25$

$y-0.30*t\geq 0$

$y-0.30*t\leq 0.25$

The halfspaces corresponding to the second block are formed in a similar manner. In the 2D case we have a total of 8 halfspaces and in the 3D case we have a total of 12 halfspaces. Figure 18 illustrates the 2D case. Tables 2 and 3 contain the results of the evaluations of the 3D and 4D trees at varying resolutions. Note that for these examples, the maximum sizes of the universes are 2^{33} and 2^{36} voxels, respectively.

The above experiments verify the intuitively plausible fact that bintrees of objects in motion tend to become very large when represented in the (location, time) space. However, often when analyzing motion, we are only interested in checking whether or not an interference exists. For this purpose we modified the CSG tree conversion algorithm so that it searches only for the first BLACK leaf, and outputs a tree where all the rest of the universe is WHITE. Unlike the previous experiments, voxels are classified according to the color of their centroid.

Tables 4 and 5 show the results of experiments identical to those of Tables 2 and 3, respectively,

Table 1. Conversion times (CPU seconds) for different discs

Number of halfspaces	Resolution				
	256	512	1024	2048	4096
5	1.2	2.1	4.2	8.5	16.2
11	1.7	2.9	4.7	9.4	17.1
19	2.5	3.8	6.1	10.8	18.6

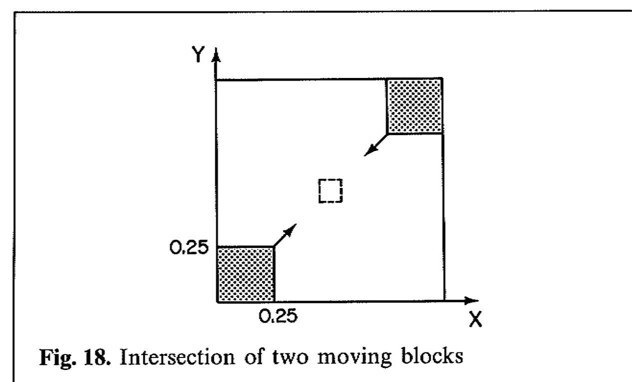


Fig. 18. Intersection of two moving blocks

Table 2. Intersecting two moving two-dimensional blocks

Resolution	CPU seconds	BIN nodes	Halfspace evaluations	CSG evaluations
64	0.7	826	1641	3124
128	1.3	2898	4357	7300
256	3.2	10866	13552	19358
512	10.3	42514	47684	59254
1024	37.6	172802	183335	207248
2048	144.1	699362	720631	769768

Table 3. Intersecting two moving three-dimensional blocks

Resolution	CPU seconds	BIN nodes	Halfspace evaluations	CSG evaluations
16	0.8	370	1846	3900
32	1.2	898	3354	7008
64	3.1	4658	10471	21326
128	12.5	31458	49969	90614
256	67.4	231570	295662	430730
512	428.0	1826466	2071158	2695692

Table 4. Interference test for two moving two-dimensional blocks

Resolution	CPU seconds	BIN nodes	Halfspace evaluations	CSG evaluations
64	0.3	130	511	1036
128	0.3	154	585	1190
256	0.3	178	661	1344
512	0.4	202	736	1498
1024	0.5	226	811	1652
2048	0.4	250	886	1806
4096	0.5	290	998	2030

Table 5. Interference test for two moving three-dimensional blocks

Resolution	CPU seconds	BIN nodes	Halfspace evaluations	CSG evaluations
16	0.4	54	343	712
32	0.4	70	409	856
64	0.4	70	419	872
128	0.5	86	485	1016
256	0.5	86	507	1048
512	0.6	102	573	1192

except that the interference-detection program has been used instead of the conversion program. These experiments demonstrate that interference detection employing the proposed methods is not expensive, and that a high resolution can be used when necessary without an appreciable increase in cost.

Figure 19 shows the number of halfspace evaluations per bintree node as a function of resolution in the experiment of Table 2. Notice that the asymptotical bound obtained in Sect. 5 does hold in this case. Nevertheless, inspection of Tables 2 and 3 shows that the convergence to this bound is not necessarily very fast in the high dimensional (location, time) space. Therefore, this method should be used primarily when the CSG tree is expected to evaluate to NULL, in which case the performance will be similar to that depicted by Tables 4 and 5.

As an example of the conversion of a more complicated 3D object, consider an icosahedron (i.e., a

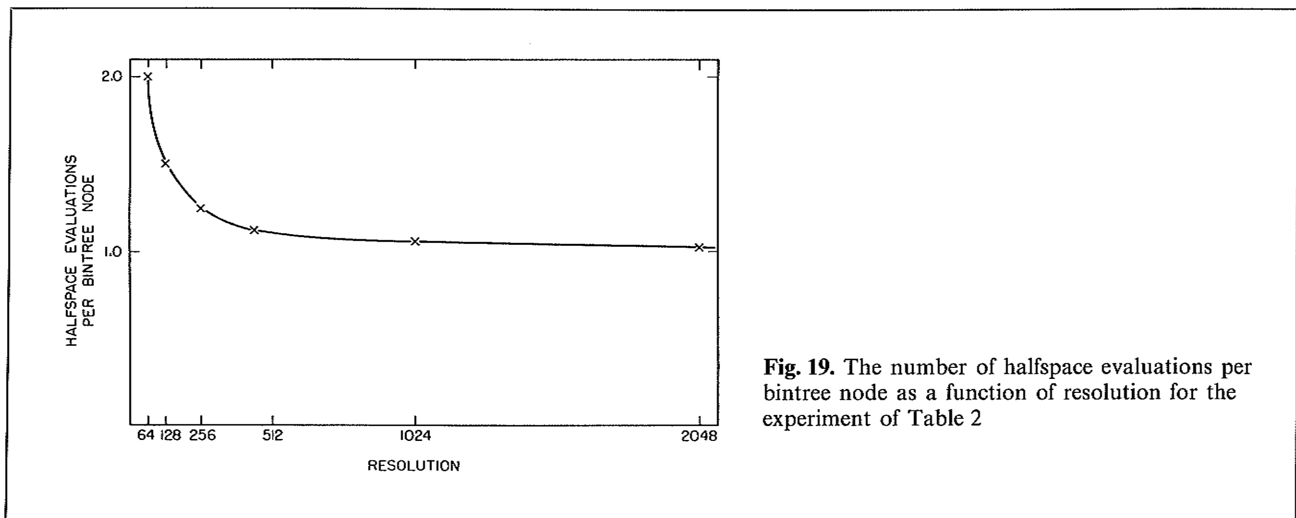


Fig. 19. The number of halfspace evaluations per bintree node as a function of resolution for the experiment of Table 2

Table 6. Converting an icosahedron

Resolution	CPU seconds	BIN nodes	Halfspace evaluations
64	22.7	51102	76331
128	72.1	204638	255257
256	212.0	818864	919187

Table 7. Icosahedron in motion

Resolution	CPU seconds	BIN nodes	Halfspace evaluations
16	9.9	6380	35132
32	34.8	42296	128917
64	173.1	320042	607340
128	1019.0	2522602	3553646

Table 8. Two icosahedra in opposite motion

Resolution	CPU seconds	BIN nodes	Halfspace evaluations
16	4.6	1014	13801
32	7.2	2830	23559
64	14.9	10888	49529
128	43.2	62190	153018

Table 9. Interference test of two icosahedra in opposite motion

Resolution	CPU seconds	BIN nodes	Halfspace evaluations
16	1.0	92	1285
32	1.0	144	1447
64	1.1	262	1942
128	1.1	360	2090
256	1.3	422	2195
512	1.3	474	2289
1024	1.4	604	2479

Table 10. Two icosahedra in touching motion

Resolution	CPU seconds	BIN nodes	Halfspace evaluations
16	3.7	556	8429
32	4.8	968	11260
64	5.9	1816	15660
128	8.8	4302	25359
256	19.2	15338	56020
512	61.3	85743	191647

20-faced regular solid) similar to that shown in Fig. 1. Table 6 summarizes the conversion effort for the icosahedron, while Table 7 describes the 4D bintree conversion related to its motion (scaled by 0.5) when moving from one octant to the opposite octant along the main diagonal of the unit cube. Again, it is seen that 4D bintrees become large, so that swept area should, whenever possible, be determined by more efficient methods than by first forming the (location, time) bintree and then projecting it back to location space.

The size of a (location, time) bintree of a 3D object in motion depends on the 4D extent of the object, as measured by the volume of its 3D boundary. In the experiment of Table 7 the extent of the object is "big", as compared to the experiments reported in Tables 8 and 10. Table 8 describes the (location, time) object obtained when two identical icosahedra move towards each other along opposite courses: one moves from the first to the last octant, the other one from the last to the first octant. The 4D object of Table 8 is described as the intersection of 40 halfspaces. The extent of the intersection of the 4D objects is obviously smaller than each one of them, i.e., what is depicted in Table 7. This is reflected in the descriptors of the amount of conversion effort in Table 8. For the purpose of comparison, Table 9 reports the amount of resources required to just detect interference.

The final icosahedron experiment consists of intersecting the (location, time) object of Table 7 with another one, obtained as follows:

Move the centroid of the icosahedron in the last octant of the unit cube from location (0.75,0.75,0.75) to location (0.25,0.25,0.75) – i.e., along a diagonal parallel to the (x, y)-coordinate plane.

Note that the two objects do not move at the same velocity. The original object of Table 7 moves a distance of $\sqrt{3}/2$ in a time unit, while the new one moves a distance of $\sqrt{2}/2$ in the same time. From the description of the experiment, it is not obvious whether the two moving objects meet at any instant (they do). The results of the experiment are reported in Table 10. Because the objects will barely touch, the 4D extent of their intersection is small. However, the coarseness of approximation at low resolutions means that all of the voxels in which there is a potential for overlap are evaluated – a relatively large number. Therefore, as resolution

inverse halfspaces, it can be replaced by a BLACK leaf.

Our algorithms have several useful applications aside from volume-like computations and interference checking. Viewing 3D CSG models is a prime application (Koistinen et al. 1985). In this case bintree conversion would be performed solely for the sake of generating shaded output. The method of viewing 3D bintrees in the direction of a coordinate axis described by Tamminen et al. (1984) can be used in this case because the eye-point dependent operations (e.g., perspective transformation of halfspaces, etc.) can precede bintree conversion. Shading would be generated from the normals of the halfspaces active at each visible node at voxel level. Evaluation would proceed from front to back and be combined with projection so that the nodes known to be covered would not be generated. The simplicity of our conversion algorithm is such that, especially with suitable hardware support, it might provide a practical alternative as a system for viewing 3D CSG trees (Atherton 1983).

Acknowledgements. We are deeply indebted to Petri Koistinen who produced the shaded figures (i.e., Figs. 1, 2, and 21) and performed the experiments with CSG trees with quadratic halfspaces. We are grateful to Prof. A.R. Forrest for his encouragement. We thank Jarmo Alander, Olli Karonen, Petri Koistinen, Walter Kropatsch, Martti Mantyla, Reijo Sulonen, and Robert E. Webber for their comments. Erik Jansen at Delft University of Technology provided us with the CSG model shown in Fig. 2.

References

Alander J (1984) Interval arithmetic methods in the processing of curves and sculptured surfaces. Proc 6th Int Symp CAD/CAM, Zagreb, Yugoslavia

Alander J, Hyytia K, Hamalainen J, Jaatinen A, Karonen O, Rekola P, Tikkanen M (1984) Programmer's manual of interval package IP. Report-HTKK-TKO-B59, Laboratory of Information Processing, Helsinki University of Technology, Espoo

Atherton PR (1983) A scan-line hidden surface removal procedure for constructive solid geometry. Comput Graph 17:73–82

Boyse JW (1979) Interference detection among solids and surfaces. Commun ACM 22:3–9

Cameron SA (1984) Modelling solids in motion. PhD dissertation, Univ Edinburgh

Clemmesen M (1983) Interval arithmetic implementations using floating point arithmetic. (Institute of Datalogy Report 83/9) Univ Copenhagen, Copenhagen

Cohen J, Hickey T (1979) Two algorithms for detecting volumes of convex polyhedra. J ACM 26:401–414

Cole AJ, Morrison R (1982) Triplex: a system for interval arithmetic. Software Pract Experience 12:341–350

Hunter GM (1978) Efficient computation and data structures for graphics. PhD dissertation, Princeton University

Jackins CL, Tanimoto SL (1980) Oct-trees and their use in representing three-dimensional objects. Comput Graph Image Processing 14:249–270

Jackins C, Tanimoto SL (1983) Quad-trees, oct-trees, and k-trees – a generalized approach to recursive decomposition of Euclidean space. IEEE Trans Patt Anal Machine Intelligence 5:533–539

Jansen FW, Wijk JJ van (1984) Previewing techniques in raster graphics. Comput Graphics 8:149–161

Kawaguchi E, Endo T (1980) On a method of binary picture representation and its application to data compression. IEEE Trans Pattern Analysis Mach Intell 2:27–35

Koistinen P (1985) Viewing solid models by bintree conversion. MS Thesis, Helsinki Univ Technol

Koistinen P, Tamminen M, Samet H (1985) Viewing solid models by bintree conversion. Vandoni CE (ed) Proc EUROGRAPHICS '85 Conf, North-Holland, Amsterdam, pp 147–157

Lee YT, Requicha AAG (1982a) Algorithms for computing the volume and other integral properties of solids: I. Known methods and open issues. Commun ACM 25:635–641

Lee YT, Requicha AAG (1982b) Algorithms for computing the volume and other integral properties of solids: II. A family of algorithms based on representation conversion and cellular approximation. Commun ACM 25:642–650

Mantyla M, Sulonen R (1982) GWB: a solid modeler with Euler operators. IEEE Comput Graph Appl 2:17–31

Meagher D (1980) Octree encoding: a new technique for the representation, manipulation and display of arbitrary 3-D objects by computer. (Rep IPL-TR-80-111) Rensselaer Polytechnic Institute, Troy, NY

Meagher D (1982) Geometric modeling using octree encoding. Comput Graph Image Processing 19:129–147

Meagher D (1982) Octree generation, analysis and manipulation. (Rep IPL-TR-027) Rensselaer Polytechnic Institute, Troy NY

Meagher D (1984) The Solids engine: a processor for interactive solid modeling. Proc NICOGRAPH '84 Conf, Tokyo, November

Moore RE (1979) Methods and applications of interval analysis. SIAM, Philadelphia

Mudur SP, Koparkar PA (1984) Interval methods for processing geometric objects, IEEE Comput Graph Appl 4:7–17

Newman WM, Sproull RF (1979) Principles of interactive computer graphics, 2nd edn. McGraw Hill, New York

Okino N, Kakazu Y, Kubo H (1973) TIPS-1: technical information processing system for computer aided design, drawing and manufacturing. In: Hatvany J (ed) Computer languages for numerical control. North-Holland, Amsterdam, pp 141–150

Ratschek H, Rokne J (1984) Computer methods for the range of functions. Ellis Horwood, Chichester

Requicha AAG (1980) Representations of rigid solids: theory, methods, and systems. ACM Comput Surv 12:437–464

Requicha AAG, Voelcker HB (1982) Solid modeling: a historical summary and contemporary assessment. IEEE Comput Graph Appl 2:9–24

Requicha AAG, Voelcker HB (1983) Solid modeling: current status and research directions. IEEE Comput Graph Appl 3:25–37

Samet H (1990a) The design and analysis of spatial data structures. Addison-Wesley, Reading, Mass

Samet H (1990b) Applications of spatial data structures: computer graphics, image processing, and GIS. Addison-Wesley, Reading, Mass

Samet H, Tamminen M (1985) Computing geometric properties of images represented by linear quadtrees. IEEE Trans Pattern Anal Mach Intell 7:229–240

Srihari SN (1981) Representation of three-dimensional digital images. ACM Comput Surv 13:399–424

Tamminen M, Samet H (1984) Efficient octree conversion by connectivity labeling. Comput Graphics 18:43–51 (also presented at the SIGGRAPH '84 Conf, Minneapolis, July 1984)

Tamminen M, Koistinen P, Hamalainen J, Karonen O, Korhonen P, Raunio R, Rekola P (1984) Bintree: a dimension independent image processing system. (Report-HTKK-TKO-C9) Helsinki Univ Technol, Espoo

Tilove RB (1981) Exploiting spatial and structural locality in geometric modeling, TM-38. Production Automation project, Univ Rochester

Tilove RB (1984) A null-object detection algorithm for constructive solid geometry. Commun ACM 27:684–694

Voelcker HB, Requicha AAG (1977) Geometric modeling of mechanical parts and processes. IEEE Comput 10:48–57

Wallis AF, Woodwark JR (1984) Creating large solid models for NC toolpath verification. Proc CAD 84

Weng J, Ahuja N (1987) Octrees of objects in arbitrary motion: representation and efficiency. Comput Vision Graph Image Processing 39:167–185

Woodwark JR, Quinlan KM (1982) Reducing the effect of complexity on volume model evaluation. Computer-aided Design 14:89–95

Yau M, Srihari SN (1983) A hierarchical data structure for multidimensional digital images. Commun ACM 26:504–515

HANAN SAMET received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. Degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.
In 1975 he joined the Computer Science Department at the University of Maryland, College Park, where he is now a Professor. He is a member of the Computer Vision Laboratory of the Center for Automation Research and also has an appointment in the University of Maryland Institute for Advanced Computer Studies.
His research interests are data structures, computer graphics, geographic information systems, computer vision, robotics, programming languages, artificial intelligence, and database management systems. He is the author of the books *The Design and Analysis of Spatial Data Structures*, and *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS* both published by Addison-Wesley, Reading, MA, 1990.

MARKKU TAMMINEN (M'83) received the M.Sc. degree in applied mathematics in 1966 from the University of Helsinki, Finland, and the Ph.D. degree in computer science from the Helsinki University of Technology in 1982.
He had been with the Laboratory of Information Processing Science, Helsinki University of Technology, since 1980. From 1973 to 1980 he was Chief Mathematician at the Data Center of the Helsinki Metropolitan Area. His research interests include data structures based on address computation, computational geometry, image data structures and algorithms, computer aided design, and the management of spatially referenced data.
Dr. Tamminen was a member of the Association for Computing Machinery, the IEEE Computer Society, and the Operations Research Society of America.
The world of computer science, especially computer graphics, has lost one of its main contributors.