Blist: A Boolean list formulation of CSG trees

Jarek Rossignac
GVU Center
Georgia Institute of Technology

Abstract

Set membership classification algorithms visit nodes of a CSG tree through a recursive divide-and-conquer process, which stores intermediate results in a stack, whose depth equals the height, H, of the tree. During this process, the candidate sets is usually subdivided into uniform cells, whose interior is disjoint from primitives' boundaries. Cells inside the CSG object are identified by combining the binary results of classifying them against the primitives. In parallel systems, which allocate a different process to each leaf of the tree, and in algorithms that classify large collections of regularly spaced candidate sets (points, pixels, voxels, rays, or cross-sections) against the primitives using forward differences, a separate stack is associated with each candidate or cell. Our new representation for CSG trees, called Blist, distributes the merging operation to the primitives and reduces the storage requirement for each cell to log(H+1) bits. Blist can represent any Boolean expression as a list of primitives, each containing a reference to the primitive's description (type, parameter, transformation), a sign, a stamp, and a name. During set membership classification, a label is attached to each cell and passed to the successive primitives in the Blist. When the name written on the label matches the primitive's name, the cell is classified against the primitive. If the result matches the primitive's sign, the name stored in the primitive's stamp is put on the label—if not, a zero name is used. The elimination of the intermediate CSG nodes and of the recursive merging operations make the Blist architecture particularly well suited for parallel hardware configurations. We provide a simple algorithm for converting CSG expressions to Blists. It uses rotations on the positive form of the tree to reduce the number of bits needed for each label.

Introduction

Various representations of solids are surveyed in [Rossignac94]. We focus here on CSG representations, which combine primitive shapes through regularized Boolean expressions. The primitive shapes define regularized solids [Requicha80, Mantyla88], which may be represented as parameterized primitives, such as cylinders or blocks, or as more general boundary or procedural representations. With each primitive is associated a transformation, often restricted to be a rigid body motion or linear map. In what follows, primitives' descriptions, which may involve the primitive type, parameters, and transformation coefficients, are stored in a table indexed an integer, p, identifying a primitive. The Boolean expression combines the primitive shapes through union ($\cup$), regularized intersection ($\cap$), and regularized difference ($-$) operations. It may be represented as a binary tree, whose interior nodes store the Boolean operators and whose leaves store integer references to the primitives in the table.

To make things more precise, we assume that the input CSG model is represented a binary tree. With each node, n, is associated a structure with several fields. The field n.type specifies the type of the node: leaf or internal.

Internal nodes contain the following fields:

- n.type, which is equal to *node*
- n.operator, an operator ($\cup$, $\cap$, or $-$)
- n.leftChild, a pointer to the left child
- n.rightChild, a pointer to the right child
- n.parent, a pointer to the parent node (null for the root)

Leaf-nodes contain the following fields:

- n.type, which is equal to *leaf*
- n.parent, a pointer to the parent node
- n.primitiveReference, which identifies the corresponding primitive in a Table of Primitives, which contains a complete description of the primitive, for example: the primitive's type, parameters, scale, position, orientation, and color.
- n.primitiveID, an integer used during the CSG-to-Blist conversion to identify the corresponding entry in the Blist table.

Many other, popular CSG representation may be converted to this simple format. For instance, rooted, directed, acyclic CSG graphs may be expanded into binary CSG trees. CSG trees with transformation nodes may be converted to trees with only Boolean nodes by composing the transformations that are applied to each primitive and by storing the result in the table of primitives.

Let S be an r-set [Requicha80]. Let X be a candidate set: curve, surface, volume, edge-neighborhood [Requicha85, Banerjee96]. A set membership classification process [Tilove80] segments a candidate set, X, into three subsets: X∩iS; X–S; and X∩bS, where iS and bS stand respectively for the interior and the boundary of S [Alexandroff61]. All set membership classification algorithms perform two tasks: (1) subdivide X into cells of uniform classification against primitives and (2) combine the binary results of classifying these cells against the primitives according to the corresponding Boolean expression. CSG-to-boundary conversion algorithms are often based on such set-membership classifications [Requicha85, Mantyla86, Hoffmann89, Rossignac86]. Subdivision typically involves computing intersections between the carrier of X (i.e. its supporting manifold [Rossignac98b]) and the surfaces that bound the primitives. Classification may in general be expressed as a combination of binary values [Rossignac86, Rossignac89, Banerjee96], which represent the results of classifying the cell against the interior of the primitives. The combination uses Boolean operators (OR, AND, and AND NOT) for the corresponding CSG operators (∪, ∩, and –).

The standard way of implementing the evaluation of such expressions [Requicha85] is illustrated by the pseudo-code of Procedure *ClassifyAgainstTree*, below, which is invoked using the cell and the root-node of the tree as arguments.

```
PROCEDURE ClassifyAgainstTree(cell,node)
    IF node.type = leaf
        THEN RETURN InPrimitive(node.primitiveReference, cell)
        ELSE RETURN Combine(ClassifyAgainstTree(cell,node.leftChild),
                            node.operator, ClassifyAgainstTree(cell,node.right.child));
```

When the set membership classification process implements such a recursive procedure of evaluating the Boolean expression, it requires a stack of binary values, whose length equals the height, H, of the binary tree (see Fig. 1).

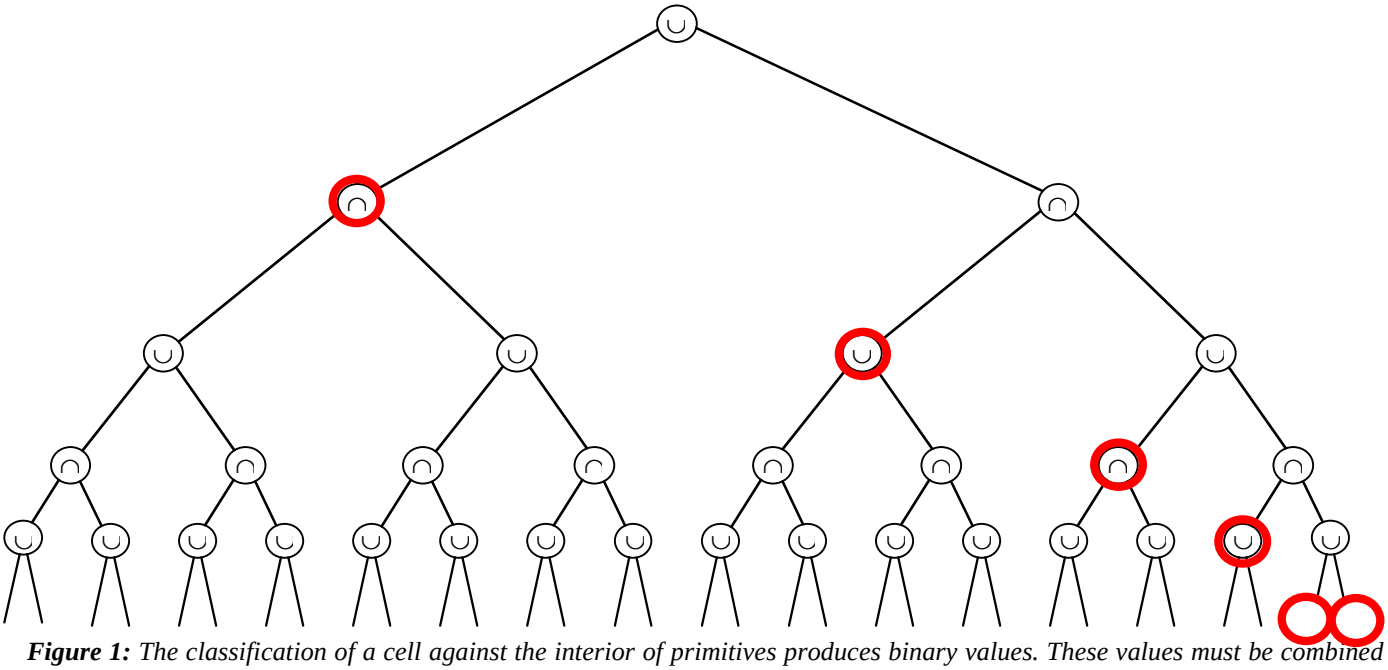

Figure 1: *The classification of a cell against the interior of primitives produces binary values. These values must be combined according to a Boolean expression, which may be represented by a binary tree. The combining process uses a stack to store the results of previous combine operations. Fat red circles indicate a set of nodes whose value may be contained in the stack at some point of the classification. When the tree is full (as shown here), the depth of the stack is the height of the tree.*

Note that the recursive procedure produced above may be improved by avoiding the *ClassifyAgainstTree*(cell,node.right.child)) call when its result is irrelevant. For example, if A returns FALSE, then A∩B is FALSE no matter what B returns. Similarly, when A is TRUE, then A∪B returns true, no matter what B returns. The Blist technique proposed here avoids all these unnecessary calls.

The CSG tree described above could be represented in different ways. For example, using the inverse Polish notation, the CSG expression *(A∩(B∪C)) – ((D∪(E–F))∩G)* could be represented by the sequence of operations ###∪∩###–∪#∩–, where # pushes a new value on the stack and where the other operators combine the two top elements of the stack.. The expression could be evaluated without recursion. Nevertheless, the evaluation would use a stack, whose length would be equal to the height of the tree, at least when the CSG tree is full (i.e. when all leaves are at the same depth).

The storage required for this stack is usually not a problem when the cells are classified one at a time against the entire CSG model. However, when the cells form a regular array of points in space or on the boundary of a primitive [Rossignac86, Goldfeather86] or

when they form a pencil of rays [Bronsvoort84, Ellis91], it may be more efficient to classify all cells against one primitive, before classifying them against the next primitive. For example, rasterization [Fuchs82] techniques may be used to classify all pixels or all voxels against a single primitive and forward difference techniques may be used to compute the intersections of a family of rays with an algebraic surface [Kedem84]. In such cases, either all the results of cell-primitive classification must be stored for all primitives before they are combined, or each step of the combine process may be performed as soon as its arguments are available. The latter solution requires storing a stack of intermediate results for each cell.

Parallel implementations of algorithms that classify large amounts of cells (points, voxels, or ray segments) may assign one primitive to each processor or thread. A cell may be either classified simultaneously by all threads [Kedem84b] or may be classified against the primitives one-by-one in a pipeline fashion. The pixel-planes architecture [Fuchs82] combines both approaches. In any case, these binary classification results for each cell must be combined according to the Boolean expression that defines the CSG solid. The combine process requires passing the results of the cell-primitive classifications between processors, and may be implemented as a hardware combine-tree, an array of p by log(p) processors [Kedem84b, Kedem88]. Because the array does not grow linearly with the number of primitives, it is difficult to extend such architectures to support larger CSG models without performing several passes or breaking the tree into subsets, for which intermediate results must be stored and recycled.

We propose in this paper a new representation for Boolean expressions. We call it Blist, because it may be represented as a list of primitive, instead of a tree, and may be evaluated in a pipeline fashion, combining at each step the result of classifying the cell against the current primitive with the result of the previous classifications. The fundamental breakthrough provided here lies in the fact that the result of the previous classifications does not require the list of values of cell-primitive classification results, nor a stack of intermediate results of evaluating sub-expressions. Instead, Blist passes from one primitive to the next a simple label, which may be stored using at most $\lceil \log(H+1) \rceil$ bits, where H is the height of the CSG tree. Note that log(H) equals log(log(|P|)), where |P| is the total number of primitives.

Using a Blist representation, we can evaluate any Boolean expression without using recursion or a stack. In fact, the whole binary merging process is replaced by a simple comparison between the content of the label that is attached to the cell and the name of the primitive.

In the remainder of the paper, we discuss prior art; define the structure of the Blist table; propose a simple algorithm for converting CSG trees to a Blist form while minimizing the number of bits needed for each label; and provide an intuitive explanation of the essence of our approach. We demonstrate the conversion process and the evaluation algorithm on an example.

Prior art

Over the last 15 years, an impressive number of techniques were proposed for accelerating the rendering of CSG models. The few mentioned here can not be substituted for a comprehensive survey of this area. Images of CSG solids may be computed by casting one ray at a time against a CSG model [Bronsvoort84, Ellis91], by processing the primitives one by one and by adaptively classifying points on their boundary against the CSG tree [Rossignac86, Rossignac96] or by subdividing space adaptively [Morris85, Woodwark82]. Several hardware architectures have been proposed for rendering CSG models [Fuchs82, Meagher84, Soto85, Jansen86, Goldfeather86, Jansen87, Kedem88, Rossignac90b]. They either cast rays against the CSG tree or classify points on the boundary of primitives. Many of these approaches could be considerably improved by using our Blist representation.

The work reported here stems from the author's attempts to generalize the solid capping approach described in [Rossignac92] to situations where the assembly of solids was clipped by an arbitrary Boolean combination of planar half-spaces.

The number of bits that must be associated with each cell to store the results of the previous Boolean evaluations before they are merged may be significantly reduced by expanding the CSG tree into a disjunctive form [Rossignac94b] or even a more general form, which is the union of lists [Goldfeather88]. Unfortunately, such expansions generate a large number of primitives' repetitions. Although these repetitions can often be reduced by identifying empty products or list, they have a significant impact on the overall performance of the rendering algorithms. The number of repetitions may be reduced by pre-computing some geometric intersections and by using the fact that a z-buffer rendering architecture renders correctly unions of overlapping solids [Rossignac86, Rappoport97]

The Blist method proposed here transforms in some sense the CSG tree into a decision graph. A primitive classifies a candidate cell and depending on the result, forwards the cell to one or another primitive. The same principle is used in Binary Space Partition (BSP) Trees [Naylor86]. Such trees may be merged through Boolean operations [Naylor90] and used to perform Boolean operations on polyhedral sets [Thibault97]. Nevertheless, a BSP tree formulation of a general CSG model with n primitives would require 2^n nodes, unless geometric test were used to identify which of these nodes correspond to empty sets.

Blist representation

Blist represents a CSG expression as a table, called BL, of primitive entries. The entry BL[p] associated with primitive number p contains:

- BL[p].primitiveReference: The ***reference*** to the primitive's description, which includes its type, parameters, and associated transformation
- BL[p].sign: The ***sign*** (binary value) indicating, when set, that the result of classifying a cell against the primitive should be complemented
- BL[p].name: The ***name*** associated with the primitive (several primitives may have the same name and many primitives have no name)
- BL[p].stamp: The ***stamp***, which contains the name of the next primitive in the list that should classify the cell that are inside the current primitive if its sign is positive, or the cells that are outside of the current primitive, if its sign is negative

CSG-to-Blist conversion

Although the CSG-to-Blist conversion may be performed more efficiently by combining the steps proposed below, we have separated the steps here for sake of simplicity. The CSG-to-Blist conversion process takes as input the root-node of the binary tree, T, and produces the corresponding BL table. Both structures have been described above. The conversion performs the following steps:

1. Convert T into a positive form by applying deMorgan's laws and propagating complements to the leaves
2. Rotate the tree by switching the left and right children at each node to make the tree left heavy
3. Visit the leaves from left to right and for each leaf, p, fill in the corresponding fields of BL[p]

For unbalanced trees, Step 2 may reduce the total number of bits needed for each label to less than $\lceil \log(H+1) \rceil$.

We describe the details of these steps below using as example T= (A∩(B∪C)) – ((D∪(E–F))∩G). The literals, A, B…G, denote integer primitive references. Parsing this expression yields a binary tree shown Fig. 2 (left).

We first convert that tree to its positive form (as in [Rossignac89] and in [Goldfeather88]). This conversion process traverses the tree top-down and applies the deMorgan transformations: A–B→A∩B', (A∩B)'→A'∪B', (A∩B)'→A'∪B', and (A')'→A, where X' denotes the complement of set X. The result (Fig.2, right) is a tree with the same structure and no difference operators. Note that some of its leaves (indicated by an apostrophe) have been negated, i.e., replaced by they complement.

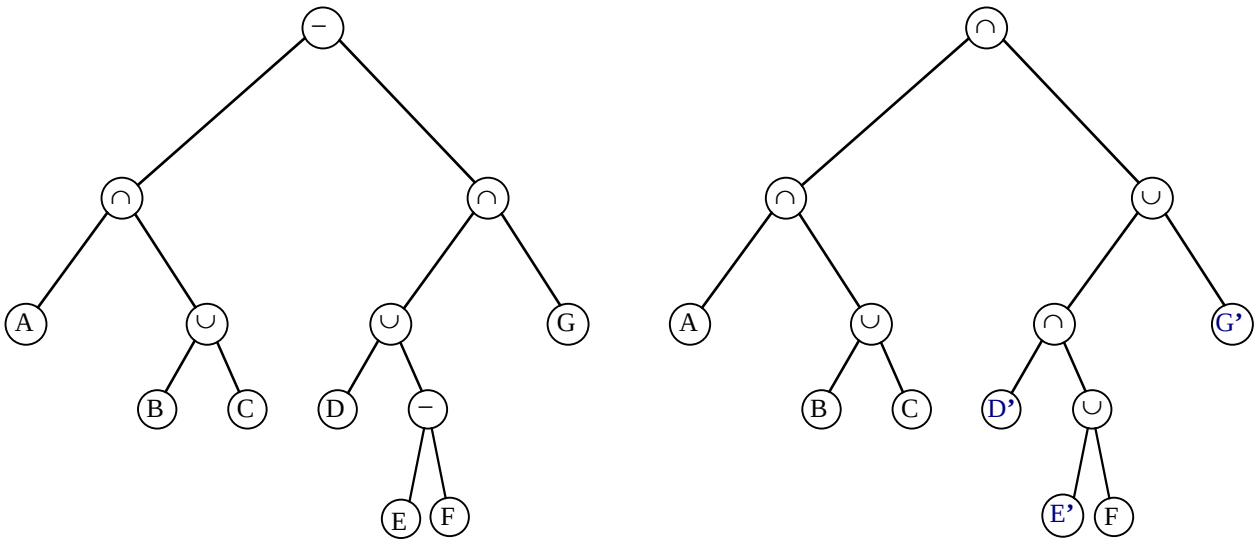

Figure 2: *The binary tree corresponding to the Boolean expression: (A∩(B∪C)) – ((D∪(E–F))∩G) is shown left. Its positive form is shown right. Note the complemented primitives are indicated using apostrophes and a blue color.*

Then, we exploit the fact that both ∪ and ∩ are commutative (A∪B=B∪A and A∩B=B∩A) to switch the left and right child of nodes when the right child is a higher sub-tree (i.e. has a superior maximum path length from its root to its leaves). The switch is performed during a recursive traversal of the positive tree, switching first the lower-level nodes and reporting their height to the parent node, before we consider switching the parent node. The result is illustrated Fig. 3 (left).

Finally, we insert the resulting tree, T, as the left-most leaf of a two level tree: (T ∪ OUT) ∩ IN, as shown Fig. 3 (right) and we traverse the new tree and assign consecutive integer ID's, P.primitiveID, to each leaf, P, in left-to-right order.

Now we are ready for the final phase (Fig. 4), which fills in the Blist table BL. During that phase, we initialize the content of BL to zero and, once more, traverse the rotated positive version of the tree T recursively. At each leaf, P, we invoke the procedure *Match*(P,BL) illustrated by the pseudo-code below.

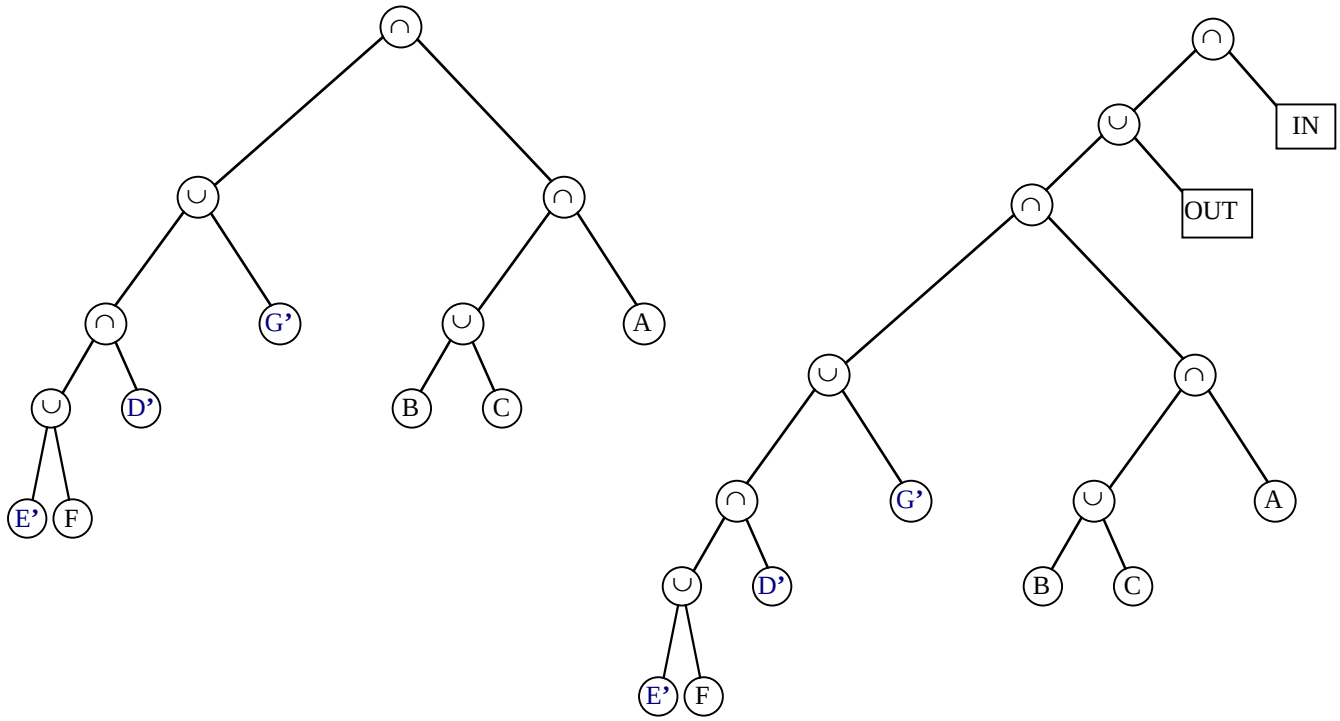

Figure 3: *The binary tree of Fig. 2 has been rotated to make it left heavy (left). The result is inserted as the left-most leaf into a small tree with special IN and OUT nodes marked by rectangles (right).*

```
        Procedure Match(P,BL)
(a)         p := P.primitiveID;
(a)         IF BL[p].name ≠ 0 THEN releaseIntegerName(BL[p].name);
(c)         BL[p].sign := P.sign;
(d)         BL[p].primitiveReference := P.primitiveReference;
(e)         M := P;
(f)         WHILE M ≠ M.parent.leftChild DO M := M.parent;
(g)         op := M.parent.operator;
(h)         IF  op = "∩" THEN BL[p].sign := NOT BL[p].sign;
(i)         M := M.parent;
(j)         WHILE (M ≠ M.parent.leftChild) OR (M.parent.operator = op) DO M := M.parent;
(k)         M := M.parent.rightChild;
(l)         WHILE M.type ≠ leaf DO M := M.leftChild;
(m)         m := M.primitiveID;
(n)         IF BL[m].name = 0 THEN BL[m].name := lockLowestAvailbleIntegerName;
(o)         BL[p].stamp := BL[m].name;
```

Procedure *Match* is based on the following observation. Consider a sub-tree S=((P∪B)∪C)∩((M∪E)∩F) in T. If we are classifying the cell c against S and discover that it is outside of primitive P, then we must classify it against primitive B, which is the next one in the Blist representation. Therefore, marking the cell with a zero label will indicate to the next primitive, B, that it should process the cell. If, however, we discover that c ⊂ P, we can skip primitives (or sub-trees) B and C and go directly to M, which is the left-most leaf of the sub-tree ((M∪E)∩F). Consequently, we must write on the label of c the name used by M.

If M does not yet have a name, we use the procedure *lockLowestAvailbleIntegerName (line n)* to obtain the lowest strictly positive integer that is not yet in use as the name of any primitive coming after P in the Blist table. Note that when later we reach M, that

integer is released using procedure *releaseIntegerName* (line b), so that it can be used as a name for another primitive that comes after M. The same name is often used multiple times. This strategy helps reduce the maximum number of bits needed for labels.

Given the current primitive, P, we locate its match, M, by moving up the tree (line f), until we reach the left child of a node N_1. We detect this situation because it is the first time that M is the left child of its parent. We save the operator of N_1 in the variable: op. In the above example of the sub-tree S, the operator for N_1 is $\cup$.

However, if it was $\cap$, as for example in the sub-tree $S=((P\cap B)\cap C)\cup((M\cup E)\cap F)$, we would jump to M, only if $c \not\subset P$, or would continue to the next primitive, if $c \subset P$. To distinguish between these two situations, we toggle the sign associated with P (line h), when the variable op is "$\cap$".

Then, we keep walking up the tree, until we reach a node, N_2, that is the left-child of a node, whose operator is different from op (line j). The desired "match" leaf, M, is the left-most leaf in the sub-tree that is the right child of N_2. We move to the right child first (line k), and then walk down the tree, always turning left (line l). M is the leaf where this journey ends.

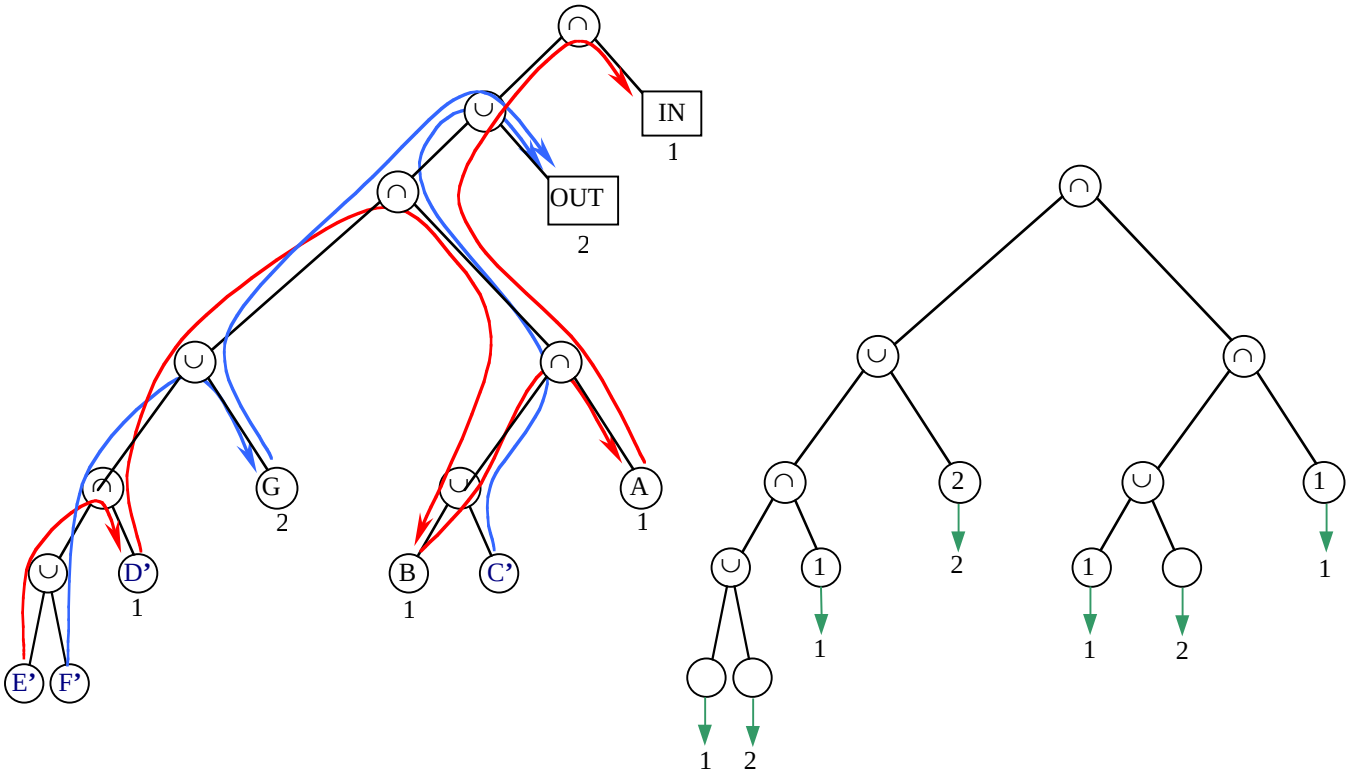

Figure 4: *The leaves of right tree of Fig. 3 are visited in the left-to-right order. We number the leaves with increasing strictly positive integers. For each leaf, numbered p, we compute its match, M, indicated by the red and blue arrows (left). Note that each arrow first traces a path upwards, first reaching a node N_1 coming from its left child, then reaching a node N_2 with a different operator still coming from the left child. If N_2.operator=$\cup$, the sign of p, Blist[p].sign, is inverted (see leaves, where blue lines start in the left figure). Then, the arrows follow the pointer to N_2.rightChild and then take the left-most child at each internal node, until they reach the matching leaf M.. Then, if M does not yet have a name, it grab the lowest available strictly positive integer and use it as its name. We also store that name as the stamp, Blist[p].stamp, of p. On the right, we show the resulting names for the leaves (inside the circles) and their stamps (under the green arrows). Note that three leaves do not have a name and that only two names were needed here for the entire tree.*

If the name attributed to the IN node was not 1, but x, we simply switch all uses of x'es and 1's in the names and stamps stored in BL, so as to follow the convention that all cells marked 1 at the end of the classification process are in.

The Blist table resulting from the conversion process for the example of Fig. 4 is:

p	BL[p].name	BL[p].sign	BL[p].PrimitiveReference	BL[p].stamp
1	0	−	E	1
2	0	−	F	2
3	1	−	D	1

4	2	+	G	2
5	1	+	B	1
6	0	−	C	2
7	1	+	A	1

Cell classification using Blist

During set membership classification, a ***label*** is attached to each cell and passed to the successive primitives in the Blist. When the label matches the primitive's name, the cell is classified against the primitive. If the result of this classification matches the primitive's sign, the name on the primitive's stamp is put on the label—if not, a zero name is put on the label of the cell.

The following procedure describes how a single cell, c, is classified against a CSG model represented by a Blist array BL. |P| defines the total number of primitives in BL.

```
PROCEDURE ClassifyAgainstBlist(c,BL)
        c.label := 0;
        FOR p := 1 TO |P| DO
                IF (c.label = 0) OR (c.label = BL[p].name)
                        THEN IF InPrimitive(BL[p].primitiveReference , c) = BL[p].sign
                                THEN c.label := BL[p].stamp
                                ELSE c.label := 0;
```

At the end of this process, if the label on the cell is 1 the cell is inside the CSG solid. Otherwise, it is out.

To illustrate how this classification works, consider for example a cell whose classification against the sequence of primitives, ABCDEFG, yields the following pattern of bits: 1010101, where a 1 (short for true) indicates that the cell lies inside the corresponding primitive. Classifying the cell against the CSG tree (A∩(B∪C))−((D∪(E−F))∩G) yields: (1 AND (0 OR 1)) AND NOT ((0 OR (1 AND NOT 0)) AND 1), which evaluates to 0 and indicates that the cell is out.

We describe below how the Blist table, produced by CSG-to-Blist conversion algorithm for this example, is used to classify the cell. To help the reader follow the process, we have appended two columns to the table:

- *Classification*, which indicates whether the cell lies inside (1) or outside (0) of the primitive.
- *Label*, which indicates the content of the after the primitive has processed the cell

A "*" after the primitive's name indicates that it matches the incoming label.

p	BL[p].name	BL[p].sign	BL[p].primitiveReference	BL[p].stamp	Classification	Label
1	0*	−	E	1	1	0
2	0*	−	F	2	0	2
3	1	−	D	1	0	2
4	2*	+	G	2	1	2
5	1	+	B	1	0	2
6	0	−	C	2	1	2
7	1	+	A	1	1	2

Originally, the label was 0. The cell lies inside the first primitive (E), but the primitive is inverted in BL (it was a negative primitive in the positive formulation of T), consequently, we do not jump, but pass the cell to the next primitive in the list. To do this, we set the label to 0. The cell is out of the second primitive (F) and since the primitive is inverted (BL[2].sign = "–"), we set the label to contain BL[2].stamp, which is 2. We skip primitive 3, because its name does not match the label. At primitive 4, we have a match between the label and the name, and the classification of the cell against G yields 1. We write the content of BL[4].stamp into the label, which is again set to 2. We skip primitives 5, 6, and 7, because their name is not 2. At the end of this process, the label has value 2, which indicates that the cell is out of the CSG model.

To classify a set C of cells, we propose a simple extension of the above procedure:

```
PROCEDURE ClassifyCellsAgainstBlist(C,BL)
        FOREACH cell c IN C DO c.label := 0;
        FOR p := 1 TO |P| DO
                FOREACH cell c IN C DO
```

```
IF (c.name = 0) OR (c.label = BL[p].name)
        THEN IF InPrimitive(BL[p].primitiveReference , c) = BL[p].sign
                THEN c.labe := BL[p].stamp
                ELSE c.label := 0;
```

Conclusions and future work

We have introduced a new representation for CSG trees—and in fact for arbitrary Boolean expressions. This representation, called Blist, may be evaluated very efficiently by updating a label, when its value matches the primitive's name. Deciding on how to update the label requires only one comparison. Consequently, a Boolean expression may be evaluated one primitive at a time. The combining steps used with the traditional recursive evaluation are not necessary.

Because the label requires at most $\lceil \log(H+1) \rceil$ bits, where H is the height of the tree, the use of a Blist formulation reduces the storage necessary for evaluating large collections of cells in parallel architectures (for example in SIMD or pipelines) popular in graphics.

We have provide the details of simple algorithms for converting binary CSG trees into a Blist format and for classifying one or several cells against this new representation.

We expect the Blist model to be useful in many settings and plan to investigate its applications to a large class of solid modeling and graphic algorithms that operate on CSG models.

Furthermore, we plan to investigate extensions of Blist formulations to constructive models for a broader class of topological domains [Rossignac91, Rossignac97].

Acknowledgements

The author thanks Ying Chen from Georgia Tech for providing an implementation of this CSG-to-Blist conversion algorithm and of a prototype Blist-based rendering environment for discretized 3D CSG models.

References

[Alexandroff61] P. Alexandroff, Elementary Concepts of Topology, Dover Publications, New York, NY, 1961.

[Banerjee96] R. Banerjee and J. Rossignac, Topologically exact evaluation of polyhedra defined in CSG with loose primitives, to appear in Computers Graphics Forum, Vol. 15, No. 4, pp. 205-217, 1996.

[Bronsvoort84] W. Bronsvoort, J. Van Wijk, and F. Jansen, Methods for Improving the Efficiency of Ray Casting in Solid Modelling, Computer-Aided Design, Vol. 16, No. 1, pp. 51-55, 1984.

[Ellis91] J Ellis, G. Kedem, G. T. Lyerly, D. Thielman, R. Marisa, J. Menon, The Ray Casting Engine and ray representations, ACM Symp. on Solid Modeling Foundations and CAD/CAM Applic., ACM Press, Austin, TX, pp. 255-268, June 5-7, 1991.

[Fuchs82] H. Fuchs, J. Pulton, et al. Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System, Proc. 82 Conf. on Advanced VLSI, M.I.T., 1982.

[Goldfeather86] J. Goldfeather, J. Hultquist, and H. Fuchs,. Fast Constructive Solid Geometry Display in the Pixel-Power Graphics System, ACM SIGGRAPH '86 Proc., Computer Graphics, Vol. 20, No.4, August 1986.

[Goldfeather88] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning, IEEE Computer Graphics and Applications, Vol. 9, No. 3, pp. 20-28, May 1989.

[Hoffmann89] C. Hoffmann, *Geometric and Solid Modeling: An introduction*, Morgan Kaufmann, San Mateo, CA, 1989.

[Jansen86] F. Jansen, A Pixel-parallel Hidden Surface Algorithm for Constructive Solid Geometry, Proc. Eurograhpics '86, Ed. A. Requicha, Elsevier Science, North-Holland, Amsterdam, pp. 29-40, 1986

[Jansen87] F. Jansen, Display of Solid Models with a Multi-Processor System, Proc. Eurographics '87, Elseviers Science Publishers, pp. 377-387, Amsterdam, August 1987.

[Kedem84] G. Kedem and J. Ellis, Computer Structures for Curve-solid Classification in Geometric Modelling, Production Automation Project, Univ. Rochester, Tech. Memo 51, May 1984.

[Kedem84b] G. Kedem and J. Ellis, The Ray-casting Machine, Proc. ICCCD, pp. 533-538, October 1984.

[Kedem88] G. Kedem and J. Ellis, The Ray-casting Machine Prototype, International Conf. on Parallel Processing for Computer Vision and Display, University of Leeds, UK, January 1988.

[Mantyla86] M. Mantyla, Boolean Operations of 2-manifold Through Vertex Neighborhood Classification, ACM Trans. on Graphics, 5(1):1-29, January 1986.

[Mantyla88] M. Mantyla, *An introduction to Solid Modeling.* Computer Science Press, 1988.

[Meagher84] D. Meagher, The Solids Engine: A Processor for Interactive Solid Modeling, Proc. of Nicograph '84, Tokyo, Japan, November 1984.

[Morris85] D. Morris, An Algorithm for Direct Display of CSG Objects by Spatial Subdivision, in *Fundamental Algorithms for Computer Graphics*, Ed. R.A. Earnshaw, Springer-Verlag, Berlin, pp. 725-736, 1985.

[Naylor86] B. Naylor and W. Thibault, Application of B.S.P. Trees to Ray Tracing and CSG Evaluation, Technical Report GIT-ICS 86/03, Georgia Institute of Technology, Atlanta, GA. February 1986.

[**Naylor90**] B. Naylor, J. Amanatides, W. Thibault. Merging BSP trees yields polyhedral set operations. In Proc. ACM Siggraph'90, *Computer Graphics*, vol. 24, pp. 115-124, 1990.

[Rappoport97] A. Rappoport and S. Spitz, Interactive Boolean Operations for Conceptual Design of 3-D Solids, ACM Computer Graphics, Proceedings Siggraph, pp. 269-278, July 1997.

[Requicha80] Representation of Rigid Solids: Theory, Methods, and Systems, A.A.G. Requicha, ACM Computing Surveys, 12(4), 437:464, Dec 1980.

[Requicha85] A. Requicha and H. Voelcker, Boolean operations in solid modeling: boundary evaluation and merging algorithms, Proc. IEEE, Vol. 73, No. 1, pp. 30-44, January 1985.

[Rossignac86] J. Rossignac and A. Requicha, Depth Buffering Display Techniques for Constructive Solid Geometry, IEEE Computer Graphics and Applications, Vol. 6, No. 9, pp. 29-39, September 1986.

[Rossignac89] J. Rossignac and H. Voelcker, Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection and Shading Algorithms, ACM Transactions on Graphics, Vol. 8, pp. 51-87, 1989.

[Rossignac89b] J. Rossignac and M. O'Connor, SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries, in *Geometric Modeling for Product Engineering*, Eds. M. Wosny, J. Turner, K. Preiss, North-Holland, pp. 145-180, 1989.

[Rossignac90] J. Rossignac and J. Wu, Shading of Regularized CSG Solids Using a Depth-Interval Buffer, in *Advances in Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, Springer-Verlag, Eurographics Seminars, Eds. R. Grimsdale and A. Kaufman, Berlin, pp.117-138, 1990.

[Rossignac91] J. Rossignac, and A. Requicha, Constructive Non-Regularized Geometry, Computer-Aided Design, Vol. 23, No. 1, pp. 21-32, Jan./Feb. 1991.

[Rossignac92] J. Rossignac, A. Megahed, and B.O. Schneider, Interactive Inspection of Solids: Cross-sections and interferences, ACM Computer Graphics, Proc. SIGGRAPH'92, Vol. 26, No. 4, 1992.

[Rossignac92b] J. Rossignac and J. Wu, Correct Shading of Regularized CSG solids using a Depth-Interval Buffer, *In Advances in Computer Graphics Hardware V*, Eds. R.L. Grimsdale and A. Kaufman, Springer Verlag, pp. 117-138. 1992.

[Rossignac94] J. Rossignac, Through the cracks of the solid modeling milestone, *From Object Modelling to Advanced Visual Communication*, Eds. Coquillart, Strasser, Stucki, Springer-Verlag, pp. 1-75, 1994.

[Rossignac94b] J. Rossignac, Processing Disjunctive forms directly from CSG graphs, in *CSG 94: Set-theoretic Solid Modelling Techniques and Applications*, Information Geometers, pp. 55-70, Winchester, UK, April 1994.

[Rossignac96] J. Rossignac, CSG Formulations for Identifying and for Trimming Faces of CSG Models, CSG 96: Set-theoretic Solid Modelling Techniques and Applications, Information Geometers, Ed. J. Woodwark, Winchester, UK, April 17-19, 1994.

[Rossignac97] J. Rossignac, Structured Topological Complexes: A feature-based API for non-manifold topologies, ACM Press, C. Hoffman and W. Bronsvort, Edts., pp. 1-9, 1997.

[Schneider92] B.O. Schneider and J. Rossignac, M-Buffer: A flexible MISD Architecture for Advanced Graphics, Proc. 7th Workshop on Computer Graphics Hardware, Cambridge, UK, September 1992.

[Soto85] H. Soto, M. Ishii, K. Sato, and M. Ikesaka, Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor, ACM Computer Graphics, Proc. Siggraph '85, Vol. 19, No. 3, pp. 95-102, July 1985.

[Thibault87] W. Thibault and B. Naylor, Set operations on polyhedra using binary space partition trees. Proc. ACM Siggraph'87, Computer Graphics, vol. 21, pp. 153-162, 1987.

[Tilove80] R. Tilove, Set Membership Classification: A Unified Approach to Geometric Intersection Problems, IEEE Trans. on Computers, vol. C-29, No. 10, pp. 874-883, October 1980.

[Woodwark82] J. Woodwark and K. Quinlan, Reducing the Effect of complexity on Volume Model Evaluation, Computer-Aided Design, Vol. 14, No. 2, pp. 89-95, March, 1982.