Solid Modeling

Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning

Jack Goldfeather, Carleton College

Steven Molnar, Greg Turk, and Henry Fuchs
University of North Carolina

In this article we describe a set of algorithms for efficiently rendering a CSG-defined object directly into a frame buffer without converting first to a boundary representation. This method requires only that the frame buffer contain sufficient memory to hold two color values, two depth values, and three one-bit flags. The algorithm first converts the CSG tree to a normalized form that is analogous to the sum-of-products form for Boolean switching functions. Expanding on earlier results,[1] we develop the following:

1. The dynamic interleaving of Boolean tree normalization with bounding-box pruning, allowing efficient rendering for most CSG objects.
2. A method for extending the technique to nonconvex primitives.
3. Implementation of these ideas in an interactive CSG design system on Pixel-planes 4.

In this system the designer directly manipulates the CSG structure while continuously viewing the color rendering of the object being designed. We believe that our algorithms will attain similar speeds on many of the next-generation high-performance graphics systems that have frame buffers with many bits per pixel.

A serious drawback for designers using constructive solid geometry (CSG) is the inability to create and modify shaded CSG objects in an interactive environment. Recent advances in graphics hardware have made it possible to achieve near real-time rendering of shaded CSG objects by taking advantage of the hardware's parallelism. Attention has turned to adapting some traditional rendering algorithms to this parallel environment.

Conversion of a CSG description to a boundary representation (B-rep) allows rapid display of the boundary polygons using conventional rendering engines. However, the B-rep must be recomputed every time the object is modified. Thibault and Naylor[2] demonstrated dynamically rendered CSG images based on B-reps with a recent system using binary-space partitioning trees, but their system can handle only a limited set of modifications without incurring many seconds of delay. For example, moving a beveled hole would be time consuming.

Algorithms such as Atherton's CSG scan-line algorithm,[3] which displays directly from the CSG description, are slow when implemented on a conventional machine, but can be sped up using object-parallel hardware. For example, Kedem and Ellis currently are build-

0272-17-46/89/0500-0020$01.00

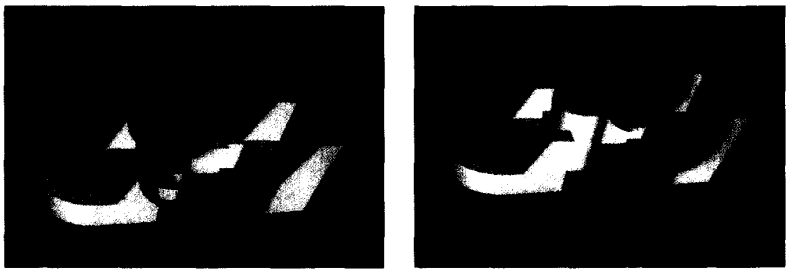

Figure 1. Consecutive frames during an editing session (0.142 seconds per frame).

ing a parallel ray-casting machine for fast scan-line rendering.[4] Their machine allocates one processor per node in the CSG tree. An arbitrary CSG tree with fewer nodes than the number of hardware processors can be rendered directly, and larger trees may be rendered using multiple passes.

Depth-buffering algorithms involve generalizations of the z-buffer hidden-surface algorithm. Speed can be gained using pixel-parallelism in special-purpose hardware. Rossignac and Requicha published an algorithm for directly rendering CSG objects using point-classification and z-buffering.[5] Jansen published a similar algorithm,[6] specifically designed for pixel-parallel machines like Pixel-planes.[7] In this algorithm a number of flags ($\log_2 n$ bits, where n is the number of primitives) are used to hold intermediate point-classification results while traversing the tree. Our approach is a depth-buffering algorithm which rearranges and prunes the CSG tree before rendering, eliminating the need for these "place-holding" bits. Okino, Kakazu, and Morimoto published a paper on depth-buffering algorithms,[8] which also may employ tree rearrangement, but we have been unable to verify this.

The algorithm we present here will render any CSG object using a constant number of bits per pixel (≤ 128). Our rendering system, based on this algorithm, displays modestly sized CSG objects in fractions of a second, even if the user changes the geometric structure of the CSG tree each and every frame (see Figure 1).

CSG rendering using the normalization method

In this section we outline a method for transforming any CSG tree into a "normalized" form. In the worst case, such a transformation can create a combinatorial explosion in the number of nodes, but in a later section we will show pruning techniques that keep the node growth under control.

Normalizing a CSG tree is analogous to converting a Boolean expression to a sum-of-products form. One reason for converting Boolean expressions into this form is to enable fast combinatorial logic with only two gate delays. In an analogous manner, a CSG expression written as a sum of products can be rendered using two image (z and color) buffer pairs. This enables any CSG object composed of convex primitives to be rendered in a frame buffer with a constant number of bits per pixel.

A CSG tree is a Boolean expression: each union ($\cup$) is a sum, each intersection ($\cap$) is a product, and each difference ($-$) is a product of a complement. The primitives in a CSG tree correspond to literals in the Boolean expression. For example, the CSG tree $((A-B)\cup C)\cap D$ can be written as $(AB'+C)D$. In particular, we will define a normalized CSG tree in terms of well-known Boolean constructs. We say a CSG tree is in normal form if its Boolean representation is in disjunctive normal form, that is, if it is a sum of products of literals and complements of literals.

In addition to allowing rendering with a constant number of bits per pixel, normalizing a CSG tree allows the rendering algorithm to be simpler than it would be otherwise. Each product in the normalized expression can be rendered using primitive/primitive interaction rather than subtree/subtree interaction. In a later section we will see that normalization also allows unnecessary portions of the CSG tree to be recognized and pruned easily.

The normalization algorithm

1. $X-(Y\cup Z) = (X-Y)-Z$
2. $X\cap(Y\cup Z) = (X\cap Y)\cup(X\cap Z)$
3. $X-(Y\cap Z) = (X-Y)\cup(X-Z)$
4. $X\cap(Y\cap Z) = (X\cap Y)\cap Z$
5. $X-(Y-Z) = (X-Y)\cup(X\cap Z)$
6. $X\cap(Y-Z) = (X\cap Y)-Z$
7. $(X\cup Y)-Z = (X-Z)\cup(Y-Z)$
8. $(X\cup Y)\cap Z = (X\cap Z)\cup(Y\cap Z)$

Figure 2. Set equivalences for normalization.

The normalization algorithm introduced earlier[1] uses the eight basic set equivalences of Figure 2 to reduce a CSG tree to normal form. These equivalences encapsulate the associative and distributive properties of set operations and were chosen because they represent all of the possible unnormalized configurations at a single node. Equivalences 2, 3, 5, 7, and 8 decompose an expression into sums, and the remaining equivalences replace

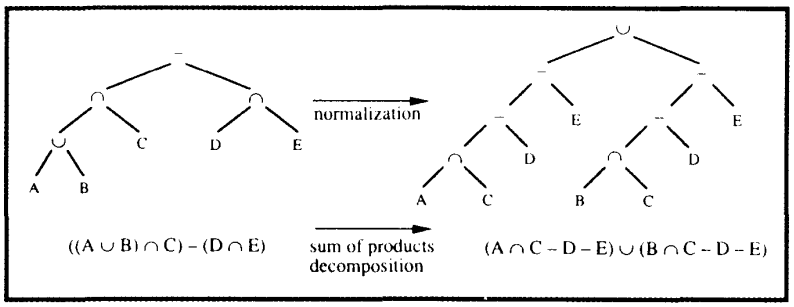

Figure 3. CSG tree before and after normalization.

right association with left association. The normalization algorithm can be written recursively as follows:

```
procedure normalize(T : tree);
{ reduce a CSG tree to sum-of-products form }
begin
    if T = PRIMITIVE then return;
    repeat
        while T matches left side of any set equivalence
            replace with right side, using
            equivalences 1 – 6 before 7 or 8;
        normalize(T.left);
    until T = ∪ or (T.right = PRIMITIVE and T.left ≠ ∪)
    normalize(T.right);
end normalize;
```

This version corrects an error in an earlier presentation of the algorithm.[1] Figure 3 shows a sample CSG tree before and after normalization. The algorithm above is not the only possible normalization algorithm, since there are many ways to convert a Boolean expression to disjunctive normal form. We have proved, however, that *normalize* has the following desirable properties:

1. It terminates given any CSG tree as input.
2. Upon termination, it leaves the CSG in normal form.
3. Each restructuring step requires only local information (node type and child node types).
4. If the initial tree contains no redundant subtrees or repeated primitives, *normalize* will not add redundant product terms or repeat primitives within a product.

Property 3 follows directly from the set equivalences. Formal proofs for properties 1, 2, and 4 are not included here due to lack of space.[9]

Displaying a CSG tree in normal form

Rendering a CSG tree in normal form requires sufficient memory at each pixel for two z/color buffer pairs, (*ztemp,ctemp*) and (*zfinal,cfinal*), and three one-bit flags. We will describe the process for convex primitive solids now and refer the reader to a later section for an extension of the rendering algorithm to nonconvex solids. The basic idea is to break each product into separate terms, render each term into the (*ztemp,ctemp*) image buffer, then composite the terms into the (*zfinal,cfinal*) image buffer using a standard z-buffer algorithm. A product with *n* primitives will contain *n* terms. Rendering each term consists of rendering the surface of a primitive and trimming it by the remaining $n-1$ primitives in the product.

The following algorithm performs a standard in/out classification[10] of points on primitive boundaries to render a product of primitives. A primitive is *complemented* if the corresponding literal in the normalized Boolean expression is complemented. *Zfar* is a constant equal to the largest value representable in the z-buffers. The front surface of a primitive refers to all of the points on the exterior of a primitive that are visible from the eye point. In our case this is simply the set of front-facing polygons.

```
procedure front(P : primitive);
{ place front surface of P into (ztemp,ctemp) }

procedure back(P : primitive);
{ place back surface of P into (ztemp,ctemp) }

procedure intersect(P : primitive);
{ if ztemp not between front and back surfaces of P }
{ then set ztemp = zfar }

procedure subtract(P : primitive);
{ if ztemp between front and back surfaces of P }
{ then set ztemp = zfar }

procedure render(M : product);
{ render a product into the final image buffer }
begin
    for each primitive P in M do begin
        if P is uncomplemented then front(P)
                              else back(P);
        for each primitive Q (Q ≠ P) in M
            if Q is uncomplemented then intersect(Q)
                                   else subtract(Q);
        composite (ztemp,ctemp) into (zfinal,cfinal);
    end;
end render;
```

Figure 4 illustrates the entire rendering process for the

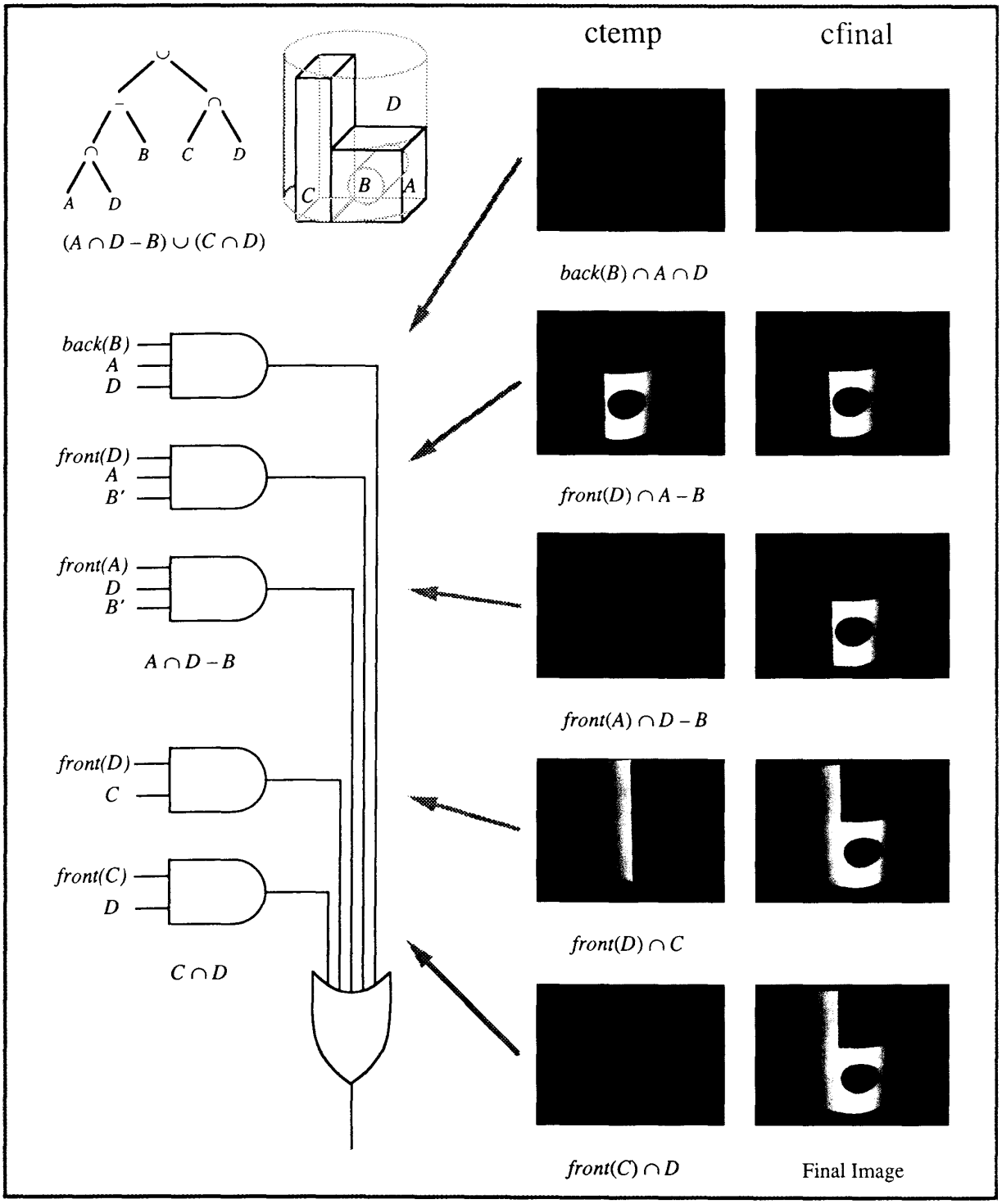

Figure 4. Rendering a CSG tree.

CSG tree $((A-B)\cup C)\cap D$. The photographs illustrate the contents of the temporary and final image buffers at each stage in the rendering process.

Geometric pruning

Normalization can add many primitive leaf nodes to a CSG tree; for certain trees the increase can be dramatic

	Object Description		Number of primitives			Faces sent to frame buffer		Average product length (k)	Rendering time in seconds
	Name	Source	user tree	normalized tree	after pruning	no diff pruning	with diff pruning		
1.	Tube	Okino[1]	11	15	15	75	75	3.0	0.19
2.	Cut Tube	Okino[1]	12	20	20	140	136	4.0	0.24
3.	MBB	Okino[1]	23	30	30	150	150	2.0	0.14
4.	Tie Rod	BRL[2]	19	19	19	127	91	3.2	0.14
5.	Phone	Randy Brown	32	62	31	852	456	4.6	0.11
6.	Joystick	Randy Brown	45	75	47	151	139	1.7	0.82
7.	Die	Greg Turk	44	44	44	380	252	1.9	4.02
8.	GEB[3]	Steven Molnar	29	297	164	2252	2036	8.2	0.71
9.	Enterprise	BRL[2]	165	165	164	884	820	2.4	6.46
10.	Truck	BRL[2]	180	180	180	352	352	1.3	2.77
11.	Rod - MBB	(above)	42	69120	96	1152	928	6.1	0.59

[1] Okino, Kakazu and Morimoto (see references)
[2] US Army Ballistic Research Laboratory
[3] inspired by cover of Godel, Escher, Bach, by Douglas Hofstadter

Figure 5. Rendering times and statistics.

(sample object 11 in Figure 5 expanded from 42 primitives before normalization to 69,120 primitives after normalization). However, in most cases, large subtrees in the normalized tree will not contribute to the final image, since primitives within them may not intersect. Information about which primitives overlap provides us with "don't care" conditions similar to those used to minimize logic expressions.

Bounding boxes

A standard approach to pruning a CSG tree is to use bounding boxes.[6,12] We review this technique and, in addition, describe how the bounding-box pruning process can be integrated with tree normalization to minimize tree growth.

The idea behind bounding-box pruning is to calculate a few numbers that bound the region occupied by a CSG object. Only if the bounding boxes intersect do we need to compute the actual intersection between objects. Similarly for subtraction, if the bounding box of an object to be subtracted does not intersect the bounding box of the object from which it will be subtracted, the subtraction need not be computed. Entire subtrees can be eliminated as well as primitives, since each subtree has a bounding box derived from the subtrees below it.

To compute bounding boxes for primitive nodes, we must first transform them into a common coordinate system. We then compute a bounding box for each primitive by finding the minimum and maximum x, y, and z coordinates of the vertex set. *Min(x,y,z)* and *Max(x,y,z)* define the corners of a box (aligned with the coordinate axes) that completely encloses the primitive.

We compute bounding boxes for an operator node from the bounding boxes of its children using the following rules:

1. $\text{Bound}(A \cup B) = \text{Bound}(\text{Bound}(A) \cup \text{Bound}(B))$
2. $\text{Bound}(A \cap B) = \text{Bound}(A) \cap \text{Bound}(B)$
3. $\text{Bound}(A - B) = \text{Bound}(A)$

Intersection and difference nodes can potentially be pruned. If the operator is $\cap$, we compare *Min(x,y,z)* with *Max(x,y,z)*. If the minimum in any coordinate exceeds the maximum, the bounding box describes a null volume and we prune the subtree $A \cap B$. If the operator is $-$, we compare Bound(A) with Bound(B). If they do not intersect, we replace the subtree $A - B$ with A. We can prune subtrees that do not intersect the viewing frustum as well.

Pruning and normalization

Normalization, even though tending to increase the number of tree nodes, separates large subtrees (with large bounding boxes) into a set of products (generally with smaller bounding boxes). This makes pruning very effective on normalized trees, in many cases reducing the number of primitive leaf nodes by orders of magnitude. For example, geometric pruning reduced the number of primitives from 69,120 to 96 in sample object 11 of Figure 5.

An even more effective approach is to prune the tree

before it ever reaches its maximum size. We can do this by pruning during normalization rather than after. Integrating pruning into the tree-normalization process restricts the growth of the normalized tree. This integration is achieved by computing new bounding boxes every time a set equivalence is applied or after normalizing a left or right subtree. The pruning operation is fast, since it depends only on bounding-box information for at most two nodes.

The dynamic pruning process can be made even more effective by adding one more set equivalence to the normalization algorithm:

6.5 $(X - Y) \cap Z = (X \cap Z) - Y$

This transformation forces intersections to the left in a product. Since intersection can reduce the size of bounding boxes, but subtraction and union cannot, this allows subtracted subtrees to be pruned which might not be pruned otherwise. It also allows null subtrees to be detected as early as possible, rather than having the normalization algorithm manipulate them many times before pruning. For a subtree of the form given in equivalence 6.5, if the bounding boxes for Y and Z are disjoint, then the form of the tree on the left will not result in pruning Y. Transforming the tree into the form on the right will result in having Y pruned from the tree when the bounding box for Y is compared with that of $X \cap Z$.

Recall that the rendering process for products with n primitives requires that each primitive boundary (front or back) be trimmed by all of the other primitives in the product. Once the dynamic pruning process is complete, a further optimization, called *difference pruning*, can simplify some of the terms to be rendered. This is achieved by comparing the bounding boxes of each subtractive primitive within the product. When the bounding boxes of two subtractive primitives are disjoint, certain of the terms to be rendered simplify. For example, when rendering $A - B - C$, if the bounding boxes of B and C do not intersect, the term $back(B) \cap A - C$ is equivalent to $back(B) \cap A$. Difference pruning results in a significant speedup for product terms with disjoint subtraction primitives, since trimming a primitive with another requires comparisons with both front and back surfaces of the subtractive primitive.

From here on we will refer to the process outlined above as the "normalization and geometric pruning" algorithm (NGP).

Rendering nonconvex primitives

We say that a solid is k-convex if a ray intersecting the solid can enter and exit it at most k times. A 1-convex solid is convex in the usual sense and 2-convex solids include primitives such as the torus and the helix primitive (that is, one turn of a coil spring).

To render a k-convex primitive, its boundary surface must be divided into entirely front-facing and back-facing subsurfaces. Planar polygons satisfy this condition. We assume primitives are tiled with polygons in the algorithm below.

The *front* and *back* routines now must send the polygons of a k-convex primitive to the frame buffer k times. In each pass we retain a portion of the primitive's surface, perform in/out classification with respect to the other primitives in the product term, and then composite what remains into the final image buffer. We use a small buffer (of size $\log_2 k$) called *count* in the frame buffer to ensure that all potentially visible points on the primitive are captured in at least one of the passes. If a surface point is captured more than once, no harm will be done. The following two algorithms are called within loops inside *front* and *back*, respectively, and perform one front- or back-surface pass:

```
procedure front_pass(P : k-convex primitive, n : integer);
{ called by front k times for }
{ uncomplemented k-convex primitives }
begin
    count := 0;
    for each front-facing polygon of P do
        for each pixel in polygon do begin
            count := count + 1;
            if count = n then
                scan polygon into (ztemp,ctemp);
        end;
end front_pass;

procedure back_pass(P : k-convex primitive, n : integer);
{ same as front_pass, but for }
{ complemented k-convex primitives }
```

A 1-bit parity flag at each pixel is used to perform in/out classification on a k-convex primitive:

```
procedure in_out_classify(P : k-convex primitive);
{ toggle parity each time a surface }
{ is encountered with z < ztemp }
begin
    parity := 0;
    for each polygon of P do
        for each pixel in polygon do
            if z of polygon < ztemp then toggle parity;
end in_out_classify;

procedure intersect(P : k-convex primitive);
{ trim a term using an }
{ uncomplemented k-convex primitive }
begin
    in_out_classify(P);
    for each pixel do
        if parity = 0 then ztemp := zfar;
end intersect;
```

```
procedure subtract(P : k-convex primitive);
{ trim a term using a complemented k-convex primitive }
begin
    in_out_classify(P);
    for each pixel do
        if parity = 1 then ztemp := zfar;
end subtract;
```

The procedures *front_pass* and *back_pass* are called k times from the procedures *front* and *back* in the section on displaying a CSG tree in normal form. *Intersect* and *subtract* replace the corresponding routines in that section. These routines allow 1-convex and k-convex primitives to be rendered simultaneously without sacrificing any of the speed of the 1-convex algorithms.

Time complexity

The time complexity of a depth-buffering algorithm in a pixel-parallel machine is proportional to the number of times that a front or back face of a primitive needs to be sent to the frame buffer, since the various pixel-oriented "bookkeeping" operations are fast. In calculating the time complexity of the NGP algorithm, we ignore the time required to normalize and prune the tree, since we have found this to be negligible for all of the objects we have rendered.

Assuming that the normalized tree has j products, each of length k, the time complexity for the NGP algorithm is $2jk(k-1)$. The relationship between j and k varies depending on an object's CSG expression and the geometry of the primitives. Thus the algorithm's overall time complexity depends on the structure of the tree as well as the number of primitives. Pathological objects will always exist for which the NGP algorithm's time complexity explodes. Nevertheless, its performance on actual objects has been very good. From the limited data we have taken, it appears to perform somewhere between $O(n)$ and $O(n^2)$ in the number of primitives.

We have obtained CSG data sets from a number of sources, and local students have used our interactive modeler to design a number of their own objects. Figure 5 gives statistics and rendering times for 11 objects ranging from 11 to 180 primitives. The table illustrates how small the average length of each product term (k) is after pruning. With k-values so small and relatively independent of the number of primitives, the NGP algorithm's time complexity approaches $O(n)$.

An interactive CSG modeler

We have incorporated these algorithms into an interactive CSG modeler. The modeler displays a smooth-shaded, hidden-surface-removed image of the objects on Pixel-planes 4's monitor. The user interface runs on the host DEC VaxStation II/GPX running Ultrix 2.2 using X Windows. It allows users to manipulate both the geometric descriptions of primitives and the structure of the CSG tree interactively.

Figure 6 shows the Pixel-planes 4 monitor and the user interface screen of our modeling system. The user can modify the CSG tree by dragging icons and can alter the sizes and positions of primitives and subtrees using a pair of joysticks.[13] Changes are displayed by the system in fractions of a second for objects between 20 and 50 primitives. Figure 7 shows images and rendering times for a number of the objects described in Figure 5.

Pixel-planes 4's front-end processor is a single fast floating-point processor based on the Weitek XL floating-point chip set. We have found this to be the bottleneck in our system. The majority of the CSG objects we have rendered use only approximately 10 percent of Pixel-planes 4's frame-buffer cycles. This makes us believe that the NGP algorithm may perform better on systems with more front-end processing power, in particular, high-performance graphics workstations with large-grain parallelism among front-end processors.

Future extensions

We intend to increase the speed of our renderer in two ways. First, we can increase the power of our front-end processor. The NGP algorithm adapts well to the large-grain MIMD front-end paradigm. Jansen describes an algorithm for a multiprocessor system that divides the frame buffer into multiple regions, allocating a processor per region.[11] Each processor maintains its own version of the CSG tree and prunes it using its particular viewing frustum as a bounding volume. We intend to implement this algorithm on Pixel-planes 5 (scheduled for completion in summer 1989).[14] Second, we can use the quadratic expression evaluator of Pixel-planes 5 to render primitives with quadratic surfaces. This will allow us to render primitives with curved surfaces rapidly.

Conclusion

We have presented a set of algorithms for efficiently rendering CSG-defined objects directly into a frame buffer without first converting to a boundary representation. Unlike other depth-buffering CSG algorithms, the NGP algorithm requires only a constant number of bits per pixel (two image buffers plus three flag bits) to render objects with convex primitives. It also can take advantage of pixel-parallelism available in such new-generation high-performance graphics workstations as the AT&T Pixel Machine and the Stellar GS-1000 graphics workstation. Although the NGP algorithm has a worst-case time complexity much poorer than the $O(n^2)$ complexity of other algorithms,[5] its time complexity compares favorably on all of the data sets we have encountered. We have incorporated the NGP algorithm into an interactive CSG modeling system that renders shaded images of moderately sized objects in fractions of a second. ■

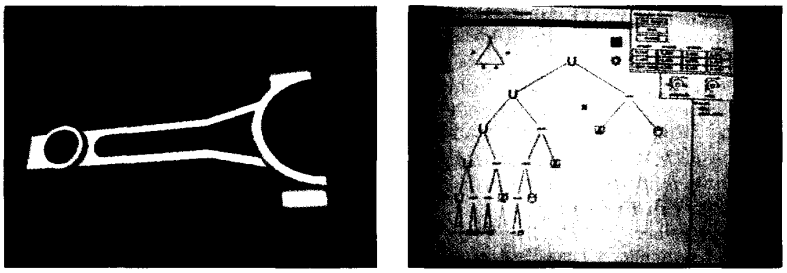

Figure 6. Pixel-planes 4 monitor and modeler interface screen.

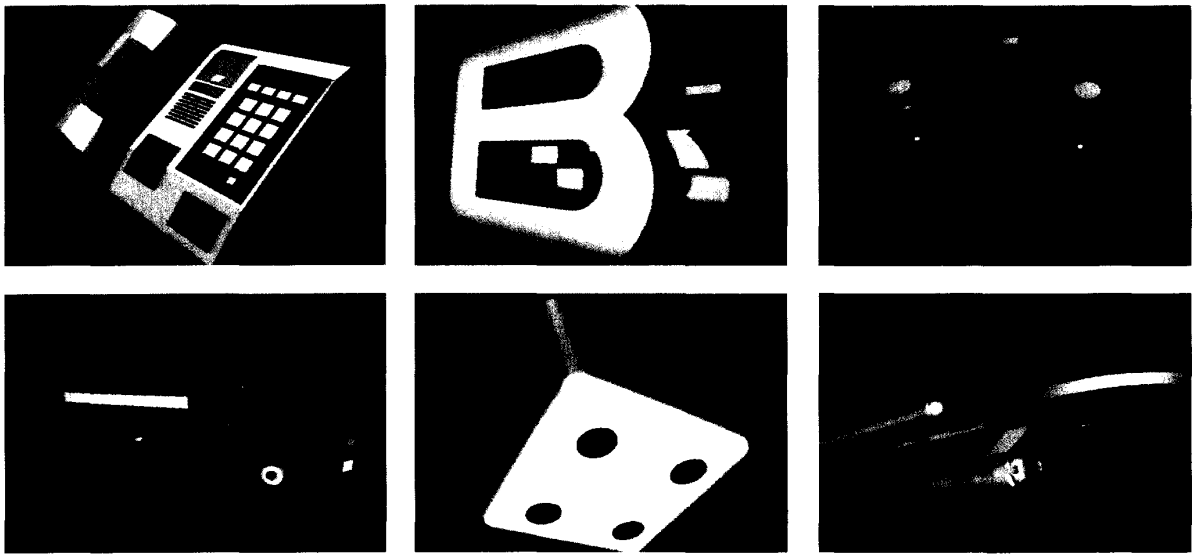

Figure 7. Images generated with the Pixel-planes 4 solid modeler.

Acknowledgments

We would like to thank Clare Durand for her participation in creating the interactive modeler, Don Stanat for help in developing the tree-normalization algorithm, Fred Brooks for encouragement and suggestions about the modeler, and the reviewers of this article for valuable suggestions for improvement. We also thank David Banks, Randy Brown, and Brice Tebbs for being early users of our system, and the members of the Pixel-planes team for creating an exciting graphics environment.

This work was supported by the Defense Advanced Research Projects Agency, ISTO order no. 6090, the National Science Foundation, grant no. MIP-8601152, and the Office of Naval Research, contract no. N00014-86-K-0680.

References

1. J. Goldfeather, J.P.M. Hultquist, and H. Fuchs, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System," *Computer Graphics* (Proc. SIGGRAPH), Vol. 20, No. 4, Aug. 1986, pp. 107-116.
2. W.C. Thibault and B.F. Naylor, "Set Operations on Polyhedra Using Binary Space Partitioning Trees," *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp. 153-162.
3. P.R. Atherton, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry," *Computer Graphics* (Proc. SIGGRAPH), Vol. 17, No. 3, July 1983, pp. 73-82.
4. G. Kedem and J.L. Ellis, "Computer Structures for Curve-Solid Classification in Geometric Modeling," Tech. Report TR84-37, Microelectronic Center of North Carolina, Research Triangle Park, N.C., 1984.
5. J.R. Rossignac and A.A.G. Requicha, "Depth-Buffering Display Techniques for Constructive Solid Geometry," *CG&A*, Vol. 6, No. 9, Sept. 1986, pp. 29-39.

6. F.W. Jansen, "A Pixel-Parallel Hidden Surface Algorithm for Constructive Solid Geometry," *Proc. Eurographics 86*, Elseviers Science Publ., New York, 1986, pp. 29-40.

7. H. Fuchs et al., "Fast Spheres, Shadows, Textures, Transparencies and Image Enhancement in Pixel-Planes," *Computer Graphics* (Proc. SIGGRAPH), Vol. 19, No. 3, July 1985, pp. 111-120.

8. N. Okino, Y. Kakazu, and M. Morimoto, "Extended Depth-Buffer Algorithms for Hidden-Surface Visualization," *CG&A*, Vol. 4, No. 5, May 1984, pp. 79-88.

9. J. Goldfeather and S. Molnar, "CSG Tree Restructuring," tech. report, Computer Science Dept., Univ. of North Carolina at Chapel Hill, 1989.

10 R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. on Computers*, C-29, No. 10, Piscataway, NJ, Oct. 1980, pp.874-883.

11. F.W. Jansen and R.J. Sutherland, "Display of Solid Models with a Multi-Processor System," *Proc. Eurographics 87*, Elsevier Science Publ., 1987, pp.377-387.

12. K. Bouatouch et al., "A New Algorithm of Space Tracing Using a CSG Model," *Proc. Eurographics 87*, Elsevier Science Publ., New York, 1987, pp. 65-77.

13. C. Durand and S. Molnar, "CSG User's Manual," Computer Science Dept., Univ. of North Carolina at Chapel Hill, 1987.

14. H. Fuchs et al., "A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," Tech. Report TR89-005, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, 1989.

Jack Goldfeather is associate professor of mathematics and computer science at Carleton College in Northfield, Minnesota, where he teaches a variety of undergraduate mathematics and computer science courses. Trained as an algebraic topologist, he has recently turned his attention to mathematical problems related to hardware and software design for graphics systems. In 1984 he began working as a mathematics consultant on the Pixel-planes project directed by Henry Fuchs.

Goldfeather received a BA in mathematics from Rutgers University in 1969 and an MS and PhD in mathematics from Purdue University in 1971 and 1975, respectively. He taught at the University of Wisconsin at Milwaukee from 1975 to 1977 before joining the Carleton faculty.

Goldfeather can be contacted at Carleton College, Dept. of Mathematics, 1 North College St., Northfield, MN 55057.

Steven Molnar is a research associate in computer science at the University of North Carolina at Chapel Hill. His research interests include architectures and algorithms for real-time 3D graphics. He is currently involved in hardware and software development for the Pixel-planes 5 graphics system, a VLSI-based architecture for raster graphics being built at UNC Chapel Hill.

Molnar received a BS in electrical engineering from the California Institute of Technology in computer science from the University of North e is a member of ACM.

Greg Turk is a graduate student in the Computer Science Department at the University of North Carolina at Chapel Hill. His research interests include computer animation, image rendering techniques, and virtual worlds.

Turk received a BA in mathematics from UCLA in 1984. He is a member of ACM and IEEE Computer Society.

Henry Fuchs is Federico Gil Professor of computer science at the University of North Carolina at Chapel Hill. He teaches graduate courses in computer graphics and directs the research of PhD students and research associates in graphics algorithms and VLSI architectures. He is principal investigator for several research projects funded by DARPA/ISTO, NIH, and NSF. He has consulted for a variety of industrial organizations, and he is presently a member of the Technical Advisory Board for Stellar Computer, a company producing high-performance graphics workstations. He served as chairman of the 1985 Chapel Hill Conference on VLSI and the 1986 Workshop on Interactive 3D Graphics held at UNC-Chapel Hill.

Fuchs received a BA from the University of California at Santa Cruz in 1970 and a PhD from the University of Utah in 1975.

Molnar, Turk, and Fuchs can be contacted at the Computer Science Department, Sitterson Hall, University of North Carolina, Chapel Hill, North Carolina 27599.