# Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System

Jack Goldfeather
Carleton College, Northfield, MN

Jeff P.M. Hultquist
Henry Fuchs
University of North Carolina at Chapel Hill

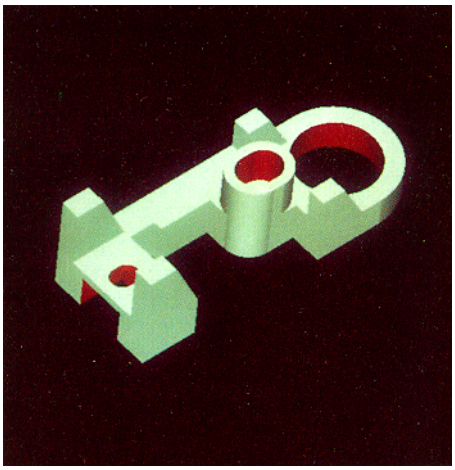
## ABSTRACT

We present two algorithms for the display of CSG-defined objects on Pixel-Powers, an extension of the Pixel-Planes logic-enhanced memory architecture, which calculates for each and every pixel on the screen (in parallel) the value of any quadratic function in the screen coordinates $(x, y)$. The first algorithm restructures any CSG tree into an equivalent, but possibly larger, tree whose display can be achieved by the second algorithm. The second algorithm traverses the restructured tree and generates quadratic coefficients and opcodes for Pixel-Powers. These opcodes instruct Pixel-Powers to generate the boundaries of primitives and perform set operations using the standard Z-buffer algorithm.

Several externally-supplied CSG data sets have been processed with the new tree-traversal algorithm and an associated Pixel-Powers simulator. The resulting images indicate that good results can be obtained very rapidly with the new system. For example, the commonly used MBB test part (at right) with 24 primitives is translated into approximately 1900 quadratic equations. On a Pixel-Powers system running at 10MHz (the speed at which our current Pixel-Planes memories run), the image should be rendered in about 7.5 milliseconds.

**CR Categories and Subject Descriptors:** I.3.1 [**Computer Graphics**]: Hardware Architecture — *raster display devices*; I.3.3 [**Computer Graphics**]: Picture/-Image Generation — *display algorithms*; J.7 [**Computer Applications**]: Computer-Aided Engineering — *computer-aided design*;
**General Terms:** algorithms
**Additional Key Words and Phrases:** constructive solid geometry, SIMD processor, frame buffer





**MBB test part from Pixel-Powers simulator.** The Pixel-Powers graphics system should render this image in 7.5 milliseconds.

## I. Introduction

We are designing Pixel-Powers, an enhancement of the Pixel-Planes graphics system [2][6], by replacing the multiplier tree that evaluated linear expressions by one that evaluates quadratic expressions [3]. This Quadratic Expression Evaluator (QEE) is used to evaluate expressions of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F$ simultaneously for each pixel $(x, y)$ on the screen. We estimate that the QEE will calculate bit-sequentially a 30 bit value of this expression for each and every pixel on the screen in under 4 microseconds. The speed at which Pixel-Powers will render convex polyhedra, as well as smooth-shaded cylinders, cones, and ellipsoids, has led us to explore the possibility of using Pixel-Powers for real-time rendering of smooth-shaded Constructive Solid Geometry objects constructed from quadratic primitives. A CSG object is defined by starting with a set of solid primitives and constructing an expression tree in which the leaves are primitives and the non-leaf nodes are set operations. The CSG object is constructed recursively by performing each set operation on the objects defined by its left and right subtrees [7].

In this paper we describe a general method for displaying any CSG object using a frame buffer that is 128 bits deep. Our method differs from other CSG display methods [1] in that we compute on the fly the boundary representation of each primitive in terms of the viewpoint. While this can be a disadvantage in some systems, we will show how it can be implemented efficiently in Pixel-Powers by making use of the Quadratic Expression Evaluator and the general parallelism of the system. In particular, we will describe an algorithm for fast rendering of smooth-shaded CSG objects based on quadratic primitives. Our approach, parallel on all pixels but processing CSG primitives sequentially, contrasts with another system by Kedem [4] that allocates a processing element for each primitive and renders the images sequentially by pixel in raster-scan order.

Just as in the development of the Pixel-Planes system, we have implemented software simulators that enable us to develop display algorithms before the actual chip is completely designed and committed to silicon. All of the images in this paper are from the Pixel-Powers simulator.

## II. A Simple Example

In this section we describe a method for displaying any CSG object with the aid of a deep frame buffer. The present working Pixel-Planes system has a 72 bit deep frame buffer. A Pixel-Powers system with a depth of 128 bits was our model when we were analyzing the problem, but the algorithm should be implementable in any computer with a deep frame buffer. The memory requirements are:

- Three flag registers: F1, F2, and F3
  (one bit each)
- Two depth buffers: ZTEMP and ZMIN
  (20-30 bits each)
- One color buffer: COLOR
  (24 bits)
  (If double buffering is desired,
  two color buffers are needed.)

We defer until Sections III and IV the discussion of the particular Pixel-Powers implementation of these algorithms for CSG objects defined with convex primitive solids whose boundary surfaces can be defined using quadratic or linear equations in $x$, $y$, and $z$ (e.g. cylinders, ellipsoids, and cones). In this section we outline a general method of display that will work for any set of convex primitives and any display system that can do both of the following:

(a) Scan convert front- and back-facing surfaces of each primitive in screen space. That is, a flag F at each pixel can be set to 1 if it is inside the region on the screen determined by the projection of the surface on the screen. Note that the front and back face of a surface depends on the viewpoint. The front surface of a cylinder consists of all points on the cylinder surface (including the ends) which face toward the viewer.

(b) Calculate and store in each pixel memory with F=1 the depth and color values of the front- or back-facing surfaces of a primitive.

In Section III, a general algorithm is derived based only on the assumptions (a) and (b) above. We illustrate the ideas behind this algorithm by examining the simple cases of union, difference, and intersection of two cylinders.

### II.a. *Cylinder1* ∪ *Cylinder2*

This is displayed by applying the standard Z-buffer algorithm. If *Front(obj)* denotes the (viewpoint dependent) visible part of an object's surface, then *Front(Cylinder1)* will, in general, be the visible part of the curved portion of the cylinder together with one of the two planar ends. We begin by calculating the $Z$ values and color values of *Front(Cylinder1)* and storing them in ZMIN and COLOR. Since later in this paper we will be decomposing more complicated objects into unions of simpler ones, we will describe carefully how *Cylinder2* is added to the partial image:

**Step 1:** At each pixel, set the flag F1 if it is inside the region determined by *Front(Cylinder2)*, and clear it otherwise (figure 1a).

**Step 2:** Calculate and store $Z$ values for *Front(Cylinder2)* in ZTEMP.

**Step 3:** For each pixel with F1 set, compare ZTEMP to ZMIN and if ZTEMP > ZMIN then clear F1 (figure 1b).

**Step 4:** For each pixel with F1 still set, replace the contents of COLOR with the color of *Front(Cylinder2)* and replace ZMIN by ZTEMP (figure 1c).

Note that this algorithm does not require that the unioned objects be primitive. As long as scan conversion, depth values, and colors can be calculated, any objects can be unioned together by this simple method. This technique of composing objects with Z-buffers has been used in many previous systems.

### II.b. *Cylinder1* − *Cylinder2*

This can be displayed by first recognizing that its image is identical to the image of:

$$(Front(Cylinder1) - Cylinder2) \cup (Back(Cylinder2) \cap Cylinder1).$$

The general algorithm for generating such decompositions is described in Section IV. As we saw in the union process above, it suffices to generate the first term in the union and then add the second term to this partial image. The first term, *Front(Cylinder1)* − *Cylinder2* is rendered as follows:

**Step 1:** Set F1 for all pixels inside the projection of *Cylinder1* onto the screen (figure 2a).

**Step 2:** For pixels at which F1 is set, store the depth of *Front(Cylinder1)* in ZTEMP.

**Step 3:** Set F2 everywhere. Clear F2 at any pixel outside *Cylinder2*. A pixel $(x, y)$ is outside *Cylinder2* if its ZTEMP does not lie between the values of *Front(Cylinder2)* and *Back(Cylinder2)* (figure 2b). Replace F1 by (F1 *xor* F2) (figure 2c).

**Step 4:** We now transfer the value of ZTEMP to ZMIN for each pixel at which F1 is set. For these same pixels, we update the contents of COLOR with the color of *Front(Cylinder1)* at that location.

*Front(Cylinder1)* − *Cylinder2* is now finished. Next we add *Back(Cylinder2)* ∩ *Cylinder1* to this partial image.

**Step 5:** Set F1 for all pixels inside the projection of *Back(Cylinder2)* on the screen (figure 2d).

**Step 6:** For those pixels in which F1 is set, set ZTEMP to the depth of *Back(Cylinder2)*.

**Step 7:** Clear F2 everywhere. Set F2 for all pixels which are inside *Cylinder1* in a manner similar to step 3 (figure 2d). Replace F1 by (F1 *and* F2). F1 is now set for all pixels which display the back wall of the hole which *Cylinder2* bores into *Cylinder1* (figure 2e).

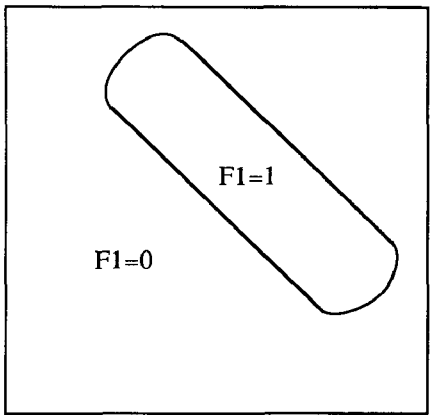



Figure 1a

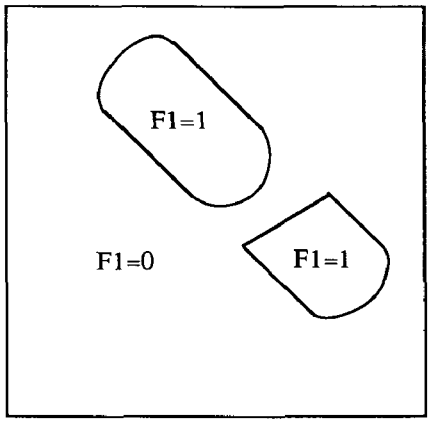



Figure 1b

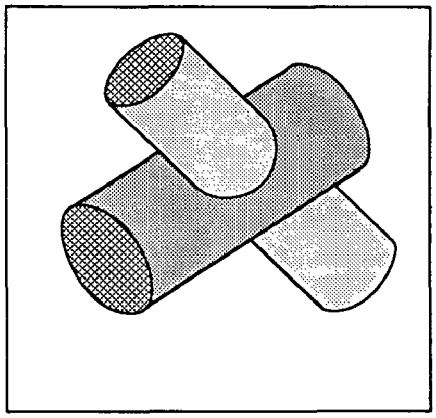


Figure 1c

**Step 8:** For these pixels, we clear F1 if ZTEMP > ZMIN. We now transfer the value of ZTEMP to ZMIN for each pixel at which F1 remains set. For these same pixels, we update the contents of COLOR with the color of *Back*(*Cylinder2*) at that location (figure 2f).

II.c. *Cylinder1* ∩ *Cylinder2*

This can be decomposed into

(*Front*(*Cylinder1*) ∩ *Cylinder2*) ∪
(*Front*(*Cylinder2*) ∩ *Cylinder1*)

The terms in this union are generated in a manner similar to the terms in the decomposition of the difference of the cylinders.

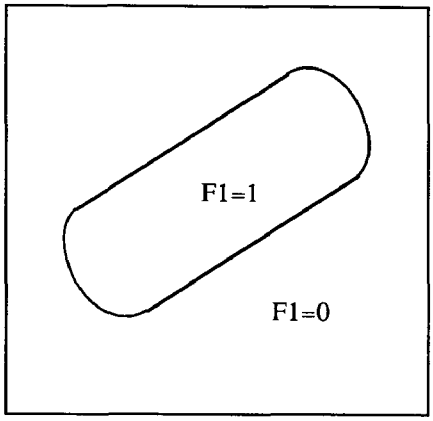



Figure 2a

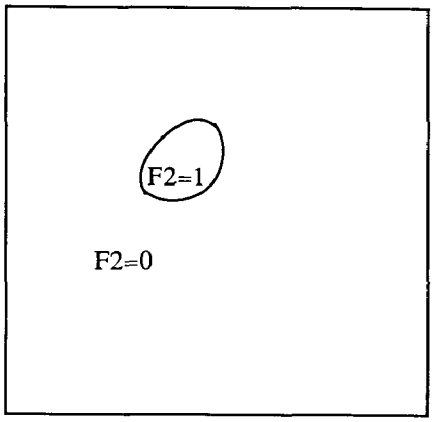



Figure 2b

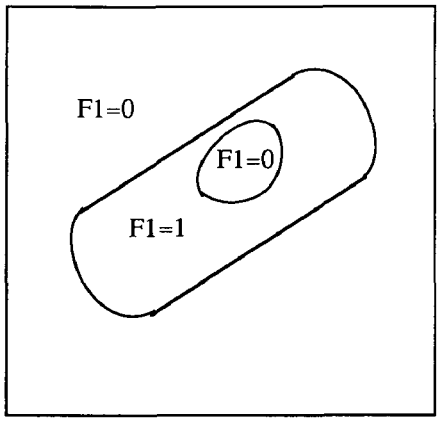



Figure 2c

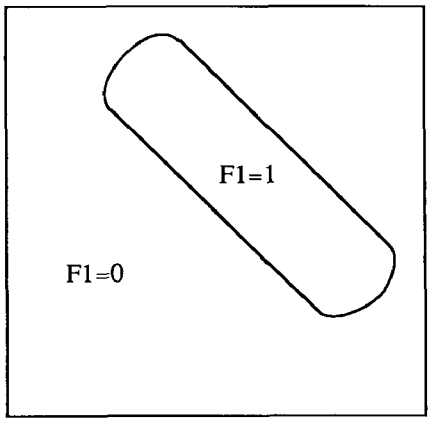



Figure 2d

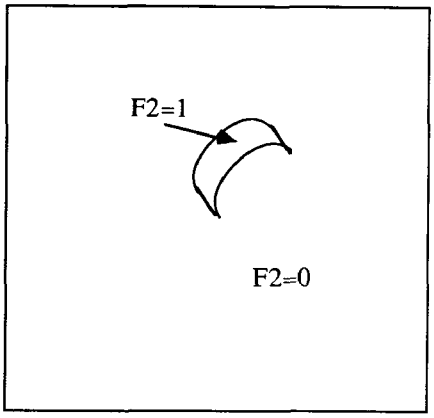



Figure 2e

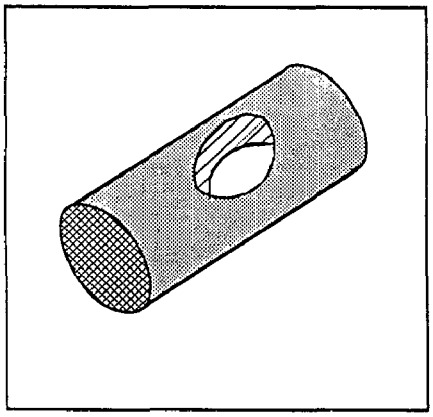


Figure 2f

### III. Example Implemented with Pixel-Powers

We will see in the following sections that this method is particularly suitable for implementation in a machine such as Pixel-Powers that has a small fixed amount of memory at each pixel. The dramatic speed in Pixel-Powers is due in large part to the Quadratic Expression Evaluator which evaluates quadratic expressions in $x$ and $y$ simultaneously at each pixel. The architecture of this Evaluator is more fully described in [3]. For the purposes of this discussion, it is sufficient to assert that the Pixel-Powers system will consist of a enhanced frame buffer memory. Each pixel is located at a leaf of the Evaluator, which receives the coefficients A,B,C,D,E, and F as input and evaluates the expression $Q(x,y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F$. The speed of Pixel-Powers is due in large part to the fact that this calculation is done simultaneously at each pixel when the coefficients are broadcast to the system. One bit of the function value is calculated for each and every pixel at each clock cycle. As with the current Pixel-Planes chips in 3 micron nMOS, we expect a 100 nsec clock cycle. Each pixel will have a single-bit ALU and 128-bits of randomly-addressable memory. This memory is also scanned out by the video controller.

For the particular algorithms described here, the memory is logically configured into ZMIN, ZTEMP, and COLOR registers, and also one-bit flags F1, F2, and F3. The Host processes the CSG tree to produce a sequence of instructions that drive the Evaluator and the ALUs. All geometric transformations and clipping are calculated in the host as well as the translating of the information in the CSG tree into the sequence of opcodes and the quadratic equations.

In this section, we will describe a way to implement in Pixel-Powers the basic operations listed in Section II:

- Scan conversion of primitives
- Computation of depth values
- Determination of "inside" or "outside"
- Calculation of color

We illustrate the procedure with part of the preceeding example: *Front(curved part of Cylinder1) − Cylinder2.* We omit the calculations involving the end of the cylinder as they are similar.

**Step 1:** Scan Conversion

We begin by writing the equations of the bounding curves of *Front(Cylinder1)* in screen coordinates $(x,y)$ (figure 3a). The elliptical ends are defined by quadratic equations $Q_1(x,y) = 0$ and $Q_2(x,y) = 0$. The lines of intersection of the front- and back-facing surfaces have linear equations $L_1(x,y) = 0$ and $L_2(x,y) = 0$. The lines $L_3$ and $L_4$ indicated in figure 3a have linear equations $L_3(x,y) = 0$ and $L_4(x,y) = 0$. We combine $L_1$ and $L_2$ to create the quadratic equation $Q(x,y) = L_1(x,y)L_2(x,y) = 0$, and we combine $L_3$ and $L_4$ to create the quadratic equation $Q_3(x,y) = L_3(x,y)L_4(x,y) = 0$.

Each of the curves $Q, Q_1, Q_2, Q_3$ separate the plane into pieces and a pixel can determine which piece it is in by simply checking the sign of the result. Different choices of the coefficients will produce different signs for these expressions, so the selection must be made to conform to the signs indicated in figure 3a. The Host computes the coefficient sets for each of the four quadratic curves and broadcasts them to the Quadratic Expression Evaluator. Three one-bit flags are used to enable or disable pixels according to the sign of the evaluated expression at that location.

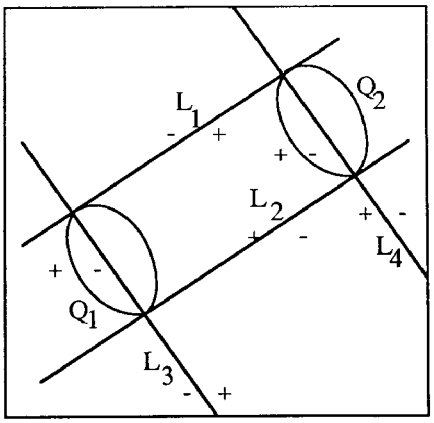



Figure 3a

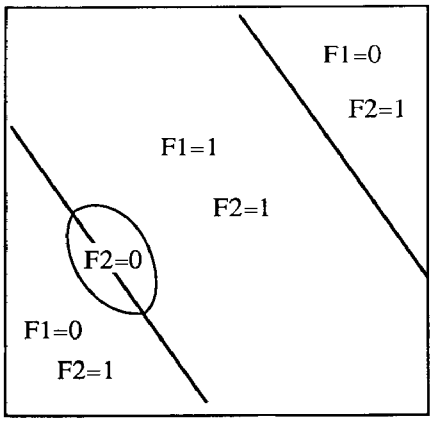



Figure 3b

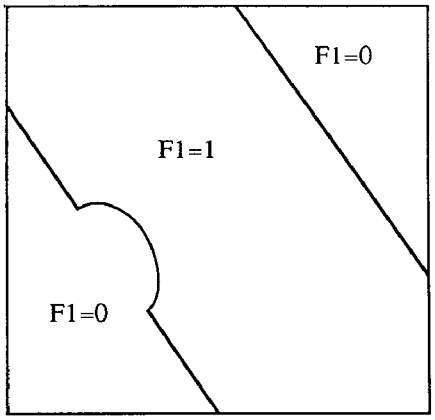



Figure 3c

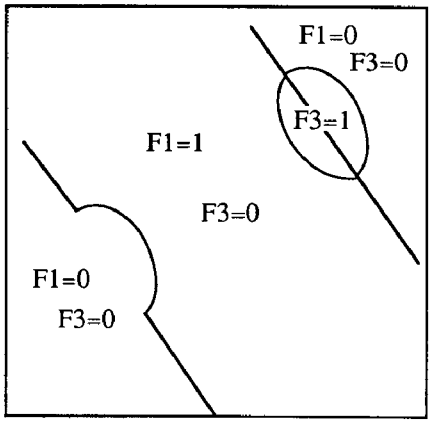



Figure 3d

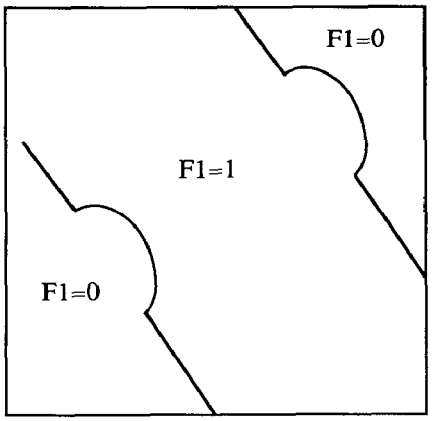



Figure 3e

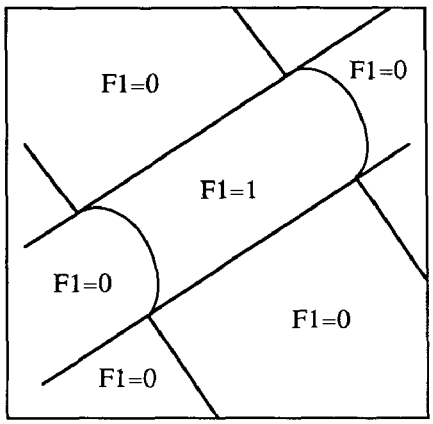



Figure 3f

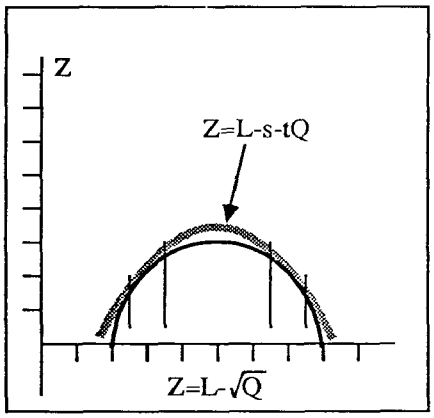



Figure 4a

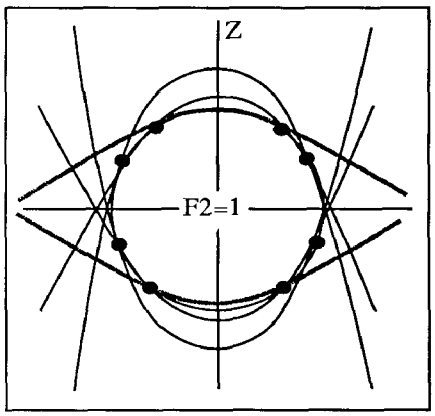



Figure 4b

The specific sequence in our example is:

(a) Clear flags F1, F2, and F3 everywhere.

(b) For each pixel $(x,y)$, set F1 if $Q_3(x,y) > 0$, and set F2 if $Q_1(x,y) > 0$. Replace F1 by (F1 *and* F2) (figures 3b and 3c).

(c) For each pixel $(x,y)$, set F3 if $Q_2(x,y) < 0$. Replace F1 by (F1 *or* F3) (figures 3d and 3e).

(d) For each pixel $(x,y)$, clear F1 if $Q(x,y) < 0$ (figure 3f).

Note that this scan conversion process requires that the coefficient sets for $Q$, $Q_1$, $Q_2$, and $Q_3$ be broadcast only once each.

**Step 2:** Z-Buffer

The equation of *Front(curved part of Cylinder1)* when solved for $z$ is of the form $z = L - \sqrt{Q}$, where $L$ is linear and $Q$ is quadratic in $x$ and $y$. (The function $Q$ is the same one from step 1.) Since the QEE cannot evaluate square roots directly, an approximation to $\sqrt{Q}$ must be made. This approximation is of the form $s + tQ$ where $s$ and $t$ are constants, and we replace $z = L - \sqrt{Q}$ by $Zapprox = L - s - tQ$ which is quadratic in $(x,y)$. By choosing $s$ and $t$ carefully, this approximation is very accurate in strips down the length of the cylinder. Geometrically, the surface with equation $Zapprox = L - s - tQ$ is a "parabolic" cylinder. Figure 4a illustrates how it passes near to the actual cylinder surface. The magnitude of the error tolerance determines the size of the strips in which the approximation is within this tolerance.

We begin by choosing an error tolerance for the $Z$ approximation. The Host determines the number of strips needed to guarantee this accuracy across the entire scan converted region. The constants $s$ and $t$ are computed for each such strip pair. Geometrically, the set of parabolic cylinders (one for each $(s,t)$) forms an "envelope" of the actual cylinder. Further, as indicated in figure 4b, for each $(x,y)$, the largest *Zapprox* is the one that best approximates the actual $Z$ for that pixel $(x,y)$. The Host simply broadcasts the coefficients for all of the parabolic cylinder approximations and each pixel $(x,y)$ saves in ZTEMP the largest *Zapprox* for that pixel. Note that for back facing surfaces, the pixel saves the smallest *Zapprox*.

The number of strips needed depends on the size of the object in screen space. It might seem that many strips would be needed to guarantee reasonable accuracy, but in many images that we have generated using the functional simulator, sufficient accuracy can be achieved with a small number of strips (1 to 8). This small number is due to the fact that we are approximating a curved surface by another curved surface, so that we do not need nearly as many subdivisions as would be necessary if we were approximating the same surface with polygons.

**Step 3:** Subtracting *Cylinder2*

(a) Subdivide *Cylinder2* into strips for accurate Z calculation as in Step 2. Compute the quadratic expressions $Q_i$ that represent the parabolic cylinder approximations for these strips.

(b) Set F2 at each pixel. For each parabolic cylinder $C_i$, broadcast the coefficients of $Q_i$ and clear F2 if the ZTEMP stored at the pixel $(x,y)$ is less than $Q_i(x,y)$ and $C_i$ is front-facing or if ZTEMP $> Q_i(x,y)$ and $C_i$ is back-facing (figure 4b).

(c) Only those pixels with both F1 and F2 still set are inside *Cylinder2*. Replace F1 with (F1 *xor* F2).

**Step 4:** Shading

If we compute the exact diffuse shade at $(x, y)$ using the unit normal to the surface then the expression we have to evaluate is of the form $shade(x, y) = (L + \sqrt{Q})/\sqrt{W}$ where $L$ is linear, $Q$ is quadratic in $(x, y)$ and $W$ is a relatively complicated expression in $(x, y)$ that comes from turning an arbitrary normal to the surface into a unit vector. We approximate the numerator as in the Z-buffer step except that we only use a single parabolic cylinder for $Q$. We approximate the denominator by a single constant. Although these approximations may seem coarse, the effect is smooth shaded.

## IV. Tree Restructuring

In this section we describe a method for transforming any CSG tree into an equivalent one that is a union of simpler subtrees. (Similar work is briefly discussed in [8].) We will then describe how each of these simple subtrees can be displayed by further dividing them into the union of pieces which can be displayed by starting with the boundary of a primitive and paring it with other primitives, using the set operations of intersection and difference. This transformation and display process builds up the image without the use of large amounts of intermediate information stored at each pixel. This method is particularly appropriate for a system like Pixel-Powers with limited memory available at each pixel.

There are two major difficulties with trying to display arbitrary CSG trees without any transformation. First, the paring part, that is, the piece that is subtracted or intersected with a previously constructed piece, might be complicated. In particular, it might be hard to determine the inside or outside in an efficient manner. Second, paring may reveal parts of an object previously obscured. Both of these difficulties can be overcome by the transformation process that restructures the CSG tree into an equivalent one in which the paring objects are always primitives.

The transformation produces a new tree which we call a *normal form* for the tree which has the properties (i) at every node where there is an intersection or difference the right branch is primitive, and (ii) no node where there is a union is on a path from a difference or intersection. This new tree can be broken into simpler subtrees that are unioned together. Although the transformation process may increase the size of the tree, each of the simple subtrees can be displayed with a minimum of calculation and merged into a single image using the union process described in Section II. The simple subtrees are of the form:

$$X_0\, op_1\, X_1\, op_2 \ldots op_k\, X_k$$

where each $X_i$ is a primitive, $op_i$ is either $-$ or $\cap$, and the absence of parentheses indicates that association is from left to right. A normal form for a CSG tree is created using the 8 basic equivalences in figure 5 together with the following recursive algorithm. The execution of this algorithm is demonstrated in figure 6.

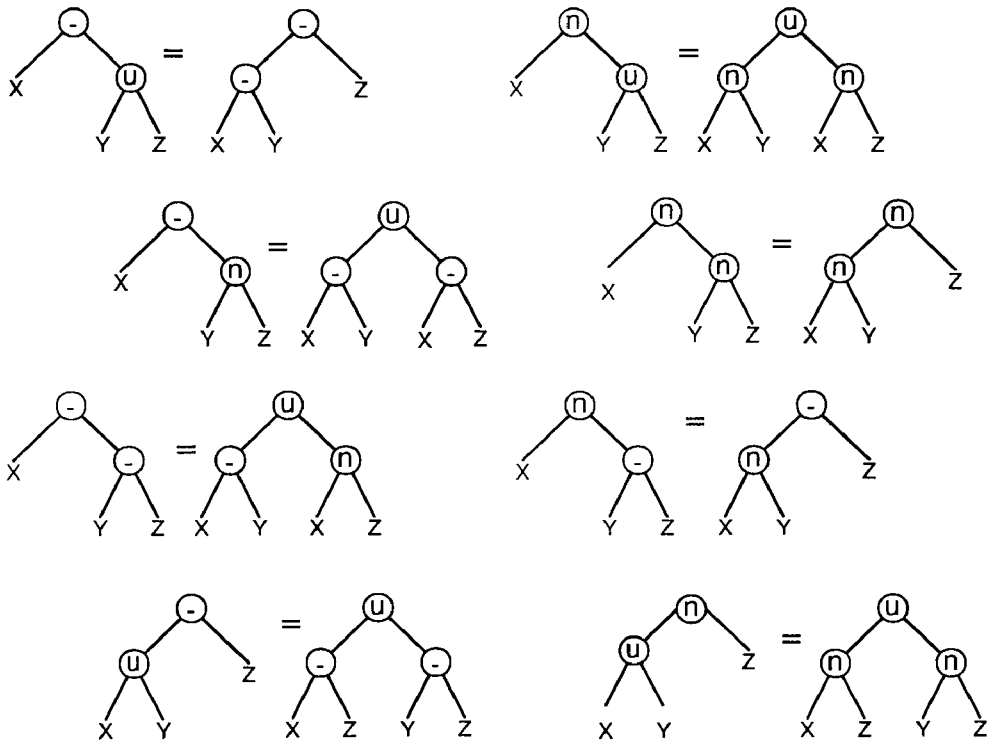



**Figure 5.** In each pair, the tree on the left can be transformed to the equivalent form to its right. The new tree will have the same image as the original.

```
procedure Normalize (T);
begin
   Redo(T);
   case (T.type) of begin
      primitive:
         return T;
         break;
      ∪ :
         Normalize (T.L);
         Normalize (T.R);
         break;
      −,∩ :
         while (T.type ≠ primitive) and
               (T ≠ ∪) and
               (T.R.type ≠ primitive) do begin
            Redo (T);
         end;
         Normalize (T.R);
         Normalize (T.L);
         Redo(T);
         break;
   end;
end;
```

```
procedure Redo(T)
begin
   if T does not have any of the patterns
     in figure 5 then begin
      return T;
   end else begin
      restructure the top nodes of T
        using equivalent patterns in figure 5;
      return newT;
   end;
end;
```

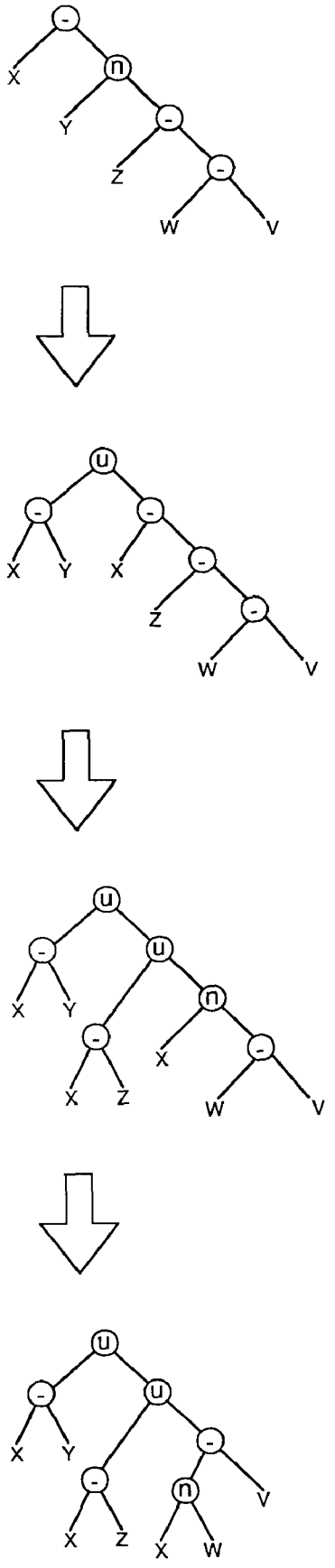



**Figure 6.** The trees on this page demonstrate the execution of the code above using the equivalences shown in figure 5 (at left). At each step, one interior node of the tree is restructured. This processing continues recursively until the tree is in normal form.

Once the tree has been normalized, the problem of display is reduced to that of simple trees. Let $D(X)$, $D_f(X)$, and $D_b(X)$ denote the boundary of a solid X, the front-facing boundary of X, and the back-facing boundary of X, respectively. In order to display a solid X it suffices, of course, to display $D(X)$. We are left then with the problem of displaying

$$D(X_0\, op_1\, X_1\, op_2 \ldots op_k\, X_k)$$

In order to derive the general display algorithm, it is necessary to know how the CSG operations interact with the boundary operators $D$, $D_f$, and $D_b$.

**Theorem 1:** From the point of view of 2-D display:

(a) $D(X) = D_f(X)$
(b) $D(X \cup Y) = D_f(X) \cup D_f Y$
(c) $D(X \cap Y) = (D_f(X) \cap Y) \cup (D_f(Y) \cap X)$
(d) $D(X - Y) = (D_f(X) - Y) \cup (D_b(Y) \cap X)$

For example, if we want to display the tree A-B-C, we apply Theorem 1(d) twice and use the set identity $X \cap (Y - Z) = X \cap Y - Z$:

$$\begin{aligned} & D(A - B - C) \\ = \; & (D_f(A - B) - C) \;\cup \\ & (D_b(C) \cap (A - B)) \end{aligned}$$

by applying Theorem 1(d) with $X = A - B$ and $Y = C$

$$\begin{aligned} = \; & (D_f(A) - B - C) \;\cup \\ & (D_b(B) \cap A - C) \;\cup \\ & (D_b(C) \cap A - B) \end{aligned}$$

by applying Theorem 1(d) again and using the above set identity.

The terms in the union are rendered one at a time and merged into the partial object being built up. The first term is rendered by storing $D_f(A)$ and paring it down with the objects B and C. This is essentially how the example in Section II was done. The other terms are rendered similarly.

We will adopt the convention that there is an operator $op_0$ equal to $\cap$ preceding $X_0$ in the simple tree $X_0\, op_1\, x_1\, op_2 \ldots op_k\, X_k$ and define for each $i = 0, \ldots, k$:

$$D_p(X_i) = \begin{cases} D_f(X), & \text{if } op_i = \cap \\ D_b(X), & \text{if } op_i = - \end{cases}$$

Then we can apply the theorem recursively to obtain:

**Theorem 2:** $D(X_0\, op_1\, X_1 \ldots op_k\, X_k)$ is the union ($i = 0, \ldots, k$) of:

$$D_p(X_i)\, op_1\, X_1 \ldots op_{i-1}\, X_{i-1}\, op_{i+1}\, X_{i+1} \ldots op_k\, X_k$$

The individual terms in this union are displayed as in the example in Section II. To summarize, the normalization process that reduces an arbitrary CSG tree to a union of simple trees together with the further subdivision using Theorem 2 produces a decomposition that allows images to be drawn without sending anything more complicated than a primitive to the system. This is essential for graphics systems with limited frame buffer memory.

## V. Results

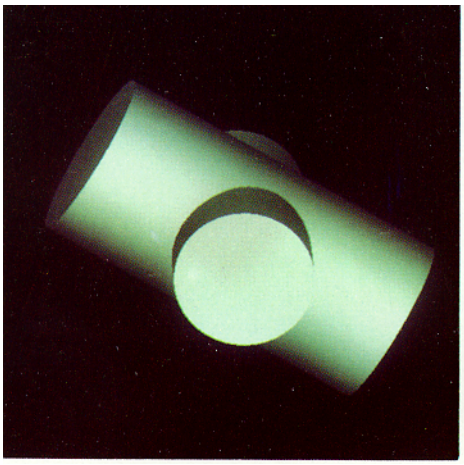
We have implemented (in C on a VAX-11/780 running 4.2bsd UNIX) and show results here of (1) a tree traverser that processes a union of "simple" trees and generates opcodes and quadratic coefficients to a Pixel-Powers memory system, and (2) a simulator for a Pixel-Powers memory system that accepts opcodes and quadratic coefficients and generates for each pixel the various image buffer-related values ((r,g,b), z, flags, etc.) for display on a conventional raster screen. This set of software modules was exercised with externally supplied data sets from the US Army Ballistic Research Laboratory and Hokkaido University [5].

We have been surprised to find no need yet for the CSG restructuring algorithm, so we have not as yet implemented it. Of the handful of data sets we have received we have found none yet whose CSG tree needed to be restructured before processing for Pixel-Powers. That is, all the trees were already "simple" according to the definition given in Section IV above. Thus the tree traverser could process all of these data sets directly and generate opcodes and coefficients for Pixel-Powers.

We ran the tree traverser on the various data sets and ran the Pixel-Powers simulator on the output from the tree traverser. Table 1 gives, for various data sets, the number of Pixel-Powers operations generated by the tree traversal process and the estimated time for Pixel-Powers to generate the images from these data sets shown in the photographs. It is important to note when considering these results, however, that the estimated image generation times given in the table are for the 10MHz Pixel-Powers logic-enhanced memories themselves. It is assumed that the rest of the system, the "front end" (the viewing transformation engine and the tree traverser) can run fast enough to keep up with the 10MHz Pixel-Powers memories. We hope to achieve this by transferring the implementation to our fast arithmetic processors, which are Mercury Systems ZIP 3232s.

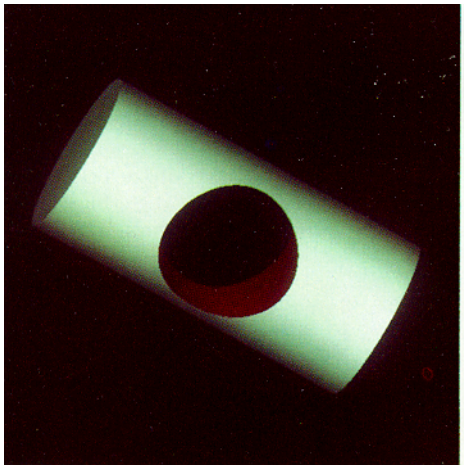
| Part Name | Source | Primitives | Opcodes | Equations | Time |
|---|---|---|---|---|---|
| Union | local | 2 | 54 | 46 | .19 msec |
| Difference | local | 2 | 182 | 170 | .68 |
| Intersection | local | 2 | 178 | 170 | .68 |
| Tube | [Okino 84] | 11 | 1205 | 1065 | 4.3 |
| Cut Tube | [Okino 84] | 12 | 1969 | 1733 | 7.0 |
| MBB | [Okino 84] | 24 | 2139 | 1854 | 7.5 |
| Tie Rod | BRL | 17 | 2660 | 2309 | 9.3 |

**Table 1: Estimated Image Generation Time**

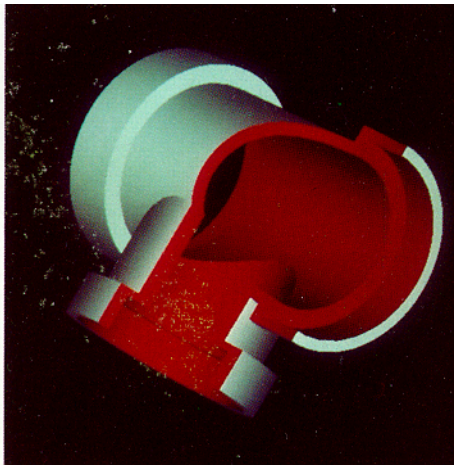


**Union**
estimated time: 0.19 msec

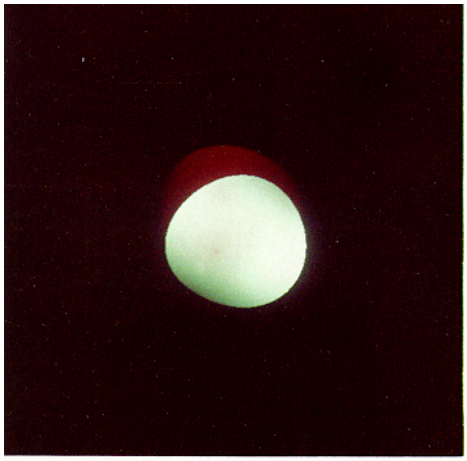
**Tube**
estimated time: 4.3 msec



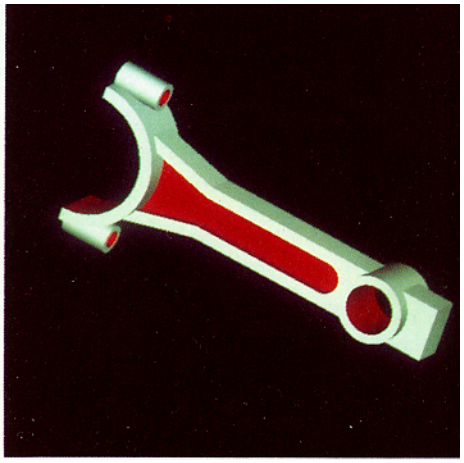
**Difference**
estimated time: 0.68 msec



**Cut Tube**
estimated time: 7.0 msec



**Intersection**
estimated time: 0.68 msec



**Tie Rod**
estimated time: 9.3 msec

**Images from Pixel-Powers Simulator**

## VI. Future Work

We hope to implement a Pixel-Powers system in stages by enhancing the next generation Pixel-Planes chips and by casting much of the CSG tree traverser into microcode for our fast arithmetic processors. The enhancement to the Pixel-Planes chips involves substituting the Quadratic Expression Evaluator tree for the current Linear Expression Evaluation tree and increasing the memory per chip from the 72 bits in the present Pixel-Planes chips to 128 bits.

We also hope to develop more sophisticated algorithms for CSG-defined objects: algorithms for generating shadows and algorithms for calculating shadings on curved surfaces more rapidly according to more sophisticated lighting models such as the popular one due to Phong. We also hope to develop techniques for rendering higher order surfaces such as cubic patches. Already two approaches for this are evident: the quadratic expression evaluator on the memory chip could be expanded into a cubic expression evaluator (we can already see how to do this, but the size would be enormous) or we can approximate each of the cubic curves by combination of many quadratic curves. We also plan to implement with the CSG restructuring algorithm the well-known "bounding-box" techniques to trim the restructured tree to the smallest possible size. For example, if the bounding boxes of $A$ and $B$ do not intersect, then $(A - B)$ is equivalent to $A$.

## VII. Summary

We have shown that CSG-defined objects can be efficiently rendered in a logic-enhanced frame buffer memory with fast quadratic expression evaluation for each pixel. Such rendering can be efficiently generated by first restructuring the tree, if necessary, into a union of simple trees and then traversing these trees to generate a sequence of quadratic coefficients and operation codes for the logic-enhanced memories. Resulting images from a software implementation of the tree traverser and display simulator illustrate the methods and allow estimation of its speed with an expected hardware implementation. The method's speed promises real-time interactions for complex CSG objects and the ability to handle objects of arbitrary complexity by building up the image during the traversal of the CSG tree.

## VIII. Acknowledgements

We thank the other members of the Pixel-Planes team — particularly John Poulton, John Eyles, John Austin, and Wayne Dettloff — for stimulating discussions and suggestions. We thank John Eyles also for developing a detailed logic-level simulator of the Quadratic Expression Evaluator and improving the QEE design in the process.

We also wish to thank our colleagues who graciously sent us CSG data sets: Paul Stay and Paul Deitz of the US Army Ballistic Research Laboratory and Professor Ari Requicha, Director of the Production Automation Project at the University of Rochester. Testing our algorithms on these externally-supplied data sets considerably increased our confidence in the algorithms and their implementations. Also we thank Norio Okino and his colleagues for publishing their data.

This research was supported in part by the DARPA contract DAAG29-83-K-0148 (monitored by the US Army Research Office, Research Triangle Park, NC) and the NSF Grant ECS-8300970. Jeff Hultquist was supported with a grant from the UNC Board of Governors.

Finally, we thank Mary Hultquist for her help with the photographs and text.

## IX. References

[1] Atherton, P.R., "A Scan-line Hidden Surface Removal Procedure for Constructive Solid Geometry" *Computer Graphics*, Vol. 17, No. 3, pp. 73–82, 1983. (Proceedings of SIGGRAPH '83)

[2] Fuchs, H., J. Goldfeather, J.P. Hultquist, S. Spach, J. Austin, F.P. Brooks, Jr., J. Eyles, and J.Poulton. "Fast Spheres, Textures, Transparencies, and Image Enhancements in Pixel-Planes" *Computer Graphics*, Vol. 19, No. 3, pp. 111–120, 1985. (Proceedings of SIGGRAPH '85)

[3] Goldfeather, J., H. Fuchs. "Quadratic Surface Rendering on a Logic-Enhanced Frame-Buffer Memory" *IEEE Computer Graphics and Applications*, pp. 48–59, January, 1986.

[4] Kedem, G., J.L. Ellis. "Computer Structures for Curve-Solid Classification in Geometric Modelling" Technical Report TR84-37, Microelectronic Center of North Carolina, Research Triangle Park, N.C., 1984.

[5] Okino, N., Y. Kakazu, M. Morimoto. "Extended Depth Buffer Algorithms for Hidden Surface Visualization" *IEEE Computer Graphics and Applications*, pp. 79–88, May, 1984.

[6] Poulton, J., H. Fuchs, J.D. Austin, J.G. Eyles, J. Heinecke, C. Hsieh, J. Goldfeather, J.P. Hultquist, and S. Spach. "PIXEL-PLANES: Building a VLSI Based Raster Graphics System" *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pp. 35–60.

[7] Requicha, A.A.G. "Representation for Rigid Objects: Theory, Methods, and Systems" *ACM Computing Surveys*, Vol. 12, No. 4, Dec. 1980, pp. 437–464.

[8] Sato, H., H. Ishihata, M. Ishii, M. Kakimoto, K. Sato, K. Hirota, M. Ikesaka, K. Inoue. "Fast Image Generation of Constructive Solid Geometry Using A Cellular Array Processor" *Computer Graphics*, Vol. 19, No. 3, pp. 95–102, 1985. (Proceedings of SIGGRAPH '85)