PAR: A New Representation Scheme for Rotational Parts *

Yung-Chia Lee and Kuen-Fang Jack Jea

Department of Electrical Engineering and Computer Science

The University of Michigan, Ann Arbor, MI 48109-1109

July 28, 1986

Abstract

A new representation scheme PAR (Principal Axis Representation) for rotational parts is proposed as an internal representation scheme for Constructive Solid Geometry (CSG). The key idea of PAR is to represent an object by its principal axis and a set of boundary curves. Based on a mathematical framework, an algorithm is designed to convert a CSG tree into a PAR, which represents the same object as the CSG tree does but is in an evaluated form. Geometrical properties of parts can then be computed more directly and efficiently from this evaluated representation than from the original CSG tree. In addition to its computational efficiency, the PAR is a *unique* representation scheme.

*This work is supported in part by the NSF Grant No. DMC-8504790 and by the SME Grant No. 585-895.

Contents

1 Introduction **1**

2 The Problem and Proposed Approach **3**

2.1 Problem 3

2.2 Basic Ideas 4

3 Formulation of Principal Axis Representation **5**

3.1 Terminology 5

3.2 Represent Primitive Solids by PAR 8

3.3 Operations on PAR 9

3.4 Relation between PAR and CSG 12

4 Algorithm to Convert CSG Representation to PAR **17**

4.1 Data Structure for PAR 17

4.2 Conversion from CSG Representation to PAR 17

4.3 Union and Difference of Two PAR's 19

5 Illustrative Examples **27**

6 Conclusion and Discussion **29**

6.1 Conclusion 29

6.2 Discussion and Future Work 30

1 Introduction

Solid modeling techniques have been considered as one of the keys to the integration of computer-aided design and manufacturing (CAD/CAM)[Bee82]. Two of the prevailing geometric schemes in solid modeling are ***Boundary Representation*** (B-rep) scheme and ***Constructive Solid Geometry*** (CSG) scheme[ReV82]. B-rep represents an object by segmenting its boundary into a finite number of bounded subsets usually called faces or patches, and representing each face by (for example) its bounding edges and vertices. CSG scheme represents objects as constructions or combinations, via the regularized set operators, of solid components. While B-rep in general is in an evaluated form, which explicitly shows what and where each geometric entity is, CSG is not. The much more concise CSG is easy to construct and to check the validity of the object being constructed. The tradeoff between the evaluatedness and conciseness seems to be clear but the choice between B-rep and CSG has never been easy.

As a means to integrate design and manufacturing, solid modeling techniques not only display objects but also compute important properties of objects. A more comprehensive understanding of the geometric representation by the computer becomes so important that shape features can then be extracted, parts classified, processes planned, and etc.. Several interesting researches have been done along the direction of enhancing machine's understanding of part geometric representations. Woo[Woo77] focused on cavity recognition in studying problems of transforming volumetric designs of parts into numerical control (NC) descriptions. Staley et al.[StHA83], without specific application emphasized, also dealt with cavity recognition but using syntactic pattern recognition techniques. Grayer[Gra76] compares the part in B-rep with its initial work part in order to compute NC tool paths. In addition to NC path generation, Armstrong[ArCP84] also considered the determination of fixture orientation. For global understanding of part geometry, Woo[Woo82] suggested a convex hull technique which transforms a boundary representation into an expression of alternating sum of volumes. To generate a part code for group technology, Kyprianou[Kyp80] used syntactic pattern recognition techniques to extract features characterized by protrusions and depressions. Jakubowski[Jak82] also used syntactic methods to develop a part description language primarily for the determination of part shapes. Henderson[Hen84] was able to use PROLOG and expert system techniques to extract features and organize them in a feature graph so as to facilitate automatic process planning.

While most of the work mentioned above is based on B-rep, very little has been done on CSG. In CSG, a machine part is represented as a tree with some primitive solids as terminal nodes and the regularized set operations (*union* and *difference*) and *movement* operations as non-terminal nodes. Properties of the machine part can be computed by evaluating the CSG tree. The evaluation procedure is basically a tree traversal routine that combines CSG subtrees into higher level CSG subtrees. The algorithm is simple but the computation involved in calculating the intersection (the major computation of *union* and *difference* operations) of two solids are complex and time-consuming.

Without evaluation, Lee and Fu[LeK86] proposed an approach to the problem of feature extraction. The approach is based on the spatial relationships among axes of primitive solids. They also suggested a unification process using tree reconstruction to unify feature representations so that properties can be computed more easily. As more general cases for this approach are yet to be studied, there are algorithms that convert CSG into B-rep[BoG82]. Not surprisingly, such algorithms are often called for when certain properties must be computed from a CSG tree[KaO84]. However, the conversion often appears to be computationally expensive or, being worse, a waste to simply compute the specific property.

There are part families for which CSG should be an excellent representation scheme and, furthermore, a transformation into B-rep can actually be avioded. A such example, rotational parts, is investigated in this study. Instead of B-rep, a new representation scheme PAR (Principal Axis Representation) is proposed as an internal representation scheme for rotational parts that are initially described by CSG. The key idea of PAR is to represent an object by a principal axis and a set of boundary curves. Based on a mathematical framework, an algorithm is designed and implemented to convert a CSG tree into a PAR, which represents the same object as the CSG tree does but is in an evaluated form. From PAR, the profile of the part can be efficiently computed. Other properties such as length and maximum diameter can also be obtained easily.

The conversion algorithm is much simpler than the one converting CSG into B-rep. The main reason is that one dimensional curves (e.g. lines and arcs) are used to characterize the rotational parts and the evaluation of PAR is performed on two dimensional space, e.g. the intersection of two lines on the same plane rather than that of two solids on a three dimensional space. In addition to its computational ease, the PAR is proved to be a unique representation scheme. Being unique, it facilitates feature definition and thus simplifies the tasks of feature extraction. The possible extension of PAR is discussed in Section 6.

2 The Problem and Proposed Approach

2.1 Problem

Rotational parts include all parts that are symmetrical with respect to their principal axes. In this study, however, the primitive solids are limited to cylinders, cones, and tori only. The problem is formulated as follows:
Given a CSG tree of a machine part, which is

1. axis-symmetrical, and

2. constructed from cylinders, cones and tori in an arbitrary order of combination such as union and difference.

can we:

1. by utilizing the property of axis-symmetry, efficiently derive its profile and some geometric properties such as length and diameter so that part classification and process planning can possibly be supported?

2. develop an internal scheme which not only supports the above computation but also represents the part uniquely?

2.2 Basic Ideas

If a machine part is axis-symmetrical, it can be viewed as a $2\frac{1}{2}$D object and represented by the union of components, each being generated by rotating its boundary curve segments with respect to the principal axis. For example, a cylinder can be represented by rotating a line segment around its axis where the axis and the line segment are parallel to each other and their distance is the radius of the cylinder. Other primitive solids like cones and tori can also be represented in a similar way. Since the machine part is axis-symmetrical, each component solid depicted by a CSG subtree can also be represented by a collection of principal axis segments, each being associated with several pairs of boundary curves. For each boundary curve pair, rotating it around the associated axis segment would generate a volume layer for the corresponding component solid.

This idea of representing axis-symmetrical machine parts by a principal axis and a set of bounded curves can be exploited to develop efficient algorithms to evaluate CSG trees and deduce their geometrical properties because the computational complexity of the algorithms can be reduced significantly. For example, in the course of evaluating CSG trees, instead of testing the intersection of two three dimensional solids, we need only to test the intersection of a few one dimensional curves on the same plane, where the number of the 1D curves needed to be tested depends on the complexity of the two original 3D solids involved.

To evaluate a CSG tree and convert it into this kind of representation, new operations such as *union*, *difference*, and *movement* operated on this new representation scheme must be defined in order to maintain the actual semantics as their counterpart operations do in the CSG tree representation scheme. To deal with the *movement* operation is easy; if all the coordinates of the segments and boundary curves are relative to a principal axis, then the result of the *movement* operation is merely applying the *movement* transformation to the principal axis, and others remain unchanged. But the *union* and *difference* operations are not so simple to deal with, as more elaborate processing would be required.

After a CSG tree is converted to the new representation, the profile and other geometric properties of the corresponding part can be easily computed. These properties may be computed all at the same time after the CSG tree is converted into the new representation. To compute the desired geometric properties directly from the CSG tree, on the other hand, is complex and time-consuming.

In summary, the proposed approach includes the following:

1. To define a new representation scheme (Principal Axis Representation) for axis-symmetrical machine parts

2. To convert CSG trees into their new representations.

3. To derive desired geometrical properties from this new representation, and it is believed to be easier and more efficient.

4. To prove that the Principal Axis Representation is a unique representation scheme for axis-symmetrical machine parts.

3 Formulation of Principal Axis Representation

3.1 Terminology

Definition: Principal Axis

A *Principal Axis* A of an axis-symmetrical object O is a line segment in the three dimensional space such that the object O is represented in a way that starts from and ends at the two end points of A, and O is symmetrical with respect to A.

Definition: Axis Segment

An *Axis Segment* S is a line segment (i.e. subset) of a Principal Axis which has two end points.

Definition: Principal Axis Coordinate System

A curve C and a principal axis A form a coordinate system if they are co-plane and, on this plane, the horizontal axis is the line containing A and its vertical axis is the line perpendicular to A and passing through the start point of A. This 2-D coordinate system is called ***principal axis coordinate system.***

Definition: Bounded Curve

A curve C is bounded at [a b] with respect to a principal axis A iff in the principal axis coordinate system formed from C and A, there exist two real numbers a and b (a≤b) such that the curve C is differentiable within the interval (a b). a and b are called the *bound points* of the curve C , and [a b] is called a *pair of bounds* of C.

Definition: Range of a Bounded Curve

If C is a bounded curve within [a b] with respect to a principal axis A, the *range* of C at x , x ∈ [a b], on the principal axis coordinate system is C(x), i.e. the mapping of C from x.

Definition: Arc

An *Arc* is a bounded curve which is also a subset of a circle.

Note that : a line segment is a bounded curve.

Definition: Bounded Curve Set

A *Bounded Curve Set* is a set of *Arcs* and/or *Line Segments.*

Definition: Principal Axis Representation (PAR)

If an object O is an axis-symmetrical machine part which can be characterized by a Principal Axis A, then its *Principal Axis Representation* PAR(O, A) can be defined as a set of tuples (S_i, C_i) , i = 1,.., n , n ∈ N, such that

1. S_i is an Axis Segment, and $S_i \subseteq$ A, $\bigcup_{i=1}^{n} S_i =$ A, $\exists i, j \leq n$, $S_i \cup S_j$ is either a point (i.e. their common bound point) if $j = i + 1$ or $\emptyset$, otherwise; that is, { S_i } is a partition of A.

2. $C_i = \{ C_{ij} \mid j = 1, .., 2m_i \}$ is a Bounded Curve Set.

3. $\exists a, b \in R$ such that all C_{ij} 's and S_i have the same bound [a b] with respect to A.

4. all C_{ij} 's do not intersect within their bound interval (a b).

5. all C_{ij} 's are differentiable (i.e. their first derivatives exist) within (a b), and $\exists$ a curve $C_{ik} \in C_i$ such that C_{ik} is not differentiable at the bound point a, and $\exists$ a curve $C_{il} \in C_i$ such that C_{il} is not differentiable at the bound point b.

Theorem 1:

For a Principal Axis Representation of an axis-symmetrical machine part O and its Principal Axis A, PAR(O, A) = { (S_i, C_i) | i=1,..,n }. $\forall i \leq n$, all C_{ij} 's ($C_{ij} \in C_i$, j = 1,..,$2r_i$), within (a b) which is the bound of S_i with respect to A, form a *total ordering.*

Proof: let's denote C_{ij}(x) to be the range of the bounded curve C_{ij} at x, x $\in$ (a b).

For C_{ik} and $C_{im} \in C_i$, $k \neq m$, $\exists x \in$(a b), C_{ik}(x) and C_{im}(x) $\in$ R, and $C_{ik}(x) \neq C_{im}(x)$; otherwise C_{ik} and C_{im} intersect at x within (a b) and this contradicts to the definition of PAR(O, A). Therefore either $C_{ik}(x) < C_{im}(x)$ or $C_{ik}(x) > C_{im}(x)$ is true. And we prove that all C_{ij}(x), $C_{ij} \in C_i$, form a total ordering at some x, x $\in$ (a b).

Next, we will prove that the total orderings of all C_{ij} (j = 1, 2, .., $2r_i$) at all x, x $\in$ (a b) are consistent.

For x_1 and $x_2 \in$ (a b), $x_1 \neq x_2$, if the total ordering of all C_{ij} (j = 1, 2, .., $2r_i$) at x_1 and that of all C_{ij} (j = 1, 2, .., $2r_i$) at x_2 are not consistent, then there must exist two different curves, C_{ik} and C_{il}, such that $C_{ik}(x_1) < C_{il}(x_1)$ and $C_{ik}(x_2) > C_{il}(x_2)$. It means that there must exist some value x_3 between x_1 and x_2 such that C_{ik} and C_{il} intersect at x_3. This contradicts to the definition of PAR. Therefore the total orderings of all C_{ij} (j = 1, 2, .., $2r_i$) at all x, x $\in$ (a b), must be consistent, and we may conclude that all C_{ij} (j = 1, 2, .., $2r_i$) within (a b) form a unique total ordering. **Q.E.D.**

Note: this total ordering will be used to determine the relation among those curves within some bound.

Using the concept of Theorem 1, we may define the concept of a layer.

Definition: Layer and Layer Set

For PAR(O, A), S_i is an Axis Segment and C_i is its associated bounded curve set. C_{ij} 's (j =1,..,$2m_i$) are those bounded curves in C_i and they are sorted in such a sequence : $C_{i1} > C_{i2} > C_{i3} > .. > C_{i2m_i}$. Then a *layer* within this Axis Segment S is defined as a pair of curves C_{ik} and C_{ik+1} ,denoted as $[C_{ik}C_{ik+1}]$, k =1, 3,..., $2m_i$-1. The set of layers { $[C_{ik}C_{ik+1}]$ | k =1, 3,..., $2m_i$-1 } is called a *layer set.*

Note: a layer with its bound actually generates a volume of the object by rotating the curves in the layer with respect to its corresponding Axis Segment.

3.2 Represent Primitive Solids by PAR

1. Use PAR to represent cylinder

PAR(cylinder, A) = { (S, C) | C = { C_1, C_2 }, C_1 and C_2 are line segments; S = A = C_2; $C_1 \parallel C_2$; | $C_1 - C_2$ | = radius of the cylinder; [a b] is the bound for S and C where a and b are the horizontal coordinates of the starting and ending points, respectively, of A in the Principal Axis Coordinate System formed by A and C_1. }

Notice that the statement "$C_1 \parallel C_2$" means that the line segment C_1 is parallel to the line segment C_2, and the statement "| $C_1 - C_2$ |" denotes the distance between C_1 and C_2.

2. Use PAR to represent cone

PAR(cone, A) = { (S, C) | C = { C_1, C_2 }, C_1 and C_2 are line segment; S = A = C_2; [a b] is the bound for S and C where a and b are the horizontal coordinates of the starting and ending points, respectively, of A in the Principal Axis Coordinate System formed by A and C_1; C_1 and C_2 intersect at b, and the distance of C_1 and C_2 at a is the radius of the cone. }

3. Use PAR to represent torus

PAR(torus, A) = { (S, C) | S = A; C = { C_1, C_2 }, [a b] is the bound for S and C where a and b are the horizontal coordinates of the starting and ending points, respectively, of A in the Principal Axis Coordinate System formed by A and C; C_1, C_2 are Arcs of the circle with the radius equal to the *local radius* of the torus and with its center located at (x

y) in the Principal Axis Coordinate System where $x = (a+b)/2$, and y is the *global radius* of the torus, C_1 is the upper half and C_2 is the lower half of the circle. }

3.3 Operations on PAR

Definition: Overlap of Two Layers

The *overlap* of two layers $[c_{11}c_{12}]$ and $[c_{21}c_{22}]$ with respect to some bound [a b] is defined as either a layer $[c_{31}c_{32}]$ with respect to [a b] if $c_{31} = \min(c_{11}, c_{21}), c_{32} = \max(c_{12}, c_{22})$ and $c_{31} > c_{32}$; or $\emptyset$, otherwise.

Definition: Layer-Union of Two Layers

The *layer-union* of two layers $[c_{11}c_{12}]$ and $[c_{21}c_{22}]$ with respect to some bound [a b] is defined as either the layer $[c_{31}c_{32}]$ where $c_{31} = \max(c_{11}, c_{21}), c_{32} = \min(c_{12}, c_{22})$ if the overlap of $[c_{11}c_{12}]$ and $[c_{21}c_{22}]$ is not $\emptyset$; or the two layers $[c_{11}c_{12}]$ and $[c_{21}c_{22}]$, otherwise. The result layer or layers are with respect to the same bound[a b].

Definition: Layer-Union of Two Layer Sets

The *layer-union* of *two layer sets* L_1 and L_2 is defined as a layer set which is the *set union* of all layer-union's of two layers, one from L_1 and the other from L_2, respectively. L_1, L_2 and the resulting layer set are all with respect to the same bound.

Definition: Maximum Union-able Layer Set

The *maximum union-able layer set* of two layer sets L_1 and L_2 is defined as a layer set which is the transitive closure of applying the layer-union operation to the resulting layers of the layer-union of L_1 and L_2; that is, if L_3 is the layer-union of L_1 and L_2, then the *maximum union-able layer set* of L_1 and L_2 is the results of repeatedly applying the layer-union operation to the layers in L_3 until the resulting set is no more changed. L_1, L_2 and the maximum union-able layer set are all with respect to the same bound.

Definition: Union of Two PAR's

For two PAR's, PAR(O_1, A_1) and PAR(O_2, A_2), if A_1 and A_2 are subsets of a line, then the *union* of PAR(O_1, A_1) and PAR(O_2, A_2) is defined as a mapping from PAR × PAR to PAR such that if PAR(O_1, A_1) = { (S_i^1, C_i^1) | i = 1,..,n_1, $[a_i^1 b_i^1]$ is the bound of S_i^1 }, PAR(O_2, A_2) = { (S_i^2, C_i^2) | i = 1,..,n_2 , $[a_i^2 b_i^2]$ is the bound of S_i^2 }, and there exists a PAR(O_3, A_3) = { (S_i^3, C_i^3) | i = 1,..,n , $[a_i^3 b_i^3]$ is the bound of S_i^3 }, then

1. $O_3 = O_1 \cup O_2$; $A_3 = A_1 \cup A_2$;
2. S_i^3 is a line segment of A_3 with bounds $[a_i^3 b_i^3]$, i=1,..,n; and there exists a set of bounds $[d_m^3 e_m^3]$, m = 1, 2, ..,n_3 ($n_3 \geq n$) which is a partition of the set of bounds $[a_i^3 b_i^3]$ and there exists a bounded curve set K_m^3 for each bound $[d_m^3 e_m^3]$ where
   $K_m^3 = C_j^1$ with adjusted bound $[d_m^3 e_m^3]$ if $\nexists\ [a_l^2 b_l^2] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_2$
   $K_m^3 = C_k^2$ with adjusted bound $[d_m^3 e_m^3]$ if $\nexists\ [a_l^1 b_l^1] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_1$
   K_m^3 = maximum union-able layer set of C_j^1 and C_k^2 with respect to a bound $[d_m^3 e_m^3]$ if $[a_j^1 b_j^1] \supseteq [d_m^3 e_m^3]$, and $[a_k^2 b_k^2] \supseteq [d_m^3 e_m^3]$ for some j, k.
3. $C_i^3 = \bigcup_{m=x_i}^{y_i} K_m^3$,(x_i, y_i are integers) if $[d_m^3 e_m^3]$ for m = x_i,..., y_i is a partition of $[a_i^3 b_i^3]$

NOTE: This mapping is from PAR × PAR to PAR, the non-differentiable properties of curves across the bound $[a_i^3 b_i^3]$ for the resulting PAR should be preserved by the definition of PAR.

Definition: Layer-Difference of Two Layers

The *layer-difference* of two layers $[c_{11} c_{12}]$ and $[c_{21} c_{22}]$ with respect to some bound [a b] is defined as

1. the layer $[c_{11} c_{12}]$ if $c_{11} \leq c_{22}$ or $c_{12} \geq c_{21}$;
2. the layer $[c_{22} c_{12}]$ if $c_{21} \geq c_{11} \geq c_{22} > c_{12}$;
3. the layer $[c_{11} c_{21}]$ if $c_{11} > c_{21} \geq c_{12} \geq c_{22}$;
4. the layers $[c_{11} c_{21}]$ and $[c_{22} c_{12}]$ if $c_{11} > c_{21} > c_{22} > c_{12}$;

5. $\emptyset$ if $c_{21} \geq c_{11} > c_{12} \geq c_{22}$;

The result layer or layers is with respect to the same bound[a b].

Definition: Layer-Difference of Two Layer Sets

The *layer-difference* of *two layer sets* L_1 and L_2 (assume L_1 = { $[c_i c_{i+1}]$ | i = 1, 3,..., 2n-1 } is the first operand and L_2 = { $[d_i d_{i+1}]$ | i = 1, 3,..., 2m-1 } is the second operand) is defined as a layer set which is the *set union* of all the set OV_k (k = 1, 2,..., n) where each OV_k is the overlap of all the layers l_{kj} (j = 1, 2,..., m), l_{kj} = the layer-difference of $[c_{2k-1}c_{2k}]$ to $[d_{2j-1}d_{2j}]$. L_1, L_2 and the resulting layer set are all with respect to the same bound.

Definition: Minimum Difference-able Layer Set

The *minimum difference-able layer set* of two layer sets L_1 and L_2 with respect to some bound [a b] is the *layer-difference* of L_1 to L_2 with respect to the same bound [a b].

Definition: Difference of Two PAR's

For two PAR's, PAR(O_1, A_1) and PAR(O_2, A_2), if A_1 and A_2 are subsets of a line, then the *difference* of PAR(O_1, A_1) and PAR(O_2, A_2) is defined as a mapping from PAR × PAR to PAR such that if PAR(O_1, A_1) = { (S_i^1, C_i^1) | i = 1,..,n_1, $[a_i^1 b_i^1]$ is the bound of S_i^1 }, PAR(O_2, A_2) = { (S_i^2, C_i^2) | i = 1,..,n_2 , $[a_i^2 b_i^2]$ is the bound of S_i^2 }, and there exists a PAR(O_3, A_3) = { (S_i^3, C_i^3) | i = 1,..,n , $[a_i^3 b_i^3]$ is the bound of S_i^3 }, then

1. $O_3 = O_1 - O_2$; $A_3 \subseteq A_1$;

2. S_i^3 is a line segment of A_3 with bounds $[a_i^3 b_i^3]$, i=1,..,n; and there exists a set of bounds $[d_m^3 e_m^3]$, m = 1, 2, ..,n_3 ($n_3 \geq n$) which is a partition of the set of bounds $[a_i^3 b_i^3]$ and there exists a bounded curve set K_m^3 for each bound $[d_m^3 e_m^3]$ where
$K_m^3 = C_j^1$ with adjusted bound $[d_m^3 e_m^3]$ if $\nexists$ $[a_l^2 b_l^2] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_2$
$K_m^3 = \emptyset$ if $\nexists$ $[a_l^1 b_l^1] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_1$
K_m^3 = minimun difference-able layer set of C_j^1 and C_k^2 with respect to a bound $[d_m^3 e_m^3]$ if $[a_j^1 b_j^1] \supseteq [d_m^3 e_m^3]$, and $[a_k^2 b_k^2] \supseteq [d_m^3 e_m^3]$ for some j, k.

3. $C_i^3 = \bigcup_{m=x_i}^{y_i} K_m^3$, (x_i, y_i are integers) if $[d_m^3 e_m^3]$ for m = x_i,..., y_i is a partition of $[a_i^3 b_i^3]$.

Definition: Movement of a PAR

The *movement* of a PAR(O, A) is defined as a mapping from PAR to PAR such that the new PAR(O,B) after movement is the same as PAR(O,A) except that A is transformed to B in 3-D space by applying the movement operation specified.

Note: The bounds in PAR are with respect to A ,and therefore movement-invariant.

3.4 Relation between PAR and CSG

In this subsection, the relation between CSG and PAR representation schemes is investigated. It is the basis of our algorithm to convert a CSG tree into a PAR.

Theorem 2.

For a CSG tree with *union* as its root, the object represented by this CSG tree is identical to that represented by a PAR which is the *union* of two PAR's, each corresponding to either the left- or right- subtree of the original CSG tree.

Proof : Assume that $O_3 = O_1 \cup O_2$ in CSG domain where O_3 is the CSG tree, and O_1 and O_2 are its left- and right- subtrees, respectively. In the PAR domain, assume O_1 and O_2 are represented by PAR(O_1,A_1) and PAR(O_2,A_2), respectively. Also assume $A_3 = A_1 \cup A_2$.

Now we want to prove PAR(O_1,A_1)$\cup$PAR(O_2,A_2) represents O_3. In this proof, we may view an object as a set of points in 3-D space and use the notation V(layers) to denote the volume generated by rotating the layers with respect to a principal axis in 3-D space.

Step 1: To prove $O_3 \subseteq$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))).

For every point p of O_3, p $\in O_1$ or p $\in O_2$ is true in CSG domain because $O_3 = O_1 \cup O_2$. But O_1 and O_2 can be represented by PAR(O_1,A_1) and PAR(O_2,A_2), respectively. Therefore, p $\in$ V(layers of (PAR(O_1,A_1)) or p $\in$ V(layers of (PAR(O_2,A_2)). More specifically, on the principal axis coordinate system, the horizontal coordinate of p must lie

5. $\emptyset$ if $c_{21} \geq c_{11} > c_{12} \geq c_{22}$;

The result layer or layers is with respect to the same bound[a b].

Definition: Layer-Difference of Two Layer Sets

The *layer-difference* of *two layer sets* L_1 and L_2 (assume $L_1 = \{ [c_i c_{i+1}] \mid$ i = 1, 3,..., 2n-1 $\}$ is the first operand and $L_2 = \{ [d_i d_{i+1}] \mid$ i = 1, 3,..., 2m-1 $\}$ is the second operand) is defined as a layer set which is the *set union* of all the set OV_k (k = 1, 2,..., n) where each OV_k is the overlap of all the layers l_{kj} (j = 1, 2,..., m), l_{kj} = the layer-difference of $[c_{2k-1}c_{2k}]$ to $[d_{2j-1}d_{2j}]$. L_1, L_2 and the resulting layer set are all with respect to the same bound.

Definition: Minimum Difference-able Layer Set

The *minimum difference-able layer set* of two layer sets L_1 and L_2 with respect to some bound [a b] is the *layer-difference* of L_1 to L_2 with respect to the same bound [a b].

Definition: Difference of Two PAR's

For two PAR's, PAR(O_1, A_1) and PAR(O_2, A_2), if A_1 and A_2 are subsets of a line, then the *difference* of PAR(O_1, A_1) and PAR(O_2, A_2) is defined as a mapping from PAR × PAR to PAR such that if PAR(O_1, A_1) = $\{ (S_i^1, C_i^1) \mid$ i = 1,..,n_1, $[a_i^1 b_i^1]$ is the bound of S_i^1 $\}$, PAR(O_2, A_2) = $\{ (S_i^2, C_i^2) \mid$ i = 1,..,n_2 , $[a_i^2 b_i^2]$ is the bound of S_i^2 $\}$, and there exists a PAR(O_3, A_3) = $\{ (S_i^3, C_i^3) \mid$ i = 1,..,n , $[a_i^3 b_i^3]$ is the bound of S_i^3 $\}$, then

1. $O_3 = O_1 - O_2$; $A_3 \subseteq A_1$;

2. S_i^3 is a line segment of A_3 with bounds $[a_i^3 b_i^3]$, i=1,..,n; and there exists a set of bounds $[d_m^3 e_m^3]$, m = 1, 2, ..,n_3 ($n_3 \geq n$) which is a partition of the set of bounds $[a_i^3 b_i^3]$ and there exists a bounded curve set K_m^3 for each bound $[d_m^3 e_m^3]$ where
   $K_m^3 = C_j^1$ with adjusted bound $[d_m^3 e_m^3]$ if $\nexists$ $[a_l^2 b_l^2] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_2$
   $K_m^3 = \emptyset$ if $\nexists$ $[a_l^1 b_l^1] \supseteq [d_m^3 e_m^3]$, l = 1,..,$n_1$
   K_m^3 = minimun difference-able layer set of C_j^1 and C_k^2 with respect to a bound $[d_m^3 e_m^3]$ if $[a_j^1 b_j^1] \supseteq [d_m^3 e_m^3]$, and $[a_k^2 b_k^2] \supseteq [d_m^3 e_m^3]$ for some j, k.

3. $C_i^3 = \bigcup_{m=x_i}^{y_i} K_m^3$, (x_i, y_i are integers) if $[d_m^3 e_m^3]$ for m = x_i,..., y_i is a partition of $[a_i^3 b_i^3]$.

Definition: Movement of a PAR

The *movement* of a PAR(O, A) is defined as a mapping from PAR to PAR such that the new PAR(O,B) after movement is the same as PAR(O,A) except that A is transformed to B in 3-D space by applying the movement operation specified.

Note: The bounds in PAR are with respect to A ,and therefore movement-invariant.

3.4 Relation between PAR and CSG

In this subsection, the relation between CSG and PAR representation schemes is investigated. It is the basis of our algorithm to convert a CSG tree into a PAR.

Theorem 2.

For a CSG tree with *union* as its root, the object represented by this CSG tree is identical to that represented by a PAR which is the *union* of two PAR's, each corresponding to either the left- or right- subtree of the original CSG tree.

Proof : Assume that $O_3 = O_1 \cup O_2$ in CSG domain where O_3 is the CSG tree, and O_1 and O_2 are its left- and right- subtrees, respectively. In the PAR domain, assume O_1 and O_2 are represented by PAR(O_1,A_1) and PAR(O_2,A_2), respectively. Also assume $A_3 = A_1 \cup A_2$.

Now we want to prove PAR(O_1,A_1)∪PAR(O_2,A_2) represents O_3. In this proof, we may view an object as a set of points in 3-D space and use the notation V(layers) to denote the volume generated by rotating the layers with respect to a principal axis in 3-D space.

Step 1: To prove $O_3 \subseteq$ V(layers of (PAR(O_1,A_1) ∪ PAR(O_2,A_2))).

For every point p of O_3, p $\in O_1$ or p $\in O_2$ is true in CSG domain because $O_3 = O_1 \cup O_2$. But O_1 and O_2 can be represented by PAR(O_1,A_1) and PAR(O_2,A_2), respectively. Therefore, p $\in$ V(layers of (PAR(O_1,A_1)) or p $\in$ V(layers of (PAR(O_2,A_2)). More specifically, on the principal axis coordinate system, the horizontal coordinate of p must lie in some bounds $[a_i b_i]$, which is a subset of $[a_j^1 b_j^1]$ of PAR(O_1,A_1) or a subset of $[a_k^2 b_k^2]$ of PAR(O_2,A_2) for some j or k.

1. if $[a_i b_i]$ is a subset of $[a_j^1 b_j^1]$ but not a subset of $[a_k^2 b_k^2]$, then p $\in$ V(layers specified by C_j^1 of PAR(O_1,A_1)).

2. if $[a_i b_i]$ is a subset of $[a_k^2 b_k^2]$ but not a subset of $[a_j^1 b_j^1]$, then p $\in$ V(layers specified by C_k^2 of PAR(O_2,A_2)).

3. if $[a_i b_i]$ is both a subset of $[a_k^2 b_k^2]$ and a subset of $[a_j^1 b_j^1]$, then p $\in$ V($[c_m c_{m+1}]$) or p $\in$ V($[d_n d_{n+1}]$) where $[c_m c_{m+1}]$ and $[d_n d_{n+1}]$ are some layers of PAR(O_1,A_1) and PAR(O_2,A_2), respectively. By the definition of *layer-union* of two layers and *Maximum Union-able Layer Set*, p $\in$ V(layer-union of $[c_m c_{m+1}]$ and $[d_n d_{n+1}]$), and therefore p $\in$ V(maximum union-able layer set of L_1 and L_2) where L_1 and L_2 are the layer sets with the bound $[a_i b_i]$ that contain $[c_m c_{m+1}]$ and $[d_n d_{n+1}]$, respectively.

Thus, from the definition of the *union* of two PAR's, p $\in$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))). And we prove that $O_3 \subseteq$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))).

Step 2: To prove $O_3 \supseteq$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))).

For every point p, p $\in$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))), by the definition of the *union* of two PAR's, there are three possible cases for p:

1. p $\in$ V($[c_m c_{m+1}]$) where $[c_m c_{m+1}]$, which is bounded by some bound [a b] $\subseteq [a_i^1 b_i^1]$ but $\not\subseteq [a_j^2 b_j^2]$, is a layer of PAR($O_1$,$A_1$). In this case, since PAR(O_1,A_1) represents O_1, p $\in O_1$ and therefore p $\in O_1 \cup O_2 = O_3$.

2. p $\in$ V($[d_n d_{n+1}]$) where $[d_n d_{n+1}]$, which is bounded by some bound [a b] $\subseteq [a_j^2 b_j^2]$ but $\not\subseteq [a_i^1 b_i^1]$, is a layer of PAR($O_2$,$A_2$). In this case, since PAR(O_2,A_2) represents O_2, p $\in O_2$ and therefore p $\in O_1 \cup O_2 = O_3$.

3. p $\in$ V(maximum union-able layer set of L_1 and L_2) where L_1 and L_2 are the sets of layers of PAR(O_1,A_1) and PAR(O_2,A_2) at some bounds [a b], respectively. By the definition of the maximum union-able layer set, there are three possible cases for p:

(a) p $\in$ V($[c_m c_{m+1}] \subseteq L_1$ but $\not\subseteq L_2$). Thus, p $\in O_1 \subseteq (O_1 \cup O_2) = O_3$.

(b) p $\in$ V($[d_n d_{n+1}] \subseteq L_2$ but $\not\subseteq L_1$). Thus, p $\in O_2 \subseteq (O_1 \cup O_2) = O_3$.

(c) p $\in$ V($[e_l e_{l+1}]$) where $[e_l\ e_{l+1}] \subseteq (L_1 \cap L_2)$. In this case, p $\in$ (V(L_1) $\cap$ V(L_2)). That is p $\in (O_1 \cap O_2) \subseteq (O_1 \cup O_2) = O_3$.

Therefore we prove that $O_3 \supseteq$ V(layers of (PAR(O_1,A_1) $\cup$ PAR(O_2,A_2))).

From Step 1 and Step 2, we prove the theorem. **Q.E.D.**

Notice that the union operator on PAR performs the actual semantics of the union operator on CSG.

Theorem 3.

For a CSG tree with *difference* as its root, the object represented by this CSG tree is identical to that represented by a PAR which is the *difference* of two PAR's, which correspond to the left- and right- subtrees of the original CSG tree.

Proof : Instead of using the concept of *maximum union-able layer set* and the definition of the *union* of two PAR's, the concept of *minimum difference-able layer set* and the definition of the *difference* of two PAR's may be used here to prove this theorem in a similar way as in **Theorem 2**. **Q.E.D.**

Notice that the difference operator on PAR performs the actual semantics of the difference operator on CSG.

Theorem 4.

For a CSG tree with *movement* as its root, the object represented by this CSG tree is identical to that represented by a PAR which is the *movement* of a PAR, which corresponds to the subtree of the original CSG tree.

Proof : By the definition of *movement* of a PAR, the object represented by the PAR after applying a *movement* operation is the same as the object represented by the opiginal PAR (i.e. the structure of the object is not changed) except that the location and orientation of the object are changed. In the CSG representation scheme, an object represented by a CSG subtree has the same structure as the object represented by the CSG tree after applying a *movement* operation to the original CSG subtree except these two objects have different location and orientation in 3-D space.

Therefore, after the same transformation matrix (i.e. *movement* operation) is applied to both the PAR and the original CSG subtree which represent the same object, the resulting objects should be identical; that is, they not only have the same structure but also have the same location and orientation in the 3-D space. **Q.E.D.**

Notice that the movement operator on PAR performs the actual semantics of the movement operator on CSG.

Theorem 5.

An axis-symmetrical machine part represented by a CSG tree can be evaluated on PAR domain and the final resulting PAR after evaluation represents the same object as the CSG tree represents.

Proof : A CSG tree is composed of operators (*union, difference, movement*) as non-terminal nodes and primitive solids (*cylinder, cone, torus*) as terminal nodes in this study. It is shown in Section 3.2 that these primitive solids can be represented by their corresponding PAR's. If the CSG tree is converted in bottom-up fashion from leaves to root, by Theorem 2, 3 and 4, the final resulting PAR should represent the identical object as the CSG tree represents. **Q.E.D.**

Notice that by this theorem, a given CSG tree can be evaluated in PAR domain to obtain an identical object as the CSG tree represents. This theorem is the theoretical basis of our algorithms in Section 4.

Theorem 6. Uniqueness

Any axis-symmetrical machine part has a *unique* PAR representation.

Proof : Our theorem may be rephrased as follows:

For two representations, PAR(O_1,A_1) and PAR(O_2,A_2), if $O_1 = O_2$ then PAR(O_1,A_1) = PAR(O_2,A_2).

Assume that the object O_1 (or O_2) is in some 3-D coordinate system. Then there must be a unique Principal Axis for the object in this coordinate system, thus $A_1 = A_2$.

Our proof procedure consists of two steps:

Step 1: To prove PAR(O_1,A_1) and PAR(O_2,A_2) have the same Axis Segment bounds with respect to A_1 and A_2, respectively. That is, if PAR(O_1,A_1) = { (S_i^1,C_i^1) | i =1,..,n; S_i^1 is bounded by $[a_i^1\ b_i^1]$ }, PAR(O_2,A_2) = { (S_i^2,C_i^2) | i =1,..,m; S_i^2 is bounded by $[a_i^2\ b_i^2]$ } then we want to prove n = m, and a_i^1=a_i^2 and b_i^1=b_i^2 , for all i= 1,..,n.

By the definition of PAR, all the curves within a bound must be differentiable and there exists at least one curve non-differentiable at one of its lower and upper bounds. This means that the bounds on the Principal Axis are uniquely defined by the discontinuity of the first derivatives of the curves specifying the object. Because the object has fixed shape and the bounded curve to specify some part of the shape of an object can be either an arc or a line segment but not both, the bounded curve to describe the object shape and thus the discontinuity points of the bounded curve can be uniquely specified in PAR. Therefore, there is a unique way to partition the Principal Axis in PAR and this unique partition decides a unique set of bounds.

Step 2: Within each bound of (S_i^1,C_i^1) and (S_i^2,C_i^2) of PAR(O_1,A_1) and PAR(O_2,A_2), respectively, we want to prove $C_i^1 = C_i^2$ if $S_i^1 = S_i^2$.

This can be proved by contradition. Assume $C_i^1 \neq C_i^2$. Since C_i^1 and C_i^2, both are composed of a set of layers, if $C_i^1 \neq C_i^2$, there must exist, for some j, a layer $[c_{ij}^1 c_{ij+1}^1]$ in C_i^1 and its corresponding layer $[c_{ij}^2 c_{ij+1}^2]$ in C_i^2 such that $[c_{ij}^1 c_{ij+1}^1] \neq [c_{ij}^2 c_{ij+1}^2]$, and generate different volumes for O_1 and O_2. In this case $O_1 \neq O_2$, which contradicts to our initial assumption $O_1 = O_2$. Therefore, $C_i^1 = C_i^2$ if S_i^1=S_i^2.

From the proofs of Step1 and Step2, in PAR, there is a unique set of bounds on the Principal Axis for an object, and on each pair of bounds there is a unique set of layers (i.e. pairs of curves) to represent the object. Therefore, we conclude that there is only one unique PAR for an axis-symmetrical object and further PAR is a unique representation scheme. **Q.E.D.**

4 Algorithm to Convert CSG Representation to PAR

In this section, we will describe how to convert a CSG representation to its corresponding Principal Axis Repesentation (PAR). Section 4.1 first describes the data structure for PAR. The algorithm is described in Section 4.2, while Section 4.3 describes how to combine (either *union* or *difference*) two PAR's into one. The combination procedure is definitely a key component of the conversion algorithm.

4.1 Data Structure for PAR

Since a PAR is composed of a set of tuples (S_i, C_i), we may use a record to represent each tuple and a link list to link these records together. Being a pair of bounds on the principal axis, each S_i can thus be represented by a pair of real numbers and stored in the tuple record. Each C_i consists of a set of curves C_{ij}, and can therefore be represented by a pointer to a link list where each element stores the parameters of one curve pair C_{ij} and C_{ij+1}, that is, a record representing a layer of the machine part.

We will assume the link list for tuples is sorted in the order of their bounds. Within each record for a tuple, the link list for the pairs of curves C_{ij} is also kept sorted in their ordering (from Theorem 1). This makes the concept of layers easy to deal with because each pair of the curves enumerating from the beginning of the sorted sequence is just a layer.

4.2 Conversion from CSG Representation to PAR

The conversion algorithm basically traverses the CSG tree from bottom to top. It first converts the CSG leaf nodes to their corresponding PAR's, and then combines (either *union* or *difference*) the PAR subtrees into composite PAR subtrees in higher levels. The combination procedure is repeated until the root is visited. The following recurive procedure describes this algorithm.

```
function EvaluateCSG ( T : CSG_tree ) : principal_axis_rep;
var
        P, P1, P2 : principal_axis_rep;
```

```
begin

      case T.type of

           movement   : P := EvaluateCSG ( T.left_child^ );
                        Transform ( T.movement, P );
                        return ( P );
           union,
           difference: P1 := EvaluateCSG ( T.left_child^ );
                        P2 := EvaluateCSG ( T.right_child^ );
                        P := Combine ( T.type, P1, P2 );
                        return ( P );
           primitive  : P := BuildAxisRep ( T );
                        return ( P );
      end; { case }

end; { EvaluateCSG }
```

To convert a CSG tree, its root can be passed to this function, which will recursively walk the whole CSG tree and transform it into a PAR. We assume that each node of the CSG tree is either a primitive solid (cylinder, cube, or torus) or an operator (*movement*, *union* or *difference*).

Primitives (i.e. the CSG-tree leaf nodes) are converted to corresponding PAR's by *BuildAxisRep* procedure, which is based on the framework of Section 3.2. Using the data structure representation in Section4.1, the *BuildAxisRep* procedure simply creates one record to represent the primitive . In this record, the pair of bounds (in terms of the line parameter values of its principal axis) of the primitive solid are stored. A pair of curves representing a layer is also stored within the record. For a cylinder or cone, the outer curve of the pair is the line specifying the outer shape of the primitive solid and the inner curve is just the principal axis. For a torus, this pair is just the outer and inner half circles of the torus.

For the *movement* node in the CSG tree, its subtree (i.e. left-subtree) is evaluated first, then the transformation matrix stored within the *movement* node is applied to the evaluated subtree. The transformation matrix is a 4 by 4 matrix denoting the translation and rotation components of the movement. Since all the curves in PAR are specified relative to the principal axis, procedure *Transform* needs only transform the principal axis, not the whole set of curves. In fact, the transformation matrix is applied only to the two ending points of the principal axis.

To deal with the *union* or *difference* node, its two subtrees are evaluated first, then the *Combine* procedure is called to union or difference them together. To implement the definitions of union and difference of two PAR's developed in Section 3.3, the *Combine* procedure includes four passes. We will describe these four passes in the next section.

4.3 Union and Difference of Two PAR's

This algorithm (the procedure *Combine*) basically employs the split-and-merge paradigm, splitting the two composite axes into segments, computing the union or difference of two segments and then merging all the resulting segments into a new PAR. It has four steps (passes).

Step 1. Find New Pairs of Bounds

In this stage, the two operand subtrees being combined are represented in the PAR form, each being a link list of segments (records) in the order of the values of the bounded pairs. The function of this step is to compute the new bounded pairs for the resulting PAR.

Since the values of the bounds in each operand PAR are in terms of line parameters of its own principal axis, we must first convert these values so that they are in terms of the line parameters of the resulting principal axis. Depending on the operation to be performed, the resulting principal axis can be chosen either as that of the first operand for difference operation or as the union of those of the two operands for the union operation. This selection process can be easily accomplished by simply checking the two ending points of the two operand principal axes. We assume that the two operands have the principal axes in the same or opposite direction if they are to be combined. (Note that the opposite direction may occur because *cones* are not symmetrical in its top and buttom and thus have two directions.)

After the resulting principal axis is roughly chosen and the values of the bounds are adjusted in term of this resulting axis, we may determine the pairs of all bounds for the resulting PAR. This process is simply the merge-and-sort procedure by repeatedly inputing two values of the bounds from each operand PAR and choosing the smaller value for the new bound. The state transition diagrams of this procedure are shown in Figure 1 and Figure 2.

In Figure 1 and Figure 2, we may imagine that there are two stacks for two operand PAR's. Each stack stores the bounds of one operand PAR which are in ascending order with the smallest one on the top of the stack. A pair of bounds is just an even-odd pair of values on the stacks if the top element of the stack is numbered from zero. This implies that a common bound of two neighboring segments has duplicate values on the stack. Let's also assume that a is the top element of the first operand stack A and b is the top element of the second stack B, respectively.

Now the following four states in the state transition diagrams can be defined.

- State "-0" means a upper bound of a pair in stack A and a upper bound of a pair in stack B have just been popped out.
- State "+0" means a lower bound of a pair in stack A and a upper bound of a pair in stack B have just been popped out.
- State "+1" means a lower bound of a pair in stack A and a lower bound of a pair in stack B have just been popped out.
- State "-1" means a upper bound of a pair in stack A and a lower bound of a pair in stack B have just been popped out.

The starting and final state is State "-0". The notation { *condition/ action1*; *action2*; .... } is used in the state transition diagrams to indicate that, if the *condition* is satisfied, the

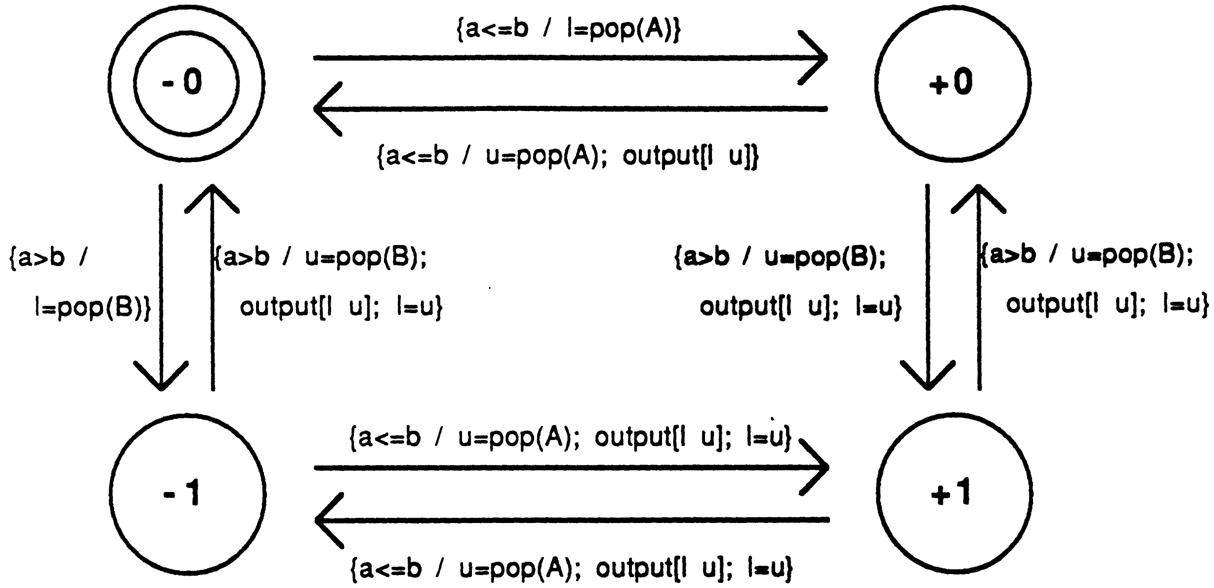


Figure 1: State Transition to Compute the New Bounds for Union.

actions specified in the bracket are executed and the new state is pointed to by the arrow sign. Let's also use l and u to denote lower bound and upper bound of a bound pair. Note that these state transition diagrams may generate null bound pairs like [c c], where c is a real number, and they should be discarded.

Step 2. Refine the Pairs of Bounds

After Step 1, a set of pairs of bounds is obtained for the resulting PAR. For those pairs of bounds which are not subintervals common to both operands, the resulting shape within them can be immediately determined by the curves from the original (i.e. operand PAR) bound pairs. For those new bound pairs which are subintervals of both original operand PARs, the story is, however, not so simple. They should be refined. That is, subdivision of them must be considered because the resulting shape within a pair of bounds might be determined by curves from both original PARs.

To refine the pairs of bounds which are the subintervals of both original PARs, we compute the intersection points of the curves within them. After sorting these intersecting points, a set of refined pairs of bounds may be created. The resulting curves within these refined pairs of bounds are then determined by the subcurves of the original curves.

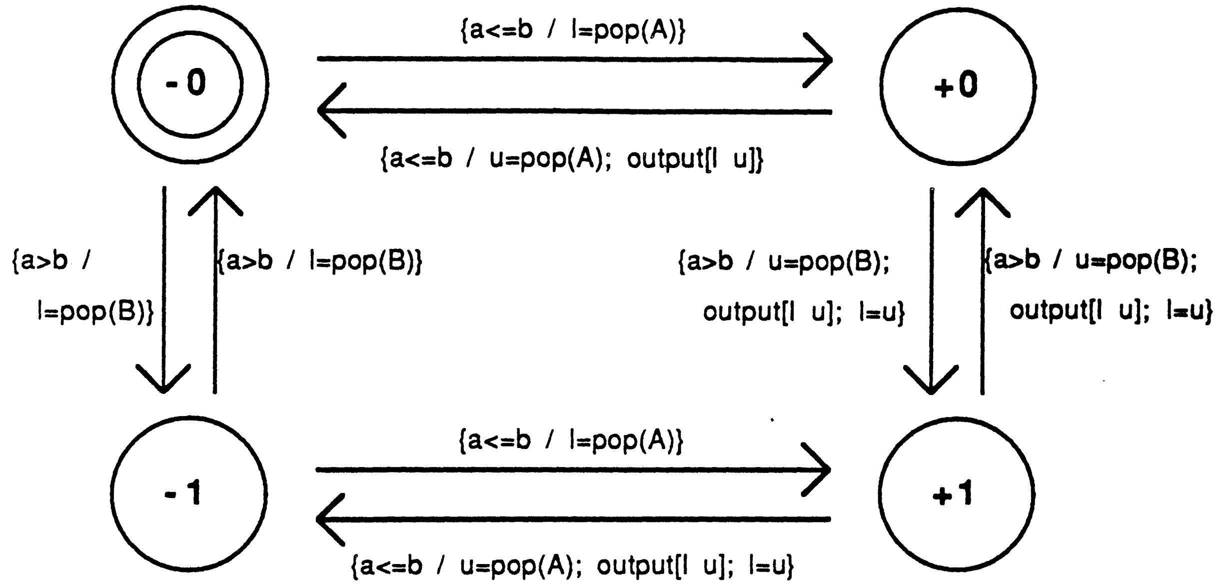


Figure 2: State Transition to Compute the New Bounds for Difference.

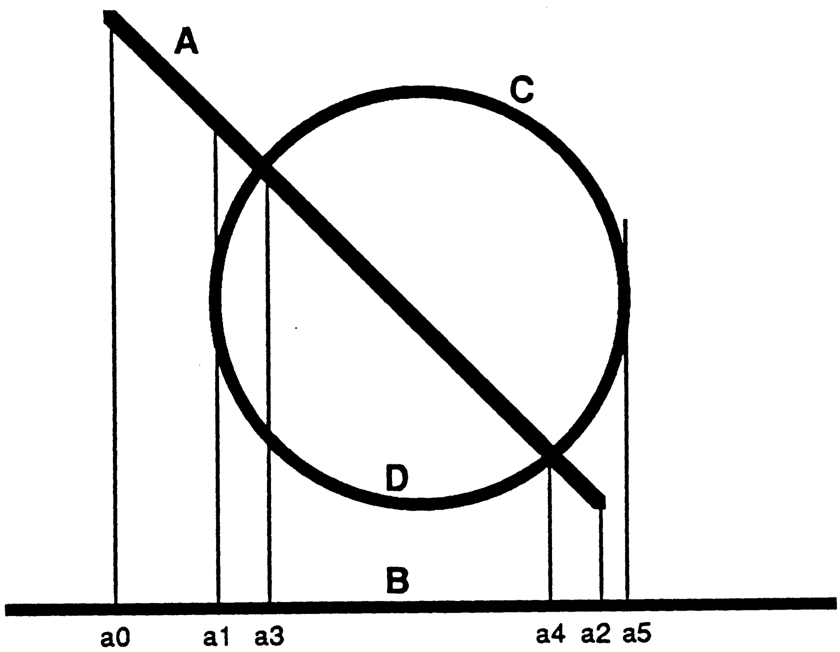


Figure 3: Refine a Pair of Bounds by Computing Intersection Points.

Consider the example of Figure 3. A cone and a torus are combined (either *union* or *difference*). The cone is specified by lines A and B, and the torus is specified by the upper half circle C and lower half circle D within the bounds $[a_1\ a_2]$. Since $[a_1\ a_2]$ obtained after Step 1 is both a subinterval of the $[a_0\ a_2]$ of the cone and of the $[a_1\ a_5]$ of the torus, we compute the intersection points for the curves A, B, C, D. Two intersection points a_2 and a_3 are obtained, then we create three refined bound pairs $[a_1\ a_3]$, $[a_3\ a_4]$, and $[a_4\ a_2]$ to replace the $[a_1\ a_2]$. The curve shape within these three refined intervals can then be uniquely determined, which depends on the operation to be performed. We will discuss it in the next step of the algorithm.

Step 3. Apply the Union or Difference Operation

In this stage, the resulting curve shape within each pair of bounds can be determined. For those pairs of bounds created from Step 1 but not refined by Step2, that is, they are subintervals that belong to only one original pair of bounds, the curves within the original pair of bounds are directly copied into the newly created bound pair with the two ending points of the curves adjusted so as to be consistent with the new bounds. This works for the union operation. For the difference operation, the bound pairs contributed solely by the second operand are simply thrown away, but those bound pairs from the first operand should be kept and the curves defined within these bound pairs should also be copied.

For those refined bound pairs from Step 2, it is assured that there is no curve intersection within them, thus from Theorem 1, the curves within them form a total ordering. This property of total ordering together with the concepts of *maximum union-able* and *minimum difference-able layer sets* is useful for determining which curves from the original ones contribute to the resulting curves.

Here we use the concept of layer to determine the resulting curves. A layer is an area bounded by two curves within some interval. Because the curves of layers of the two operands within the bound pair form a total ordering, they are topologically eqivalent to lines which are parallel to the principal axis within that bound pair. At any point within the bound pair, a line, passing this point and perpendicular to the principal axis, intersects all

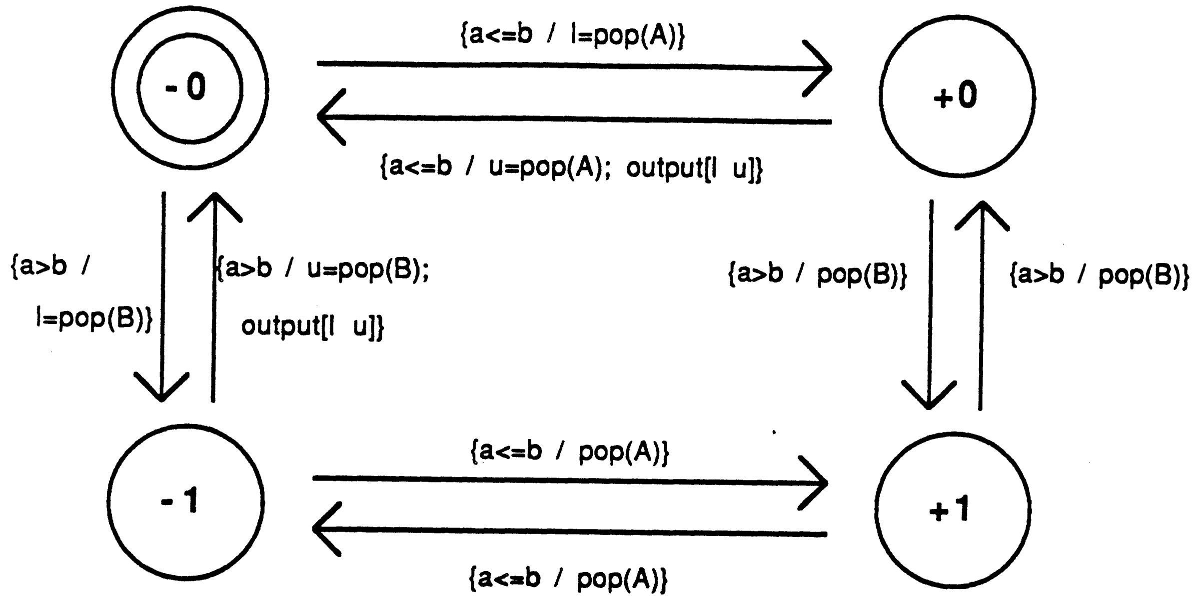


Figure 4: State Transition Diagram for Union.

the curves within that bound pair. Layers can then be represented by segments (determined by these intersection points) on this line. If the intersection point of this line and the principal axis is assumed to be the zero point, then the coordinates of the original curves and those of segments for layers can thus be uniquely determined in this line coordinate.

By using this line coordinate system, computing the union or difference of the line segments on this line is a simple task. It is similar to the procedure of merge-and-sort in Step 1. The coordinates of the line segments (for layers), which are kept sorted, from two operands are compared one by one, and depending on the operation performed (either *union* or *difference*), the resulting curves and layers can be determined. The following state transition diagrams (Figure 4 and Figure 5) demonstrate this algorithm. The meaning of the states and symbols is very similar to that in Figure 1 and Figure 2 except that the concept of bound pairs is replaced by the concept of layers and the values of the bounds are replaced by the coordinates of the curves in the line coordinate system.

Consider again the example of Figure 3. We assume the cone is operated with the torus, the cone is the first operand (i.e. left subtree) and the torus is the second operand (i.e. right subtree). Within the bounds $[a_1\ a_3]$, lines A and B form a layer for the cone and curves C and D form another layer for the torus. Since the layer bounded by A and B

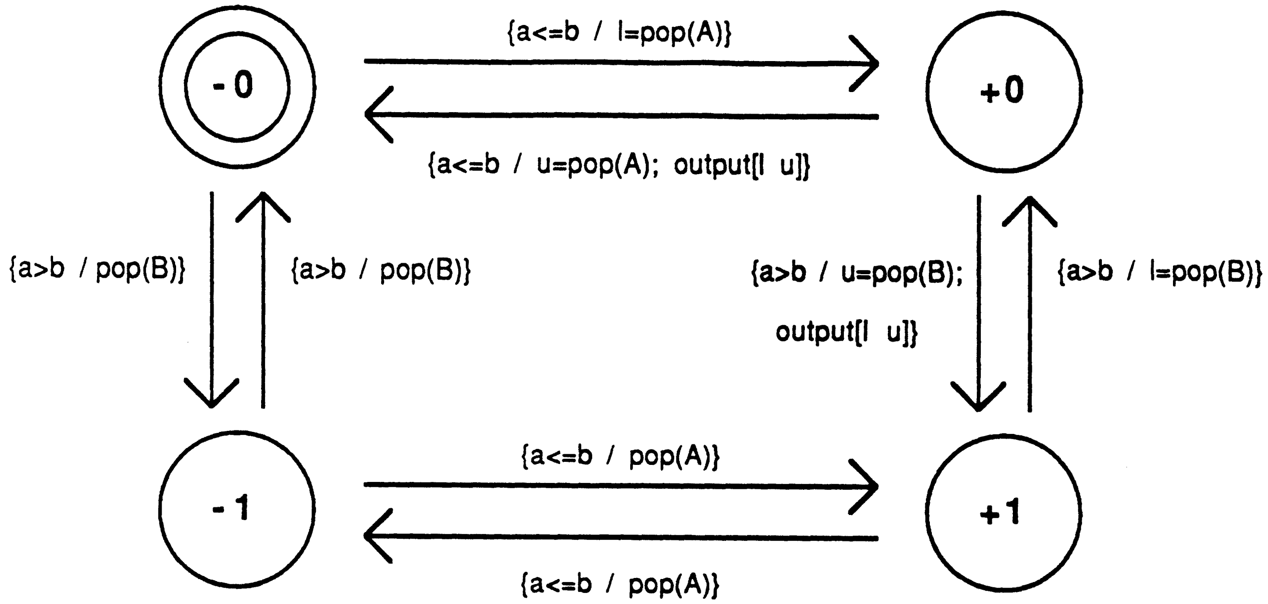


Figure 5: State Transition Diagram for Difference.

covers that bounded by C and D (this can be checked by their line coordinates), if the union operation is performed, the resulting layer would be that bounded by A and B. However, if the difference operation is executed, the results would be the layer bounded by A and C, and the layer bounded by D and B.

Similarly, within the bounds $[a_3\ a_4]$, the result of the union operation is the layer bounded by the curves C and B, and the result of the difference operation is the layer bounded by D and B. Within the bounds $[a_4\ a_2]$, the result of the union operation is the layer bounded by C and D and the layer bounded by A and B. The result of the difference operation would be the layer bounded by A and B.

Step 4. Merge Neighboring Segments If Possible

After the processing of the previous steps, the resulting shape of the combined object is obtained. However, it might not be in the PAR form because the curves at one of the two bound points might be differentiable. This may happen if some curves are seperated due to the bound pair refinement and later only those seperated subcurves are saved to be the resulting curves by the selection process of Step 3. In this case, the bound point at which all curves are differentiable is not a really breakpoint and it does not satisfy the non-differentiable requirement of the PAR definition at the bound points. Therefore further processing is necessary to make the result a PAR.

The actual work in this step is to go through the link list and check if the current segment is possible to be merged with its neighboring segments. The conditions for merging two neighboring segments are that

1. two neighboring segments have one common bound point

2. all the curves in one segment are one-to-one connected to all curves in another segment at the common bound point.

3. the two curves from both segments which are connected at the common point must have the common curve type and must belong to (i.e. subcurves) a curve whose curve type is identical to the common curve type.

4. two curves which satisfy condition (3) must be differentiable at the common bound point.

After testing the merge condition, if two neighboring segments are mergeable, a segment is deleted from the list and the bound point of the extant segment is extended to cover the range of the deleted segment. Since all the curves in the original two segments are one-to-one correspondance and the corresponding curves have the same curve parameters, nothing the bounds of the curves need to be updated.

This step makes sure that the result of operating two PAR's is still a PAR. It is very important because it assures that, from Theorem 6, PAR is a unique representational scheme. The uniqueness property of a representational scheme is very useful because it minimizes the redundancy of storing data and eliminates the concern of data inconsistency in a database. Furthermore, it saves programmers' effort of writing code to analyze each of the possible representations for the same object and makes the access of objects in the database more efficient.

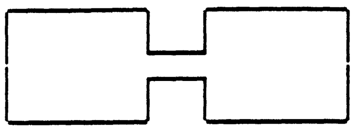


Figure 6.1

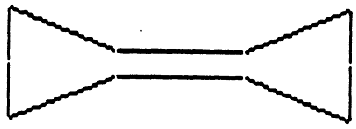


Figure 6.2

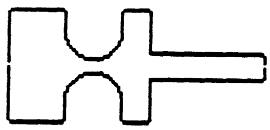


Figure 6.3

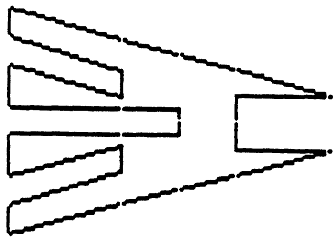


Figure 6.4

Figure 6: Examples.

5 Illustrative Examples

In this section, the testing results of several examples by our program are shown in Figure 6 and Figure 7. This program implements the algorithms described in Section 4. Also implemented are the algorithms to compute the length and maximum diameter of a symmetrical part and to compute and plot its profile from PAR. The whole program code, which has more than 2,500 lines Pascal code on Apollo computer, can be found in Appendix 1, while Appendix 2 shows the input data for the example "bottle" of Figure 7. Appendix 3 shows all the pictures of the machine parts in the example of Figure 6 and 7. These pictures are generated from a *ray tracing* program with CSG trees as input data.

Figure 6.1 shows the profile of a machine part which is joined together from three cylinders of different sizes. Some part of the middle cylinder originally overlapping with the other two is removed after processing, therefore it is not seen from the profile. Figure 6.2 shows the profile of a machine part which is unioned from two cones and a cylinder. Notice that the two cones are in opposite directions, that is, one of them is rotated 180° with respect to y-axis from its originial orientation. Figure 6.3 is a profile of a neck. The neck is made from a cylinder, cut off a pipe and further a torus from its outer surface. In Figure 6.4, a pipe and two holes are drilled from a cone. The computation for this object

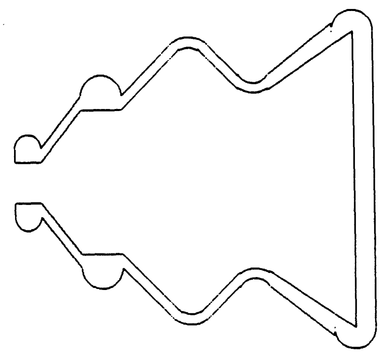


Figure 7: Example of Bottle.

involves many *layer difference* operations. Its profile clearly displays the layer information.

Figure 7 displays the profile of a "bottle". Its length and diameter are also computed by the program (Length = 5.30 units, Diameter = 4.905 units). This "bottle" is composed of 27 primitive solids involing 8 cones, 12 cylinders and 7 tori (see Appendix 2, *prmitive* file). Its CSG tree contains 53 nodes (*object* file). Since there is no movement node used in this example, its *movement* file is empty. Many *principal axis union* and *difference* operations as well as *layer union* and *layer difference* operations are involved in converting the CSG tree into its PAR representation. Its length, diameter information and profile are then computed from this PAR.

To compute a profile of a machine part, its algorithm may take advantages of the PAR representation. Since the layer information which consists of its outer, inner curves, and starting and ending points exists in the PAR representation, the profile of a machine part can be generated by drawing the four boundary curves (two vertical lines for the starting and endinding line boundaries, and two for the inner and outer curves) for each layer in an *exclusive or* plotting mode. In such mode, the common boundary of two layers will dissappear, rather than being plotted twice.

This algorithm is efficient and easy to implement, it has some problems, however. It relies on the *exclusive or* operation to eliminate the common boundary of two layers, but the *exclusive or* operation also eliminates the conner points of actual boundary because the intersection point of two boundary curves is plotted twice. In general, a *right-turn* algorithm [WeA77] can be used to solve this problem. The vertices of layers and their relations in the *principal axis coordinate system* can be easily obtained from the PAR. Then the *right-turn* algorithm can be applied to these vertices to get the profile. At this time, we only implement the profile computation using the *exclusive or* algorithm. Close examination of Figure 6 and Figure 7 reveals the fact of the missing common corner points.

6 Conclusion and Discussion

In this section, some concluding remarks are made first in Section 6.1, then several issues about PAR and its future extension are discussed in Section 6.2.

6.1 Conclusion

As a new representational scheme toward *representational uniqueness*, the Principal Axis Representation (PAR) is developed for *axis symmetrical* machine parts. Based on the CSG, the machine parts can be constructed by applying the *regularized set operations union, difference* and *movement* on the primitive solids *cylinder, cone* and *torus*. The PAR can represent the machine parts composed of primitives that have the same *Principal Axis*.

The PAR is first defined in this report. The operations for operating (combining) PAR's : *union, difference* and *movement* are then defined. Also shown is how to use PAR to represent primitive solids: cylinder, cone and torus. Based on this formulation, several theorems are proved, which show that a CSG tree can be converted into a PAR representing the same axis-symmetrical object.

An algorithm based on the mathematical formulation of PAR to convert a CSG tree has been designed and implemented in this study. It first converts the CSG leaf nodes into their PAR representation and then operates these PAR's (using *union, difference, movement* operations of PAR) on the CSG tree from bottom to top until the root of the CSG tree is visited. Several examples and testing results of this algorithm are shown in Section 5.

To support CAD applications, for example, generating the profile of machine parts for Numerical Control and computing the geometrical properties of a machine part such as its length and maximum diameter for part classification and CAPP (Computer Aided Process Planning), the PAR seems to be more efficient than its counterpart: the CSG scheme. This is due to the fact that, to represent an object, the CSG tree is left unevaluated while the PAR is already the result after evaluation. Algorithms that compute the profile and the length as well as diameter of a machine part from PAR have also been implemented. Its results are shown in Section 5.

In addition to its computational efficiency, the PAR is proved to possess yet another important property as a representational scheme, that is, representational uniqueness. A unique representatation scheme allows much simpler *feature definition* and therefore *feature extraction* or *object recognition* because only one representation for a feature or an object is required to deal with[LeK86].

6.2 Discussion and Future Work

The key ideas of PAR are to represent an object by its principal axis and boundary curves, and to resolve the overlap (intersection) of two composite solids, i.e. to compute the *union* and *difference* of layers. At present, PAR works for axis-symmetrical machine parts that are constructed from primitives such as cylinder, cone and torus. To make it more general as a unique representational scheme, several extensions are currently under investigation:

- relax the assumption of common principal axis requirement. Based on the PAR, a hierarchical PAR could be defined as a set of PAR's, each having its own common principal axis. But the problem of how to characterize the overlap parts of two PAR's during *union* and *difference* operations to ensure that the generalized PAR is also a uniqueness representation needs to be studied.

- include more primitives like *sphere* and *cube*. In fact, *sphere* is a special case of *torus*. But naively integrating *sphere* into PAR would invalidate the uniqueness property of the PAR because a *sphere* has two different representations in PAR, i.e. using *sphere* primitive or *torus* primitive. Including *cube* into PAR relaxes the constraint of axis-symmetry to a large extent. How to characterize the boundary curves, however, requires further study.

To make the PAR more useful, several areas of application are also under study:

- support feature extraction and object recognition. Since the PAR is representational uniqueness, object features such as *round, fillet, keyway, hole* can be defined on the PAR more easily than on the CSG tree. Developing algorithms to extract features or recognize object based on the PAR should be simpler because less cases need to be analyzed.

- support CAD (Computer Aided Design) applications. In this report, we have shown the profile generation of machine parts for NC (Numerical Control) and the length and diameter computation for machine parts to support part classification and CAPP (Computer Aided Process Planning). Other geometrical properties and geometrical codes [KaO84] could also be computed easily from the PAR. We are working on exploit it now.

References

[ArCP84] G.T. Armstrong, G.C. Carey, and A. de Pennington, "Numerical Code Generation from A Geometric Modeling System," in *Solid Modeling by Computers: From Theory to Applications*, ed. by M.S. Pickett and J.W. Boyse, Plenum Press, 1984.

[Bee82] W. Beeby, "The Future of Integrated CAD/CAM Systems: the Boeing Perspective," *Computer Graphics and Applications*, Vol.2, No.1, Jan.1982, pp51-56.

[BoG82] J.W. Boyse and J.E. Gilchrist, "GMSolid: Interactive Modeling for Design and Analysis of Solids," *Computer Graphics and Applications*, Vol.2, No.2, March 1982, pp27-40.

[Gra76] A.R. Grayer, "The Automatic Production of Machined Components Starting from A Stored Geometric Description," *PROLAMAT Proceedings*, North Holland Publishing Co., 1976.

[Hen84] M.R. Henderson, "Extraction of Feature Information from Three Dimensional CAD Data," Ph.D. Thesis, School of Mechanical Engineering, Purdue University, May, 1984.

[Jak82] R. Jakubowski, "Syntactic Characterization of Machine Parts Shapes," *Cybernetics and Systems: An International Journal*, 13, 1982, pp1-24.

[KaO84] Y. Kakazu and N. Okino, "Pattern Recognition Approach to GT Code Generation on CSG," Proc. 16th CIRP International Seminar on Manufacturing Systems, Tokyo, 1984, pp. 10-18.

[Kyp80] L.K. Kyprianou, "Shape Features in Computer-Aided Design," Ph.D. Thesis, University of Cambridge, Cambridge, England, July, 1980.

[LeK86] Y.C. Lee and K.S. Fu, "Machine Understanding of CSG: Extraction and Unification of Manufacturing Features," *Computer Graphics and Applications*, (to appear as a regular paper in December, 1986)

[ReV82] A. A. G. Requicha and H. B. Voelcker, "Solid Modeling : A Historical Summary and Contemporary Assessment," *Computer Graphics and Applications*, Vol.2, No.2, March 1982, pp. 9-24.

[StHA83] S.M. Staley, M.R. Henderson, and D.C. Anderson, "Using Syntactic Pattern Recognition to Extract Feature Information from a Solid Geometric Data Base," *Computers in Mechanical Engineering*, Vol.2, No.2, Sept. 1983, pp61-66.

[WeA77] K. Weiler and P. Atherton, "Hidden Surface Removal Using Polygon Area Sorting," *Computer Graphics*, Vol.11, No.2, March 1977, pp. 9-24.

[Woo77] T.C. Woo, "Computer Aided Recognition of Volumetric Designs," in *Advances in Computer-Aided Manufacture*, edited by D. Mcpherson, North-Holland Publishing Company, 1977, pp121-136.

[Woo82] T.C. Woo, "Feature Extraction by Volume Decomposition," Proc. Conference on CAD/CAM Technology in Mechanical Engineering, MIT, Mar. 1982, pp76-94.

Appendix 1. Program Listing

```
(******************************************************)
(*                                                    *)
(*      This program implements how to convert a CSG  *)
(*  input tree into a PAR (Principal Axis Representa- *)
(*  tion) that represents the identical object as the *)
(*  original CSG tree does.                           *)
(*      All the algorithms and procedures implemented *)
(*  here are based on the definitions of PAR, its     *)
(*  union, difference and movement operations.        *)
(*      Geometrical properties such as the length and *)
(*  diameter of objects are computed from the result- *)
(*  ing PAR to support part classification and process*)
(*  planning.  Profile is also generated to support   *)
(*  Numerical Control application.                    *)
(*                                                    *)
(*      Author :  K. F. Jack Jea                      *)
(*      Date   :  6/12/1986                           *)
(*                                                    *)
(******************************************************)

PROGRAM AXIS( input, output );

%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/gpr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;

CONST
        { part number of the csg tree root }
    CSG_ROOT        = -1;
    PI              =  3.1415926;

TYPE

        { normalized parametric value on the axis }
    t_value         = real;

        { value of vertical axis on principal axis coordinate }
    y_value         = real;

        { disparity of directions of two axes }
    direction       = (same_dir, opposite_dir, non_co_linear);

        { basic primitive solids currently implemented }
    solid_kind      = ( cylinder, cone, torus );

        { node types of the CSG tree }
    node_kind       = ( primitive, movement, union_op, diff_op );

        { axis operation type of two principal axes }
    axis_op_kind    = ( union, difference );

        { lower or upper bounds }
    lower_or_upper = ( lower, upper );
```



```

        { upper or lower part of a circle }
    up_or_down      = ( up, down );

        { curve type }
    curve_kind      = ( line, arc );

        { possible states when two axes are operated }
    state           = ( pluszero, minuszero, plusone, minusone );

        { point coordinate on the principal axis coordinate }
    D2_point        = record
                            x_coord : t_value;
                            y_coord : y_value;
                      end;

        { actual coordinate in the 3-D space }
    D3_point        = record
                            x_coord,
                            y_coord,
                            z_coord : real;
                      end;

        { chain to link intersection points of two axes }
    intersect_rec_ptr  = ^intersect_rec;

        { record for storing the intersection points }
    intersect_rec   = record
                            intersection   : t_value;
                            next_intersect : intersect_rec_ptr;
                      end;

        { record to store a bounded curve }
    curve     = record
                    curve_start,
                    curve_end    : y_value ;
                    case curve_type : curve_kind of
                        line : ( );
                        arc  : (
                                   center      : D2_point;
                                   radius      : real;
                                   which_half  : up_or_down );
                end;

        { pointer to a bounded curve }
    curve_ptr   = ^curve;

        { pointer to a layer }
    layer_ptr       = ^layer;

        { a layer which is bounded by two curves }
    layer       = record
                      next_layer   : layer_ptr;
                      inner_curve,
                      outer_curve  : curve_ptr;
                  end;
```

```

        { pointer to a segment of the axis }
    segment_ptr     = ^segment;

        { segment record, part of axis }
    segment             = record
                            lower_bound,
                            upper_bound   : t_value;
                            next_segment : segment_ptr;
                            layer_head    : layer_ptr;
                          end;

        { information of the principal axis }
    principal_axis  = record
                            start_point,
                            end_point     : D3_point;
                            segment_head : segment_ptr;
                          end;

        { transformation matrix for movement }
    xform_matrix  = record
                        translate_x,
                        translate_y,
                        translate_z : real;
                        rotate_x,
                        rotate_y,
                        rotate_z : real;
                    end;

        { information of a primitive solid }
    solid    = record
                    base, top  : D3_point;
                    case solid_type : solid_kind of
                        cylinder : ( radius : real );
                        cone     : ( radius_b,
                                     radius_t : real );
                        torus    : ( inner_radius,
                                     outer_radius :real );
               end;  { solid }

        { pointer to a CSG tree }
    CSG_tree_ptr        = ^CSG_tree;

        { CSG tree node information }
    CSG_tree = record
                    node_type : node_kind;
                    partno    : integer;
                    case node_kind of
                        primitive : ( prim_solid : solid );
                        movement  : ( move : xform_matrix;
                                      child : CSG_tree_ptr );
                        union_op,
                        diff_op   : ( left_child,
                                      right_child: CSG_tree_ptr);
               end;

```



```
        { global variables for main program use }
VAR
            { original input CSG tree }
        CSG : CSG_tree_ptr;
            { resulting PAR corresponding to CSG above }
        P : principal_axis;
            { length and maximum diameter of the object }
        L, D : real;
            { input data for creating the CSG }
        obj_file, mov_file, prim_file : text;
            { starting and ending points of a combined axis }
        min_point, max_point : D3_point;

            { switches for debugging use }
        DEBUG_INPUT : boolean;
        DEBUG_EVALUATE_CSG : boolean;
        DEBUG_BUILD_AXIS : boolean;
        DEBUG_LD : boolean;
        DEBUG_DRAW : boolean;
        DEBUG_COMPUTE_T : boolean;
        DEBUG_CURVE_CURVE, DEBUG_INTERSECT : boolean;
        DEBUG_AXIS_OP, DEBUG_NORMALIZATION,
        DEBUG_PARTITION, DEBUG_MERGE_INTERVAL : boolean;
        DEBUG_LAYER_DIFFERENCE, DEBUG_COMPUTE_LAYER,
                                 DEBUG_ADD_LAYER : boolean;
        TRACE_AXIS : boolean;

procedure dump_axis( PX :principal_axis);
{ dump the principal axis; for debugging purpose }
var
    s : segment_ptr;
    l : layer_ptr;
begin
    writeln('@@@@@@@  Dump Principal Axis @@@@@@@');
    writeln('start point =', PX.start_point.x_coord,
            PX.start_point.y_coord, PX.start_point.z_coord);
    writeln('end point =', PX.end_point.x_coord,
            PX.end_point.y_coord, PX.end_point.z_coord);

    { go thru the axis and print its content }
    s := PX.segment_head;
    while s <> nil do
    begin

        writeln('lower and upper bounds=',
                s^.lower_bound, s^.upper_bound);

        l := s^.layer_head;
        while l <> nil do
            begin

                writeln('outer curve, start and end y-value=',
                        l^.outer_curve^.curve_start,
                        l^.outer_curve^.curve_end);
                writeln('inner curve, start and end y-value=',
                        l^.inner_curve^.curve_start,
```

```
                          l^.inner_curve^.curve_end);

                  l := l^.next_layer;
              end;

          s := s^.next_segment;
      end;

      writeln('@@@@@@@  End of Dumping Principal Axis @@@@@@@');

end; { dump_axis }


function compute_t ( pt1, pt2, point : D3_point ) : t_value;
{ calculate the t parameter for point in line segment
  bounded by pt1 and pt2.  Note: we do not check the linearity
  of point, pt1 and pt2 in this version; it will be refined in
  later version.  }
const
        { numerical error range }
    EPS = 0.0001;
var
        t : t_value;
begin

    if DEBUG_COMPUTE_T then writeln(
        '%%%%%%% In compute_t, pt1, pt2, point=',
        pt1.x_coord, pt1.y_coord, pt1.z_coord,
        pt2.x_coord, pt2.y_coord, pt2.z_coord,
        point.x_coord, point.y_coord, point.z_coord );

    if abs(pt1.x_coord - pt2.x_coord) >= EPS  then
              t := (point.x_coord - pt1.x_coord) /
                    (pt2.x_coord - pt1.x_coord)
    else if abs(pt1.y_coord - pt2.y_coord) >= EPS  then
              t := (point.y_coord - pt1.y_coord) /
                    (pt2.y_coord - pt1.y_coord)
    else if abs(pt1.z_coord - pt2.z_coord) >= EPS  then
              t := (point.z_coord - pt1.z_coord) /
                   (pt2.z_coord - pt1.z_coord)
    else  writeln( 'ERROR: pt1 and pt2 are the same point,',
                     'in compute_t procedure');

    if DEBUG_COMPUTE_T then writeln('%%%%%%% In compute_t, t ='
                                   , t);

    compute_t := t;

end; { compute_t }

procedure modularize_t ( var t : t_value;
                         f1, f2, m1, m2 : t_value );
{ actually modularize t here;
  t' = m1 + ( t - f1 ) * ( m2 - m1 ) / ( f2 - f1 ) }
begin
```



```
        t := m1 + ( t - f1 ) * ( m2 - m1 ) / ( f2 - f1 );

end; { modularize_t }

procedure adjust_t_value( var P : principal_axis;
                          f1, f2, m1, m2 : t_value );
{ This routine use m1 and m2 as the factors to adjust the
  t parameters in the principal axis P; formula :
  t' = m1 + ( t - f1 ) * ( m2 - m1 ) / ( f2 - f1 ) }
var
        s : segment_ptr;
        l : layer_ptr;
begin

        { go thru each bound and t-value dependent item. }
        s := P.segment_head;
        while ( s <> nil ) do
            begin

                modularize_t ( s^.lower_bound, f1, f2, m1, m2 );
                modularize_t ( s^.upper_bound, f1, f2, m1, m2 );

                l := s^.layer_head;
                while ( l <> nil ) do
                begin
                    if l^.inner_curve^.curve_type = arc then
                        modularize_t (
                            l^.inner_curve^.center.x_coord,
                            f1, f2, m1, m2 );
                    if l^.outer_curve^.curve_type = arc then
                        modularize_t (
                            l^.outer_curve^.center.x_coord,
                            f1, f2, m1, m2 );
                    l := l^.next_layer;
                end; { inner while }

                s := s^.next_segment;

            end; { while }

end; { adjust_t_value }

function check_direction( P1, P2 : principal_axis ) :direction;
{ check to see if P1 and P2 are in the same, opposite or
  non-co-linear direction }
const
    EPS = 0.0001;
var
    x1, y1, z1, x2, y2, z2, d1, d2 : real;
begin

    { get three vector components of P1 and P2 }
    x1 := P1.start_point.x_coord - P1.end_point.x_coord;
    y1 := P1.start_point.y_coord - P1.end_point.y_coord;
    z1 := P1.start_point.z_coord - P1.end_point.z_coord;
    d1 := sqrt( x1 * x1 + y1 * y1 + z1 * z1 );
```

```
    xl := xl / dl;
    yl := yl / dl;
    zl := zl / dl;

    x2 := P2.start_point.x_coord - P2.end_point.x_coord;
    y2 := P2.start_point.y_coord - P2.end_point.y_coord;
    z2 := P2.start_point.z_coord - P2.end_point.z_coord;
    d2 := sqrt( x2 * x2 + y2 * y2 + z2 * z2 );
    x2 := x2 / d2;
    y2 := y2 / d2;
    z2 := z2 / d2;

    { check linearity }
    if ( abs(xl - x2) <= EPS) and ( abs(yl - y2) <= EPS) and
       ( abs(zl - z2) <= EPS) then
            check_direction := same_dir
    else if ( abs(xl + x2) <= EPS) and ( abs(yl + y2) <= EPS)
          and ( abs(zl + z2) <= EPS) then
            check_direction := opposite_dir
    else    check_direction := non_co_linear;

end;  { check_direction }

procedure  reverse_direction( var P: principal_axis );
{ adjust the t-parameters in P; make all the t-value reverse }
var
    s, ps, ns : segment_ptr;
    l : layer_ptr;
    pt : D3_point;
    t : t_value;
    y : y_value;
begin

    pt := P.start_point;
    P.start_point := P.end_point;
    P.end_point := pt;

    s := P.segment_head;
    ps := nil;

    while ( s <> nil ) do
    begin
        t :=  s^.lower_bound ;
              s^.lower_bound := 1.0 - s^.upper_bound;
        s^.upper_bound := 1.0 - t;

        l := s^.layer_head;
        while ( l <> nil ) do
        begin

            y := l^.inner_curve^.curve_start;
            l^.inner_curve^.curve_start :=
                        l^.inner_curve^.curve_end;
            l^.inner_curve^.curve_end := y;

            if l^.inner_curve^.curve_type = arc then
```



```
                l^.inner_curve^.center.x_coord := 1.0 -
                        l^.inner_curve^.center.x_coord;

            y := l^.outer_curve^.curve_start;
            l^.outer_curve^.curve_start :=
                        l^.outer_curve^.curve_end;
            l^.outer_curve^.curve_end := y;

            if l^.outer_curve^.curve_type = arc then
                l^.outer_curve^.center.x_coord := 1.0 -
                        l^.outer_curve^.center.x_coord;

            l := l^.next_layer;
        end; { inner while }

        ns := s^.next_segment;
        s^.next_segment := ps;
        ps := s;
        s := ns;

    end; { while }

    P.segment_head := ps;

end; { reverse_direction }

procedure normalization ( var Pl, P2 : principal_axis;
                          var ptl, pt2 : D3_point );
{ normalize both Pl and P2 to have the same t-parameter
  basis; note: after difference operation, the principal
  axis should re-normalized.  }
var
        tl, t2 : t_value;
        ml, m2 : real;
        dir    : direction;
begin

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% before normalization,Pl.start and end point=',
        Pl.start_point.x_coord, Pl.start_point.y_coord,
        Pl.start_point.z_coord, Pl.end_point.x_coord,
        Pl.end_point.y_coord,Pl.end_point.z_coord);

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% before normalization,P2.start and end point=',
        P2.start_point.x_coord, P2.start_point.y_coord,
        P2.start_point.z_coord, P2.end_point.x_coord,
        P2.end_point.y_coord,P2.end_point.z_coord);

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% before normalization,P2.low, up =',
        P2.segment_head^.lower_bound,
        P2.segment_head^.upper_bound);

    { check the direction of two axes, ok if the same;
      inverse one if in reverse direction, reject it
```

```
      if not co_linear  }
    dir := check_direction( P1, P2 );
    case dir of
        same_dir :  { ok; go thru it }
                    if DEBUG_NORMALIZATION then writeln(
                        '%%%%% In normalization,',
                        'two axes same direction' ) ;
        opposite_dir : if (P1.start_point.z_coord >
                           P1.end_point.z_coord) then
                           reverse_direction( P1 )
                       else
                           reverse_direction( P2 );
        non_co_linear : writeln('---->  error,',
                         ' two axes do not co-linear');
    end; { case }

        { compute the new starting and ending points }
    t1 := compute_t( P1.start_point, P1.end_point,
                     P2.start_point );
    t2 := compute_t( P1.start_point, P1.end_point,
                     P2.end_point );

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% IN normalization, t1, t2 =', t1, t2 );

    if 0.0 <= t1 then pt1 := P1.start_point
                 else pt1 := P2.start_point;
    if 1.0 >= t2 then pt2 := P1.end_point
                 else pt2 := P2.end_point;

    { adjust P1 and P2 using the new starting and
      ending points }
    m1 := compute_t( pt1, pt2, P1.start_point );
    m2 := compute_t( pt1, pt2, P1.end_point );
    adjust_t_value( P1, 0.0, 1.0, m1, m2 );

    m1 := compute_t( pt1, pt2, P2.start_point );
    m2 := compute_t( pt1, pt2, P2.end_point );
    adjust_t_value( P2, 0.0, 1.0, m1, m2 );

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% after normalization,P2.low, up =',
        P2.segment_head^.lower_bound,
        P2.segment_head^.upper_bound);

    if DEBUG_NORMALIZATION then writeln(
        '%%%%% result of normalization,P1.start and end point=',
        P1.start_point.x_coord, P1.start_point.y_coord,
        P1.start_point.z_coord, P1.end_point.x_coord,
        P1.end_point.y_coord,P1.end_point.z_coord);

end;  { normalization }

function get_curve( var p : layer_ptr;
              var flag : lower_or_upper ) : curve_ptr;
{ get next curve from the layers;  if none, then return nil.
```



```
  this routine is somewhat like get_next function, but it
  will be used by compute_intersection and compute_layer.  }
begin

    case flag of
        upper : if p = nil then
                    get_curve := nil
                else begin
                        flag := lower;
                        get_curve := p^.outer_curve ;
                    end;
        lower : begin
                    flag := upper;
                    get_curve := p^.inner_curve ;
                    p := p^.next_layer;
                end;
    end; { case }

end;  { get_curve }

procedure line_line( C1, C2 : curve_ptr;
                     S1, S2 : segment_ptr;
                     a, b : t_value;
                     var Iptr : intersect_rec_ptr );
{ compute the intersection of two lines C1 and C2;
  results are appended to Iptr.  }
label
        R;
const
    SLOPE_EPSILON = 0.001;
var
        x1, x2, x3, x4, dx, t1, x : t_value;
        y1, y2, y3, y4, dy : real;
        p : intersect_rec_ptr;
begin

        { get the first line segment }
    x1 := S1^.lower_bound;
    x2 := S1^.upper_bound;
    y1 := C1^.curve_start;
    y2 := C1^.curve_end;


    { get the second line segment }
    x3 := S2^.lower_bound;
    x4 := S2^.upper_bound;
    y3 := C2^.curve_start;
    y4 := C2^.curve_end;

    if DEBUG_CURVE_CURVE then
        writeln('$$$$$$ Enter In line_line,',
                'x1,y1,x2,y2,x3,y3,x4,y4',
                 x1,y1,x2,y2,x3,y3,x4,y4 );

    if ( ( x1 = x3 ) and ( y1 = y3 ) ) and
          ( ( x2 = x4 ) and ( y2 = y4 ) ) then
```

```
            { lines coincidence }
        begin
            { return }
        end
    else if (x1 = x2) and (x3 = x4)  then
            begin
                {return}
            end
    else begin
            if (x1 <> x2) and (x3 <> x4) then
                begin
                    if abs( (y2 - y1)/(x2 - x1) -
                        (y4 -y3)/(x4 - x3) ) <= SLOPE_EPSILON
                    then begin
                            goto R;
                        end;
                end;
            dy := y4 - y3;
            dx := x4 - x3;

            if DEBUG_CURVE_CURVE then writeln(
                '$$$$$$ In line_line, dx, dy, y1, y2 =',
                dx, dy, y1, y2 );

            t1 := dy * (x3 - x1) - dx * (y3 - y1);
            t1 := t1 / ( dy * (x2 - x1) - dx * (y2 - y1) );
            x := x1 + t1 * ( x2 - x1 );
            if ( a < x ) and ( x < b ) then
                begin
                    new( p );
                    p^.intersection := x;
                    p^.next_intersect := Iptr;
                    Iptr := p;

                    if DEBUG_CURVE_CURVE then writeln(
                        '$$$$$$ In line_line, intersection t ='
                        , x );
                end;
        end;
    R: ;

end; { line_line }

procedure line_arc( C1, C2 : curve_ptr;
                    S1, S2 : segment_ptr; a, b : t_value;
                    var Iptr : intersect_rec_ptr );
{ compute the intersection of a line C1 and an arc C2;
  results are appended to Iptr.
}
var
        x0, x1, x2, x3, x4, dx, x, length : real;
        y0, y1, y2, y3, y4, dy, t1, t2, AA, d : real;
        p : intersect_rec_ptr;
        theta, ans : array[1..2] of real;
        num, i : integer;
begin
```



```

    length := sqrt( (max_point.x_coord - min_point.x_coord)
                   *(max_point.x_coord - min_point.x_coord)
                   +(max_point.y_coord - min_point.y_coord)
                   *(max_point.y_coord - min_point.y_coord)
                   +(max_point.z_coord - min_point.z_coord)
                   *(max_point.z_coord - min_point.z_coord) );

        { get the first line segment }
    x1 := S1^.lower_bound * length;
    x2 := S1^.upper_bound * length;
    y1 := C1^.curve_start;
    y2 := C1^.curve_end;


        { get the second arc segment }
    x3 := S2^.lower_bound * length;
    x4 := S2^.upper_bound * length;
    y3 := C2^.curve_start;
    y4 := C2^.curve_end;
    x0 := C2^.center.x_coord * length;
    y0 := C2^.center.y_coord;

    if DEBUG_CURVE_CURVE then
            writeln('$$$$$$ In line_arc,x1,y1,x2,y2',
                    x1,y1,x2,y2 );
    if DEBUG_CURVE_CURVE then
            writeln('$$$$$$ In line_arc,x0,y0,x3,y3,x4,y4',
                    x0,y0,x3,y3,x4,y4 );

    if ( ( x1 = x3 ) and ( y1 = y3 ) ) and
           ( ( x2 = x4 ) and ( y2 = y4 ) ) then
             {curves intersect at end point}
        begin
            { return  }
        end
    else begin  { computate intersection of line and arc }
            dx := x2 - x1;
            dy := y2 - y1;
            AA := ( (y0 - y1) * dx - (x0 - x1) * dy ) /
                    C2^.radius;
            d := dx * dx + dy * dy - AA * AA;

           if DEBUG_CURVE_CURVE then writeln(
               '$$$$$$ In line_arc, dx, dy, AA, d',
               dx, dy, AA, d );

           if d >= 0.0 then
              begin
                  if abs( AA + dy ) <= 0.000001 then
                     begin
                         theta[1] := PI;
                         theta[2] := -PI;
                     end
                  else begin
                          t1 := (-dx + sqrt(d) ) / ( AA + dy );
```

```
                        t2 := (-dx - sqrt(d) ) / ( AA + dy );
                        theta[1] := 2.0 * arctan( t1 );
                        theta[2] := 2.0 * arctan( t2 );
                     end;

                num := 0;
                case C2^.which_half of
                    down  :  { take negative angles }
                            for i := 1 to 2 do
                               if theta[i] <= 0.0 then
                                   begin
                                       num := num + 1;
                                       ans[num] := theta[i];
                                   end;
                    up    :  { take positive angles }
                            for i := 1 to 2 do
                                if theta[i] >= 0.0 then
                                   begin
                                       num := num + 1;
                                       ans[num] := theta[i];
                                   end;
                 end; { case }

                 for i := 1 to num do
                     begin
                         x := x0 + cos( ans[i] ) * C2^.radius;
                         if ( a < (x/length) ) and
                            ( x < (b/length) ) then
                             begin
                                new( p );
                                p^.intersection := x / length;
                                p^.next_intersect := Iptr;
                                Iptr := p;
                             end;
                     end; { for }
               end; { if }
         end; { else }

end; { line_arc }

procedure arc_arc( C1, C2 : curve_ptr;
                S1, S2 : segment_ptr; a, b : t_value;
                var Iptr : intersect_rec_ptr );
{ compute the intersection of an arc C1 and an arc C2;
  results are appended to Iptr.
}
var
        xc1, xc2, x1, x2, x3, x4, dx, x, length : real;
        yc1, yc2, y1, y2, y3, y4, dy, t1, t2, AA, d : real;
        p : intersect_rec_ptr;
        theta, ans : array[1..2] of real;
        num, i : integer;
begin

    length := sqrt( (max_point.x_coord - min_point.x_coord)
                   *(max_point.x_coord - min_point.x_coord)
```



```
                   +(max_point.y_coord - min_point.y_coord)
                   *(max_point.y_coord - min_point.y_coord)
                   +(max_point.z_coord - min_point.z_coord)
                   *(max_point.z_coord - min_point.z_coord) );

    { get the first arc segment }
    x1 := S1^.lower_bound * length;
    x2 := S1^.upper_bound * length;
    y1 := C1^.curve_start;
    y2 := C1^.curve_end;
    xc1 := C1^.center.x_coord * length;
    yc1 := C1^.center.y_coord;

    { get the second arc segment }
    x3 := S2^.lower_bound * length;
    x4 := S2^.upper_bound * length;
    y3 := C2^.curve_start;
    y4 := C2^.curve_end;
    xc2 := C2^.center.x_coord * length;
    yc2 := C2^.center.y_coord;

    if ( ( x1 = x3 ) and ( y1 = y3 ) ) and
       ( ( x2 = x4 ) and ( y2 = y4 ) ) then
       {curves intersect at end point}
        begin
            { return }
        end
    else begin  { computate intersection of line and arc }
            dx := xc2 - xc1;
            dy := yc2 - yc1;

            if DEBUG_EVALUATE_CSG then writeln(
                '$$$$$$ In arc-arc, C2 radius =', C2^.radius );

            AA := ( C1^.radius * C1^.radius - C2^.radius *
                    C2^.radius - dx * dx - dy * dy ) /
                    ( 2.0 * C2^.radius );
            d :=  dx * dx + dy * dy - AA * AA ;

            if d >= 0.0 then
                begin
                    d := sqrt( d );
                    if abs( AA + dy ) <= 0.000001 then
                        begin
                            theta[1] := PI;
                            theta[2] := -PI;
                        end
                    else begin
                            t1 := ( dy + d ) / ( AA + dx );
                            t2 := ( dy - d ) / ( AA + dx );
                            theta[1] := 2.0 * arctan( t1 );
                            theta[2] := 2.0 * arctan( t2 );
                         end;

                    num := 0;
                    case C2^.which_half of
```

```
                         down   :  { take negative angles }
                                 for i := 1 to 2 do
                                    if theta[i] <= 0.0 then
                                       begin
                                          num := num + 1;
                                          ans[num] := theta[i];
                                       end;
                         up     :  { take positive angles }
                                 for i := 1 to 2 do
                                    if theta[i] >= 0.0 then
                                       begin
                                           num := num + 1;
                                           ans[num] := theta[i];
                                       end;
                     end; { case }

                     for i := 1 to num do
                         begin
                             x := xc2 + cos( ans[i] ) *
                                  C2^.radius;
                             if (( a < (x/length) ) and
                                 ( x < (b/length) )) and
                                (( x1 < x ) and ( x < x2 )) and
                                (( x3 < x ) and ( x < x4 )) then
                                begin
                                    new( p );
                                    p^.intersection := x / length;
                                    p^.next_intersect := Iptr;
                                    Iptr := p;
                                end;
                         end; { for }
                 end; { if }
          end; { else }

end; { arc_arc }

procedure compute_intersection( S1, S2 : segment_ptr;
                                var Iptr :intersect_rec_ptr;
                                a, b : t_value);
{ compute the intersection points of curves in S1 and S2;
  results are stored in the list pointed by Iptr.  }
var
        p1, p2 : layer_ptr;
        flag1, flag2 : lower_or_upper;
        curve1, curve2 : curve_ptr;
begin

        if ( S1 = nil ) or ( S2 = nil ) then
             Iptr := nil
        else begin

                 Iptr := nil ;

                 p1 := S1^.layer_head;
                 flag1 := upper;
                 curve1 := get_curve( p1, flag1 );
```



```

                 while ( curve1 <> nil ) do
                 begin
                    p2 := S2^.layer_head;
                    flag2 := upper;
                    curve2 := get_curve( p2, flag2 );

                    while ( curve2 <> nil ) do
                    begin
                        case curve1^.curve_type of
                            line : if curve2^.curve_type = line
                                   then
                                     line_line( curve1, curve2,
                                            S1, S2, a, b, Iptr )
                                    else
                                     line_arc( curve1, curve2,
                                           S1, S2, a, b, Iptr );
                            arc  : if curve2^.curve_type = line
                                   then
                                     line_arc( curve2, curve1,
                                           S1, S2, a, b, Iptr )
                                   else
                                     arc_arc( curve1, curve2,
                                         S1, S2, a, b, Iptr );
                        end; { case }

                        curve2 := get_curve( p2, flag2 );
                    end;  { curve2 }

                    curve1 := get_curve( p1, flag1 );

                end; { curve1 }

                 end; { else }

end; { compute_intersection }

procedure sort_intersection( var Iptr : intersect_rec_ptr );
{ sort the intersection points is ascending order;
  linear sort is implemented here.  }
var
        p, q, I1, I2 : intersect_rec_ptr;
        done : boolean;
begin

    if DEBUG_INTERSECT then
    begin
       writeln('#### In sort_intersection, begin dump Iptr list');
       p := Iptr;
       while ( p<> nil ) do
           begin
               writeln('#### In sort_intersection, intersection ='
                         , p^.intersection );
               p := p^.next_intersect;
           end;
       writeln('#### In sort_intersection, end dump Iptr list');
```

```
    end;

    if Iptr <> nil then
        begin
            p := Iptr;
            p := p^.next_intersect;
            while ( p <> nil ) do
            begin
                q := p;
                if q^.intersection <= Iptr^.intersection then
                    begin
                        p := p^.next_intersect;
                        q^.next_intersect := Iptr;
                        Iptr := q;
                    end
                else begin
                         I2 := Iptr;
                         I1 := I2^.next_intersect;
                         done := false;
                         while ( I1 <> nil ) and ( not done ) do
                         begin
                             if I1^.intersection >=
                                 q^.intersection then
                                    done := true
                             else begin
                                     I2 := I1;
                                     I1 := I1^.next_intersect;
                                 end;
                         end; { while }

                         p := p^.next_intersect;
                         q^.next_intersect := I1;
                         I2^.next_intersect := q;
                     end;
            end; { while }
        end; { if }

end; { sort_intersection }

procedure screen_intersection( var Iptr : intersect_rec_ptr;
                               a, b : t_value );
{ insert a and b into the intersection list and make sure that
  this list is bounded by a and b, and there are no duplicate
  intersection points there.  }
var
        p, q, I1, I2 : intersect_rec_ptr;
begin

    new( I1 );
    I1^.intersection := a;
    I1^.next_intersect := Iptr;
    new( I2 );
    I2^.intersection := b;
    I2^.next_intersect := nil;
    p := Iptr;
    Iptr := I1;
```



```
    q := I1;

    while ( p <> nil ) do
    begin
        if ( p^.intersection <= I1^.intersection ) or
           ( p^.intersection = q^.intersection )  then
            begin
                p := p^.next_intersect;
                q^.next_intersect := p;
            end
        else if p^.intersection >= I2^.intersection then
                begin
                    p := nil;
                end
        else    begin
                    q := p;
                    p := p^.next_intersect;
                end;
    end;  { while }

    q^.next_intersect := I2;

end; { screen_intersection }

function compute_point( t : t_value; C : curve_ptr;
                        x1, x2 : t_value ) :y_value;
{ given a parameter t, compute the value associated with t
  on the curve C; i.e. C(t) }
var
        m, length : real;
        y : y_value;
begin

    case C^.curve_type of
         line : begin
                    m := ( C^.curve_end - C^.curve_start ) /
                         ( x2 - x1 );
                    y := C^.curve_start + m * ( t - x1 );
                    compute_point :=  y ;
                end;
         arc:   begin
                    length := sqrt(
                     (max_point.x_coord - min_point.x_coord)
                     * (max_point.x_coord - min_point.x_coord)
                     + (max_point.y_coord - min_point.y_coord)
                     * (max_point.y_coord - min_point.y_coord)
                     + (max_point.z_coord - min_point.z_coord)
                     * (max_point.z_coord - min_point.z_coord));

                    m := (t - C^.center.x_coord) *
                         (t - C^.center.x_coord) *
                         length * length;
                    m := C^.radius * C^.radius - m;
                    m := sqrt(abs( m ));
                    if C^.which_half = up then
                        y := C^.center.y_coord + m
```

```
                    else
                        y := C^.center.y_coord - m;
                    compute_point := y ;
                end;
    end; { case }

end; { compute_point }

procedure copy( C, D : curve_ptr );
{ copy the curve pointed by C to D; }
begin

    D^.curve_type := C^.curve_type;
    case C^.curve_type of
        line : begin
                    D^.curve_start := C^.curve_start;
                    D^.curve_end := C^.curve_end;
               end;
        arc  : begin
                    D^.curve_start := C^.curve_start;
                    D^.curve_end := C^.curve_end;
                    D^.center.x_coord := C^.center.x_coord;
                    D^.center.y_coord := C^.center.y_coord;
                    D^.radius := C^.radius;
                    D^.which_half := C^.which_half;
               end;
    end; { case }

end; { copy }

function fix_curve( C : curve_ptr; low, upp : t_value;
                    S : segment_ptr ) : curve_ptr;
{  fix the start and end points of the curve C }
var
        x1, x2 : t_value;
        D : curve_ptr;
begin

    new( D );
    copy( C, D );
    x1 := S^.lower_bound;
    x2 := S^.upper_bound;
    D^.curve_start := compute_point( low, C, x1, x2 );
    D^.curve_end   := compute_point( upp, C, x1, x2 );

    fix_curve :=  D ;

end; { fix_curve }

procedure fix_bound( var q : layer_ptr; low, upp : t_value;
                     S : segment_ptr );
{ This is for direct copy of intervals ( segments S ).
  go thru the q chain to fix the strat and end points
  of the curves.  }
var
        p, r : layer_ptr;
```



```
begin

    {  must save q (i.e. copy list ) before operation  }
    p := q;
    r := nil;
    while p <> nil do
    begin
        if r = nil then
            begin
                new( q );
                r := q;
            end
        else begin
                new( r^.next_layer );
                r := r^.next_layer;
             end;

        r^.inner_curve := fix_curve( p^.inner_curve,
                                     low, upp, S );
        r^.outer_curve := fix_curve( p^.outer_curve,
                                     low, upp, S );
        r^.next_layer := nil;

        p := p^.next_layer;
    end; { while }

end; { fix_bound }

function distance_to_axis( C : curve_ptr; low, upp : t_value;
                           S : segment_ptr) : real;
{ compute the distance from the mid-point of the curve C
  to principal axis; if C = nil, (means no data) then a
  minimal distance is assumed.  }
const
        min_distance = -1.0;
begin

    if C = nil then
        distance_to_axis := min_distance
    else begin

            distance_to_axis := compute_point(
                        (low + upp) / 2.0 , C,
                        S^.lower_bound, S^.upper_bound ) ;
         end;

end; { distance_to_axis }

function null_layer( a, b : curve_ptr ) : boolean;
{ to test if curve a and b form a null layer }
const
    EPS = 0.0001;
var
    result : boolean;
begin

```

```
    result := false;
    if ( abs (a^.curve_start - b^.curve_start) <= EPS )  and
       ( abs (a^.curve_end - b^.curve_end) <= EPS     )  and
       ( a^.curve_type = b^.curve_type ) then
       begin
          case a^.curve_type of
               line : result := true;
               arc  : result := (a^.radius = b^.radius) and
                        (a^.which_half = b^.which_half) and
                        ( abs(a^.center.x_coord -
                        b^.center.x_coord) <= EPS) and
                        ( abs(a^.center.y_coord -
                        b^.center.y_coord) <= EPS);
          end; { case }
       end;
    null_layer := result;

end; { null_layer }

procedure add_layer( upper_range, lower_range :curve_ptr;
                     var q, r :layer_ptr );
{ add a new layer bounded by [upper_range  lower_range]
  to the q chain ; r always points to the head of the q chain. }
begin

    if not null_layer( upper_range, lower_range ) then
        { null layer will be removed }
    begin

        if DEBUG_ADD_LAYER then  writeln(
            '$$$$ In add_layer, uppercurve(start,end), ',
            'lowercurve(start,end)=',
            upper_range^.curve_start, upper_range^.curve_end,
            lower_range^.curve_start, lower_range^.curve_end );

        if q = nil then  { the first one }
            begin
                new( q );
                q^.next_layer := nil;
                q^.outer_curve := upper_range;
                q^.inner_curve := lower_range;
                r := q;
            end
        else
            begin   { normal process }
                new( r^.next_layer );
                r := r^.next_layer;
                r^.outer_curve := upper_range;
                r^.inner_curve := lower_range;
                r^.next_layer := nil;
            end;
    end;

end; { add_layer }

procedure layer_union ( var q : layer_ptr; low, upp : t_value;
```



```
                        S1, S2 : segment_ptr );
{ actually union of layers specified in S1 and S2; both S1 and
  S2 are not nil; new layers are pointed by q and bounded by
  low and upp; }
var
        flag1, flag2 : lower_or_upper;
        p1, p2, r : layer_ptr;
        C1, C2 : curve_ptr;
        next_state : state;
        d1, d2 : real;
        a, b : curve_ptr;
begin

    flag1 := upper;
    flag2 := upper;
    p1 := S1^.layer_head;
    p2 := S2^.layer_head;
    C1 := get_curve( p1, flag1 );
    C2 := get_curve( p2, flag2 );
    d1 := distance_to_axis( C1, low, upp, S1 );
    d2 := distance_to_axis( C2, low, upp, S2 );
    q := nil;
    next_state := minuszero;

    repeat
      case next_state of
         minuszero :
              if d1 >= d2 then  { + }
                  begin
                      a := fix_curve( C1, low, upp, S1 );
                      next_state := pluszero;
                      C1 := get_curve( p1, flag1 );
                      d1 := distance_to_axis(C1,low,upp,S1);
                  end
              else
                  begin  { 1 }
                      a := fix_curve( C2, low, upp, S2 );
                      next_state := minusone;
                      C2 := get_curve( p2, flag2 );
                      d2 := distance_to_axis(C2,low,upp,S2 );
                  end;
         pluszero  :
              if d1 >= d2 then  { - }
                  begin
                      b := fix_curve( C1, low, upp, S1 );
                      add_layer( a, b, q, r );
                      next_state := minuszero;
                      C1 := get_curve( p1, flag1 );
                      d1 := distance_to_axis(C1,low,upp,S1);
                  end
              else
                  begin  { 1 }
                      next_state := plusone;
                      C2 := get_curve( p2, flag2 );
                      d2 := distance_to_axis(C2,low,upp,S2 );
                  end;
```

```
            plusone   :
                if dl >= d2 then  { - }
                    begin
                        next_state := minusone;
                        Cl := get_curve( pl, flagl );
                        dl := distance_to_axis(Cl,low,upp,Sl);
                    end
                else
                    begin  { 0 }
                        next_state := pluszero;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
            minusone  :
                if dl >= d2 then  { + }
                    begin
                        next_state := plusone;
                        Cl := get_curve( pl, flagl );
                        dl := distance_to_axis(Cl,low,upp,Sl);
                    end
                else
                    begin  { 0 }
                        b := fix_curve( C2, low, upp, S2 );
                        add_layer( a, b, q, r );
                        next_state := minuszero;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
        end; { case }
    until  ( Cl = nil ) and ( C2 = nil ) ;

end; { layer_union }

procedure layer_difference ( var q : layer_ptr;
                             low, upp : t_value;
                             Sl, S2 : segment_ptr );
{ actually difference of layers specified in Sl and S2;
  both Sl and S2 are not nil; new layers are pointed by q
  and bounded by low and upp; }
var
        flagl, flag2 : lower_or_upper;
        pl, p2, r : layer_ptr;
        Cl, C2 : curve_ptr;
        next_state : state;
        dl, d2 : real;
        a, b : curve_ptr;
begin

    flagl := upper;
    flag2 := upper;
    pl := Sl^.layer_head;
    p2 := S2^.layer_head;
    Cl := get_curve( pl, flagl );
    C2 := get_curve( p2, flag2 );
    dl := distance_to_axis( Cl, low, upp, Sl );
    d2 := distance to axis( C2, low, upp, S2 );
```



```
    q := nil;
    next_state := minuszero;

    repeat

        if DEBUG_LAYER_DIFFERENCE then writeln(
            '###### In layer difference, dl, d2=', dl, d2);

        case next_state of
            minuszero :
                if dl >= d2 then  { + }
                    begin
                        a := fix_curve( Cl, low, upp, Sl );
                        next_state := pluszero;
                        Cl := get_curve( pl, flagl );
                        dl := distance_to_axis(Cl,low,upp,Sl);
                    end
                else
                    begin  { 1 }
                        next_state := minusone;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
            pluszero  :
                if dl >= d2 then  { - }
                    begin
                        b := fix_curve( Cl, low, upp, Sl );
                        add_layer( a, b, q, r );
                        next_state := minuszero;
                        Cl := get_curve( pl, flagl );
                        dl := distance_to_axis(Cl,low,upp,Sl);
                    end
                else
                    begin  { 1 }
                        b := fix_curve( C2, low, upp, S2 );
                        add_layer( a, b, q, r );
                        next_state := plusone;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
            plusone   :
                if dl >= d2 then  { - }
                    begin
                        next_state := minusone;
                        Cl := get_curve( pl, flagl );
                        dl := distance_to_axis(Cl,low,upp,Sl);
                    end
                else
                    begin  { 0 }
                        a := fix_curve( C2, low, upp, S2 );
                        next_state := pluszero;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
            minusone  :
                if dl >= d2 then  { + }
```

```
                    begin
                        next_state := plusone;
                        C1 := get_curve( p1, flag1 );
                        d1 := distance_to_axis(C1,low,upp,S1);
                    end
                else
                    begin  { 0 }
                        next_state := minuszero;
                        C2 := get_curve( p2, flag2 );
                        d2 := distance_to_axis(C2,low,upp,S2 );
                    end;
        end; { case }
    until  ( C1 = nil ) and ( C2 = nil ) ;

    if DEBUG_LAYER_DIFFERENCE then writeln(
        '**** In layer_difference, before fix_bound ****');

end; { layer_differenc }

procedure compute_layer( operation : axis_op_kind;
                         var q : layer_ptr; low, upp : t_value;
                         S1, S2 : segment_ptr );
{ compute the new layers by S1 operation S2 and bounded by
  low and upp; results should be sorted according to depth.
  S1 or S2 may be nil, in this case direct copy the curve is
  possible which depends on what operation is applied.  New
  layer is pointed by q.  }
var
        S : segment_ptr;
begin

    case operation of
         union : begin

                 if DEBUG_COMPUTE_LAYER then writeln(
                    '$$$$ compute_layer, union op, low, upp='
                    ,low, upp);

                 if ( S1 <> nil ) and ( S2 <> nil ) then
                        layer_union( q, low, upp, S1, S2 )
                 else if ( S1 = nil ) and ( S2 = nil ) then
                        q := nil
                 else begin
                          if S1 = nil then
                               S := S2
                          else if S2 = nil then
                               S := S1;
                          q := S^.layer_head;
                          fix_bound( q, low, upp, S );
                      end;
                 end;
         difference :
                 begin

                     if DEBUG_COMPUTE_LAYER then writeln(
                        '$$$$ compute_layer, difference op,'
```



```
                        ,'low, upp=',low, upp);

                     if S1 = nil then
                          q := nil
                     else if S2 <> nil  then
                       layer_difference( q, low, upp, S1, S2 )
                     else  begin
                                q := S1^.layer_head;
                                fix_bound( q, low, upp, S1 );
                           end;
                 end;
    end; { case }

end; { compute_layer }

procedure add_segment( operation : axis_op_kind;
                       a, b : t_value; var p : segment_ptr;
                       S1, S2 : segment_ptr );
{ add a newly created segment [a b] into the principal axis P;
  where p is the segment pointer.  This routine should include
  interval intersection computation and layer merge operation.}
var
        Iptr : intersect_rec_ptr;
        low, upp : t_value;
begin

    if a >= b then
          begin
              { return }  { null interval; do nothing }
          end
    else  begin

         compute_intersection( S1, S2, Iptr, a, b );
         sort_intersection( Iptr );
         screen_intersection( Iptr, a, b );

         low := Iptr^.intersection;
         Iptr := Iptr^.next_intersect;

         repeat
             upp := Iptr^.intersection;
             p^.lower_bound := low;
             p^.upper_bound := upp;
             compute_layer(operation, p^.layer_head,
                           low, upp,S1,S2);
             if p^.layer_head <> nil then
                  begin
                       new( p^.next_segment );
                       p := p^.next_segment;
                       p^.next_segment := nil;
                  end;

             low := upp;
             Iptr := Iptr^.next_intersect;
         until Iptr = nil;

```

```
    end;

end; { add_segment }

function compute_D3_point( pt1, pt2 : D3_point;
                           t : t_value ) : D3_point;
{ given two end points pt1 and pt2, and a parameter t,
  compute its coordinate }
begin

    compute_D3_point.x_coord := pt1.x_coord +
                  t * (pt2.x_coord-pt1.x_coord);
    compute_D3_point.y_coord := pt1.y_coord +
                  t * (pt2.y_coord-pt1.y_coord);
    compute_D3_point.z_coord := pt1.z_coord +
                  t * (pt2.z_coord-pt1.z_coord);

end;  { compute_D3_point }

function get_next( var flag : lower_or_upper;
                   var done : boolean;
                   var  S, NS : segment_ptr ) : t_value;
{ get next interval value for use, done is set while no more
  can be obtained.  S should always point to the interval
  currently dealt with.  }
const
        max_t_value = 2.0;
begin

    if done then
         get_next := max_t_value
    else begin
            case flag of
                lower : begin
                            S := NS;
                            flag := upper;
                            if S <> nil then
                               get_next :=  S^.lower_bound
                            else begin
                                    done := true;
                                    get_next := max_t_value;
                                 end;
                        end;
                upper : begin
                            flag := lower;
                            NS := S^.next_segment;
                            get_next := S^.upper_bound
                        end;
            end; { case }
        end;

end; { get_next }

function partition ( operation : axis_op_kind;
                     P1, P2 : principal_axis ) : principal_axis;
{ compute new intervals for union/difference operation;
```



```
  basically it performs the merge step of merge_sort algorithm.}
var
        PX : principal_axis;
        S1, S2, NS1, NS2 : segment_ptr;
        flag1, flag2 : lower_or_upper;
        done1, done2 : boolean;
        a, b, a1, a2, actual_t, first_t, last_t : t_value;
        pt1, pt2 : D3_point;
        p : segment_ptr;
        next_state : state;
begin

    { need code to initialize PX.start_point and PX.end_point }
    PX.start_point := min_point;
    PX.end_point := max_point;

    new( PX.segment_head );
    p := PX.segment_head;
    p^.next_segment := nil;
    NS1 := P1.segment_head;
    NS2 := P2.segment_head;
    flag1 := lower;
    flag2 := lower;
    done1 := false;
    done2 := false;

    a1 := get_next( flag1, done1, S1, NS1 );
    a2 := get_next( flag2, done2, S2, NS2 );
    next_state := minuszero;

    repeat
        case next_state of
            minuszero :
                if a1 <= a2 then
                     begin
                        a := a1;
                        next_state := pluszero;
                        a1 := get_next( flag1, done1, S1, NS1 );
                     end
                else begin
                        a := a2;
                        next_state := minusone;
                        a2 := get_next( flag2, done2, S2, NS2 );
                     end;
            pluszero :
                if a1 <= a2 then
                     begin
                        b := a1;
                        next_state := minuszero;
                        add_segment(operation, a, b, p, S1,nil);
                        a1 := get_next( flag1, done1, S1, NS1 );
                     end
                else begin
                        b := a2;
                        next_state := plusone;
                        add_segment(operation, a, b, p, S1, nil);
```

```
                    a := a2;
                    a2 := get_next( flag2, done2, S2, NS2 );
                 end;
          plusone :
              if al <= a2 then
                 begin
                    b := al;
                    next_state := minusone;
                    add_segment(operation, a, b, p, Sl, S2);
                    a := al;
                    al := get_next( flagl, donel, Sl, NSl );
                 end
              else begin
                    b := a2;
                    next_state := pluszero;
                    add_segment(operation, a, b, p, Sl, S2);
                    a := a2;
                    a2 := get_next( flag2, done2, S2, NS2 );
                 end;
          minusone :
              if al <= a2 then
                 begin
                    b := al;
                    next_state := plusone;
                    if operation = union then
                        add_segment( operation, a, b,
                                     p, nil, S2 );
                    a := al;
                    al := get_next( flagl, donel, Sl, NSl );
                 end
              else begin
                    b := a2;
                    next_state := minuszero;
                    if operation = union then
                        add_segment( operation, a, b, p,
                                     nil, S2 );
                    a := a2;
                    a2 := get_next( flag2, done2, S2, NS2 );
                 end;
      end; { case }
   until donel and done2;

   { need code to delete the last element of the segment
     ( null segment ) }
   p := PX.segment_head;
   if p^.next_segment = nil then
       begin
           PX.segment_head := nil;
           actual_t := 0.0;
       end;

   while ( p <> nil ) do
   begin
       if p^.next_segment^.next_segment = nil then
       begin
           p^.next_segment := nil;
```



```
               actual_t := p^.upper_bound;
           end;
           p := p^.next_segment;
       end;

       first_t := PX.segment_head^.lower_bound;
       last_t  := actual_t;

       { if operation = difference, should reshape the t
         parameters because the first and last several segments
         might be removed.  recalucate the end point and the t
         parameters.  }
       if ( operation = difference ) and
          (( last_t < 1.0 ) or ( first_t > 0.0 )) then
           begin
               ptl := PX.start_point;
               pt2 := PX.end_point;
               PX.start_point := compute_D3_point(
                                     ptl, pt2, first_t );
               PX.end_point := compute_D3_point(
                                     ptl, pt2, last_t );
               adjust_t_value( PX, first_t, last_t, 0.0, 1.0 );
           end;

       partition :=  PX ;

       if DEBUG_PARTITION then writeln(
           '%%%%% In partition, PX.start and end point=',
           PX.start_point.x_coord, PX.start_point.y_coord,
           PX.start_point.z_coord, PX.end_point.x_coord,
           PX.end_point.y_coord,PX.end_point.z_coord);

   end; { partition }

   function smooth_curve( xl, x2 : t_value; Cl : curve_ptr ;
                    x3, x4 : t_value; C2 : curve_ptr ) : boolean;
   { check to see if the curve Cl and C2 are smoothly connected.
     note that Cl and C2 have the same curve type here.  }
   const
           EPSILON = 0.00001;
           INFINITY = 99999.99999;
   var
           ml, m2 : real;
   begin

       case Cl^.curve_type of
            line : begin  { check if they have the same slope }
                       if abs(xl -x2) <= EPSILON then
                              ml := INFINITY
                       else   ml := ( Cl^.curve_start -
                                  Cl^.curve_end )/( xl - x2 );
                       if abs(x3 -x4) <= EPSILON then
                              m2 := INFINITY
                       else   m2 := ( C2^.curve_start -
                                  C2^.curve_end )/( x3 - x4 );
                       if abs(ml - m2) <= EPSILON then
```

```
                                   smooth_curve := true
                        else smooth_curve := false ;
                    end;
            arc   : begin
                        if C1^.radius <> C2^.radius then
                              smooth_curve := false
                        else if C1^.which_half <> C2^.which_half
                             then smooth_curve := false
                        else if ( C1^.center.x_coord <>
                                  C2^.center.x_coord ) or
                                ( C1^.center.y_coord <>
                                  C2^.center.y_coord ) then
                                 smooth_curve := false
                        else     smooth_curve := true ;
                    end;
      end; { case }

end; { smooth_curve }

function check_smooth( q, r : segment_ptr;
                       a, b : layer_ptr ) : boolean;
{ within segments q and r, check if layer a and b
  are smoothly connected.  }
var
        flag : boolean;
        x1, x2, x3, x4 : t_value;
begin

    x1 := q^.lower_bound;
    x2 := q^.upper_bound;
    x3 := r^.lower_bound;
    x4 := r^.upper_bound;

    flag := smooth_curve( x1, x2, a^.inner_curve,
                          x3, x4, b^.inner_curve );
    if flag  then
         flag := smooth_curve( x1, x2, a^.outer_curve,
                               x3, x4, b^.outer_curve );
    check_smooth :=  flag ;

end; { check_smooth }

function check_connect( q, r : segment_ptr ) : boolean;
{ To check if q and r are connected; check the following
  condition :
        the same connecting point, the same curve type,
        smooth at the connecting point,
        the same number of layers.  }
var
        a, b : layer_ptr;
        done : boolean;
begin

    if q^.upper_bound <> r^.lower_bound  then
        check_connect := false
    else
```



```
        begin
            a := q^.layer_head;
            b := r^.layer_head;
            done := false;
            while  not done do
            begin
               if (a^.inner_curve^.curve_type <>
                   b^.inner_curve^.curve_type) or
                  (a^.outer_curve^.curve_type <>
                   b^.outer_curve^.curve_type)
               then  begin
                         done := true;
                         check_connect := false ;
                     end
               else  if ( a^.inner_curve^.curve_end <>
                          b^.inner_curve^.curve_start ) or
                        ( a^.outer_curve^.curve_end <>
                          b^.outer_curve^.curve_start ) then
                          begin
                              done := true;
                              check_connect := false ;
                          end
               else if  not check_smooth( q, r, a, b ) then
                          begin
                              done := true;
                              check_connect := false ;
                          end
               else  { ok for this layer, let's try next }
                         begin
                             a := a^.next_layer;
                             b := b^.next_layer;
                             if ( a = nil ) and ( b <> nil ) or
                                ( b = nil ) and ( a <> nil ) then
                                 begin
                                     done := true;
                                     check_connect := false ;
                                 end
                             else if (a = nil) and (b = nil) then
                                 begin
                                     done := true;
                                     check_connect := true  ;
                                 end;
                         end;
            end;  { while }
        end; { else }

end; { check_connect }

procedure merge_curve( C1, C2 : curve_ptr );
{ merge the curve C1 and C2; result is in C1; }
begin

    case C1^.curve_type of
        line : C1^.curve_end := C2^.curve_end;
        arc  : C1^.curve_end := C2^.curve_end;
    end; { case }
```

```

end; { merge_curve }

procedure connect( q, r : segment_ptr );
{  merge q and r into a new interval pointed by q.
}
var
        p,s : layer_ptr;
begin

    { merge segment }
    q^.upper_bound := r^.upper_bound;
    q^.next_segment := r^.next_segment;

    { merge each layer }
    p := q^.layer_head;
    s := r^.layer_head;
    while ( p <> nil ) do
    begin
        merge_curve( p^.inner_curve, s^.inner_curve );
        merge_curve( p^.outer_curve, s^.outer_curve );

        p := p^.next_layer;
        s := s^.next_layer;
    end; { while }

end; { connect }

procedure merge_interval ( var P : principal_axis );
{ try to merge consecutive intervals into a larger one;
  this step is important to enforce the uniqueness of
  this principal-axis representation. }
var
        q, r : segment_ptr;
begin

    q := P.segment_head;
    r := q^.next_segment;
    while ( r <> nil ) do
    begin
        if check_connect( q, r ) then
            begin
                connect( q, r );
                r := q^.next_segment;
            end
        else
            begin
                q := r;
                r := q^.next_segment;
            end;
    end;  { end }

    if DEBUG_MERGE_INTERVAL then writeln(
       '%%%%% In merge_interval,P.start and end point=',
       P.start_point.x_coord, P.start_point.y_coord,
       P.start_point.z_coord, P.end_point.x_coord,
```



```
       P.end_point.y_coord,P.end_point.z_coord);

end;  { merge_interval }

function axis_operation ( operation : axis_op_kind;
                   P1, P2 : principal_axis ) : principal_axis;
{  union/difference two axes P1 and P2 into a new axis;
   this operation performs the actual union/difference of
   two CSG trees;  it first calls normalization procedure
   to make two trees have t-parameters on the same basis,
   then applies the merge routine of merge-sort algorithm
   to union/difference them together. }
var
        P : principal_axis;
begin

    if DEBUG_AXIS_OP then writeln('%%%% enter normalization');

    normalization ( P1, P2, min_point, max_point );

    if DEBUG_AXIS_OP then writeln('%%%% leave normalization');

    if DEBUG_AXIS_OP then writeln('%%%% enter partition');

    P := partition ( operation, P1, P2 );

    if DEBUG_AXIS_OP then writeln('%%%% leave partition');

    if TRACE_AXIS then dump_axis( P );

    if DEBUG_AXIS_OP then writeln('%%%% enter merge_interval');

    merge_interval ( P );

    if DEBUG_AXIS_OP then writeln('%%%% leave merge_interval');

    if TRACE_AXIS then dump_axis( P );

    axis_operation :=  P ;

end; { axis_operation }

procedure rot_x( angle : real; var p : D3_point;
                 p0 : D3_point );
{ rotate the point p about x-axis by amount of angle;
  since the rotation is relative to local coordinate system
  of its own, translation must be done before and after
  rotation. }
var
        y, z : real;
begin

    { translation to the origin of local coordinate }
    p.x_coord := p.x_coord - p0.x_coord ;
    p.y_coord := p.y_coord - p0.y_coord ;
    p.z_coord := p.z_coord - p0.z_coord ;
```

```

    { rotate }
    y := p.y_coord;
    z := p.z_coord;
    angle := angle * PI / 180.0 ; { convert to radian degree }
    p.y_coord := y * cos( angle ) - z * sin( angle );
    p.z_coord := y * sin( angle ) + z * cos( angle );

    { translation back to the global coordinate }
    p.x_coord := p.x_coord + p0.x_coord ;
    p.y_coord := p.y_coord + p0.y_coord ;
    p.z_coord := p.z_coord + p0.z_coord ;

end; { rot_x }

procedure rot_y( angle : real; var p : D3_point;
                 p0 : D3_point );
{ rotate the point p about y-axis by amount of angle,
  since the rotation is relative to local coordinate system
  of its own, translation must be done before and after
  rotation. }
var
      x, z : real;
begin

    { translation to the origin of local coordinate }
    p.x_coord := p.x_coord - p0.x_coord ;
    p.y_coord := p.y_coord - p0.y_coord ;
    p.z_coord := p.z_coord - p0.z_coord ;

    { rotate }
    x := p.x_coord;
    z := p.z_coord;
    angle := angle * PI / 180.0 ; { convert to radian degree }
    p.x_coord := x * cos( angle ) + z * sin( angle );
    p.z_coord := - x * sin( angle ) + z * cos( angle );

    { translation back to the global coordinate }
    p.x_coord := p.x_coord + p0.x_coord ;
    p.y_coord := p.y_coord + p0.y_coord ;
    p.z_coord := p.z_coord + p0.z_coord ;

end; { rot_y }

procedure rot_z( angle : real; var p : D3_point;
                 p0 : D3_point );
{ rotate the point p about z-axis by amount of angle,
  since the rotation is relative to local coordinate system
  of its own, translation must be done before and after
  rotation. }
var
      x, y : real;
begin

    { translation to the origin of local coordinate }
    p.x_coord := p.x_coord - p0.x_coord ;
```



```
    p.y_coord := p.y_coord - p0.y_coord ;
    p.z_coord := p.z_coord - p0.z_coord ;

    { rotate }
    x := p.x_coord;
    y := p.y_coord;
    angle := angle * PI / 180.0 ; { convert to radian degree }
    p.x_coord := x * cos( angle ) - y * sin( angle );
    p.y_coord := x * sin( angle ) + y * cos( angle );

    { translation back to the global coordinate }
    p.x_coord := p.x_coord + p0.x_coord ;
    p.y_coord := p.y_coord + p0.y_coord ;
    p.z_coord := p.z_coord + p0.z_coord ;

end; { rot_z }

procedure translate( xform : xform_matrix ; var p : D3_point );
{ translate the point p by the amount of the translation part
  of xform }
begin

    p.x_coord := p.x_coord + xform.translate_x;
    p.y_coord := p.y_coord + xform.translate_y;
    p.z_coord := p.z_coord + xform.translate_z;

end; { translate }

procedure transform ( xform : xform_matrix;
                      var P : principal_axis );
{ transform the tree by a transformation matrix;
  actually the principal axis is transformed;
  more specifically only the two end points are
  transformed.  }

begin

    { P.start_point := P.start_point * transform_matrix;}
    { P.end_point   := P.end_point   * transform_matrix;}
    { note other variables are transformation invariant }

    { let's do the rotation first }
    if xform.rotate_x <> 0.0 then
        rot_x( xform.rotate_x, P.end_point, P.start_point);

    if xform.rotate_y <> 0.0 then
        rot_y( xform.rotate_y, P.end_point, P.start_point);

    if xform.rotate_z <> 0.0 then
        rot_z( xform.rotate_z, P.end_point, P.start_point);

    { and then do the translation }
    if ( xform.translate_x <> 0.0 ) or
       ( xform.translate_y <> 0.0 ) or
       ( xform.translate_z <> 0.0 )    then
       begin
```

```
            translate( xform, P.start_point );
            translate( xform, P.end_point );
        end;

end; { transform }

function build_axis ( T : CSG_tree ) : principal_axis;
{ convert the primitive solid T to the principal_axis
  type representation }
var
        P : principal_axis;
        s : segment_ptr;
        l : layer_ptr;
        height : real;
begin

    P.start_point := T.prim_solid.base;
    P.end_point   := T.prim_solid.top;

    if DEBUG_BUILD_AXIS then writeln(
        '**** In build_axis,P.start and end point=',
        P.start_point.x_coord, P.start_point.y_coord,
        P.start_point.z_coord, P.end_point.x_coord,
        P.end_point.y_coord, P.end_point.z_coord);

    if DEBUG_BUILD_AXIS then writeln(
       '**** In build_axis,Prim_solid.base and top point=',
       T.prim_solid.base.x_coord,T.prim_solid.base.y_coord,
       T.prim_solid.base.z_coord, T.prim_solid.top.x_coord,
       T.prim_solid.top.y_coord,T.prim_solid.top.z_coord);

    new( P.segment_head );

    s := P.segment_head;
    s^.lower_bound := 0.0;
    s^.upper_bound := 1.0;
    s^.next_segment := nil;
    new( s^.layer_head );

    l := s^.layer_head;
    l^.next_layer := nil;
    new( l^.inner_curve );
    new( l^.outer_curve );

    case T.prim_solid.solid_type of

         cylinder :
             begin
                 l^.outer_curve^.curve_type  := line;
                 l^.outer_curve^.curve_start :=
                                 T.prim_solid.radius;
                 l^.outer_curve^.curve_end   :=
                                 T.prim_solid.radius;

                 l^.inner_curve^.curve_type  := line;
                 l^.inner_curve^.curve_start := 0.0;
```



```
                 l^.inner_curve^.curve_end   := 0.0;
             end;

         cone :
             begin
                 l^.outer_curve^.curve_type  := line;
                 l^.outer_curve^.curve_start :=
                                 T.prim_solid.radius_b;
                 l^.outer_curve^.curve_end   :=
                                 T.prim_solid.radius_t;

                 l^.inner_curve^.curve_type  := line;
                 l^.inner_curve^.curve_start := 0.0;
                 l^.inner_curve^.curve_end   := 0.0;
             end;

         torus :
             begin
                 l^.outer_curve^.curve_type  := arc;
                 height := ( T.prim_solid.inner_radius +
                       T.prim_solid.outer_radius ) / 2.0;
                 l^.outer_curve^.curve_start := height;
                 l^.outer_curve^.curve_end   := height;
                 l^.outer_curve^.center.x_coord := 0.5;
                 l^.outer_curve^.center.y_coord := height;
                 l^.outer_curve^.radius      :=
                       T.prim_solid.outer_radius - height;
                 l^.outer_curve^.which_half  := up;

                 l^.inner_curve^.curve_type  := arc;
                 l^.inner_curve^.curve_start := height;
                 l^.inner_curve^.curve_end   := height;
                 l^.inner_curve^.center.x_coord := 0.5;
                 l^.inner_curve^.center.y_coord := height;
                 l^.inner_curve^.radius      := height -
                                 T.prim_solid.inner_radius;
                 l^.inner_curve^.which_half  := down;
             end;

    end; { case }

    build_axis :=  P ;

end; { build_axis }

function evaluate_CSG ( T : CSG_tree_ptr ) : principal_axis;
{ recursive call itself to evaluate the CSG tree T^;
  it is basically a tree traversal algorithm.
  Our assumption is that the two operands at this point must
  be coaxial although they might not be coaxial at the lower
  part of the tree;    }
var
        P, P1, P2 : principal_axis;
begin

    if DEBUG_EVALUATE_CSG then writeln(
```

```
        '*** in evaluate_csg , partno =', T^.partno );

   case T^.node_type of
       movement :
           begin
               P := evaluate_CSG ( T^.child );
               transform( T^.move, P );
               evaluate_CSG := P ;

               if DEBUG_EVALUATE_CSG then writeln(
                   '*** finish evaluate_csg , partno =',
                   T^.partno );

           end;
       union_op :
           begin
               P1 := evaluate_CSG ( T^.left_child );
               P2 := evaluate_CSG ( T^.right_child );

               if DEBUG_EVALUATE_CSG then writeln(
                   '#### axis operation for partno =',
                   T^.partno );

               P := axis_operation ( union, P1, P2 );
               evaluate_CSG := P ;

               if DEBUG_EVALUATE_CSG then writeln(
                   '*** finish evaluate_csg , partno =',
                   T^.partno );

               if DEBUG_EVALUATE_CSG then writeln(
                  '!!!! result of union,P.start and end point=',
                  P.start_point.x_coord,P.start_point.y_coord,
                  P.start_point.z_coord, P.end_point.x_coord,
                  P.end_point.y_coord,P.end_point.z_coord);

           end;
       diff_op  :
           begin
               P1 := evaluate_CSG ( T^.left_child );
               P2 := evaluate_CSG ( T^.right_child );

               if DEBUG_EVALUATE_CSG then writeln(
                   '#### axis operation for partno =',
                   T^.partno );

               P := axis_operation ( difference, P1, P2 );
               evaluate_CSG := P ;

               if DEBUG_EVALUATE_CSG then writeln(
                   '*** finish evaluate_csg , partno =',
                   T^.partno );

           end;
       primitive:
           begin
```



```
               P := build_axis ( T^ );
               evaluate_CSG :=  P ;

               if DEBUG_EVALUATE_CSG then writeln(
                   '*** finish evaluate_csg , partno =',
                   T^.partno );

           end;
   end; { case }

end; { evaluate_CSG }

procedure get_input_data( var CSG : CSG_tree_ptr );
{ get the input CSG tree from outside world; it may be
  from the output of an interactive graphical geometrical
  modeller; EECS487 term project, for example. }
const
    obj_path  = 'object';
    mov_path  = 'movement';
    prim_path = 'primitives';

var
    openstatus : status_$t;

    procedure read_tree( T : CSG_tree_ptr );
    { recurvely read data from object, primitives, and
      movement files to construct a CSG tree T }
    var
        part1, part2, part3 : integer;
        ch : char;
        op : array[1..9] of char;
        prm : array[1..8] of char;
        ax, ay, az, x, y, z, x1, y1, z1 : real;

    begin

        readln( obj_file, part1, ch, ch, ch, ch, ch, ch, ch,
                          op, part2, part3 );

        if DEBUG_INPUT then writeln(
            '**** read object, objno =',part1, op);

        if (T^.partno <> CSG_ROOT) and
           ( T^.partno <> part1 ) then
          writeln('----> read_tree error,',
                  ' input tree structure incorrect!!');

        if      op = '    union' then T^.node_type := union_op
        else if op = '   differ' then T^.node_type := diff_op
        else if op = 'primitive' then T^.node_type := primitive
        else if op = 'moved_obj' then T^.node_type := movement
        else writeln('illegal CSG node type!!');

    { T^.partno := part1;  }

        case T^.node_type of
```

```
        union_op,
        diff_op   :
            begin

                new( T^.left_child );
                T^.left_child^.partno := part2;
                read_tree( T^.left_child );

                new( T^.right_child );
                T^.right_child^.partno := part3;
                read_tree( T^.right_child );

            end;
        primitive :
            begin
                readln( prim_file, partl, ch,
                        ch, ch, ch, ch, ch, prm,
                        xl, yl, zl, x, y, z,
                        ax, ay, az );

                if DEBUG_INPUT then writeln(
                    '**** read primitive, no =',
                    partl,'  ',prm);

                if ( partl <> T^.partno ) then
                    writeln('----> read_tree error,',
                    'input tree structucture incorrect!!');

                { must treat cone specially }
                if prm = 'cylinder' then
                    T^.prim_solid.solid_type := cylinder
                else if prm = '    cone' then
                    T^.prim_solid.solid_type := cone
                else if prm = '   torus' then
                    T^.prim_solid.solid_type := torus
                else writeln( 'illegal CSG ',
                     'primitive node type!!');

                case T^.prim_solid.solid_type of
                    cylinder :
                        begin
                        T^.prim_solid.base.x_coord := x;
                        T^.prim_solid.base.y_coord := y;
                        T^.prim_solid.base.z_coord := z;
                        T^.prim_solid.top.x_coord := x;
                        T^.prim_solid.top.y_coord := y;
                        T^.prim_solid.top.z_coord := z+xl;
                        T^.prim_solid.radius := yl;
                        end;
                    cone     :
                        begin
                          T^.prim_solid.base.x_coord := x;
                          T^.prim_solid.base.y_coord := y;
                          T^.prim_solid.base.z_coord := z;
                          T^.prim_solid.top.x_coord := x;
                          T^.prim_solid.top.y_coord := y;
```



```
                          T^.prim_solid.top.z_coord := z+xl;
                          T^.prim_solid.radius_b := yl;
                          T^.prim_solid.radius_t := 0.0;
                        end;
                    torus    :
                        begin
                          T^.prim_solid.base.x_coord := x;
                          T^.prim_solid.base.y_coord := y;
                          T^.prim_solid.base.z_coord := z-yl;
                          T^.prim_solid.top.x_coord := x;
                          T^.prim_solid.top.y_coord := y;
                          T^.prim_solid.top.z_coord := z+yl;
                          T^.prim_solid.inner_radius :=xl-yl;
                          T^.prim_solid.outer_radius :=xl+yl;
                        end;
                end; { case }

                if ax <> 0.0 then
                    rot_x( ax, T^.prim_solid.top,
                           T^.prim_solid.base  );

                if ay <> 0.0 then
                    rot_y( ay, T^.prim_solid.top,
                           T^.prim_solid.base  );

                if az <> 0.0 then
                    rot_z( az, T^.prim_solid.top,
                           T^.prim_solid.base  );

            end;
        movement  :
            begin
                readln( mov_file, partl,
                        T^.move.translate_x,
                        T^.move.translate_y,
                        T^.move.translate_z,
                        T^.move.rotate_x,
                        T^.move.rotate_y,
                        T^.move.rotate_z );

                if DEBUG_INPUT then writeln(
                    '**** read movement, no =', partl);

                if ( partl <> part3 ) then
                    writeln('----> read_tree error,',
                   'input tree structucture incorrect!!');

                new( T^.child );
                T^.child^.partno := part2;
                read_tree( T^.child );
            end;
    end; { case }

end; { read_tree }

begin
```

```

    { read data and construct them as a CSG tree;
      there are three data files; one is the movement,
      another is primitives, the third one is the
      construction . }
    open( obj_file, obj_path, 'OLD', openstatus.all );
    open( mov_file, mov_path, 'OLD', openstatus.all );
    open( prim_file, prim_path, 'OLD', openstatus.all );

        reset( obj_file );
        reset( mov_file );
        reset( prim_file );

    if DEBUG_INPUT then writeln(
        '*** open files in get_input_data');

    new( CSG );
    CSG^.partno := CSG_ROOT;
    read_tree( CSG );

    if DEBUG_INPUT then writeln(
        '*** finish get_input_data');

end; { get_input_data }

procedure compute_length_diameter( PX : principal_axis;
                                   var L, D : real );
{ compute the maximal length L of PX; note:
  PX might be broken into several segments.  Compute the
  extremal boundary (diameter D) from the axis }
var
        p, q        : segment_ptr;
        len, maxlen : t_value;
        xl, yl, zl  : real;
        maxht, height, local_height : real;
        C : curve_ptr;
begin

    maxlen := 0.0;
    len := 0.0;
    maxht := 0.0;
    local_height := 0.0;
    p := PX.segment_head;

    while p <> nil do
    begin
        C := p^.layer_head^.outer_curve;
        case C^.curve_type of
        line : if C^.curve_start > C^.curve_end then
                        height := C^.curve_start
                 else     height := C^.curve_end;
        arc  : if C^.which_half = up then
                   begin
                       if ( C^.center.x_coord >=
                            p^.lower_bound ) and
                          ( C^.center.x_coord <=
```



```
                            p^.upper_bound ) then
                          height := C^.center.y_coord +
                                    C^.radius
                       else if C^.center.x_coord >=
                                p^.upper_bound then
                          height := C^.curve_end
                       else
                          height := C^.curve_start;
                   end
                 else  { down }
                   begin
                       if C^.curve_start > C^.curve_end then
                               height := C^.curve_start
                       else     height := C^.curve_end;
                   end;
        end; { case }

        if height > local_height then
            local_height := height;

        len := len + p^.upper_bound - p^.lower_bound;
        q := p^.next_segment;
        if q <> nil then
        begin
            if ( q^.lower_bound - p^.upper_bound ) > 0.0001
            then
                begin  { non-continuous segments }
                    if len > maxlen then
                        begin
                            maxlen := len;
                            maxht := local_height;
                        end;
                    len := 0.0;
                    local_height := 0.0;
                end;
        end;
        p := q;
    end;

    if len > maxlen then
        begin
        maxlen := len;
        maxht := local_height;
        end;

    xl := PX.start_point.x_coord - PX.end_point.x_coord;
    yl := PX.start_point.y_coord - PX.end_point.y_coord;
    zl := PX.start_point.z_coord - PX.end_point.z_coord;

    if DEBUG_LD then writeln(
        '*** Compute L&D, xl, yl, zl, maxlen =',
                  xl, yl, zl, maxlen );

    L := maxlen * sqrt( xl * xl + yl * yl + zl * zl );

    D := 2.0 * maxht;
```

```
        file_name := 'output.data';
        name_size := 11;

        { open the disk file for external storage }

        gpr_$open_bitmap_file( gpr_$create,file_name,
                name_size,version,window.window_size,
                groups,header,attribs,filebm,created,status);
        if status.all <> status_$ok then writeln(
            '----> In produce_bitmap, Can not open the ',
            'bitmap file for the specified file name');

        { set current bitmap for block transfer }

        gpr_$set_bitmap( filebm, status);
        if status.all <> status_$ok then writeln(
            '----> In produce_bitmap, Can not set ',
            'the bitmap file');

        { blok transfer data from current bitmap to disk }

        gpr_$pixel_blt( init_bitmap, window,
                        window.window_base, status);
        if status.all <> status_$ok then writeln(
            '----> In produce_bitmap, Can not Block ',
            'transfer bitmap file');

         { set back the original display bitmap }

        gpr_$set_bitmap( init_bitmap, status);


     end;  { produce_bitmap_file }


    procedure terminate_graph;
    { terminate the graphic environment ;
      probably make a hard copy. }
    begin

        gpr_$terminate( false, status );

    end; { terminate_graph }


    procedure draw_line( x1, y1, x2, y2 : real );
    { draw a line from ( x1, y1 ) to ( x2, y2 );
      this procedure should interface graphic routine
    }
    var
        xx1, yy1, xx2, yy2 : integer;
    begin

        if DEBUG_DRAW then writeln(
            '&&&& draw_line,x1,y1,x2,y2',x1,y1,x2,y2);

```



```
        xx1 := trunc( x1 + 0.5 ) + x_origin;
        yy1 := trunc( y1 + 0.5 ) + y_origin;
        xx2 := trunc( x2 + 0.5 ) + x_origin;
        yy2 := trunc( y2 + 0.5 ) + y_origin;

        { call graphic routine to draw aline from
          ( xx1, yy1 ) to ( xx2, yy2 ) }
        gpr_$move( xx1, yy1, status );
        gpr_$line( xx2, yy2, status );


    end;  { draw_line }

    procedure draw_arc( xc, yc, radii : real;
                        half : up_or_down;
                        x1, y1, x2, y2 : real );
    {  draw an arc of a circle where its center is
       at ( xc, yc ) of radius radii; the arc starts from
       ( x1, y1 ) to ( x2, y2 ) and half specifies either
       the upper part or the lower part in the circle
       this arc belongs to.  This routine should interface
       the graphic routine to draw arc, circle or lines
       if approximation is used.  }
    var
        x3, y3, dy : real;
        first, middle, last : gpr_$position_t;
    begin

        if DEBUG_DRAW then writeln(
            '&&&& draw_arc,xc,yc,x1,y1,x2,y2',
            xc,yc,x1,y1,x2,y2);

        { compute the middle point of the arc }
        x3 := ( x1 + x2 ) / 2.0;
        dy := sqrt( radii * radii -
                  ( x3 - xc ) * ( x3 - xc ) );
        if half = up then y3 := yc + dy
                  else y3 := yc - dy;

        { convert real coordinates to integer coordinates }
        first.x_coord := trunc( x1 + 0.5 ) + x_origin ;
        first.y_coord := trunc( y1 + 0.5 ) + y_origin ;
        middle.x_coord := trunc( x3 + 0.5 ) + x_origin;
        middle.y_coord := trunc( y3 + 0.5 ) + y_origin;
        last.x_coord := trunc( x2 + 0.5 ) + x_origin;
        last.y_coord := trunc( y2 + 0.5 ) + y_origin;

        { call graphic routine , check Apollo Domain
          graphic routine }
        gpr_$move( first.x_coord , first.y_coord , status );
        gpr_$arc_3p( middle, last, status );


    end; { draw_arc }

    procedure draw_curve( C : curve_ptr; x1, x2 : real );
```

```

    writeln('-----> length of the part =', L);
    writeln('-----> diameter of the part =', D);

end; { compute_length_diameter }

procedure compute_profile ( PX : principal_axis );
{ obtain profile of part from PX; intend to support
  NC application }
const
        { to adjust the output on the center of the screen }
        x_origin = 24;
        y_origin = 512;
var
        p : segment_ptr;
        ratio, xl, yl, zl, x_position : real;
        aa, bb : real;
        status : status_$t;
        disp_bm_size : gpr_$offset_t;
        init_bitmap :  gpr_$bitmap_desc_t;
        window : gpr_$window_t;

    procedure init_graph;
    { initialize the Apollo Domain graphic environment.
      Clipping to window is implemented; be careful !! }
    begin

        disp_bm_size.x_size := 1024;
        disp_bm_size.y_size := 1024;
        gpr_$init( gpr_$borrow, 1, disp_bm_size, 0,
                   init_bitmap, status );

        { set window clipping }
        window.window_base.x_coord := 0;
        window.window_base.y_coord := 0;
        window.window_size.x_size  := 1024;
        window.window_size.y_size  := 1024;

        { set 'exclusive or' raster operation }
        gpr_$set_raster_op( 0, 6, status );

        gpr_$set_clip_window( window, status );
        gpr_$set_clipping_active( true, status );
        gpr_$clear( -2, status );

    end; { init_graph }

    procedure hold_screen;
    { hold screen for a moment; say 30 seconds }
    const
        one_second = 250000;
    var
        wait_time : time_$clock_t;
        status : status_$t;
    begin

```



```
        wait_time.high16 := 0;
        wait_time.low32 := 30 * one_second;
        time_$wait( time_$relative, wait_time, status);

    end; { hold_screen }


    procedure produce_bitmap_file;
    { store the bitmap of the whole screen into disk,
      open the disk file for it
      if file exists or other errors, write error messages }
    var
        window:     gpr_$window_t   ;
            { window of the display bitmap }
        hi_plane:   gpr_$plane_t    ;
        groups:     integer         ;
            { the number of groups in external bitmap }
        status:     status_$t;
            { returned status }
        version:    gpr_$version_t;
            { version number of bitmap file }
        header:     gpr_$bmf_group_header_array_t;
            { descriptor of external group bitmap header }
        attribs:    gpr_$attribute_desc_t;
            { attributes which the bitmap will use }
        filebm:     gpr_$bitmap_desc_t;
            { descriptor of bitmap }
        created:    boolean;
            { specifies whether the bitmap file was created }
        file_name:  name_$pname_t;
            { name of the external file for bitmap }
        name_size:  integer;
            { length of the file name }

    begin

        {  set parameters for the window of operation }

        hi_plane := 0;
        groups := 1;
        window.window_base.x_coord := 0;
        window.window_base.y_coord := 0;
        window.window_size.x_size := 1000;
        window.window_size.y_size := 800;
        with header[0] do
        begin
            n_sects := hi_plane + 1;
            pixel_size := 1;
            allocated_size := 1;
            bytes_per_line := 0;
            bytes_per_sect := 0;
        end;  {with }

        gpr_$allocate_attribute_block( attribs, status);

        { get_file_name( file_name, name_size); }
```

```
    {  draw a curve specified by C;
       xl ans x2 are horizontal bounds.  }
    var
       xc, yc : real;
    begin

       case C^.curve_type of
               line : draw_line( xl, C^.curve_start,
                                 x2, C^.curve_end );
               arc  : begin
                         xc := ratio * C^.center.x_coord;
                         yc := C^.center.y_coord;
                         draw_arc( xc, yc, C^.radius,
                                   C^.which_half, xl,
                                   C^.curve_start, x2,
                                   C^.curve_end );
                      end;
       end; { case }

    end;  { draw_curve }

    procedure draw_curve_buddy( C : curve_ptr; xl, x2 : real );
    {  draw the counterpart of a curve specified by C;
       xl ans x2 are horizontal bounds.  }
    var
       D : curve_ptr;
    begin

       new( D );
       D^.curve_type := C^.curve_type;
       case C^.curve_type of
               line : begin
                         D^.curve_start := - C^.curve_start;
                         D^.curve_end := - C^.curve_end;
                      end;
               arc  : begin
                         D^.curve_start := - C^.curve_start;
                         D^.curve_end := - C^.curve_end;
                         D^.center.x_coord := C^.center.x_coord;
                         D^.center.y_coord := -C^.center.y_coord;
                         D^.radius := C^.radius;
                         if C^.which_half = up then
                               D^.which_half := down
                         else
                               D^.which_half := up;
                      end;
       end; { case }

       draw_curve( D, xl, x2 );

    end;  { draw_curve_buddy }


    procedure draw_layer( q : layer_ptr; xl, x2 : real );
    { to draw the layer curves by going thru q chain;
      each counterpart of the curve in q must also be drawn;
```



```
    }
    begin

       while q <> nil do
       begin
          draw_curve( q^.outer_curve, xl, x2 );
          draw_curve( q^.inner_curve, xl, x2 );
          draw_line( xl, q^.outer_curve^.curve_start,
                     xl, q^.inner_curve^.curve_start );
          draw_line( x2, q^.outer_curve^.curve_end,
                     x2, q^.inner_curve^.curve_end );

          draw_curve_buddy( q^.inner_curve, xl, x2 );
          draw_curve_buddy( q^.outer_curve, xl, x2 );
          draw_line( xl, -q^.outer_curve^.curve_start,
                     xl, -q^.inner_curve^.curve_start );
          draw_line( x2, -q^.outer_curve^.curve_end,
                     x2, -q^.inner_curve^.curve_end );

          q := q^.next_layer;
       end;

    end; { draw_layer }



begin

       { initialize the Apollo graphic environment }
       init_graph;

       xl := PX.end_point.x_coord - PX.start_point.x_coord;
       yl := PX.end_point.y_coord - PX.start_point.y_coord;
       zl := PX.end_point.z_coord - PX.start_point.z_coord;
       ratio := sqrt( xl * xl + yl * yl + zl * zl );
       x_position := 0.0;

       p := PX.segment_head;

       while p <> nil do
       begin
          x_position := ratio * p^.lower_bound;
          xl := ratio * p^.upper_bound;
          draw_layer( p^.layer_head, x_position, xl );

          p := p^.next_segment;
       end; { while }

    { hold the screen for a movement }
    hold_screen;

    { generate the bitmap file for dump }
    produce_bitmap_file;

    { terminate the graphic environment }
    terminate_graph;
```

```

end; { compute_profile }

{  MAIN PROCEDURE   }
{ show the applications }
{ P is the principal axis; CSG is the CSG tree to be evaluated;
  L is the length of this part and D is the diameter of the part;
  other properties for various code assignment can also be derived.
}
begin

        { debugging switches }
        DEBUG_INPUT := false;
        DEBUG_EVALUATE_CSG := false;
        DEBUG_BUILD_AXIS := false;
        DEBUG_COMPUTE_T := false;
        DEBUG_CURVE_CURVE := false;
        DEBUG_INTERSECT := false;
        DEBUG_AXIS_OP := false;
        DEBUG_NORMALIZATION := false;
        DEBUG_PARTITION := false;
        DEBUG_ADD_LAYER := false;
        DEBUG_COMPUTE_LAYER := false;
        DEBUG_LAYER_DIFFERENCE := false;
        DEBUG_MERGE_INTERVAL := false;
        DEBUG_LD := false;
        DEBUG_DRAW := false;
        TRACE_AXIS := false;

        { get input CSG tree }
        get_input_data ( CSG );

        { transform the CSG tree into axis representation }
        P := eval

        { find applications }
        { support part/shape classification and
                process planning }
        compute_length_diameter( P, L, D );

        { support NC }
        compute_profile( P );

end.  { main }

```

Appendix 2. Input CSG Data for "Bottle"

****
BOTTLE
****

OBJECT

O#	OP	L_O#	R_O#/MV#/dummy
-1	differ	99	98
99	union	44	45
44	union	36	37
36	differ	34	35
34	differ	33	30
33	union	27	29
27	union	17	18
17	union	11	12
11	union	5	6
5	differ	3	4
3	union	1	2
1	primitive	1	1
2	primitive	2	2
4	primitive	4	4
6	differ	9	8
9	differ	7	10
7	primitive	7	7
10	primitive	10	10
8	primitive	8	8
12	differ	13	14
13	union	15	16
15	primitive	15	15
16	primitive	16	16
14	primitive	14	14
18	differ	19	20
19	union	21	22
21	differ	23	24
23	primitive	23	23
24	primitive	24	24
22	differ	25	26
25	primitive	25	25
26	primitive	26	26
20	primitive	20	20
29	differ	31	32
31	primitive	31	31
32	primitive	32	32
30	primitive	30	30
35	primitive	35	35
37	differ	40	42
40	differ	38	39
38	primitive	38	38
39	primitive	39	39
42	differ	41	43
41	primitive	41	41
43	primitive	43	43
45	differ	52	53
52	union	50	51
50	union	48	49

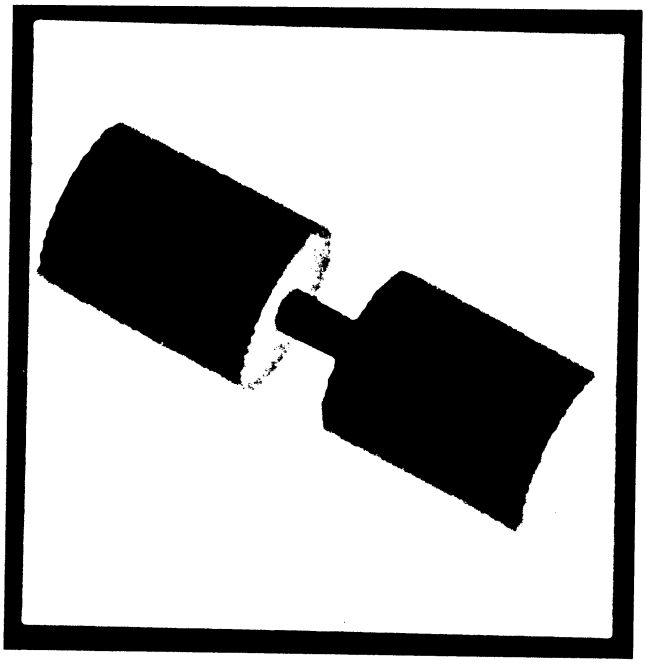
48	differ	46	47
46	primitive	46	46
47	primitive	47	47
49	primitive	49	49
51	primitive	51	51
53	primitive	53	53
98	primitive	98	98

PRIMITIVE

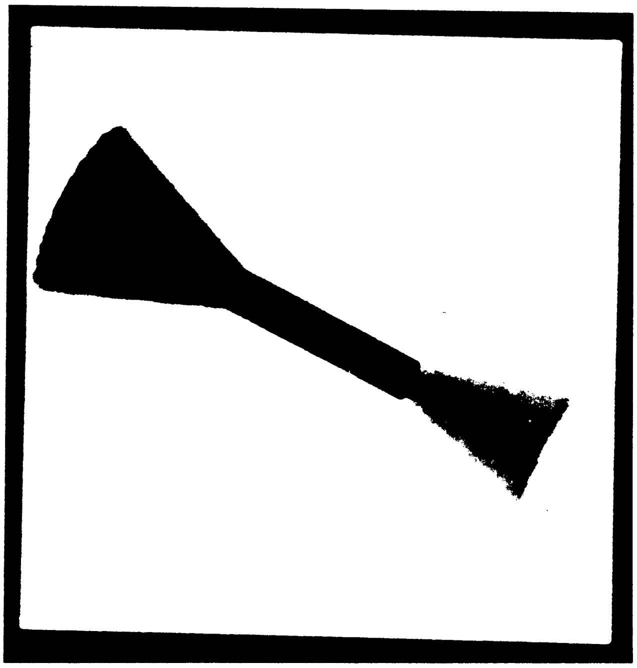
O#	GC	XL	YL	ZL	X	Y	Z	aX	aY	aZ
1	torus	0.5000	0.2000	0.0000	0.0000	0.0000	0.2000	0.0000	0.0000	0.0000
2	cylinder	0.4000	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
4	cylinder	0.4000	0.3000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
7	cone	1.0000	1.2500	0.0000	0.0000	0.0000	1.0000	180.00	0.0000	0.0000
10	cone	0.8400	1.0500	0.0000	0.0000	0.0000	1.0000	180.00	0.0000	0.0000
8	cylinder	0.4000	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
15	torus	1.2500	0.3000	0.0000	0.0000	0.0000	1.3000	0.0000	0.0000	0.0000
16	cylinder	0.6000	1.2500	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000
14	cylinder	0.6000	1.0500	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000
23	cone	1.9672	1.9672	0.0000	0.0000	0.0000	2.3172	180.00	0.0000	0.0000
24	cylinder	1.6000	1.2500	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
25	torus	1.6844	0.4000	0.0000	0.0000	0.0000	2.6000	0.0000	0.0000	0.0000
26	torus	1.6844	0.2500	0.0000	0.0000	0.0000	2.6000	0.0000	0.0000	0.0000
20	cone	1.9086	1.9086	0.0000	0.0000	0.0000	2.4586	180.00	0.0000	0.0000
31	cone	1.9672	1.9672	0.0000	0.0000	0.0000	2.8828	0.0000	0.0000	0.0000
32	cylinder	1.6000	2.0000	0.0000	0.0000	0.0000	3.3172	0.0000	0.0000	0.0000
30	cone	1.9086	1.9086	0.0000	0.0000	0.0000	2.7414	0.0000	0.0000	0.0000
35	cylinder	0.6000	1.6844	0.0000	0.0000	0.0000	2.3000	0.0000	0.0000	0.0000
38	torus	1.6844	0.4000	0.0000	0.0000	0.0000	3.5400	0.0000	0.0000	0.0000
39	torus	1.6844	0.2600	0.0000	0.0000	0.0000	3.5400	0.0000	0.0000	0.0000
41	cylinder	0.8800	2.2500	0.0000	0.0000	0.0000	3.1000	0.0000	0.0000	0.0000
43	cylinder	0.4412	1.6000	0.0000	0.0000	0.0000	3.3172	0.0000	0.0000	0.0000
46	cone	3.2500	2.5000	0.0000	0.0000	0.0000	5.0000	180.00	0.0000	0.0000
47	cylinder	2.0500	1.6000	0.0000	0.0000	0.0000	1.7000	0.0000	0.0000	0.0000
49	torus	2.1525	0.3000	0.0000	0.0000	0.0000	5.0000	0.0000	0.0000	0.0000
51	cylinder	0.3000	2.1525	0.0000	0.0000	0.0000	5.0000	0.0000	0.0000	0.0000
53	cone	3.2200	2.1525	0.0000	0.0000	0.0000	5.0100	180.00	0.0000	0.0000
98	cube	3.0000	3.0000	5.5000	0.0000	-3.000	[illegible].0000	0.0000	0.0000	0.0000

Appendix 3. Shaded Pictures by Ray Casting

(a) corresponds to Figure 6.1
(b) corresponds to Figure 6.2
(c) corresponds to Figure 6.3
(d) corresponds to Figure 6.4
(e) corresponds to Figure 7, the "bottle"

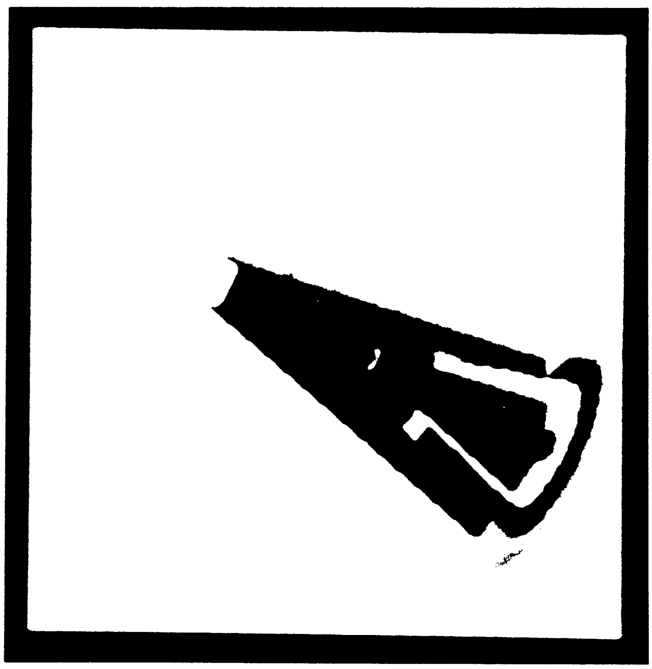


(a)

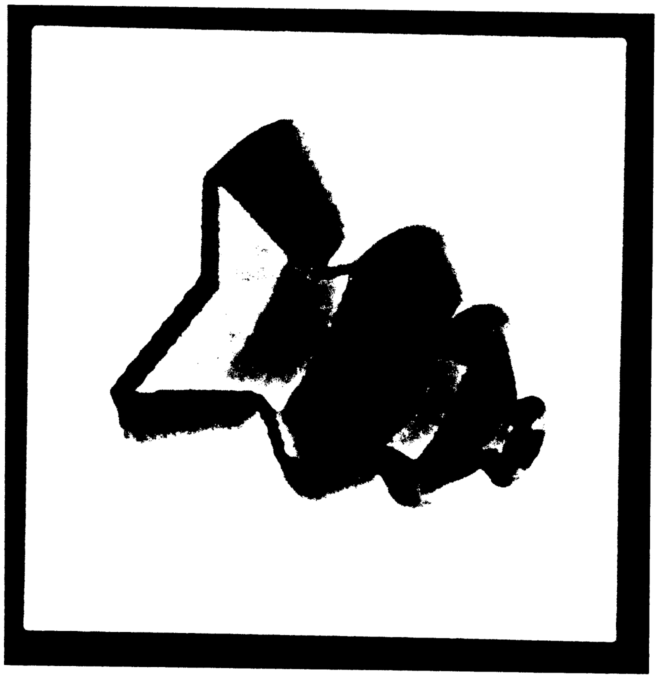


(b)

(c)



(d)



(e)