SRC TR 89-99

MARYLAND

TECHNICAL
RESEARCH
REPORT

ProtoSolid: An inside look

By

G. Vanecek, Jr.

SYSTEMS RESEARCH CENTER

UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND 20742

Protosolid:
An inside look

George Vanecek, Jr.

CSD-TR-921
October 1989

ProtoSolid: An inside look

George Vaněček, Jr.[1]
Department of Computer Science
Purdue University
West Lafayette, IN 47907

October 23, 1989

Abstract

PROTOSOLID is a solid modeler that represents solids with planar surfaces and manipulates the solids with regularized set operations. This paper looks at the internal structures and operations of PROTOSOLID and is intended as a programmer user guide.

A boundary of a solid is represented by a *fedge-based data structure*. It is developed in Common-Lisp.

[1]This work has been supported in part by an NSF Presidential Young Investigator Award to Dana Nau with matching funds provided by General Motors Research Laboratories, Texas Instruments, NSF Grant NSFD CDR-85-00108 to the University of Maryland Systems Research Center, and NSF Grant NSFD CCR-86-19817 to Purdue University.

Contents

1 Introduction

PROTOSOLID is a solid modeler that models a solid in two convenient ways, an unevaluated form and an evaluated form. The *unevaluated form* is a sequence of creation and transformation commands that when evaluated, produce the boundary of a solid; this form is similar to PADL's design language. The *evaluated form* is the boundary representation of a solid.

PROTOSOLID is a facetted modeler. The faces are embedded in planes and the edges lie on straight lines. Curved analytic surfaces such as cylinders and cones are approximated by some number of facets. In general, PROTOSOLID models a class of solids that are describable by planar polyhedra of varying genera, components, and connectivity, and includes the class of solids with non-manifold boundaries[4, 7].

In representing a boundary of a solid, PROTOSOLID maintains a graph hierarchy that separates the topology from the geometry. Each solid maintains unique topological information, but maintains the associated geometrical information in a global geometrical directory.

The representations of solids are complex data structures. The validity of the representations is maintained by restricting the creation and manipulation of solids to one of the following ways:

1. A solid is an instantiation of one of several parametrized primitives. The primitives are the block, the cylinder, the sphere, the torus, and the cone.

2. A solid is an extrusion of a non-self intersecting polygonal contour drawn in a plane with an extrusion vector pointing out of the plane.

3. A complex solid is composed out of a combination of binary Boolean set operators. These include the union, the intersection, and the subtraction of two solids. The complement of a solid is not allowed.

Although operators that manipulate the topological structures are provided, their direct use by the user is discouraged. Instead, new parameterized primitives may be added to the existing set of primitives. For example, a single validated procedure can be easily added to create a tetrahedron.

PROTOSOLID is a modeler implemented in Lisp on a TI/Explorer. Common Lisp[5] was the chosen programming language for several reasons:

1. The solid modeler needs to be portable, and easily interfacable to systems needing solid modeling services, such as the SIPS[3] process planning system, or Newton. Using Common Lisp allows that.

2. Lisp, as an implementation language, provides the necessary attributes to set up complex data structures. Programming solid modelers is, to a large extent, a play on the use of pointers, lists, and symbols. Lisp is a language designed to do just that.

3. Lisp, as a design language, provides a parser, a writer, and an interpreter for reading, writting and evaluating stored information. A design language module is an essential component of a modeler. Using an implementation language other than lisp would require additional coding of such a module. Lisp readily provides the needed interpretive environment.

4. The availability of specialized hardware to execute Lisp programs provides a software environment well-suited for rapid prototyping of code. The Explorer architecture (similarly, the Symbolics) has a dedicated 36-bit processor with run-time data-type checking, a high-resolution color bit-mapped display, and Ethernet based networking. The software environment includes an excellent editor with advanced features such as interpretation and compilation of code within the editor, incremental compilers, dynamic linking and loading, a flexible display-oriented debugging system, and other utilities.

Since PROTOSOLID strictly conforms to the current standards of Common Lisp, object oriented programming, such as Flavors or Portable Common Loops, was not used. As a result, PROTOSOLID's Lisp code can be easily translated into a language like C.

Currently PROTOSOLID has several graphical user interfaces. These include a TI/Explorer window-based interface, an S-Geometry interface on the Symbolics, and a network-based interface running on a Personal Iris 4D graphics work station. A future goal is to write a machine independent interface in X11.

This paper concentrates on the internals of PROTOSOLID. The various user interfaces are discussed in other papers.

Disclaimer: PROTOSOLID is an experimental tool continuously expanding and changing. While at the writting of this paper, the description of PROTOSOLID corresponds to what is actually inside PROTOSOLID, the reader should not view this paper as a user's manual, but rather as a guide to understanding. Inevitably, differences will be found between what is stated in this paper, and what is actually out there.

2 Coding Conventions

The implementation of PROTOSOLID follows several coding conventions:

- The modeler abides by strict Common Lisp as defined by Steele[5]. In this paper, Common Lisp code appears in a terminal-font.
- All special variables (that is, globally available variables defined by `DEFVAR`) are enclosed with asterisks.
- All special constants (that is, those defined by `DEFCONSTANT` and `DEFPARAMETER`) are enclosed in dollar-signs. For example, the constant `$EPSILON$`.
- Three types of comments appear within the code.
    1. Comments at the end of a line beginning with a single semicolon.
    2. Comment lines amidst the code beginning with two semicolons and indented with the code.
    3. Outside comments begin with three semicolons and start at the first column of a line.

This paper presents many of the functions provided by PROTOSOLID. Functions are given as follows:

- (function (THE type arg) ...) *Brief Description*
  → return-value

Underlined keywords indicate the type or the meaning of the field, as appropriate. Emphasized notes at the right margin briefly describe the function.

3 Representation of Solids

A solid model is the unambiguous and informationally complete mathematical representation of the shape of a physical object in a form that a computer can process. From a representational point of view, there is no known general model by which all solids could be denoted. As such, mathematical models must be designed that can handle a small describable subset of all solids. It is representationally convenient to model only a solid's boundary as opposed to model the volume occupied by the solid. The boundary is itself conveniently modeled as a collection of faces. The faces are described by their bordering edges, and the edges by their bordering vertices. Thus the boundary of a solid can be represented as collections of faces, edges and vertices, and the adjacency relationships between them[11].

Figure 1: Two nonmanifold solids and a pseudo-manifold.

Another method for modeling solids is the procedural method. From a few simple parameterized solids, a complex solid can be constructed by the use of transformational and set operation operators. A solid can thus be modeled in terms of the process by which it was created.

Presently, a variety of representational schemas have been developed for modeling solids. It is evident that any one schema alone cannot be used independently because certain operations are easier to compute from specific schemas, and because various types of information about a solid cannot be organized efficiently into a single schema. Therefore, many solid modeling systems incorporate several representations that are either maintained separately or are fused to form hybrid representations. PROTOSOLID uses both the boundary representation and the procedural method describe above.

Ordinary physical solids have two-dimensional boundaries (i.e., surfaces that are two-manifolds). An object is bordered by a two-manifold surface if every point on its surface is topologically equivalent to a disk. If it is a manifold, the boundary described in terms of faces, edges and vertices satisfies the following three properties:

M1) Every edge borders exactly two faces;

M2) Every vertex is surrounded by a single loop of alternating edges and faces (referred to as the edge-face loop):

M3) The object is non-self-intersecting. That is, faces may not intersect each other except touch at common edges or vertices.

It is exactly these manifold objects that are modeled by the *Euler-Poincaré* formula, and that can be created and maintained with Euler operators[2, 1]. However, there exist solids that violate properties M1 and M2. Such solids have a *nonmanifold* boundary. Two solids touching at a vertex or an edge form a single solid that is a nonmanifold. In the special case, where the nonmanifold contains a single interior component, the solid is referred to as a *pseudo-manifold*. Figures 3a and 3b show nonmanifolds. A pseudo-manifold is shown in Figure 3c. Unfortunately, such solids with nonmanifold boundaries cannot be ignored in our modeling space because manifolds are not closed under the regularized set operations. A set operation on manifolds may result in a nonmanifold and a representation must be capable of representing these solids also. This means, however, that the representation must generalize the first two properties for the manifolds stated above to read:

N1) Every edge belongs to an even number of distinct faces, and

N2) every vertex may be incident on one or more separate edge-face loops (see Figure 3).

Unfortunately, a popular data structure such as the modified winged-edge data structure will not work under these generalizations. Therefore, PROTOSOLID uses a fedge-based data structure that is capable of maintaining all solids in a consistent manner.

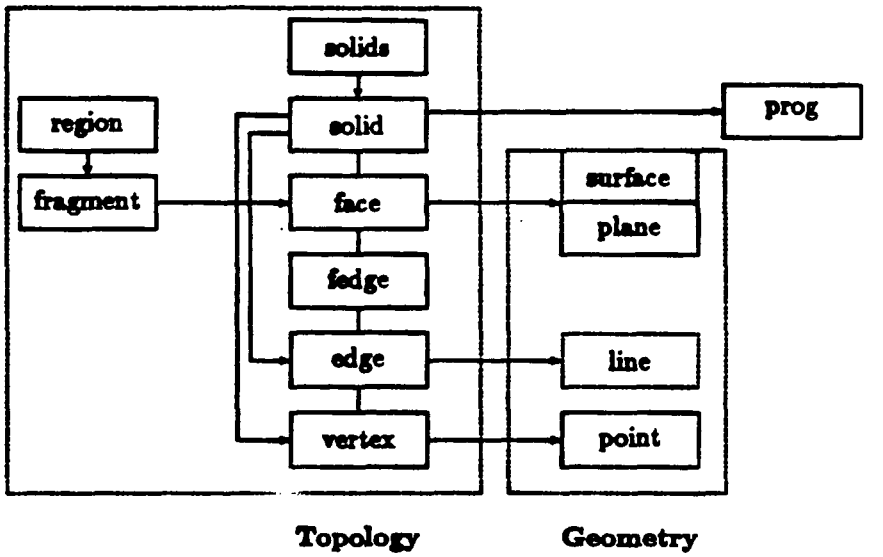

Figure 2: The hierarchy of the fedge-based data structure.

4 The Fedge-based Data Structure

The fedge-based data structure represents a solid by its boundary and by the process that created the solid. The boundary is represented in a graph hierarchy which consists of three types of data:

1. *geometric data* which consists of basic shape parameters for points, curves, and surfaces.
2. *topological data* which consists of topological entities such as the vertices, edges, and faces, and provides the adjacency relationships among these entities, and
3. *functionality* and *form feature* data. This data is essential for modeling manufacturing parts and is maintained in a separate hierarchy below the topological information.

The topological and geometrical entities of a solid are represented by a hierarchy that is shown in Figure 2. The topological data consist of six types of nodes represented as record structures, namely, the solid node, the face node, the fedge node, the edge node, and the vertex node. In addition, these record structures are interlinked to represent several of the topological adjacency relationships. The geometrical data consists of all the points, lines, planes, and surfaces. The separation of topology and geometry is such that each solid maintains its own topological structure while all solids share a common geometrical structure. This separation serves a dual purpose. First and foremost is to facilitate consistent equality checking. Second is to conserve on space and improve access efficiency.

With regard to the first purpose, consider the set of points existing in object space. Two points `P1` and `P2` referred to from different solids are equal if and only if `(EQ P1 P2)`. That is, they are the same point node. No new point in close proximity to `P1` is ever allowed to be created. The extent of this neighborhood around a given point is controlled by an epsilon value maintained in two global constants.

- `(DEFCONST $EPSILON$ 1.0E-12)`
- `(DEFCONST $-EPSILON$ -$EPSILON$)`

The second purpose for the separation of topology and geometry deals with space conservation. Consider the points as an example. Any property pertaining to a point applies equally to all solids that refer to that point. Once a point is determined to lie on one side of a plane, that information holds true for all solids requesting such a point plane classification.

Since all geometrical entities of a given type are kept in one place and their retrieval is very frequent, the different geometrical entities are ordered. The ordering allows a set of entities to be placed in a directory that can be searched in better than linear time. Currently, a directory is implemented as a binary search tree.

A directory implemented as a binary search tree consists of two node types, a header node and a tree node. The two nodes have the following structure:

```
(DEFSTRUCT DIR-TREE
    "Geometrical Directory Header Node"
    (DIR-ROOT    NIL :TYPE DIR-NODE)        Root node of tree
    (DIR-SIZE    0   :TYPE COUNTER)         Number of nodes in tree
    (DIR-EVENTS  0   :TYPE COUNTER))        Number of recent insertions

(DEFSTRUCT DIR-NODE
    "Geometrical Directory Tree Node"
    (DIR-LEFT    NIL :TYPE DIR-NODE)        Left Subtree
    (DIR-RIGHT   NIL :TYPE DIR-NODE)        Right Subtree
    (DIR-PARENT  NIL :TYPE DIR-NODE)        Parent Node
```

The DIR-NODE is used by the point, the plane and the vertex nodes. The procedure for insertion into and deletion from a directory is identical for all three node types. As a result, single locate, insert, and delete routines are used to support all directories. However, since the ordering mechanism of the nodes for each directory is different, the insertion and location routines must be provided with a comparison function for the corresponding element type. Assuming that cmpf is a binary comparison function with the range {:SMALLER,:SAME,:LARGER}, the following three routines are defined for all directories.

- `(DIR-LOCATE (THE DIRECTORY dir) x cmpf)` — *Find x in directory*
  → dir-node *or* NIL

Given a directory and a node x of the particular type of directory, DIR-LOCATE returns a directory node that contains p, if any.

- `(DIR-INSERT (THE DIRECTORY dir) n cmpf)` — *Insert n into directory*
  → dir-node

If another node already exists in the directory that contains the same informational field as n, then DIR-INSERT returns the one that is already in the directory, otherwise n is inserted and returned.

Because a directory is maintained as a binary search tree, insertion can cause an imbalance, which reduces retrieval efficiency. As a result, the directory needs to be kept balanced. This can be done by a self-balancing insertion mechanism, or by a periodic rebalancing method. In the original implementation, insertions are allowed to occur unchecked, and periodically a global rebalancing takes place to reduce the average tree depth. Each directory keeps track of how many insertions occurred in the field DIR-EVENTS. This fields is then used to estimate the need to rebalance the directory. The rebalancing is performed by an efficient linear time and space rebalancing algorithm developed by Stout and Warren[6].

- `(DIR-REBALANCE (THE DIRECTORY dir) dir-print-name)` — *Rebalance directory*
  → dir

Rebalancing occurs automatically whenever the tree grows by a prespecified percentage:

Geometrical	
CN	cones
CO	coordinates
CY	cylinders
EX	extents
LN	line
PL	planes
PO	points
SP	spheres
TR	torii

Topological	
ED	edges
EX	extents
FA	faces
FE	fedges
FR	fragments
RE	regions
SO	solids
VE	vertices

Table 1: Geometrical and Topological mnemonics.

- `(DEFVAR *DIR-REBALANCE%* 0.25)` *Percent threshold*

Assuming that node n is in the directory, `DIR-DELETE` removes n from the directory without destroying n. Note that the structure `TREE-NODE` includes a parent pointer in addition to the left and the right child pointers. The need for the parent pointer is to enable an $O(1)$ deletion operation of a node. This occurs because n is given as an argument, and the deletion occurs without the need to trace down the tree from the root node.

- `(DIR-DELETE (THE DIRECTORY dir) n)` *Remove n from directory*
  → `n`

To allow the visitation of each node in a directory using a traversal direction `:INORDER`, `:PREORDER`, or `:POSTORDER`, the following macro is provided.

- `(FOR-EACH-NODE-OF-DIR (n dir direction) body)` *Iterate*
  → `NIL`

4.1 Mnemonics

As a coding convention, access functions have mnemonic prefixes shown in Table 4.1. The use of such prefixes makes it easier to recognize the main argument of a function.

4.2 Geometry support

A fundamental construct in the geometrical support of PROTOSOLID is the `COORDS` node. It represents a row vector $[x, y, z, w]$ which can denote either a point, a vector, or a plane. It has the following structure:

```
(DEFSTRUCT COORDS
    "3D homogeneous coordinates or a row vector"
    (CO-X 0.0 :TYPE DOUBLE-FLOAT)
    (CO-Y 0.0 :TYPE DOUBLE-FLOAT)
    (CO-Z 0.0 :TYPE DOUBLE-FLOAT)
    (CO-W 1.0 :TYPE DOUBLE-FLOAT))
```

In reference to `COORDS` as a vector (i.e., considering only (x, y, z) with $w = 1$), a collection of vector operations exists.

- `(SCALAR-PROD (THE DOUBLE-FLOAT s) v &OPTIONAL (r (MAKE-COORDS)))` $r = s * v$
  → `r`

- (DOT-PROD v1 v2) $v1 \cdot v2$
  → double-float

- (CROSS-PROD v1 v2 &OPTIONAL (v3 (MAKE-COORDS))) $v3 = v1 \times v2$
  → v3

- (NORMALIZE v &OPTIONAL (u (MAKE-COORDS))) $u = v/||v||$
  → u

- (VEC-NORM v) $||v||$
  → double-float

- (DISTANCE p1 p2) $||p2 - p1||$
  → double-float

- (SUBTRACT-VEC v1 v2 &OPTIONAL (vr (MAKE-COORDS))) $vr = v1 - v2$
  → vr

- (ADD-VEC v1 v2 &OPTIONAL (vr (MAKE-COORDS))) $vr = v1 + v2$
  → vr

- (INC-VEC v w) *Increment v by w*
  → v

- (NEGATE-VEC v &OPTIONAL (n (MAKE-COORDS))) $n = -v$
  → n

- (ASSIGN-VEC v w) $v := w$
  → v

- (ZERO-VEC v) *Initialize v to (0,0,0)*
  → v

A stack of COORDS structures is provided for use in vector computations to eliminate the inefficient practice of garbage collecting the temporary structures. The stack is defined as a variable length array.

- (DEFCONST $VR-REGISTERS$ (MAKE-ARRAY 50 :VECTOR-FILL T :ELEMENT-TYPE 'COORDS))

Since the COORDS structures on the stack are preallocated, care must be taken in not leaving a reference to the stack in permanent places. A new structure can be obtained and one or more can be returned.

- (GET-VR-REGISTER) *Allocate*
  → coords

- (RELEASE-VR-REGISTERS &OPTIONAL (how-many 1)) *Deallocate*
  → NIL

For an example of using the register stack, the following program prints out the midpoint of two points P1 and P2:

```
(LET* ((S (ADD-VEC P1 P2 (GET-VR-REGISTER)))
       (M (SCALAR-PROD 0.5 S (GET-VR-REGISTER))))
  (FORMAT T "(~2,~2,~2)" (CO-X M) (CO-Y M) (CO-Z M))
  (RELEASE-VR-REGISTER 2))
```

4.2.1 Points

In PROTOSOLID a point is more than just three coordinates. A point has properties, and these are maintained in addition to the three coordinates in object space. For one, each unique point in object space is uniquely represented and its existence is retained in a point directory $POINTS$. This directory is defined as

- (DEFCONST $POINTS$ (MAKE-DIR-TREE))

and contains points having the following structure:

```
(DEFSTRUCT (POINT (:INCLUDE DIR-NODE))
    "3D Point"
    (PO-WPNT (MAKE-COORDS))                    World Coordinates
    (PO-TIME (1- *CURRENT-TIME*))              Time Stamp
    (PO-DIST 0.0)                              Distance To Some Plane
    (PO-CLASS :NONE))                          Classification
```

All the currently defined points known to PROTOSOLID can be printed.

- (PRINT-ALL-POINTS) — *Output points*
  → NIL

The three fields PO-TIME, PO-DIST and PO-CLASS will be discussed in Section 8.1. In addition to these three fields, the point structure has four other fields. One is the row vector corresponding the location of the point in the object space, and the other three are included in the DIR-NODE, namely, the DIR-LEFT, the DIR-RIGHT and the DIR-PARENT.

A three-way point comparison function needed by DIR-INSERT is

- (PO-CMPR p1 p2) — *Compare points*
  → one of :SMALLER, :SAME, or :LARGER

4.2.2 Lines

A line is represented by a point on the line and the direction unit vector:

```
(DEFSTRUCT LINE
    (LN-WPNT (MAKE-COORD) :TYPE COORDS)        Point on the Line
    (LN-DIR  (MAKE-COORD) :TYPE COORDS))       Direction Unit Vector
```

4.2.3 Planes

All planes are maintained uniquely in a directory of planes defined by

- (DEFCONST $PLANES$ (MAKE-DIR-TREE))

and has the following structure:

```
(DEFSTRUCT (APLANE (:INCLUDE DIR-NODE))
    (PL-VECT (MAKE-COORD) :TYPE COORDS))       Equation of the Plane
```

The row vector (PL-VECT p) is taken to be a four tuple $[A, B, C, D]$. One way of computing the plane equation is from the three points p1, p2 and p3:

- (FIT-PLANE p1 p2 p3) — *Compute Plane Equation*
  → (VALUES p flip)

where the plane p is represented by the coefficients $[-A,-B,-C,-D]$ if `flip` = `t`, and $[A,B,C,D]$ if `flip` = `NIL`. The coefficients reprepresent a normalised general equation of the plane for which any point q on the plane satisfies $q \cdot p^T = 0$. Furthermore, the vector (A,B,C) is the plane normal. The order of the three arguments is important. Consider the two results given the three points:

```
(MULTIPLE-VALUE-SETQ (PL1 FLIP1) (FIT-PLANE P1 P2 P3))
(MULTIPLE-VALUE-SETQ (PL2 FLIP2) (FIT-PLANE P1 P3 P2))
```

The plane's coefficients are such that `PL1` = `PL2` but `FLIP1`=(`NOT FLIP2`). The computed coefficients of a plane have the property:

$$A > 0 \vee (A = 0 \wedge (B > 0 \vee (B = 0 \wedge C > 0))). \tag{1}$$

This property is necessary to prevent having both the planes $[A,B,C,D]$ and $[-A,-B,-C,-D]$, entered in the plane directory. A three-way plane comparison function needed by `DIR-INSERT` is

- `(PL-CMPR p1 p2)` *Compare planes*
  → one of `:SMALLER`, `:SAME`, or `:LARGER`

with which a new plane p satisfying condition (1) can be placed in the plane directory by

```
(SETQ P (DIR-INSERT $PLANES$ P #'PL-CMPR)).
```

All the currently defined planes known to PROTOSOLID can be printed by

- `(PRINT-ALL-PLANES )` *Output planes*
  → `NIL`

  This is basically the following program:

```
(FOR-EACH-NODE-OF-DIR (P $PLANES$ :INORDER)
    (FORMAT T "(~F,~F,~F,~F)~%" (CO-X (PL-VECT P)) ...))
```

  Determining if a point q is on the plane p is done by

- `(PO-PL-CLASSIFICATION q p)` *Compare point/plane*
  → one of `:BELOW`, `:ON`, or `:ABOVE`

which returns the point plane relationship. The notion of below or above a plane is with respect to the plane's normal. The point is above if $q \cdot p^T >$ `$EPSILON$` and below if $q \cdot p^T <$ `$` $-$ `EPSILON$`. The point-plane relationship and the distance of the point to the plane is stored in the (`PO-CLASS p`) and (`PO-DIST p`) fields for future reference.

4.3 Extents

Extents of solids are an integral part of the geometrical support. Extents serve the purpose of quickly determining the minimum rectangular region of space containing a set of points. The extent of a set of points is the smallest box that encloses those points. Inserting a point p into an extent e is handled by

- `(UPDATE-EXTENT e p)` *Incorporate point into extent*
  → e

which updates e to include the point and returns the extent. An extent is defined by the `EXTENT` structure which contains two points. The point (`EX-MINP e`) contains the minimum coordinates of any point that has been inserted into e. The point (`EX-MAXP e`) contains the maximum coordinates of all inserted points. The two points define a diagonal of an axis aligned box that bounds all the inserted points. The extent has the following structure:

	vertex	edge	face	solid
vertex	$V:\{V\}$	$V:\{E\}$	$V:\{F\}$	$V:\{S\}$
edge	$E:\{V\}$	$E:\{E\}$	$E:\{F\}$	$E:\{S\}$
face	$F:\{V\}$	$F:\{E\}$	$F:\{F\}$	$F:\{S\}$
solid	$S:\{V\}$	$S:\{E\}$	$S:\{F\}$	$S:\{S\}$

Table 2: Topological adjacency relationships: the eight adjacency relationships used by the fedge-based data structure appear in bold.

```
(DEFSTRUCT EXTENT
    "Extent of a solid"
    (EX-MINP (MAKE-COORDS :CO-X 1.0E10 :CO-Y 1.0E10 :CO-Z 1.0E10))
    (EX-MAXP (MAKE-COORDS :CO-X -1.0E10 :CO-Y -1.0E10 :CO-Z -1.0E10)))
```

An extent is reset to the default values shown above by the function

- `(CLEAR-EXTENT extent)` — *Reset extent*
  → `extent`

and if given two extents, the first extent can be incorporated into the second extent.

- `(INCORPORATE-EXTENT extent1 extent2)` — *Enlarge e2 by e1*
  → `extent2`

Given two extents, it can be determined if they intersect. Two extents intersect if their regularised intersection is non-empty. Thus the touching of two extents does not constitute an intersection.

- `(EXTENTS-INTERSECT? extent1 extent2)` — *Check intersection*
  → boolean

Also, the center point of an extent can be obtained.

- `(EXTENT-CENTER extent &OPTIONAL (c (MAKE-COORD))` — *Center point of an extent*
  → `c`

4.4 Topological Support

The topological information consists of five different types of topological nodes, namely, the solid, face, fedge, edge, and vertex. The solid, the face, the edge and the vertex are the familiar components of a boundary representation. The *fedge* node gives a face its edge orientations. Our use of fedges is analogous to Mäntylä's use of half-edges in the half-edge data structure. The need for edge loops for representing holes in a multi-connected face has been eliminated as was done in the bridge-edge data structure used by Yamaguchi [12]. Furthermore, no shell structure is used for representing multi-shelled solids. Refer to the function SEPARATE-SOLIDS for the separation of multi-shelled solids into separate solids.

A boundary can be defined as a triple (V, E, F) where V denotes the set of vertices, E denotes the set of edges, and F denotes the set of faces. With the four topological entities, there are sixteen topological relationships. The functional notation $a : \{b\}$ represents the adjacency relationship between an entity of type a and a set of entities of type b. For example, $F : \{E\}$ specifies the face/edge adjacency relationship. The topological adjacency relationships are shown in Table 4.4. The topological adjacency relationships maintained explicitly in the fedge-based

data structure appear in bold print in the table. It is not necessary to store all the topological adjacency relationships explicitly since they can be derived. Since there exists a separate node for each topological entity in the representation, the relationship $a : \{b\}$ is maintained by double linking all the entities of type b that are adjacent to an entity of type a. All faces and edges are maintained by the solid in a doubly-linked list. The vertices of a given solid are maintained in a spatial directory to allow better than $O(n)$ retrieval.

The fedge-based data structure hierarchy is shown in Figure 2. The labeled boxes in the figure denote where in the hierarchy the entities reside. The arrows indicate pointers in one direction, while the plain lines indicate pointers in both directions. Figures 4.4 and 4 contain diagrams of the fedge-based data structure that show the fedge nodes, the bordering edge nodes, and the vertex nodes of a single face of a cube. The first figure shows the downward pointing pointers starting from a face node. The second figure shows the upward pointing pointers starting from the vertex nodes. In the remainder of this section, the vertex, the edge, the fedge, the face, and the solid node type and their functional support are detailed.

4.4.1 Vertices

Geometrically, a vertex exists at a point of intersection of three or more planes. Topologically, a vertex is incident to one or more edge-face cycles. For each vertex, the $V : \{E\}$ adjacency relationship is maintained. A vertex node points with the VE-EDGES field to an anchor edge. The anchor edge is on a unordered double-linked list of all the edges that are incident on the vertex. The number of incident edges is kept in the VE-NEDGES field.

```
(DEFSTRUCT (VERTEX (:INCLUDE DIR-NODE))
    "Vertex node of a solid."
    (VE-POINT NIL :TYPE POINT)
    (VE-NEDGES 0 :TYPE FIXNUM)                    Number of incident edges
    (VE-EDGES NIL :TYPE EDGE)                     Incident Edges
    (VE-ASSOC NIL :TYPE LIST))
```

The list of edges around a vertex is not ordered. There may be more than one edge-face cycle around each vertex (this occurs for nonmanifolds) and ordering the edges would greatly complicate the structure and add unnecessary time-complexity. Therefore, it is not worth the effort computationally and storage-wise to explicitly differentiate between the various edge-face cycles around a vertex. As it is, a particular edge-face cycle may easily be traced. Given a vertex v and one of the fedges g of an incident edge of v where (EQ v (FE-DSTV g)), the edge-face cycle around v that the edge belongs to may be traced by the following program (refer to Figure 4.4.1):

```
(DO ((LG (FE-COFEDGE G)                                   initialize
         (FE-PREV (FE-COFEDGE G))))                       step to next fedge
    ((EQ LG G))                                           terminate when G reappears
  body)                                                   process edge (FE-EDGE LG)
```

The reason one must start with a fedge node, and not an edge is because the solid need not be a manifold. Consider the example in Figure 4.4.1. The vertex is adjacent to three edge-face-loops. Giving one of the adjacent edges with four incident faces as the starting point would not provide sufficient information as to which of the two touching edge-face cycles was desired.

The solid of a vertex is given by:

```
• (VE-SOLID v)                                            Get solid of vertex
  → (FA-SOLID (FE-FACE (ED-FEDS (VE-EDGES v))))
```

4.4.2 Edges

Geometrically, an edge corresponds to a line segment lying on the line of the intersection between two (or an even number of) planes. Topologically, an edge is incident to two vertices and an even

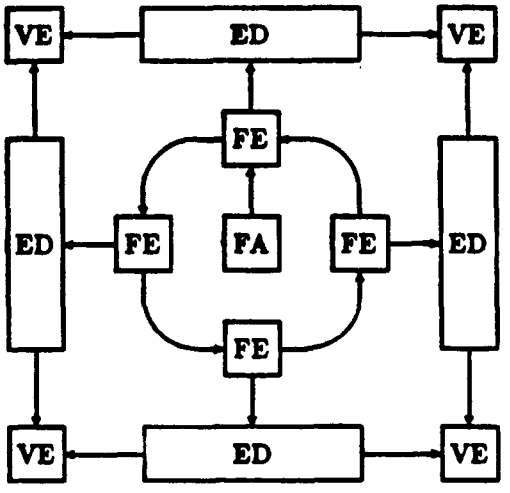

Figure 3: Diagram of the fedge-based data structure showing only the down pointers of the top face of a block. FA, FE, ED and VE stand for face, fedge, edge and vertex respectively.

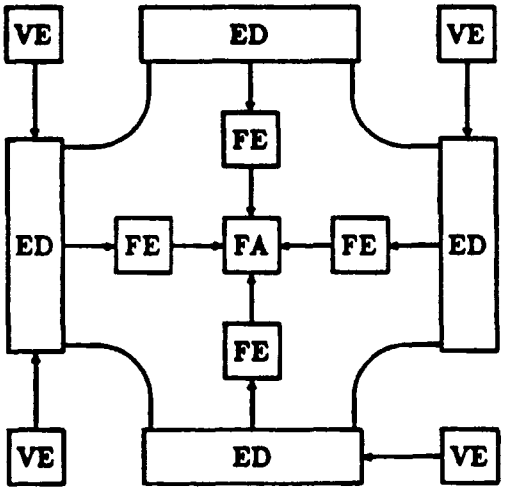

Figure 4: Diagram of data structure showing the up pointers only.

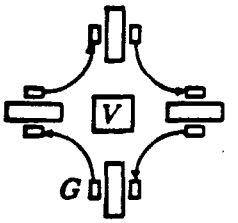

Figure 5: A single edge-face cycle around a vertex. Here, each edge has two incident fedges.

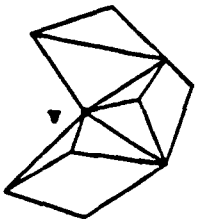

Figure 6: Multiple edge-face cycles incident on vertex v.

number of faces. The $E:\{V\}$ and the $E:\{F\}$ adjacency relationships are maintained. The edge has the following structure:

```
(DEFSTRUCT EDGE
    "Edge between two vertices"
    (ED-VRTA     NIL    :TYPE VERTEX)                    First vertex
    (ED-VRTB     NIL    :TYPE VERTEX)                    Second vertex
    (ED-FEDS     NIL    :TYPE FEDGE)                     Incident Fedges
    (ED-NFEDS    0      :TYPE FIXNUM)                    Number of Incident Fedges
    (ED-NVRA     NIL    :TYPE EDGE)                      Next Edge of VRTA
    (ED-PVRA     NIL    :TYPE EDGE)                      Previous Edge of VRTA
    (ED-NVRB     NIL    :TYPE EDGE)                      Next Edge of VRTB
    (ED-PVRB     NIL    :TYPE EDGE)                      Previous Edge of VRTB
    (ED-NEXT     NIL    :TYPE EDGE)                      Next Edge of Solid
    (ED-PREV     NIL    :TYPE EDGE)                      Previous Edge of Solid
    (ED-PSEUDO?  NIL    :TYPE BOOLEAN)
    (ED-LINE     (MAKE-LINE) :TYPE LINE)    Line passing from VRTA through VRTB
    (ED-ASSOC    NIL    :TYPE LIST))
```

The two vertices of an edge are ED-VRTA and ED-VRTB. The incident edges around vertex A, for example, are linked together cyclically by the ED-NVRA and the ED-PVRA fields. The incident edges around vertex B are likewise maintained by the ED-NVRB and the ED-PVRB pointers. These four pointers support the $V:\{E\}$ adjacency relationship. Figure 7 shows the pointers maintained by the edge node.

The $S:\{E\}$ relationship is supported by the ED-NEXT and ED-PREV pointers.

The $E:\{F\}$ relationship is kept in each edge by linking together the fedges of the faces adjacent to the edge. The anchor fedge of the fedge loop is kept in the ED-FEDS field. ED-NFEDS is the number of fedges in this loop which is also the number of faces incident on this edge.

```
• (ED-LNPO •)                                        Point on line of edge
  → (LN-NPNT (ED-LINE •))
```

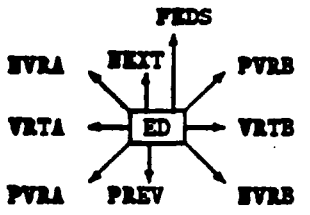

Figure 7: Edge Topology

- (ED-LNDIR e) *Direction of line of edge*
  → (LN-DIR (ED-LINE e))

- (ED-PNTA e) *1st endpoint of edge*
  → (VE-POINT (ED-VRTA e))

- (ED-PNTB e) *2nd endpoint of edge*
  → (VE-POINT (ED-VRTB e))

- (ED-SRCV e a2b?) *Source vertex*
  → (IF a2b? (ED-VRTA e) (ED-VRTB e))

- (ED-DSTV e a2b?) *Destination vertex*
  → (IF a2b? (ED-VRTB e) (ED-VRTA e))

- (ED-NVRE v e) *Next edge around vertex*
  → (IF (EQ v (ED-VRTA e)) (ED-NVRA e) (ED-NVRB e))

- (ED-PVRE v e) *Previous edge around vertex*
  → (IF (EQ v (ED-VRTA e)) (ED-PVRA e) (ED-PVRB e))

4.4.3 Fedges

Geometrically, a face is represented by the plane in which it is contained, and by the polygon on that plane that demarcates the boundary of the face. Topologically, the polygon is given by the edges. The edges are oriented in a counterclockwise order around the face when viewed from just above the face and outside the solid. However, having an cyclically ordered list of edges itself is not sufficient if it is also desirable to obtain the vertices bordering the face in a counterclockwise order. This is because the edge can have one of two orientations. It can either have the orientation from ED-VRTA to ED-VRTB or from ED-VRTB to ED-VRTA. A face handles the ordering and orientation of edges with the aid of the fedge nodes. A fedge node belongs exclusively to a face/edge pair, and provides the needed ordering and orientation of edges around the face. The link fields of a fedge are shown in Figure 8. A fedge has the following structure:

```
(DEFSTRUCT FEDGE
    "Directed Edge: Face-Edge"
    (FE-EDGE  NIL :TYPE EDGE)
    (FE-FACE  NIL :TYPE FACE)
```

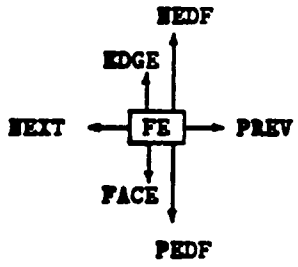

Figure 8: Fedge Topology

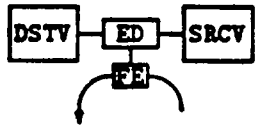

Figure 9: The source and destination vertices of a fedge.

```
(FE-A2B?  NIL :TYPE BOOLEAN)          Orientation of edge
(FE-NEXT  NIL :TYPE FEDGE)            Next Fedge around face
(FE-PREV  NIL :TYPE FEDGE)            Previous Fedge of face
(FE-NEDF  NIL :TYPE FEDGE)            Next Fedge of Edge
(FE-PEDF  NIL :TYPE FEDGE)            Previous Fedge of Edge
(FE-ASSOC NIL :TYPE LIST))
```

The fedge imposes an orientation on its edge. A source vertex of a fedge g is obtained from the edge, (FE-EDGE g), as either vertex A or vertex B depending on the orientation flag (FE-A2B? g). Figure 4.4.3 shows the source and the destination vertices with respect to a given fedge. The following four macros are provided to obtain the source and destination vertices and the source and the destination points of a fedge.

- (FE-SRCV g) — *Source vertex*
  → (IF (FE-A2B? g) (ED-VRTA (FE-EDGE g)) (ED-VRTB (FE-EDGE g)))

- (FE-DSTV g) — *Destination vertex*
  → (IF (FE-A2B? g) (ED-VRTB (FE-EDGE g)) (ED-VRTA (FE-EDGE g)))

- (FE-SRCP g) — *Source Point*
  → (VE-POINT (FE-SRCV g))

- (FE-DSTP g) — *Destination Point*
  → (VE-POINT (FE-DSTV g))

An important topological relationship which is not represented directly for nonmanifold solids is the pairing of two faces incident on a given edge bordering a solid volume. For manifold solids given a fedge g corresponding to a given face-edge relationship, the other face incident on that edge is simply (FE-FACE (FE-NEDF g)). Nonmanifold edges are rare, and so the fedges around an edge are not kept ordered.

Since the fedges of an edge can be paired up so that each fedge has one co-fedge; a FE-COFEDGE function is provided to obtain the corresponding co-fedge. From the two co-fedges, the co-faces can be easily obtained.

• **(FE-COFEDGE g)** *Find cofedge*
→ fedge

Two useful operations on fedges is the invector, and the convex predicate. The vector lying in the face and pointing into the face from an edge is called the invector.

• **(FE-INVECTOR g &OPTIONAL (v (MAKE-COORDS)))** *Compute invector*
→ v

• **(FE-CONVEX? g)** *Determine convexity of sector*
→ boolean

Given that f is the face **(FE-FACE g)**, the invector is computed as follows:

```
(IF (XOR (FE-A2B? g) (FA-FLIP? f))
    (CROSS-PROD (PL-VECT (FA-PLANE f)) (ED-LNDIR (FE-EDGE g)) v)
    (CROSS-PROD (ED-LNDIR (FE-EDGE g)) (PL-VECT (FA-PLANE f)) v))
```

The convexity of a sector is similarly computed as:

```
(LET ((v (get-vr-register)))
  (PROG1
    (IF(FE-A2B? g)
       (>= (DOT-PROD (ED-LNDIR (FE-EDGE g))
                     (FE-INVECTOR (FE-PREV g) v)) *-EPSILON*)
       (<= (DOT-PROD (ED-LNDIR (FE-EDGE g))
                     (FE-INVECTOR (FE-PREV g) v)) *EPSILON*))
    (RELEASE-VR-REGISTER)))
```

4.4.4 Faces

A face is a single connected edge-vertex loop in a plane. A face is represented by the following structure:

```
(DEFSTRUCT FACE
    "Face of a solid"
    (FA-SOLID  NIL  :TYPE SOLID)
    (FA-NEXT   NIL  :TYPE FACE)          Next Face of Solid
    (FA-PREV   NIL  :TYPE FACE)          Previous Face of Solid
    (FA-PLANE  NIL  :TYPE APLANE)
    (FA-SURF   NIL  :TYPE SURFACE)
    (FA-FLIP?  NIL  :TYPE BOOLEAN)
    (FA-FEDS   NIL  :TYPE FEDGE)         Fedge Loop of Face
    (FA-NFEDS  0    :TYPE COUNTER)       Number of Fedges
    (FA-NFRF   nil  :TYPE FACE)          Next Face of Fragment
    (FA-ASSOC  NIL  :TYPE LIST))
```

The number of fedges bordering the face is FA-NFEDS. The fedge loop around the face is anchored at FA-FEDS.

Two basic methods exist for representing multi-connected faces (i.e., faces with holes). In one method, the edges of each hole are maintained separately as loops, and each face keeps track of the loops. One loop, usually the first one, is always the outside loop. The other loops are the holes. In the other method, no loop structure is used. Instead, bridge edges connect the outer edge loop with each inner edge-loops [12]. Thus, a single edge loop results. There is no restriction on where the bridge-edges occur as long as a single complete loop connects all the edges of the

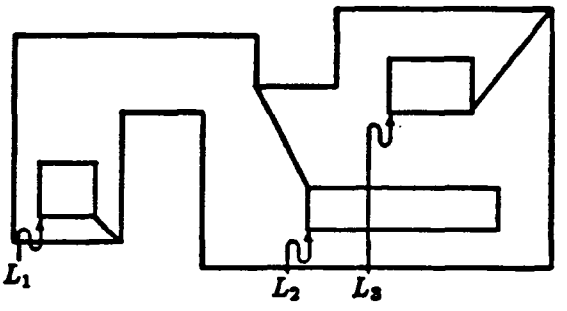

Figure 10: Three bridge edges connect the outer edge loop with three inner edge loops resulting in a single edge loop. The face has 29 fedges, 6 of which correspoond to the 3 bridge edges.

face. Figure 10 shows a face with two holes and two bridge edges. The three edge-loops are labeled as L_1, L_2, and L_3. This second method of using bridge-edges is adopted in the fedge-based data structure.

The plane can be computed from the points of the face by using Newell's averaging method.

- `(FIND-PLANE-OF-FACE f)` *Compute plane equation*
  → `(VALUES plane boolean)`

When the face is multi-connected, the number of inner holes can be computed by tracing around the face and noting the bridge edges.

- `(FA-NUM-HOLES f)` *Find number of inner loops*
  → `counter`

PROTOSOLID is a facetted modeler. In order to provide some indication of the nature of the curvature of the boundary, faces are tagged with implied surface information. For example, a face that is a result of a cylinder is given the parameters of the cylinder, namely, a point on the axis of the cylinder, the axis, and the radius. This information can be used for a variety of reasoning and display purposes. The implied surface is kept in the `FA-SURF` field. In case that the implied surface is a plane, the two fields `FA-SURF` and `FA-PLANE` are identical structures of the plane.

4.4.5 Solids

Each solid is represented by a header node with the following structure:

```
(DEFSTRUCT SOLID
    "Solid Header structure"
    (SO-HISTORY  NIL    :TYPE LIST)                    Program for the solid
    (SO-FACES    NIL    :TYPE FACE)                    Unordered set of faces
    (SO-NFACES   0      :TYPE FIXNUM)                  Number of faces
    (SO-EDGES    NIL    :TYPE EDGE)                    Unordered set of edges
    (SO-NEDGES   0      :TYPE FIXNUM)                  Number of edges
    (SO-VERTICES (MAKE-DIR-TREE) :TYPE DIR-TREE)       Vertex directory
    (SO-CLEAN?   T      :TYPE BOOLEAN)                 Fragmentation flag
    (SO-NAME     "No Name" :TYPE STRING)
    (SO-EXTENT   (MAKE-EXTENT) :TYPE EXTENT)
    (SO-ASSOC    NIL    :TYPE LIST))
```

The number of vertices in a solid are kept in the vertex-directory header.

• (SO-NVERTICES s) *Number of vertices of solid*
→ (DIR-SIZE (SO-NVERTICES s))

SO-HISTORY contains the unevaluated program code which creates this solid. SO-FACES, and SO-EDGES are the anchor nodes for the corresponding doubly-linked circular lists. SO-VERTICES is the directory header node for the vertex directory.

Internally, all existing solids are kept on a stack of solids.

• (DEFVAR *SOLIDS* NIL) *Currently existing solids*

The currently existing solids can be visited by iterating of the list of solids.

• (DOLIST (s *SOLIDS*) body) *Iterate*
→ NIL

It is assumed that this list is never manipulated by the user. Many of the operations on solids maintain this list implicitly and its integrity should not be jeopardized by the user. To allow control of the list, several functions are provided.

• (ROTATE-SOLID-STACK) *Put top solid on bottom*
→ solid

This function rotates the stack of solids by moving the top solid to the bottom and by returning the new top solid.

• (SWAP-SOLIDS-ON-STACK) *Exchange top two solids*
→ solid

This function swaps the top two solids on the stack and returns the new top solid. The entire stack of solids can be properly deleted.

• (CLEAR-EVERYTHING) *Reset everything*
→ NIL

Solids that are deleted are removed from the list of solids and its nodes are returned to corresponding free pools. Once a solid has been deleted by the DELETE-SOLID function, it is assumed that there are no outside pointers referring to any of the nodes of that solid.

5 Programming with PROTOSOLID

5.1 Looping

Frequently used flow control structures are the *for-each* macros. These macros allow a piece of code to execute once for every adjacent node of a specified node. The syntax is analogous to the DOLIST macro in Lisp. Each looping macro allows a single variable to be bound over the scope of the body. The body is treated like a block which allows abnormal exits via the RETURN-FROM command.

• (FOR-EACH-VERTEX-OF-SOLID (vertex solid order) body) *Iterate*
→ NIL

Since the vertices of a solid are organized in a directory, the vertices are spatially ordered by the x, then the y, and finally the z coordinates of the point corresponding to vertex. Furthermore, the vertices can be visited according to the specified order, namely, the :PREORDER, the :INORDER or the :POSTORDER.

• (FOR-EACH-EDGE-OF-SOLID (edge solid) body) *Iterate*
→ NIL

- **(FOR-EACH-FACE-OF-SOLID (face solid) body)** *Iterate*
  → NIL

- **(FOR-EACH-FEDGE-OF-FACE (fedge face) body)** *Iterate*
  → NIL

- **(FOR-EACH-EDGE-OF-FACE ((edge a2b?) face) body)** *Iterate*
  → NIL

- **(FOR-EACH-EDGE-OF-VERTEX (edge vertex) body)** *Iterate*
  → NIL

- **(FOR-EACH-FEDGE-OF-EDGE (fedge edge) body)** *Iterate*
  → NIL

- **(FOR-EACH-FACE-OF-EDGE (face edge) body)** *Iterate*
  → NIL

6 Memory Management

One nice property of Lisp is that it provides a garbage collector that reclaims dead space. Unfortunately when a large number of big structures is allowed to die after a short use, the increase in core image and the need for a very large swap space is the only thing accomplished. Consequently, PROTOSOLID avoids this problem by maintaining its own memory manager that reuses once deallocated structures.

Because of the design of the fedge-based data structure, free nodes are retained as lists of nodes or as trees.

```
(DEFSTRUCT FREE-LIST
    (FL-HEAD nil)
    (FL-SIZE 0))
```

(DEFSTRUCT (FREE-DIR-TREE (:INCLUDE DIR-TREE))) Once a structure is allocated from the system, it is either in use in a solid, or it resides on a free list or tree. It is not allowed to die. Each of the topological and geometrical entities are kept on an appropriate list or tree.

- **(DEFCONST $$FREE-PO$$ MAKE-FREE-DIR-TREE)** *Points*

- **(DEFCONST $$FREE-PL$$ MAKE-FREE-DIR-TREE)** *Planes*

- **(DEFCONST $$FREE-VE$$ MAKE-FREE-DIR-TREE)** *Vertices*

- **(DEFCONST $$FREE-ED$$ MAKE-FREE-LIST)** *Edges*

- **(DEFCONST $$FREE-FA$$ MAKE-FREE-LIST)** *Faces*

- **(DEFCONST $$FREE-FE$$ MAKE-FREE-LIST)** *Fedges*

- (DEFCONST $$FREE-SO$$ MAKE-FREE-LIST)

Each of the structures can be allocated and deallocated by:

- (ALLOCATE-type)
  → type-node

- (DEALLOCATE-type n) *Return* n *to the free pool*
  → type-node

7 Creating Simple Solids

PROTOSOLID currently provides several functions for creating simple solids. Since PROTOSOLID provides for the construction of complex solids to be evaluated from set operations, simple parameterized solids must be provided as the initial building blocks. The parameterized solids are easily parameterised, such as the block, the cone, and the cylinder. In addition to the parameterized solids, a class of primitives consisting of extruded solids is also provided by lifting a face into a solid.

7.1 Lifted Solids

CREATE-LIFTED-SOLID is a general primitive for creating solids by lifting a non-self-intersecting planar polygon into a solid. For example, the block and the cylinders are created by this lifting primitive. The primitive takes two required parameters. The first parameter :LIFT specifies a vector that gives the direction and distance to lift the face containing the polygon. The second parameter :POINTS gives a list of points around the bottom face. The order of the points should be counterclockwise when the face is viewed from the outside of the solid. Reversing this order will create a complement of a solid, a hole in space. Such a reversed solid has an infinite volume and is not considered a physical object. An optional third parameter :TRANSLATE is a vector that specifies a translation to be applied to the list :POINTS. (See the CREATE-BLOCK primitive for an example).

- (CREATE-LIFTED-SOLID &KEY lift points (TRANSLATE '(0.0 0.0 0.0)))
  → solid

For all the creation primitives, including the CREATE-LIFTED-SOLID, the resulting solid is created by making and then gluing into the solid the appropriate faces of that solid. This face creation is driven by the MAKE-FACE-OF-SOLID routine.

- (MAKE-FACE-OF-SOLID s points &OPTIONAL plane flip?)
  → face

This routine accepts a list of unique points lying in a common plane and given in a counterclockwise order around the face when viewed from the outside, creates a face whose vertices correspond to the points given, and glues the face to the solid s. The second argument, points, is a list of point nodes which are assumed to reside in the point directory $POINTS$. If a plane equation, plane is specified with the orientation of the normal vector, flip?, then the points should lie in that plane. If the plane equation is not specified, the equation of the plane is computed from the points. This is done by the use of FIND-PLANE-OF-FACE function.

7.2 Block

The create-block primitive generates a rectalinear block whose sides are translations of the x, the y, and the z planes. The block is specified by a corner point and three non-zero dimensional values, corresponding to the length, the width, and the height of the block.

```
• (CREATE-BLOCK &KEY
                (CORNER '(0.0 0.0 0.0))
                (DIMENSIONS '(1.0 1.0 1.0))
                (NAME "Block"))
  → solid
```

(CREATE-BLOCK :DIMENSIONS '(,x ,y ,z)) is a valid specification if x > 0 and y > 0 and z > 0. As an example of using the creation routines, consider the use of the CREATE-LIFTED-SOLID procedure to create the block. The CREATE-BLOCK procedure can be easily coded as

```
(DEFUN CREATE-BLOCK (&KEY
                     (CORNER '(0.0 0.0 0.0))
                     (DIMENSIONS '(1.0 1.0 1.0)))
   (CREATE-LIFTED-SOLID
      :LIFT '(0.0 ,(SECOND DIMENSIONS) 0.0)
      :TRANSLATE CORNER
      :POINTS '((0.0 0.0 0.0)
                (,(FIRST DIMENSIONS) 0.0 0.0)
                (,(FIRST DIMENSIONS) 0.0 ,(THIRD DIMENSIONS))
                (0.0 0.0 ,(THIRD DIMENSIONS)))))
```

7.3 Cone

A CREATE-CONE primitive generates an approximation of a cone whose bottom center point is given by the :CENTER parameter, and whose radius, height, and number of sides are given by their corresponding parameters. The cone is aligned along the positive Y-axis.

```
• (CREATE-CONE &KEY
               (CENTER '(0.0 0.0 0.0))
               (HEIGHT 1.0)
               (RADIUS 0.5)
               (SIDES 8)
               (MAJOR-AXIS :Y)
               (DEGREES 360.0)
               (NAME "Name"))
  → solid
```

(CREATE-CONE :HEIGHT H :RADIUS R :SIDES S) is a valid cone if $0 < H$ and $0 < R$ and $3 \leq s$. The base of the cone is created by inscribing a regular polygon of s sides in a circle of radius r. Although the cone is facetted, the implied surface of the cone is maintained for each of the walls of the cone within the FA-SURF fields. The cone has the following structure.

```
(DEFSTRUCT CONE
    (CN-APEX    NIL :TYPE POINT)
    (CN-AXIS    NIL :TYPE COORD)
    (CN-A-COS   0.0 :TYPE DOUBLE-FLOAT))
```

7.4 Cylinder

A CREATE-CYLINDER primitive generates an approximation of a cylinder whose bottom face is centered at a given point which is aligned along the positive Y-axis. The height, the radius, and the number of sides are given by their corresponding parameters.

```
• (CREATE-CYLINDER &KEY
                   (CENTER '(0.0 0.0 0.0))
                   (HEIGHT 1.0)
                   (RADIUS 0.5)
                   (SIDES 8)
                   (MAJOR-AXIS :Y)
                   (NAME "Cylinder"))
  → solid
```

(CREATE-CYLINDER :HEIGHT H :RADIUS R :SIDES S) is a valid cylinder if $0 < H$ and $0 < R$ and $3 \leq S$. The cylinder is created by lifting a regular polygon of s sides inscribed (or circumscribe) in a circle of radius r a distance of height h. Although the cylinder is facetted, the implied cylindrical surface is maintained for each of the walls of the cylinder within the FA-SURF fields. A cylinder has the following structure.

```
(DEFSTRUCT CYLINDER
    (CY-POINT     NIL :TYPE POINT)
    (CY-AXIS      NIL :TYPE COORD)
    (CY-RADIUS    1.0 :TYPE DOUBLE-FLOAT))
```

7.5 Sphere

A CREATE-SPHERE primitive generates an approximation of a sphere with a given center and a radius.

```
• (CREATE-SPHERE &KEY
                 (CENTER '(0 0 0))
                 (RADIUS 0.5)
                 (SIDES 8)
                 (LAYERS 8)
                 (DEGREES 360)
                 (NAME "Sphere"))
  → solid
```

(CREATE-SPHERE :RADIUS R :SIDES S :LAYERS L) if a valid sphere if $R > 0$, $S > 2$ and $L > 1$. The sphere has the following structure.

```
(DEFSTRUCT SPHERE
    (SP-CENTER    NIL :TYPE POINT)
    (SP-RADIUS    NIL :TYPE DOUBLE-FLOAT))
```

7.6 Torus

A CREATE-TORUS primitive generates an approximation of a torus.

```
• (CREATE-TORUS &KEY
                (CENTER '(0 0 0))
                (AXIS '(0 1 0))
  (MINOR 0.25)  Minor Radius
   (MAJOR 1.0)  Major Radius
```

```
                (SECTIONS 16)
                (SIDES 8)
                (DEGREES 360)
                (NAME "Torus"))
```

→ solid

The torus has the following structure.

```
(DEFSTRUCT TORUS
    (TR-CENTER   NIL :TYPE POINT)
    (TR-AXIS     NIL :TYPE COORD)
    (TR-MINOR    NIL :TYPE DOUBLE-FLOAT)
    (TR-MAJOR    NIL :TYPE DOUBLE-FLOAT))
```

7.7 Tetrahedron

A CREATE-TETRAHEDRON primitive generates a simplex given its four distinct and not all planar points.

- (CREATE-TETRAHEDRON p1 p2 p3 p4)
  → solid

8 Operations on solids

PROTOSOLID provides many operations that manipulate solids. In this section, these operations are presented.

8.1 Boolean Set Operations

The regularized set operations on two solids are the backbone of PROTOSOLID. To get a detailed exposition of the algorithm used in evaluating the set operations, refer to Vaněček and Nau[10, 8, 9]. Here I present only a short exposé on the use of the set operations.

For three dimensional objects, a classification of faces of one solid is required with respect to the other solid. This can become quite complex, especially if the solids modeled cannot be assumed to be manifolds. In general, solids are not closed under pure set theoretic operations. A set theoretic operation applied to two valid solids does not necesarily result in a valid solid. This occurs when the two solids touch in particular ways. Requicha and Voelcker have shown that set theoretic operations must be regularized. Conceptually, the regularization of a object removes all lower dimensional entities such as isolated vertices and dangling edges that do not border the interior of the object (an object is regular if it is equal to the closure of its interior). As a result, a simple approach for performing set operations on two solids is to perform a set-theoretic set operation and then to regularize the result. The efficient approach used by PROTOSOLID is to directly perform a regularized set operation. This is done in two steps. Step one cuts up the boundaries of the two solids so that a face of one of the solids intersects the other solid only at bordering edges or vertices (and therefore each face can be homogeneously classified with respect to the other solid). Step two constructs the desired solid by combining the appropriate faces of the two solids. Each of the following three functions accepts two solids and returns a regularized solid as the result.

- (UNION-SOLIDS s1 s2)
  → solid

The union of solids s1 and s2 is a solid whose interior is the combination of the interiors of s1 and s2.

- **(INTERSECT-SOLIDS s1 s2)**
  → **solid**

  The intersection of solids s1 and s2 is a solid whose interior is common to both s1 and s2.

- **(SUBTRACT-SOLIDS s1 s2)**
  → **solid**

  The subtraction operation results in a solid in which the interior of solid s2 is removed from the interior of solid s1.

  The classification step leaves the original two sets of faces partitioned into eight classification sets used subsequently by the construction step. The eight classification sets are subsets of a face set called *fragments*.

```
(DEFSTRUCT FRAGMENT
    (FR-FRONT NIL :TYPE FACE)
    (FR-REAR NIL :TYPE FACE)
    (FR-SIZE 0 :TYPE COUNTER)
    (FR-CLASS :UNKNOWN))
```

- **(DEFCONST $AinB$ (MAKE-FRAGMENT))** — *Faces of A inside B*
- **(DEFCONST $BinA$ (MAKE-FRAGMENT))** — *Faces of B inside A*
- **(DEFCONST $AoutB$ (MAKE-FRAGMENT))** — *Faces of A outside B*
- **(DEFCONST $BoutA$ (MAKE-FRAGMENT))** — *Faces of B outside A*
- **(DEFCONST $AonB$ (MAKE-FRAGMENT))** — *Faces of A with B*
- **(DEFCONST $BonA$ (MAKE-FRAGMENT))** — *Faces of B with A*
- **(DEFCONST $Aon-B$ (MAKE-FRAGMENT))** — *Faces of A anti B*
- **(DEFCONST $Bon-A$ (MAKE-FRAGMENT))** — *Faces of B anti A*

The faces in each fragment are linked together in a singly linked list by the FA-NFRF field. A face can be added to and removed from a fragment.

- **(FR-ADD-FACE f fr)** — *Add a face to fragment*
  → **fr**

- **(FR-REMOVE-FACE fr)** — *Remove front face*
  → **face**

- **(FR-APPEND-FACES sfr dfr)** — *Move faces from one to another*
  → **dfr**

- **(FR-COLLECT-FACES solid fr)** *Get all faces of a solid*
  → **fr**

The two solids that are classified are kept in the variables:

- **(DEFVAR *ASolid* NIL)** *Last solid classified in relation to B*

- **(DEFVAR *BSolid* NIL)** *Last solid classified in relation to A*

Deleting either of these two solids automatically clears the eight classification sets and the two variables. This can also be accomplished explicitly by

- **(CLEAR-CSETS)** *Clear the classification sets*
  → **NIL**

or individually by

- **(FR-CLEAR fr)** *Clear a fragment*
  → **fr**

Do to the nature of the algorithm that computes the set operations, both **s1** and **s2** are modified by having some of their faces split into subfaces. Since this face splitting is purely a topological modification of the solid, the split faces of a solid may again be merged to yield the original boundary by the following operation:

- **(CLEAN-UP-SOLID s)** *Apply topological reduction*
  → **s**

Each solid **s** maintains an indication of whether or not its faces have been split within the **(SO-CLEAN? s)** field. Using a solid as an argument to one of the set operations sets this field to true. Since the set operations expect a clean solid, a solid is automatically cleaned up if it is used in a set operation.

When merging faces during a clean-up operation, PROTOSOLID takes into account the face properties and the implied surfaces. A question as to the merging of coplanar and touching faces arises when the two face properties differ. Consider for example a stock from which two notches have been removed, one inside the other. The top notch is large, the inside notch is smaller, but one face is coplaner with a vertical face of the upper notch. The question that arises during merging is, should the two vertical and coplanar faces be merged, and if so, whose property should they inherit? The convention is that if the two faces have different properties or different implied surfaces, they cannot be merged. Thus, two faces can be merged if they have the same planes, **FA-PLANE**, the same plane orientation flags, **FA-FLIP?**, and the same surfaces, **FA-SURF**.

For a closer view of the set operations, consider the union operation. A simplified version of the union set operation code follows:

```
(DEFUN UNION-SOLIDS (S1 S2)
   "Return the union of two solids"
   (UNLESS (SO-CLEAN? S1) (CLEAN-UP-SOLID S1))
   (UNLESS (SO-CLEAN? S2) (CLEAN-UP-SOLID S2))
   (LET ((S (MAKE-SOLID)))
     (CLASSIFY-FACES-OF-SOLIDS S1 S2)
     (FOR-EACH-FACE-OF-FRAGMENT (F *AoutB*)
         (COPY-FACE-INTO-SOLID F S))
     (FOR-EACH-FACE-OF-FRAGMENT (F *BoutA)
         (COPY-FACE-INTO-SOLID F S))
     (FOR-EACH-FACE-OF-FRAGMENT (F *AonB*)
         (COPY-FACE-INTO-SOLID F S))
     (PUSH S *SOLIDS*)
     (RETURN (CLEAN-UP-SOLID S))))
```

After the two solid s1 and s2 have been classified, the resulting solid s is created by systematically gluing together the desired faces. This glueing of faces together is performed by the COPY-FACE-INTO-SOLID procedure which, when given a face of some other solid, duplicates the face in the destination solid.

- (COPY-FACE-INTO-SOLID f s)
  → face

The operation for taking the complement of a solid has not been provided. There are several reasons for this:

1. The complement of a solid is not considered a solid, because the interior is unbounded.
2. There is no practical use of the complement of a solid.
3. The set operations must handle the complement of a solid differently than solids. However, this is a minor reason, as the representation and coding can be easily modified to handle the complement.

8.2 Separation of solids

The results of set operations on two solids is not necessarly a solid consisting of a single component. A solid does not need to be a single component. This is convenient since the set operations do not need not know about multi-component solids (i.e., solids with more than one shell). As a result, if a solid is known, or suspect, to consist of multiple-components, one must be able to separate the solid into several solids consisting of a single components. The SEPARATE-SOLID operation performs this separation. It accepts a single solid and returns a list of one or more solids.

- (SEPARATE-SOLID s &OPTIONAL (number n))
  → list of solids

The need for an explicit separation of a solid operation is in part due to the convention that each solid maintains its own topology. A solid consisting of several components has a combined topology of each of the components. The separation of a given solid s having several components, results only in the partitioning of the sets (SO-VERTICES s), (SO-EDGES s), and (SO-FACES s). As an example of separation, Figure 3 shows three solids. When separated, the first two solids each separate into two tetrahedral solids. The third one, the pseudo-manifold, does not separate.

When discussing the separation of components the issue of what exactly is meant by a component cannot be ignored. If the class of solids being modeled was strictly the class of solids bounded by two-manifolds, a component could be simply defined topologically as the set of all connected faces. However, in the world of nonmanifolds, solids become a bit more difficult. Intuitively, a solid having two components can be separated if the two components are topologically disjoint, or if they touch only at isolated points or along lines. The components cannot share a common interior. That is, for any point in the interior of one component, and for any point in the interior of the other component, there cannot exist a path between the two points that are contained entirely in the interior of the solid. The SEPARATE-SOLID operation correctly handles nonmanifolds in the sense that it returns a list of non-separable solids.

When two solids do not intersect in space, they can be combined by a union operation. However, this is an expensive operation.

- (ADD-SOLID srcs dsts) *Add src solid to dst solid*
  → dsts

This operation copies the source solid into the destination solid without all the overhead of performing a full eight-way classification. Of course, if the two solids intersect, the resulting solid will be invalid, so use this operation with caution.

The ADD-SOLID operation is useful in the following setting. Assume that a given solid consists of several components and you wish to remove some of these components. First separate the solid and then combine the desired ones.

8.3 Transforming Solids

Two basic transformations can be applied to solids, a rotation and a translation. Both transformations are rigid in the sense that the solid is repositioned in object space and maintains its shape without undergoing any deformation. The transformations affect only the geometry of a solid, the topological structure remains unchanged. Unlike the same operations in other modelers, however, these operations are non-destructive. A transformation creates a new solid in the newly desired place, and so the solid is copied.

- `(TRANSLATE-SOLID s &KEY (VECTOR '(1 0 0)))`
  → solid

The TRANSLATE-SOLID operation performs the translation by the specified vector. If a null vector is specified, this operation acts like the DUPLICATE-SOLID command.

A rotation can be preformed three different ways. The first way is by specifying the degrees of rotation about each major axis. The object space in which the transformations take place is a right-handed coordinate system. This is important, as the difference between a left-handed and a right-handed coordinate system dictates the rotational direction about a given axis. The rotation about each axis is counterclockwise when viewing the solid along the desired axis towards the origin. Figure 8.3 shows the direction of a positive rotation about each axis.

- `(ROTATE-SOLID-XYZ s &KEY (CENTER '(0 0 0)) (DEGREES '(0 90 0)))`
  → solid

Rotating a solid this way is not very intuitive. The other two ways are easier to use. The second way performs a rotation about a single point and a vector.

```
• (ROTATE-SOLID-ABOUT-AXIS s &KEY
                           (POINT '(0 0.5 0))
                           (DEGREES '90)
                           (AXIS '(1 0 0)))
  → solid
```

The third way of rotating is by specifying a point and two vectors. The solid is rotated around the point so that the first vector comes in alignment with the second vector.

```
• (ROTATE-SOLID-TO-AXIS s &KEY
                        (POINT '(0 0.5 0))
                        (SAXIS '(1 0 0))
                        (DAXIS '(0 1 0)))
  → solid
```

8.4 Sectioning

The sectioning of a solid by a single cut is provided by the SPLIT-SOLID command. The form of this command is:

- `(SPLIT-SOLID n s &OPTIONAL (WHICH :BOTH))`
  → list of solids

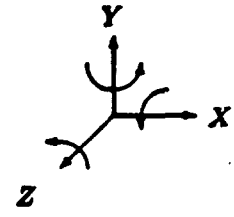

Figure 11: Positive rotation is counter-clockwise in a right-handed coordinate system.

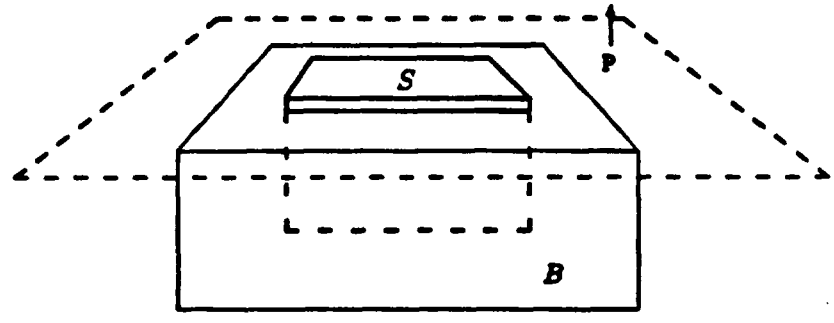

Figure 12: Cutting solid S by plane P by using a box B.

The solid s is cut by a plane specified by a point on the plane (which is currently taken to be the center of the object's extent) and the plane's normal vector n. By default, the operation returns both halves of the solid in a list. Optionally, the fourth argument can specify which half to return, either :BELOW or :ABOVE the plane, in which case only that half is returned. This operation is non-destructive. That is, s itself is not cut. Rather its two halves are partial copies of s.

The separation is performed using the set operators. A block B is constructed that completely encompasses solid s below the splitting plane P. Refer to the diagram in Figure 12. Then, the solid above the splitting plane is obtained by subtracting B from s, and the solid below the splitting plane is obtained by intersecting B and s.

8.5 Copying and Deleting

A copy of a solid can be created by the DUPLICATE-SOLID command which clones a solid s and returns the new solid. It has the following form:

- (DUPLICATE-SOLID s)
  → solid

The new solid is created by the use of the COPY-FACE-INTO-SOLID routine which duplicates a face of solid s in the new solid.

What comes must go, and so the destruction of solids is also mandated. Although the implementation is in Lisp, it is not allowed to simply lose all pointers to a solid and let the system reclaim the space through garbage collection. The removal of a solid requires the updating of other data structures dependent on the knowledge of the existence of all created solids. This is primarily required due to the global sharing of geometrical data. A solid can be properly deleted by the delete-solid command which removes the solid from existence and returns nil. It has the following form:

- (DELETE-SOLID s) *Deallocate all its nodes*
  → NIL

8.6 Storing and Loading Solids

Each solid is kept in two forms; the unevaluated CSG form and the evaluated boundary form. A solid can be stored in either of these forms. Storing only the unevaluated form is very concise, however it requires reevaluation to convert back into the explicit boundary form. On the other hand, storing the evaluated form requires no reevaluation except the proper reconstruction of the solid; however it could require large amounts of space. Both forms are valid for externally storing a solid. The following operation writes a solid to a file.

- `(SAVE-SOLID S &KEY`
  `(BREP? NIL)`
  `(FILE-NAME *DEFAULT-PART-NAME*)`
  `(DIRECTORY *PARTS-DIRECTORY*))`

  → boolean

Writing out a solid does not delete, or modify it in any way.

`(SAVE-SOLID S :BREP? T)` is equivalent to producing a file which contains the call to

- `(CREATE-SOLID history points faces)`

  → solid

which when evaluated takes the list of points and the list of faces to recreate the solid. The list of faces is a list of lists, where each sublist contains the indices of the bordering vertices of one face. As an example, a unit block can be created as follows:

```
(CREATE-SOLID
    '(create-block :so-name "Unit Block")
    '((0 (0 0 0)) (1 (1 0 0)) (2 (0 1 0)) (3 (1 1 0))
      (4 (0 0 1)) (5 (1 0 1)) (6 (0 1 1)) (7 (1 1 1)))
    '((0 2 3 1) (0 4 6 2) (0 1 5 4)
    (2 6 7 3) (4 5 7 6) (1 3 7 5)))
```

What's written can also be read back in. However the file was written by the SAVE-SOLID command, the content of the file is Lisp code. To obtain the solid in a file, simply load the file by the LOAD command.

So lets consider in more detail the unevaluated form and how it is manipulated. Programs for creating solids have one of the two following Lisp forms:

1. `(CREATE-primitive ...)`

2. `(LET (S1 ... Sn)`
   `(SETQ S1 (CREATE-primitive...))`
   `...`
   `(SETQ Sn (binaryop-SOLIDS Si Sj)))`

In form 2, the variable names are assumed to be S1 to Sn for some $n > 0$. Merging two such programs, say PR1 and PR2, with names S1..Sn and S1..Sm respectively, requires that all the names of PR2 be renamed Sn+1..Sn+m. This unevaluated form for solid description was chosen for three reasons. One, it can be simply evaluated to return the desired solid. Two, it is easier to read than is a deeply nested binary tree representing the equivalent CSG form. And three, it is easier to automatically simplify and insert `(DELETE-SOLID Si)` commands prior to saving the program in a file.

The unevaluated programs are kept in by the SO-HISTORY fields of the solids and are created by the following functions:

- `(COMBINE-PROGS binary-op prog1 prog2)`

  → prog

• (APPLY-PROG unary-op prog &REST args)
→ prog

• (GC-PROG prog) *Insert DELETE-SOLID calls*
→ prog

GC-PROG is called by SAVE-SOLID prior saving the program to insert the DELETE-SOLID calls in the right places. This is because each program is supposed to create a single solid. However in the process of creating that solid, many temporary solids might be created. It is therefore space efficient to delete any temporary solids that are no longer needed.

8.7 Detecting intersections of two solids

A useful operation on solids is the operation that detects the interference between two solids. This operation replies affirmatively to the query "Do the two solids intersect?" if there is some volume of space that is occupied by both solids. The touching of two solids does not constitute an intersection of the two solids. The answer to the non-empty intersection of two solids could be obtained by computing the actual intersection, and checking to see if the resulting intersection is empty or a solid. However, there is a price paid for constructing the intersection explicitly. The knowledge of whether or not the two solids intersect can be obtained long before the two solids can be fully intersected. For this reason, the following command returns the answer without constructing the intersecting solid:

• (SOLIDS-INTERSECT? s1 s2)
→ one of T, NIL or :TOUCH

Another useful operation is detecting whether or not two solids are equal. Here equality means that a boundary A and B share a common interior. Thus A itself does not have to be identical to B. The following command returns true if and only if the two solids represented by S1 and S2 are equal:

• (EQUAL-SOLIDS? s1 s2)
→ boolean

9 Mass Properties of solids

PROTOSOLID computes mass properties of solids such as the volume, the surface area, the centroid, and the moments of inertia.

9.1 Volume and Surface Area

In 2D, the area of a polygon can be computed by choosing a line and summing up the trapezoids formed by projecting each edge onto the line. This leads to an algorithm in 3D. Instead of using trapezoids, sum up the signed volumes of all the tetrahedrons formed by four points consisting of two adjacent points on a face, a fixed point on a face, and a point outside the solid.

• (SO-VOLUME s) *Volume of a solid*
→ double-float

The function is now given.

```
(LET ((cp (extent-center (so-extent s) (get-vr-register)))
      (volume 0.0))
  (INCF (vr-x cp) (- (vr-x (ex-maxp (so-extent s)))
```

```
                    (vr-x (ex-minp (so-extent s)))))
  (for-each-face-of-solid (f s)
     (LET ((rp (po-wpnt (fe-srcp (fa-feds f)))))
        (for-each-fedge-of-face (g f)
           (INCF volume (tetrahedron-svolume cp rp (po-wpnt (fe-srcp g))
                                                   (po-wpnt (fe-dstp g)))))))
  (release-vr-register)
  volume)
```

The function to compute the signed volume of a tetrahedron is a needed function for computing most of the mass properties.

- **(TETRAHEDRON-SVOLUME p1 p2 p3 p4)** *Signed Volume of a tetrahedron*
  → double-float

The absolute value of the result returned by TERTAHEDRON-SVOLUME is the volume. The result is positive when the points p2, p3, and p4 appear clockwise when viewed from p1. The result is negative when the three points are counter-clockwise. The signed volume v is the determinant:

$$v = \frac{1}{6}\begin{vmatrix} (x_2 - x_1) & (x_3 - x_1) & (x_4 - x_1) \\ (y_2 - y_1) & (y_3 - y_1) & (y_4 - y_1) \\ (z_2 - z_1) & (z_3 - z_1) & (z_4 - z_1) \end{vmatrix}$$

The surface area of a solid is computed similarly to that of the volume.

- **(SO-SURFACE-AREA s)**
  → double-float

9.2 Center of Mass

The center of mass of a tetrahedron is the point

$$c = \frac{1}{4}(p_1 + p_2 + p_3 + p_4).$$

- **(TETRAHEDRON-CENTROID p1 p2 p3 p4)** *The center of mass of a tetrahedron*
  → coords

The centroid of an arbitrary solid is computed by dividing the first moment by the volume.

- **(SO-CENTROID s)** *Center of mass of a solid*
  → coord

9.3 Moments of Inertia

- **(so-first-moment s &OPTIONAL (m (make-coords)))**
  → m

- **(so-second-moment s &OPTIONAL (m (make-coords)))**
  → m

- **(so-product-of-inertia s &OPTIONAL (p (make-coords)))**
  → p

- `(so-moment-of-inertia s &OPTIONAL (m (make-coords)))`
  → `m`

If most or all of the properties are desired, it is more efficient to compute them all simultaneously rather than one at a time.

- `(so-mass-properties s)` *Return all mass properties*
  → `(VALUES v c s mi m2 pi mi)`

The returned values are the volume, the centroid, the surface area, the first moment, the second moment, the product of inertia, and the moment of inertia.

Acknowledgements

I wish to thank Prof. Nau for his continued support in developing PROTOSOLID.

Bibliography

References

[1] C. M. Eastman and K. Weiler. Geometric modeling using the euler operators. In *Proceedings of First Annual Conference on Computer Graphics in CAD/CAM Systems*, pages 248–254, April 1979.

[2] M. Mäntylä. A note on the modeling space of euler operators. *Computer Vision, Graphics, and Image Processing*, 26:45–60, 1984.

[3] D. S. Nau, N. Ide, R. Karinthi, G. Vaněček Jr., and Q. Yang. Solid modeling and geometric reasoning for design and process planning. In *Third Internat. Conf. CAD/CAM, Robotics, and Factories of the Future*, Southfield, MI, August 1988.

[4] Aristides A. G. Requicha and H. B. Voelcker. Constructive solid geometry. Technical Report 25, University of Rochester, Rochester, NY, November 1977.

[5] G. L. Steele Jr. *Common LISP, The Language*. Digital Press, Burlington, MA, 1984.

[6] S. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9), September 1986.

[7] G. Vaněček Jr. *Set Operations on Polyhedra using Decomposition Methods*. PhD thesis, University of Maryland, College Park, Maryland, June 1989.

[8] G. Vaněček Jr. and D. S. Nau. Computing geometric boolean operations by input directed decomposition. Technical Report TR–87–8, Systems Research Center, University of Maryland, College Park, MD, 20742, January 1987. See Also TR 1762.

[9] G. Vaněček Jr.. and D. S. Nau. Non-regular decomposition: an efficient approach for solving the polygon intersection problem. In C. R. Liu, A. Requicha, and S. Chandrasekar, editors, *Intelligent and Integrated Manufacturing Analysis and Synthesis*, volume 25 of *PED*, pages 271–280, December 1987.

[10] G. Vaněček Jr. and D. S. Nau. A general method for performing set operations on polyhedra. Technical Report UMIACS-TR-88-48, University of Maryland Institute of Advanced Computer Studies, July 1988.

[11] K. Weiler. Edge–based data structures for solid modeling in curved–surface environments. *IEEE Computer Graphics & Applications*, pages 21–40, January 1985.

[12] F. Yamaguchi and T. Tokieda. Bridge edge and triangulation approach in solid modeling. In Tosiyasu L. Kunii, editor, *Frontiers in Computer Graphics '84*, pages 44–65. Springer-Verlag, 1985.

Index