PLASM Functional Approach to Design: Representation of Geometry

Alberto Paoluzzi
Valerio Pascucci
Michele Vicentino

Dip. di Informatica e Sistemistica
Università "La Sapienza"
Via Buonarroti 12
00185 Roma, Italy

PLASM[1] (the Programming LAnguage for Solid Modeling) is a solid-modeling-oriented design language strongly inspired by the functional language FL. In a PLASM environment, every geometrical object is generated by evaluating a suitable language expression which produces a polyhedral solid model. The language adopts a dimension-independent approach to geometry representation and algorithms. The generated objects are always geometrically consistent since the validity of geometry is guaranteed at a syntactical level. In fact (a) each well-formed expression is obtained by proper composition of well-formed subexpressions; (b) the evaluation of a well-formed (and polyhedrally typed) expression produces a valid solid model. In this paper, the representation scheme used in the language is given and some language scripts are shown and discussed.

Keywords: computer-aided architectural design, variational geometry, design language, geometric design, geometric modeling, functional programming, complexes, polyhedra, convex cells.

1 Introduction

The programming approach described in this paper supports the use of specialized macro-instructions to parametrically generate shape instances of some predefined kind, as well as to formalize the sequence of design decisions as a computer program, possibly with the aid of some interactive user-interface. In this approach a complicated design may be described in a hierarchical fashion and developed either top-down or bottom-up, or even by using some mixed strategy. In fact, with a programming approach, it is much easier to modify the design both when the changes concern some specific part and when they apply to the overall design organization. In the first case, it is possible to substitute or update some language functions, used as "generating forms" for the shape of some design parts;

[1]This work was partially supported by the Italian National Research Council, contract number 91.03226.64, within the "PF Edilizia" Project.

*U. Flemming and S. Van Wyk (eds.), **CAAD Futures '93**, pp. 127-141.*

in the second case, it is sufficient to modify some program units at suitable hierarchical levels.

In a PLASM environment (Paoluzzi et al., 1992a), every geometrical object is generated by evaluating a suitable language form, which produces a polyhedral solid model. PLASM objects are always geometrically consistent since the validity of geometry is guaranteed at a syntactical level. In fact, (a) each well-formed expression is obtained by proper composition of well-formed subexpressions; (b) the evaluation of a well-formed (polyhedrally typed) expression produces a valid solid model. This approach supports the use of a weaker representation scheme than those usually adopted in a solid modeler. Conversely, the PLASM interpreter maintains a complete mapping between the object database and the hierarchy of language forms which generate the shape. Moreover, in the language a dimension-independent approach to solid modeling is taken, which encompasses wire-frames, surfaces and solids as well as higher-dimensional objects in a simple and uniform way.

In designing the language we analyzed the constraint-based languages, which are an important approach to the formal specification of graphical objects. Some constraint-based approach, may be found in early times of computer graphics (Sutherland, 1963) but only recently constraint-based languages were identified (Leler, 1988) as an autonomous programming paradigm, similar, in some respects, to logic programming. In particular we analyzed object-based languages (Borning, 1981), logical languages (Bruderlin, 1985; Barford, 1986), constraint-based (Leler, 1988) and functional languages (Steele and Susmann, 1978). The comparison of their characteristics with architectural design methods led to adoption of a functional approach for the new language, strongly inspired by FP (Backus, 1978) and FL (Backus et al., 1988). Our approach, based on calculation of expressions in a calculus on polyhedra, supportsguaranteed constraints on the expression terms using the standard programming approach (see the examples in Section 7). The validity of geometry is always guaranteed, as explained later in the paper.

Motivation for a dimension-independent approach to solid modeling is widely discussed in (Paoluzzi et al., 1993). A functional approach to 2D graphics can be found in (Lucas et al., 1988; Zilles et al., 1988), where a very interesting functional system equivalent to *Postscript* is described. The interested reader is referred to Backus' Turing lecture (Backus, 1978) for an impressive motivation of programming at function level, to (Backus et al., 1988; Williams 1982) for a more introductory treatment of the topic, and to (Backus et al., 1989) for a complete description of FL syntax and semantics. The PLASM language is presented in (Paoluzzi et al., 1992a). Some design choices of the language described in the following are based on the solid modeling experience gained at “La Sapienza” in developing the 3D polyhedral modeler *Minerva* (Paoluzzi et al., 1989) and the multidimensional prototype modeler *Simplex*^{n} (Paoluzzi et al., 1993). The internal representation of geometric objects in the PLASM language is the subject of the following sections.

2 Background

FL (programming at Function Level) is an advanced language being developed by Functional Programming Group of IBM Research Division at Almaden (USA) (Backus et al., 1988; Backus et al., 1989). FL is a pure functional language based on the use of both predefined and user-definable combining forms, i.e., higher-level functions that are applied to functions to produce new functions. The language introduces an algebra over programs—mainly a set of identities (rewriting rules) between expressions—for reasoning formally about programs. For example, one may find simpler equivalent programs, both at

design time or at compilation time, with great advantages in style and efficiency of program development (Williams et al., 1991).

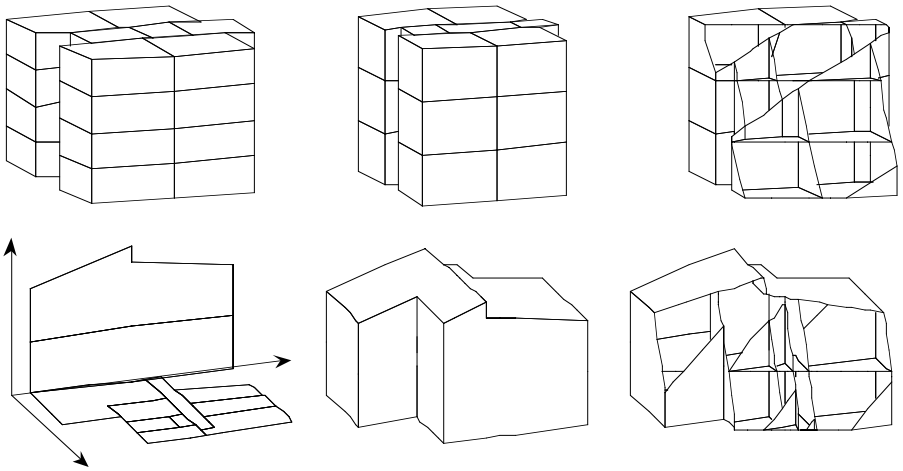

Figure 1. Two examples of variational models produced by simple PLASM functions. The first models are produced by a cartesian product between a 2D complex and a 1D complex. The second one is derived from @2~&&{120,102}:<Plan,Elevation>, where && is on intersection of extrusions, special case of the product operator described in (Bernardini et al., 1993).

The FL approach suports the easy combination of already-defined programs. New programs may be therefore obtained in a very simple and elegant way.

PLASM, the Programming LAnguage for Solid Modeling, is a functional "design language" being developed by the CAD Group at "La Sapienza" (Italy). PLASM can evaluate polyhedral expressions, i.e., expressions whose value is a polyhedron. The language is also able to combine functions to produce higher-level functions in the FL style. In its current version the language can be considered an application shell over FL. Since polyhedrally-valued expressions may appear as parameters of functions, PLASM implements a programming approach to variational geometry. In fact the geometric object resulting from the evaluation of a function may depend on the values of other geometric expressions. In other words a PLASM function can be seen as a shape prototype, or "generating form," which is able to produce infinitely many *different* geometric objects, all with some common structure. When a form is evaluated assigning specific values to parameters, often obtained by evaluation of other forms, the evaluation mechanism produces (and sometime stores) a valid solid model associated to the form.

PLASM is characterized by the use of dimension-independent complexes of polyhedra, where each polyhedron may be non-convex and non-solid. Each primitive polyhedron in a complex is represented as a solid within its affine hull, and stored by using a cell-decomposition in convex cells. The language, besides the regularized Boolean operations of union, intersection, difference (`A+B`, `A&B`, `A/B`) and the complementation (`-A`), provides some additional dimension-independent geometric operators, like the k-skeletons `@k` (k = 0, 1, 2,...) and the boundary `@` of a polyhedral complex, the offset `++` and the intersection of extrusions `&&` (Paoluzzi et al., 1992a), as well as a generalized product `*` of cell-complexes (Bernardini et al., 1993).

3 Structure as a Semilattice

Before describing some PLASM primitive objects and functions as well as the interpretation of PLASM forms, the representation scheme used within the geometrical nucleus of the language is discussed.

It is well-known since the early times of computer graphics (Sutherland, 1963) and of design methods (Alexander, 1965) that a complex object can be modeled as a semilattice, i.e., as an algebraic structure with an operation, which can be seen as containment, and with an identity element. A semilattice, later on called *lattice*, can be modeled as an oriented acyclic multigraph[2] where the source node, in the following called *root*, is the only node of indegree zero. In this multigraph the nodes represent the object parts and the arcs represent the relation of (direct) containment. In this model, any directed path from a node n_i, to a node n_j will represent a different instance of the component n_j within the component n_i. At the same time, the sub-lattice with root n_i, i.e., the part of the multigraph which is reachable from n_i, will explicitly represent a well-defined object part. Each node is represented by using a local modeling coordinate system.

An acyclic multigraph can be stored by giving for each node the list of outarcs, i.e., of oriented outgoing arcs, and by giving for each arc the ending node and the associated transformation matrix, which transforms the coordinates of the ending node into the coordinates of the starting node. The construction of an explicit model in world coordinates—the coordinates of the root—of the object represented by a lattice is called *traversal* of the lattice. The traversal can be easily performed in linear time by using either a Depth-First-Search (DFS) or a Breadth-First-Search (BFS) algorithm (Aho et al., 1983). Note that it is preferable to use acyclic multigraph and not trees, since the objects which are multiply instantiated in several contexts are so stored in memory once, as in databases.

In PLASM, a sophisticated traversal strategy, formally described in Section 5, is assumed to always maintain the geometric validity of the representation. The polyhedral complex associated with any lattice node n_i is defined by the list of sub-complexes associated to the nodes connected to it by an outgoing arc. Each of such sub-complexes is transformed into the coordinates of n_i, and made valid by subtracting from it all the *evaluated* (traversed) sub-complexes which precede it in the list. The usefulness of such a strategy in architecture and building design applications is widely discussed in (Paoluzzi et al., 1992a). In order to implement such evaluation strategy with implicit differences between evaluated sub-lattices it is necessary to execute a BFS traversal of the lattice.

In general, the BFS algorithm requires storage of a current status corresponding to a queue of sub-lattices already traversed—we say "evaluated," i.e., transformed explicitly in a valid and possibly unconnected polyhedron—after having performed the implicit differences between the sub-components. Notice that in such traversal of an acyclic multigraph with implicit differences it may be more useful not to use a queue, but to store in every node the result of the traversal of their sub-lattices. Whenever the same node is reached by different arcs it becomes indeed unnecessary to replicate the Boolean computations which may be very time-consuming.

4 Representation of Geometry

In this section, the representation scheme (Requicha, 1980) of topology and geometry of design elements within the geometrical nucleus of the prototype PLASM interpret-

[2]Graph where multiple arcs between the same pair of nodes are allowed.

er is discussed. All geometrical objects—points, polyhedra, complexes, transformation matrices—are defined in a dimension-independent way. In particular, a polyhedron P is an object in the set $\wp^{d,n}$ of polyhedra with intrinsic dimension d and embedded in $\Re^n$. For example, a polygon is an element in $\wp^{2,2}$; the boundary of a polygon is in $\wp^{1,2}$, as it is a polyhedron in $\Re^2$, whose cells are polyhedra in $\Re^1$. A complete description of topology is only assumed for elementary (i.e., atomic) polyhedra, where a decomposition in convex cells is stored. In particular, we claim that to store structure rather than topology allows a uniform representation of manifold and non-manifold objects, which are stored as assemblies of pseudo-manifolds.

4.1 Convex Cell

Cell of dimension d or d-cell, is a convex set of $\Re^d$; such a set can be defined both as a convex combination of a set of convexly independent points, called *vertices*, or as an intersection of half-spaces defined by a non-redundant set of oriented hyperplanes which coincide with the affine subspaces supporting the cell *faces*. In the first case, the cell is represented by the set of vertices given in normalized homogeneous coordinates; in the second case it is represented by the set of normalized face covectors, i.e., by the normals to the affine subspaces supporting the cell faces.

4.2 Solid Polyhedron

Polyhedron of dimension d, or d-polyhedron, is the *union* of a collection of quasi-disjoint d-cells. The intersection of any pair of cells in such collection is either empty or is a cell of dimension less than d. With such a definition, a polyhedron may be non-convex and unconnected. A polyhedron is always locally "solid," in the sense that its intrinsic dimension and the dimension of the supporting affine subspace (*affine hull*) always coincide. For example, a polyhedron in $\wp^{2,3}$ is a set of convex 2-cells contained in a plane of $\Re^3$.

Only the topology within the affine hull, i.e., the (d-1) adjacencies of convex d-cells belonging to the same d-plane, is stored in the PLASM internal representation. Notice that this assumption is much weaker than what is usually assumed in solid modeling. For example, in the d-polyhedron obtained as the boundary of a d-polyhedron, there is no storing of face adjacencies. Notice also that, conversely, this choice is consistent with the common usage in graphics and rendering, where no adjacency between 3D polygons is usually maintained.

In PLASM, a solid polyhedron is stored by using either a vertex-based or a face-based representation. Either a vertex database or a face database is maintained, respectively. In both cases a cell database is also stored. Each vertex is represented by an unique name and by the vector of homogeneous normalized coordinates $[v_0\ v_1 \ldots\ v_d]$ where $v_0 = 1$. Each face covector is analogously represented by a normalized representation $f = [f^0\ f^1 \ldots f^d]$ where $\|f\| = 1$. For a point $[x_0\ x_1 \ldots\ x_d]^T$ internal to a cell $f_x < 0$, holds for any cell face covector f, and $f_x = 0$, for some f, for the only points on the cell boundary. A point belongs to a face of codimension k (or dimension d-k) if it belongs to $h_x \geq k$ boundary hyperplanes, k of which are affinely independent.

A pair (F,C), where F is the face covector database and C is the cell database, is called *face-based representation*. For each cell an unique name used as cell index, a list of pairs (covector index, sign)[3], a list of indices of (d-1)-adjacent cells, and the list of face covector pointers are specified. Each vertex is implicitly represented by the list of faces,

[3]A face covector has opposite orientation on the two cells (d-1)-adjacent along the face.

to which it belongs. The solution of the linear system of the incident face covectors gives an explicit representation of the vertex.

4.3 Polyhedral Complex and Instance

We call polyhedral complex the multigraph representation of a lattice, where every node is the root of a sub-lattice and every node of outdegree zero (i.e., a leave) is an atomic polyhedron. A formal definition follows: a *polyhedral complex* is a multigraph node, defined in a local coordinate system, and is represented by the ordered sequence of its outgoing arcs. Each arc, called *complex instance*, is represented by the second node and a transformation matrix. For the sake of simplicity, we denote a node by the list of its outgoing arcs, and an instance by the pair (matrix, node):

$$C_i = (c_{i1}, \ldots, c_{ih}, c_k), c_k = (D_k, M_k)$$

We call *dimension* of a polyhedral complex a pair *(d,n)* of integers with $d \le n$, where *d* is the intrinsic dimension of elementary polyhedra and *n* is the dimension of the embedding space, i.e., the number of coordinates of vertices after the "evaluation" of the complex. It is assumed that in a polyhedral complex all nodes have the same *intrinsic* dimension, i.e., that any sub-complex is homogeneously *d*-dimensional. Elementary polyhedra are always assumed solid, i.e., of full dimension *(d,d)*. Since (a) a complex may have dimension *(d,d)*, (b) the transformations represented by square matrices, and (c) the elementary complex element are full dimensional polyhedra, we assume an implicit embedding from $\Re^{dd}$ to $\Re^{dn}$ that must be applied before of transformation matrix. For example, the *0*-dimensional polyhedron $o^{(0,n)}$ (the origin of $\Re^n$) is represented as the object $o^{(0,n)}$, i.e., by the vector *[1]* implicitly embedded in $\Re^n$ to *[1 0 ... 0]*, to which is applied the identity matrix $I_{(n+1)\times(n\times 1)}$.

5 Partial and Complete Evaluation

In the following, initial capital Latin letters (*A,B,C*) are used for complexes, initial small latin letters (*a,b,c*) for instances, final smallLatin letters (*p,q,r, ..., x,y,z, ...*) for polyhedra and Greek capital letters ($\Phi, \Psi, \Theta, \ldots$) for transformation matrices. For elementary polyhedra the face-based representation is used.

The sets of polyhedra, complexes, and polyhedral instances will be denoted as *P*, *C*, and *I*, respectively. Hence $C = (c_1 \ldots c_k)$ will indicate a polyhedral complex defined by *k* instances of other complexes or polyhedra, with $c_i = (\Phi_i, D_i)$ the generic instance, where D_i is a complex; and Φ_i is the affine transformation matrix which transforms D_i in the local coordinates of *C*.

The DFS traversal of a complex, with relocation of every complex or polyhedron in the coordinates of the root, will be called *forward* or *partial evaluation*, and written as $*V: C \cap P \rightarrow P*$, where *P** denotes the set of sequences of elements. The BFS traversal, with complete evaluation of complexes and computation of implicit Boolean differences, will be called *backward* or *complete evaluation*, and defined as: $V*: C \cup P \rightarrow P*$.

In general the collection of polyhedra produced by an application of the function **V* is a covering, i.e., is a family of sets with non-empty pairwise intersection. The collection of polyhedra produced by an application of the function **V* is conversely a partition of the object represented and hence can be considered as an unique well-defined polyhedron, possibly unconnected. Formally, we can define the two evaluation functions as fol-

lows, with $C = (c_1 \ \ldots \ c_k)$ a polyhedral complex and $c_i = (\Phi_i, D_i)$ a polyhedral instance, respectively:

$$C \rightarrow V^*C = \Phi_1(V^*D_1) \cup \ldots \cup \Phi_h(V^*D_h),$$

where if $D_i = p \in P$ then $V^*D_i = D_i$. Note that the symbol $\cup$ represents the set union of the arguments (sets of polyhedra), and that when we apply a transformation matrix Φ to a polyhedron represented with face covectors we have:

$$\Phi p_i = (F_i, C_i)\Phi^{-1} = (F_i\Phi^{-1}, C_i) = ((f_{i_1}\Phi^{-1}, \ldots, f_{i_n}\Phi^{-1}), C_i).$$

In fact, the coordinate transformation applied to vectors by left multiplication with a matrix Φ^{-1} is applied to dual covectors by right multiplication with the matrix. Notice that when a matrix is applied to a set or a list, we mean it is applied to every element. Analogously, for the backward evaluation, we can write:

$$C \rightarrow^* VC = \Phi_1 p_1 \cup \Phi_2 p_2 \cup \ldots \cup \Phi_h p_h,$$

where $p_1 = {}^*VD_1$,

$$p_i = {}^*VD_i - \Phi_i^{-1}\Phi_1({}^*VD_1) \cdots - \Phi_i^{-1}\Phi_{i-1}({}^*VD_{i-1}), \quad (2 \le i \le h).$$

Also in this case, the nodes of outdegree zero are not evaluated: if $D_i = p \in P$ then ${}^*VD_i = D_i$.

Example 1. *Let us consider the complex $C = (c_1,c_2,c_3)$ defined by three instances of the unit square: $c_1 = (\Phi_1, p)$, $c2 = (\Phi_2, p)$, $c3 = (\Phi_3, p)$, with $p = [0,1] \times [0,1] \subset \Re^2$ and $\Phi_1 = I_{3x3}$ and identity matrix. As we show in Figure 2, V*C is a collection of sets with a possibly non-empty pairwise intersection. Conversely, any intersection between pairs of unconnected components in *VC is empty.*

In other words, a compact representation of a polyhedral complex as a multigraph is always maintained in the geometrical nucleus of the language interpreter. A forward evaluation gives an explicit polyhedral model where the components are multiply instantiated in their context, but without the execution of progressive differences. A backward evaluation produces a complete and valid solid model, as an object partition with many adjacencies, usually unmeaning full-non-calculated. Depending on the huge quantity of computation needed to perform a complete evaluation of the model of a very complicated object (like, for example, a building), our belief is that it is useful to permanently store on

disk “many” evaluated sub-lattices. We are currently working to design intelligent decision criteria for this point.

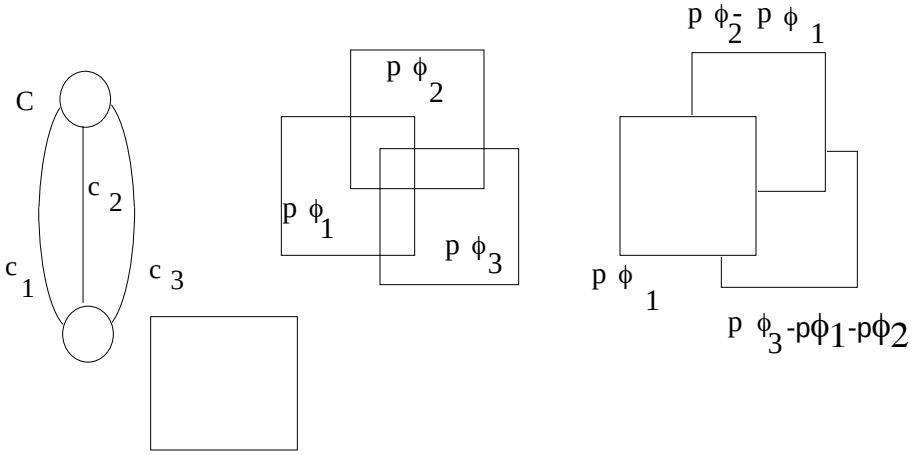

Figure 2. Partial and total evaluation of a complex made by three instances of a square.

6 Some Language Concepts

In this section, we show which geometric objects are considered primitive, indicate some predefined functions, and discuss the construction and maintenance of the design database. The current version of the syntax for the definition of functions is also given. The atoms of PLASM are the same as FL (Backus et al., 1989) with two new objects: *points* and *polyhedra*.

Points of dimension *d* result from the application of the function `Mkpt` on sequences of length of *d* numbers. APredefined constants are the *d*-dimensional zeros, *d = 0,1, ...*, expressed as `Zero{d}`. A predefined logical function on points is `IsPoint{d}`.

Polyhedra of dimension *(0,d)* result from the application of the function `Mkpol` on sequences of points of dimension *d*. A *d*-dimensional polyhedron with *m* vertices is internally represented as a complex with two nodes and arcs: the ending node of all arcs is the *0*-dimensional zero $\boldsymbol{o}^{(0,n)}$, represented in homogeneous coordinates as $_{[1]}$; at each arc is associated a $(d+1) \times 1$ matrix which transforms $\boldsymbol{o}^{(0,n)}$ in a *d*-vertex. Every polyhedron is internally represented as a complex, i.e., as a lattice with at least one node. Each lattice node with outdegree zero is represented as a pair (F,C) of sets of faces and cells.

Integer predefined functions on polyhedra are `Dim` and `Rn`, which give the intrinsic dimension and the dimension of the embedding $\Re^n$ space, i.e., the number of coordinates of vertices, respectively. The logical function on polyhedra is `IsPol{d,n}` that returns `TRUE` when applied to a polyhedron if its dimension is *(d,n)*; it returns `FALSE` if the argument is not a polyhedron or if the dimension is different.

A geometric predefined function on polyhedra is `Join`. It is evaluated either on a single polyhedron, also non-convex, or on sequences of polyhedra. It returns a single polyhedron, computed as the convex hull of the vertices in the argument. If the argument polyhedra are in different spaces, they are embedded in a suitable subspace of the space of maximum dimension, where is also given the result of the function application. Another

important function is `Struct` (see Paoluzzi et al., 1992) which is defined on sequences of affine transformations and polyhedra. Sequences of this type are called Plasm structure. The application of the `Struct` function to a PLASM structure returns a suitable complex.

Most of the PLASM syntax is identical to that of FL. The differences concern the meaning of a few characters, the PLASM use of a specification listthat is different from the list of formal parameters of a function. In both languages, function application is denoted by the character ":." The character@ is used in FL for composition, and in PLASM for skeleton. Composition is denoted in PLASM by the character "~." For example, we have `f~g:x = f:(g:x).`

A definition is a special form. When interpreted it introduces a new operator in the user functional environment. A definition may contain a list of formal parameters. When an operator is applied on a sequence of actual parameters, often obtained by evaluating other language expressions, the PLASM interpreter produces a geometric model. When the user defined operator returns a structured or elementary polyhedron it works as a "shape prototype." A definition contains:

- the keyword "Def" followed by the function identifier;
- the optional list of specification parameters, enclosed between "{"and "}" brackets, to give the user a further control over the behavior of the function;
- the optional list of formal parameters, enclosed between "("and ")" brackets, with type specification;
- an equal symbol followed by a PLASM expression. This expression is the body of the function and allows one to compute its value. The body will be called in the following "generating form;";
- an optional set of local functions between the reserved words "Where" and "End", used both for readability and for efficiency reasons in the design review steps, as discussed in (Paoluzzi et al., 1992a).
- an optional set of default value declarations for both the specification and the formal parameters, enclosed between the reserved words "Default" and "End."

PLASM operators perform a suitable composition of objects and functions. Names of functions binded to geometric models are internally represented as an acyclic multigraph, i.e., as a complex. The binding Name ↔ Complex is driven by the syntactic form definition, when the result of the body is polyhedrally typed. The use of a name within another expression is a request for generation or use of the complex associated to the generating form of the name. In particular, a name can be in other function bodies. At evaluation time the interpreter will search for the associated complex and will return it. If the value is not ready, the generating form will be calculated and the result will be stored. Storing the values will avoid unnecessary future evaluation.

7 Examples

In this section we discuss two examples of PLASM scripts. The examples are given both to show some preliminary testing of our prototype interpreter and to outline how a PLASM script is a model of the model structure. In order to introduce the examples, we give the operational semantic of some PLASM predefined functions. In this context our belief is that this is the best way to proceed. Notice that, between the keywords `WHERE END`, one can define all kinds of objects, i.e., both polyhedra and functions.

The `Struct` function takes as arguments a sequence of polyhedra, affine transformations, and applications of the `Struct` function itself. It returns a polyhedron. If `Pol1` and `Pol2` are polyhedra of $\Re^2$ then

```
STRUCT:<Pol1, STRUCT:<T{2}:4.5, Pol1>,
        T{1}:2, Pol1, S{1}:3, Pol2>
```

evaluating the expression the polyhedral complex *C* shown in Figure 3 is returned.

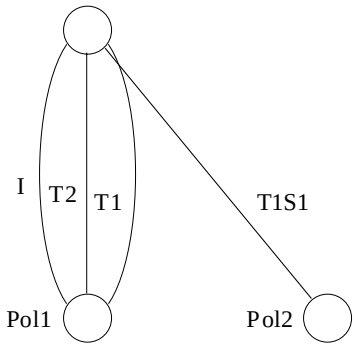

Figure 3: The complex (lattice) returned by the expression **`STRUCT:<Pol1, STRUCT:<T{2}:4.5, Pol1>, T{1}:2, Pol1, S{1}:3, Pol2>`**

The **`CUBOID`** function takes a positive real sequence as input and returns an *n*-cuboid of $\Re^n$, where *n* is the length of the sequence, such that the edge on the *i*-th axis has size s_i, where s_i is the *i*-th element of the sequence. For example, the value of the expression **`CUBOID:<1,1,1>`** is the standard unit cube of $\Re^3$.

The **`INSL`** function is a combining form which applies a binary function to a sequence of arguments. It works on the expression using the following rewriting rules:

```
INSL:DIV:<1, 2, 3, 4>
= INSL:DIV:<DIV:<1, 2>, 3, 4>
= INSL:DIV:<1/2, 3, 4>
= INSL:DIV:<DIV:<1/2, 3>, 4>
= INSL:DIV:<1/6, 4>
= DIV:<1/6, 4>
= 1/24
```

The function **`##`** is applied to a sequence of two parameters, where the first element is a non-negative integer to be intended as a repetition factor, and the second element is any expression resulting in a sequence. The result is a sequence, too:

```
##:<3, <1, 2, 3>> = <1, 2, 3, 1, 2, 3, 1, 2, 3>
```

Note that a new sequence is produced, where one can recognize the second argument repeated three times.

7.1 Spiral Stair

We want to show how to define a PLASM function to be used as the generating form of spiral stairs with a central column. What we need are only some predefined PLASM functions.

```
DEF spiral_stair (width, length, height,            % step dims %
   stairwell::ISREALPOS;                           % void radius %
   n_step::ISINTPOS)                               % step number %

= STRUCT:<stair_model,                            % spiral stair %
   CYLINDER:<                                    % internal column %
   stairwell, (height * n_step/2) + safe_column,10>

WHERE
safe_column = 5 * height,                    % makes a safe stair %
quote_all = AA:(QUOTE ~ LIST),               % a function to apply
                  'quote ~ list'to each element of a sequence %
single_step                                        % single step %
   = INSL:*:(quote_all:<length,width,height>),

my_step         % the step translated to give space to the void %
   = STRUCT:<T{1,2}:<stairwell,(-:width/2)>,single_step>,
   step_angle = 2 * ATAN:<width/2, length + stairwell>,
             % angle that provides the right alignment of steps %

stair_model =
   STRUCT ~
        CAT:<<my_step>,                      % the first step... %
             ##:<(n_step - 1),
                 <T{3}:height, R{<1,2>}:step_angle, my_step>>
   >
                            % ... the other steps translated in z
                                   and rotated in the x-y plane %
END

DEFAULT
   stairwell = 2,
   n_step    = 20,
   width     = 10,
   length    = 20,
   height    = 5
END;
```

7.2 Brick Walls

This is an example where a simple function to generate brick walls is obtained. We make extensive use of the functions POWER and QUOTE for the power of 1-dimensional complexes of polyhedra and for obtaining a 1D complex starting from a sequence of numbers, respectively (see Paoluzzi et al., 1992a).

```
DEF wall (width, length, height,             % brick dimensions %
   interstice::IsRealPos;                   % inter-brick space %
   wall_width,                                 % wall dimension %
   num_rows::IsIntPos)                       % brick row number %
```

```
= STRUCT:<
   half_wall,                                          % odd rows %
   T{1,3}:<width/2, height + interstice>,
   half_wall >                                        % even rows %

WHERE
                <% 'num_bricks' bricks in each row of the wall %
   num_bricks = INTMIN:(wall_width/(width+interstice)),
x_pattern = QUOTE ~ ##:<num_bricks, <width, -:interstice> >,
y_pattern = QUOTE:<length>,
z_pattern = QUOTE ~ ##:<num_rows, <height, -:(height +
                                              (2*interstice))>>,
   half_wall = INSL:*:<x_pattern, y_pattern, z_pattern>
                              % product of the x,y,z patterns %

END

DEFAULT
   width   = 20,                              % brick dimensions %
   length   = 10,
   height   = 10,
   interstice = 0.6,                          % brick interstice %
   wall_width = 100,                       % brick number per row %
   num_rows = 7                            % number of brick rows %
END;
```

Three examples of function calls with different parameter values follow. The solid model corresponding to the evaluated functions are also displayed in Figure 5.

```
                        % walls with different parameter values %
DEF wall1 = wall:<20, 10, 10, 1, 120, 5>;
DEF wall2 = wall:<20, df, 10, 0.7, 99, 5>;
DEF wall3 = wall:<20, 20, 10, 0.5, 70, 5>;
```

8 Conclusions

The main difference of the geometrical nucleus of the language with respect to the current solid modeling technology was produced by the decision to store as little topology as possible. We noted, in fact, that the (very simple) adjacency data structures used in our previous prototype modelers (Paoluzzi et al., 1989, Paoluzzi et al., 1993) are no used when the generated model is rendered by some graphics server, and even when the model is used to compute the integral properties of the object. Conversely, following the old good CSG approach (Requicha, 1980), we decided to associate with any model an acyclic multigraph for storing the object structure, i.e., the operations and sub-components which generated it. We believe that this may be much more interesting for the designer than to have the complete storing of adjacencies in a non-manifold subset of the boundary. In few words,

with respect to the representation of topology, geometry and structure, our language can be seen exactly mid-way between solid modeling and contemporary standard graphics.

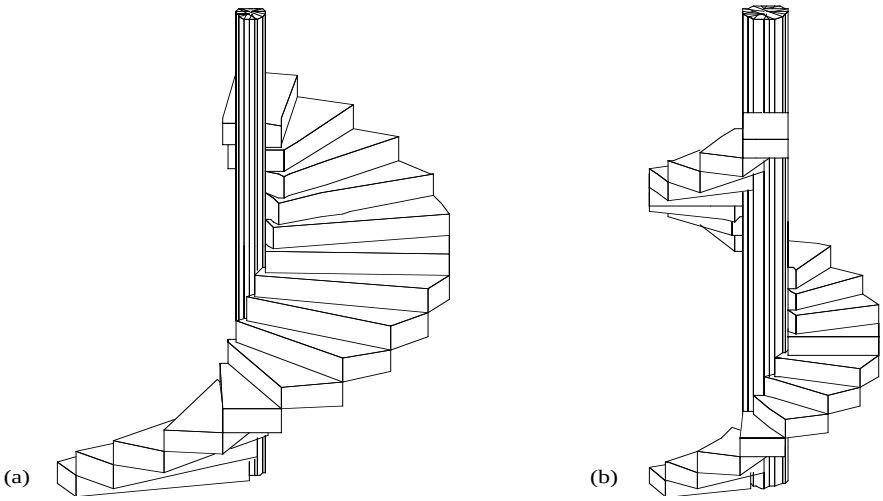

Figure 4. The value of the application SPIRAL_STAIR:<10, 20, 5, 5, df> (a); and the value of the application SPIRAL_STAIR:<20, 30, 4, 5, 15> (b).

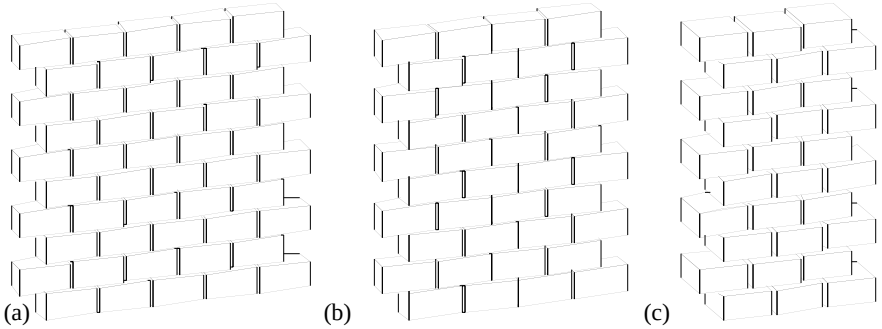

Figure 5. DEF wall1 = wall:<20,10,10,1,120, 5> (a); DEF wall2 = wall:<20, df,10, 0.7, 99, 5> (b); and DEF wall3 = wall:<20, 20,10, 0.5, 70, 5> (c).

References

Aho, A.V., Hopcroft, J.E. and Ullman, J.U., 1983. *Data Structures and Algorithms*. Reading, MA: Addison Wesley.

Alexander, C., 1965. "The Town is Not a Tree," *The Architectural Forum*, April-May.

Backus, J., 1978. "Can Programming Be Liberated from the Von Neuman's Style? A Func-

tional Style and its Algebra of Programs," (ACM Turing Award Lecture). *Communications of the ACM*, 21(8), pp. 613-641.

Backus, J., Williams, J.H., Wimmers, E.L., 1988. "An Introduction to the Programming Language FL," in D.A. Turner (ed.), *Research Topics in Functional Programming*.

Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., and Aiken, A., 1989. *FL Language Manual, Parts 1 and 2*. IBM Research Report RJ 7100 (67163).

Barford, L.A., 1986. "Representing Generic Solid Models by Constraints," *Technical Report 86-80*1. Department of Computer Science, Cornell University, Ithaca, NY, December 1986.

Bernardini, F., Ferrucci, V., Paoluzzi, A. and Pascucci, V., 1993. "Product Operator on Cell Complexes," to appear on Second ACM / IEEE Symposium on Solid Modeling and Applications, Montreal, Canada, May 1993.

Borning, A., 1981. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Applications* 3(2) (March, 1981), pp. 357-387.

Bruderlin B., 1985. "Using Prolog for Constructing Geometric Objects Defined by Constraints," *Proceedings*, EuroCal 85, North-Holland.

Gossard, D.C., Lin, V., 1983. "Representation of Part Families through Variational Geometry," in T. M. R. Ellis and O. I. Semenkov (eds.), *Advances in CAD/CAM*, Amsterdam: North-Holland, 1983.

Hoffmann, C. M., 1992. "Modeling the DARPA Diesel Engine in ProEngineer," *Technical Report CSD-TR-92.025*, Deptartment of Computer Sciences, Purdue University, April 1992.

Hoffmann, C.M., Juan, R., 1992. "EREP An Editable High-Level Representation for Geometric Design and Analysis," *Technical Report CSD-TR-92.055*, Department. of Computer Sciences, Purdue University, August 1992.

Howard, T.L.J., Hewitt, W.T., Hubbold, R.J. and Wyrwas, K.M., 1991. *A Practical Introduction to PHIGS and PHIGS PLUS*. Reading, MA: Addison Wesley.

Leler, W.M., 1988. *Constraint Programming Languages*. Reading, MA: Addison Wesley.

Light, R., Gossard, D., 1982. "Modification of Geometric Models Through Variational Geometry," *Computer Aided Design* 14(4), pp. 209-214.

Lucas P., Zilles, S.N., 1988. "Applicative Graphics Using Abstract Data Types," *IBM Research Report RJ 6198*.

Paoluzzi A.,Ramella, M. and Santarelli, A., 1989. "Boolean Algebra Over Linear Polyhedra," *Computer Aided Design* 21(8), pp. 474-484.

Paoluzzi, A., Sansoni., C., 1992. "Programming Language for Solid Variational Geometry," *Computer Aided Design* 24(7), pp. 349-366.

Paoluzzi, A., Pascucci, V. and Vicentino, M., 1992. "More Structure and Less Topology. A Linguistic Approach to Geometric Design," submitted Dip. di Informatica, Univ. di Roma "La Sapienza", November 1992.

Paoluzzi, A., Bernardini, F., Cattani, C. and Ferrucci, V., 1993. "Dimension-Independent Modeling with Simplicial Complexes, "*ACM Transactions on Graphic*s (to appear January or February, 1993).

Requicha, A.A.G., 1980. "Representations for Rigid Solids: Theory, Methods and Systems." *ACM Computing Surveys* 12(4), pp. 437-464.

Rossignac, J.R. and Requicha, A.G., 1991. "Constructive Non-Regularized Geometry." *Computer Aided Design* 23(1), pp. 8-19.

Steele, G. and Sussman, J., 1980. "Constrains: A Language for Expressing Almost Hierarchical Descriptions", *Artificial Intelligence* 31(1), pp. 1-40.

Sutherland, 1963. "SKETCHPAD: A Man-machine Graphical Communication System," *Proceedings*, IFIPS Spring Joint Computer Conference.

Williams, J.H., 1982. "Notes on the FP Style of Functional Programming", in Darlington, J. P. Henderson, and D.A. Turner (eds.), *Functional Programming and its Applications*, Cambridge: Cambridge University Press.

Williams, J.H. and Wimmers, E.L., 1991. "An Optimizing Compiler Based on Program Transformation." Internal IBM Report, IBM Almaden Research Center, March, 1991.

Zilles, S.N., Lucas, P., Linden, T.M., Lotspiech, J.B. and Harbury, A.R., 1988. "The Escher Document Imaging Model," *Proceedings*, ACM Conference on Document Processing Systems, Santa Fe, NM.