PLaSM

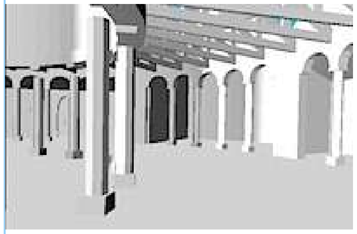
functional language for computing with geometry

NEW version !!

The PLaSM versions 5.x are based on the *new* progressive geometric kernel provided by the XGE (eXtreme Geometric Environment) library by Giorgio Scorzelli, and rely on extensive use of progressive BSPs and complete topology representation by multidimensional Hasse diagrams

S.Stefano Rotondo

The geometric model of the church of S. Stefano Rotondo in Rome is a metamodel of the heavenly Jerusalem.



Introduction to FL and PLaSM

Table of Contents

The design language PLaSM is a geometry-oriented extension of a subset of the FL language.

FL Language

FL (programming at Function Level) is an advanced language for functional programming developed by the Functional Programming Group of IBM Research Division at Almaden (USA) [BWW90,BWWLA89]. The FL language, on the line of the Backus' Turing lecture [Backus78] introduces an algebra over programs, i.e.~a set of algebraic identities between functional expressions, for formally reasoning about programs. This language enjoies some interesting features. In particular, programs are easily combined, so that new programs are obtained in a simple and elegant way; and one may find simpler equivalent programs, both at design and at compilation times. Great advantages are so obtained in style and efficiency of program prototyping.

PLaSM Language

PLaSM, (the Programming LAnguage for Solid Modeling) is a *design language*, developed by the CAD Group at the Universities of Roma *La Sapienza* and *Roma Tre* [PS92, PPV95]. The language is strongly inFLuenced by FL. With few sintactical differences, it can be considered a geometric extension of a FL subset. Among the strong points of PLaSM we cite:

- **the functional approach, which allows to compute with geometries as well with numbers and** functions, and its

- the dimension-independent implementation of geometric data structures and algorithms.

The first feature results in a very natural and powerful approach to parametric geometry. The second one, coupled with the "combinatorial engine" of the FL language, gives an amazing descriptive power when computing with geometry.

Functional Programming and Geometric Modeling

Functional programming enjoies several good properties:

- The set of rules is very small
- Each rule is very simple
- Program code is concise and clear
- The meaning of a program is well understood (no state)
- Functions both as programs and as data
- Programs connected by concatenation and nesting.

Also, a complex shape is an assembly of components, highly dependent from each other.

- each part results from computations involving other parts
- where a generating function is associated to each part
- and where geometric expressions appear as actual parameters

Resulting in a natural environment for geometric computations.

A brief outline of the FL approach to functional programming is given in the following with an introduction to the design language PLaSM. This language can evaluate polyhedral expressions, i.e. expressions whose value is a polyhedral complex. It is also able to combine functions to produce higher-level functions in the FL style, so that it can be roughly considered a geometry-oriented extension of FL.

Elements of FL Syntax

FL is a pure functional language based on combinatorial logic. The language introduces an algebra over programs (a set of algebraic identities between functional expressions) for reasoning formally about programs, so that

one may find simpler equivalent programs (both at design and at compile times); programs are easily combined, so that new programs are obtained in a simple and elegant way; great advantages in style and efficiency of program development are achieved.

Primitive objects

characters, numbers and truth values

Expressions

primitive objects, functions, applications and sequences

Sequences

expressions separated by commas and contained within a pair of angle brackets: <5, fun>

The application expression exp1:exp2

applies the function resulting from the evaluation of exp1 on the argument resulting from the evaluation of exp2. Binary functions can also be used in infix form: `:<1,3> = 1` 3 = 4

- Application associates to left: f:g:h = (f:g):h
- Application binds stronger than composition: f:g h = (f:g) h

FL combining forms and functions

The construction combining form CONS
allows to apply a sequence of functions to an argument producing the sequence of applications:

`CONS:< f1,...,fn >:x = [f1,...,fn]:x = < f1:x,...,fn:x >`
E.g. CONS:<`,->, written also [`,-], when applied to the argument <3,2> gives the sequence of applications: `[,-]:<3,2> = <`:<3,2>,-:<3,2>> = <5,1>;

The apply-to-all combining form AA
applies a function to a sequence of arguments giving a sequence of applications:

```
AA:f:< x1,...,xn > = < f:x1,...,f:xn >;
```
The identity function ID returns its argument unchanged:

```
ID:x = x;
```
The constant function K is evaluated, for whatever x2, as:

```
K:x1:x2 = x1;
```
Binary composition ~ of functions is defined as

```
(f ~ g):x = f:(g:x);
```
N-ary composition COMP of functions is also allowed:

```
COMP:< f,h,g >:x = (f ~ h ~ g):x = f:(h:(g:x));
```
The conditional form IF:< p,f,g >:x is evaluated as follows:

```
IF:< p,f,g >:x = f:x ,  if p:x = TRUE
                 g:x ,  if p:x = FALSE
```
The insert right (INSR) and insert left (INSL) allow to apply a binary function on a sequence of arguments of any length:

```
INSR:f:< x1,x2,...,xn > = f:< x1, INSR:f:< x2,...,xn > >
INSL:f:< x1,...,xn-1,xn > = f:< INSL:f:< x1,...,xn-1 >, xn >,
```
where `INSR:f:< x > = INSL:f:< x > = x`

The catenate function CAT appends any number of input sequences creating a single output sequence:

```
CAT:<< a,b,c >,< d,e >,...,< x,y,z >> = < a,b,c,d,e,...,x,y,z >
```
The distribute right (DISTR) and distribute left (DISTL) generate a sequence of pairs:

```
DISTR:<< a,b,c >, x> = << a,x >, < b,x >, < c,x >>
DISTL:< x,< a,b,c >> = << x,a >, < x,b >, < x,c >>
```
Notice finally that application associates to left, i.e.:

```
f:g:h = (f:g):h
```
and that application binds stronger than composition:

```
f:g~h = (f:g)~h
```

PLaSM Approach to Geometry

PLaSM, the Programming LAnguage for Symbolic Modeling, is a functional "design language" designed in recent years by the CAD Group at the University of Rome "La Sapienza" (Italy) and currently developed by the Geometric Computing Group at the University of Roma Tre. The PLaSM language was introduced in PS92 and later described in PPV95 in 1992 and 1995, respectively.

PLaSM features

Programming approach to "generative" modeling Geometrical objects generated by language expressions Algebraic calculus over embedded polyhedra Dim-independent approach to geometry repr and algorithms Extended variational geometry (classes of objects with varying topology and shape) Geometry-oriented extension of the functional language FL Validity of geometry syntactically guaranteed Repr. domain broader than in solid modelers (points, wire-frames, surfaces, solids, etc.) PLaSM is a geometry-oriented extension of a subset of FL:

is a functional "design language" strongly influenced by FL can evaluate expressions whose value is a polyhedral complex can produce higher-level functions in the FL style no free nesting of scopes and environments is allowed no pattern matching is provided allows identifiers for any language object The syntax of PLaSM is very similar to that one of FL. The differences only concern the meaning of few characters and the use of multiple lists of formal parameters of functions. Two new combining forms AC (apply-in-composition) and AS (apply-in-sequence) are introduced, in order to use a function with a number of actual parameters greater than the number used in the function definition.

Example

SEL is a primitive operator with a specification parameter i, such that when applied to a sequence the i-th sequence element is returned:

```
SEL:2:<13, 4.5, ID> = 4.5
```

By using the combining form AC and AS, the SEL operator can work in two different ways:

```
AC:SEL:< 3,1 >:<< 1,3,8,7 >, 89, fun>
= (SEL:3 ~ SEL:1):<< 1,3,8,7 >, 89, fun>
= 8
AS:SEL:< 3,1 >:<< 1,3,8,7 >, 89, fun>
= [SEL:3, SEL:1]:<< 1,3,8,7 >, 89, fun>
= < fun, < 1,3,8,7 >>
```

PLaSM uses a case unsensitive alphabet; Allows for overloading of operators:

```
3+2; 3*2
```

ops on numbers

```
(sin+cos):x; (sin*cos):x;
```

ops on functions

```
pol1+pol2; pol1*pol2;
```

ops on polyhedra

Allows for partially specified (curried) functions.

Predefined Generic functions

Some non geometric predefinite functions and combinatorial forms follow

Type predicates
`IsInt, IsReal, IsSeq, IsFun, IsPol, ...`
Comparison functions
`GT, GE, LT, LE, EQ, ...`
Logical functions and constants
`AND, OR, NOT, TRUE, <>, ...`
Combining Forms
`AS, AC, ...`
Mathematics functions
`SIN, COS, TAN, LN, EXP, ...`
Sequence functions
`AL, AR, FIRST, TAIL, CAT, ...`
Repetitition operator
allows for instancing n times any expression:

```
#:3:expr = < expr,expr,expr >
```

Catenation of repeated sequences
repeats and catenates sequences:

```
##:3:< a,b,c > = < a,b,c, a,b,c, a,b,c >
```

FROMTO operator
generates integer sequences between two given extremes:

```
FromTo:<2,5> = 2..5 = <2,3,4,5>
```

INTSTO operator
generates integer sequences with a given end:

```
INTSTO:5 = <1,2,3,4,5>
```

Function definition

Two kinds of functions are recognized in PLaSM:

global (or top-level) functions

Global functions may contain a definition of local functions between WHERE and END keywords. Global functions are allowed to contain formal parameters using a lambda-style. Formal parameters are specified together with a predicate wich is normally used to perform some type-checking at run-time.

local functions

The visibility of local functions is restricted to the scope of a global function.

Formal and actual parameters are given as follows:

Function definition

```
DEF f (a::type1) = bodyF
DEF g (a1,...,an::type2) = bodyG
DEF h (a1::type1)(a2::type2) = bodyH
DEF l (a1::type1; a2::type2) = bodyL
```

Function instancing

```
f:x
g:< x1, ... , xn >
h:x1:x2
l:< x1, x2 >
```

Geometric functions

Elementary shape contructors

i.e. geometric primitives

```
SIMPLEX, CUBOID, CYLINDER, ...
```

The SIMPLEX primitive

generates simplices of different intrinsic dimension:

```
SIMPLEX:5 = segment of length 5;
SIMPLEX:<1,1> = standard triangle of area 1/2!;
SIMPLEX:<1,a,1> = standard tetrahedron of volume a/3!;
SIMPLEX:<1,a,b,1> = standard 4-simplex of volume ab/4!;
```

The CUBOID primitive

generates intervals of different intrinsic dimension:

```
CUBOID:5 = segment of length 5;
CUBOID:<5,10> = rectangle of area 5×10;
CUBOID:<5,10,5> = parallelepiped of volume 5×10×5;
CUBOID:<1,1,1,1> = hypercuboid of volume 1;
```

The "Make Polyhedron" constructor MKPOL

generates polyhedral complexes of different dimension. It is the only basic geometry constructor in PLaSM. I t is a mapping from triples of sequences to polyhedral complexes:

```
MKPOL:< verts, cells, pols >
```

where

- `verts` is a sequence of points;
- `cells` is a sequence of convex cells (given as indices of points);
- `pols` is a sequence of polyhedra (given as indices of cells).

Any cell is defined as the convex hull of its vertices, any polyhedron is defined as the union set of its convex cells. This definition is quite general, and may include (complexes of) polylines, plane and space polygons, 3D polyhedra and higher dimensional geometric objects, both solid and embedded.

Example: Set of convex cells

The Example 1 gives the PLaSM code of a 2D example with three convex cells embedded in E3.

```
DEF MyPol = MKPOL:
WHERE
  verts = <<0,0,0>,<10,0,0>,<10,10,0>,<0,10,0>,
           <0,0,10>,<10,0,10>,<10,10,10>,<0,10,10>>,
  cells = <<1,2,3,4>,<1,2,5,6>,<3,4,7,8>>,
  pols  = <<1,2,3,4>>
END;
MyPol;
```

Example: Object with a hole

The Example 2 is slightly modified to give a similar object with a hole.

```
DEF MyPol = MKPOL:
WHERE
  verts = <<0,0,0>,<10,0,0>,<10,10,0>,<0,10,0>,
           <0,0,10>,<10,0,10>,<10,10,10>,<0,10,10>,
           <6,10,0>,<4,10,0>,<6,10,7>,<4,10,7>>,
  cells = <<1,2,3,4>,<1,2,5,6>,
           <3,9,11,7>,<11,12,8,7>,<10,4,8,12>>,
  pols  = <(1),(2),<3,4,5>>
END;
MyPol;
```

Affine transformations

```
T, S, R, H, MAT, ...
```

PLaSM gives some (dimension-independent) affine operators, including *translation* `T:coords:parmtrs`, *rotation* `R:coords:parmtrs`, and *scaling* `S:coords:parmtrs`, where coords denotes the coordinates affected by the transformation, and parmtrs denotes its parameters. Affine transformations can be composed and applied to polyhedra. In such a case they are equivalent to the application of a `STRUCT` operator.

For efficiency reasons, a special form for parameter specification has been implemented for some frequently used elementary geometric function. In particular, the functions MIN, MAX, SIZE, BOX and the affine transformation functions T, S, R, H can be specified both on a single value and on a sequence of values. E.g., T denotes translation, and T:3:2.5 means translation of 2.5 units on the third coordinate. Hence for a translation on more than one coordinate we can write, e.g., T:<1,3>:<2.5,6/1.5>.

Several equivalences hold for affine transformations. For instance:

```
T:<1,3>:<2.5,6/1.5> = (T:1:2.5 ~ T:3:(6/1.5))
```

The "Hierarchical structure" constructor STRUCT

```
STRUCT
```

is used to generate hierarchical assemblies of objects defined in local coordinates. It is applied to sequences of polyhedra, affine transformations and STRUCT invokations. This operator has a semantics very close to that of the structures defined in the ISO PHIGS graphics standard. At traversal time (at evaluation time, in our case) each occurrence of an object in a hierarchical network of structures is transformed from local coordinates to the coordinates of the root of the network.

Example: Linear assembly
A simple Example 3 is given, where a sequence of occurrences of the polyhedron generated in Example 2 is composed to give a linear assembly. An equivalent and more proper use of PLaSM is there shown, where the repetition operator ## is used and composed with the STRUCT operator.

```
STRUCT:< Mypol,T:1:10, Mypol,T:1:10, Mypol,T:1:10, Mypol,T:1:10, Myp
```
or, equivalently:

```
(STRUCT ~ ##:5):< Mypol,T:1:10 >;
```
Example: Array assembly of geometric objects
The structures can be nested hierarchically. So, a sort of array assembly of geometric objects, shown in Example 4, is very easily generated as:

```
(STRUCT ~ ##:3):< (STRUCT ~ ##:5):< Mypol,T:1:10 >, T:3:10 >
```
The Boundary operator

is a mapping:

@`: P_{d,n} \rightarrow P_{d-1,n}`

from the set `P_{d,n}` of polyhedra of dimension `(d,n)` to that of polyhedra of dimension `(d-1,n)`. E.g., the boundary of a (3,3)-polyhedron is a (2,3)-polyhedron (a set of polygons embeded in 3D space). The boundary of a (2,2)-polyhedron is a (1,2)-polyhedron, i.e. a set of line segments in 2D. At current time it is not yet implemented in PLaSM. It should be implemented into the next release of the language, using an algorithmic approach based on boundary-BSP trees [BC97].

The Skeleton operators

We call r-skeleton the mapping

@r`: P_{d,n} \rightarrow P_{r,n}, 0 <= r <= d`,

such that `@r:P` is the r-skeleton of the polyhedral complex `P`. Notice that `@0:P` gives the set of `P` vertices; `@1:P` gives the set of vertices and edges; `@2:P` gives the set of vertices, edges and faces. When exporting a polyhedral complex from the language environment, the 2-skeleton is implicitly extracted if the object dimension is greater than 2.

Example: Skeleton combined with product
Two simple examples of use of the skeleton extractors combined with the product operators are given in Example 7.

```
DEF OneSk = (@1 ~ CUBOID):<1,1>;
DEF Object1 = (OneSk * CUBOID:(1));
DEF ZeroSk = (@0 ~ CUBOID):(1);
DEF Object2 = CUBOID:<1,1> * ZeroSk;
Object1 LEFT Object2;
```

The QUOTE operator: constructor of 1D polyhedra

`QUOTE`
is used in PLaSM to define 1-polyhedra embedded in 1D space. Such (1,1)-polyhedra are often used by other PLaSM functions. The range of the function is the set of sequences of non-zero reals:

`QUOTE`\`: (\Re - \{\emptyset\})^* \rightarrow P_{1,1}\`

Negative elements in the sequence are used to denote empty intervals in the complex.

Example: Parameterized 2D building fronts
A function to generate fully parameterized 2D building fronts (according to a given partition of the front with full and empty spaces) by using the QUOTE operator is given in Example 8.

```
DEF XCAT = CAT ~ AA:(IF:< NOT~IsSeq,LIST,ID >);
DEF BuildingFront (a,b,c::IsRealPos)(n,m::IsIntPos) = STRUCT:<
  QUOTE:Xfull  * QUOTE:Yfull,
  QUOTE:Xempty * QUOTE:Yfull,
  QUOTE:Xfull  * QUOTE:Yempty
>
WHERE
  Xfull = XCAT:<##:n:< c,-:a >,c>,
  Yfull = XCAT:<-:a,b,##:m:< -:a,c >,b>,
  Xempty = AA:-:Xfull,
  Yempty = AA:-:Yfull
END;
BuildingFront:<1,0.5,0.2>:<5,3> * QUOTE:<0.2>;
```

A simpler example is the following:

```
@1:(QUOTE:<2,5,2, -5, 2,5,2> * QUOTE:<5,-2,5>)
```

Cartesian Product of complexes

`*`

The Product Operator

is defined as a mapping from pairs of polyhedra to polyhedra:

*`: P_(d_1,n_1) \times P_{d_2,n_2} \rightarrow P_{d_1+d_2,n_1+n_2}`

where the cells in the polyhedral output are generated by pairwise cartesian products of the cells in the polyhedral input pair. Algorithms for dimension-independent "generalized" product of both polyhedra and polyhedral complexes are given on References [BFPP93] and [PFP96], respectively.

Such a product allows as special cases

- the Cartesian product * of polyhedral complexes;
- the intersection of extrusions, denoted as &&`: <perm_1, perm_2>`, where `perm_1`, `perm_2` are two permutation of indices used to specify how to embed the lower dimensional arguments into the coordinate subspaces of the target space;
- the standard intersection of polyhedral complexes.

Intersection of extrusions

`&&`

Example: Building model
Let consider the Example 9, which evaluates in this model (here displayed as a wire-frame).

```
DEF plan = STRUCT:< QUOTE:<4,-2,4> * QUOTE:<4,4>,
                    T:<1,2>:<4,1>, QUOTE:(2) * QUOTE:<2,4,2> >;
DEF section = MKPOL:
  <<<0,0>,<10,0>,<10,3>,<5,4.5>,<0,3>>,<1..5>,<(1)>>;
plan (<1,2,0> && <1,0,2>) section
```

Parametric mapping MAP

`MAP:VectorFunction:Domain`

The "Parametric Mapping" constructor MAP allows for simplicial mapping of polyhedral domains. The syntax is:

`MAP:vfun:domain`

where `vfun` is a vector function (written using the FL selectors) and domain is a polyhedral complex. The semantics is very simple: `MAP` applies vfun to all vertices of a simplicial decomposition of the polyhedral cells of `domain`. Usually `vfun` is the `CONS` of a sequence of coordinate functions which are applied to the vertices of the simplicial decomposition to generate their images in target space `E^d`. Notice that the dimension `d` of such space will equate the number `d` of coordinate functions in the CONSed `vfun`.

Example: Graph of the sin function
A piecewise linear approximation with 32 segments of the graph of the `sin` function in the interval `[0, 2\pi]`

(where `\pi` is 3.1415) is generated by the script of Example 5. Notice the mandatory use of the FL selectors, which are used to select the needed coordinates of vertices in the domain decomposition. Vertices are in fact represented as sequences of coordinates.

```
DEF SinFun = [u, sin~u] WHERE  u = S1  END;
DEF Domain (n::IsInt) = (QUOTE ~ #:n):(2*PI/n);
MAP:SinFun:(Domain:32);
```

Example: Circumference and circle of unit radius

A piecewise linear approximation of the circumference of unit radius is generated in the Example 6. Several interesting example of PLaSM script which make use of the MAP operator can be found in [wiley book].

```
DEF Circonference (n::IsIntPos) =
  (MAP:[COS~S1,SIN~S1] ~ QUOTE) : (#:n:(2*PI/n));
DEF Circle (n::IsIntPos) = (JOIN ~ Circonference):n;
STRUCT:< Circonference:36, T:1:2, Circle:36 >;
```

Regularized Booleans Operators

```
+, &, -, \, XOR
```

Quick Positioning and scaling

```
TOP, BOTTOM, LEFT, RIGHT, ...
```

References

1. [Backus78] Backus, J. Can Programming Be Liberated from the Von Neumann's Style? A Functional Style and its Algebra of Programs - Communications of the ACM, 21(8), August 1978, 613—641. (ACM Turing Award Lecture).

2. [BWW90] Backus, J., Williams, J.H. and Wimmers, E.L. An Introduction to the Programming Language FL - In Research Topics in Functional Programming, D.A. Turner (Ed.), Addison-Wesley, Reading, MA, 1990.

3. [BWWLA89] Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., and Aiken, A.. FL Language Manual, Parts 1 and 2. - IBM Research Report, RJ 7100 (67163), 1989.

4. [Will82] Williams, J.H. Notes on the FP Style of Functional Programming - In Functional Programming and its Applications, Darlington, J., P. Henderson and D.A. Turner (Eds.), Cambridge: Cambridge Univ. Press, 1982.

5. [WW91] Williams, J.H. and Wimmers, E.L. An Optimizing Compiler Based on Program Transformation - Internal IBM report, March, 1991.

6. [LZ88] Lucas, P. and Zilles, S.N. Applicative Graphics Using Abstract Data Types - IBM Research Report, RJ 6198, 1988.
7. [ZLLLH88] Zilles, S.N., Lucas, P., Linden, T.M., Lotspiech, J.B. and Harbury, A.R.. The Escher Document Imaging Model - ACM Conference on Document Processing Systems, Santa Fe, NM, 1988.
8. [[[PPV95] Paoluzzi, A., Pascucci, V., and Vicentino, M. Geometric programming: A programming approach to geometric design. - ACM Transactions on Graphics, 14(3), July 1995, 266—306.
9. [PS92] Paoluzzi A and Sansoni C. Programming Language for Solid Variational Geometry - Computer Aided Design, 24(7), July 1992, 349—366.
10. [PFP96] Pascucci, V., Ferrucci, V., and Paoluzzi, A. Dimension-Indipendent Convex-Cell based HPC: Skeletons and Product - International Journal of Shape Modeling, 2(1), January 1996, 37—67.
11. [PBCF] Paoluzzi, A., Bernardini, F., Cattani, C. and Ferrucci, V. Dimension-Independent Modeling with Simplicial Complexes - ACM Transactions on Graphics, 12(1), January 1993, 56—102.
12. [BBCPPV] Bajaj, C., Baldazzi, C., Cutchin, S., Paoluzzi, A., Pascucci, V. and Vicentino, M. A Programming Approach for Complex Animations. Part I: Methodology. Computer Aided Design, 31 (11), 695â€“710, 1999.
13. [PA99] Paoluzzi, A. and D'Ambrogio, A. A Programming Approach for Complex Animations. Part II: Reconstruction of a Real Disaster. Computer Aided Design, 31 (11), 711â€“732, 1999.
14. [BC97] Baldazzi, C., and Paoluzzi, A. Dimension-Independent BSP (1). Section and Interior-to-Boundary Mapping - International Journal of Shape Modeling, 3 (4), 1997.
15. [BC98] Baldazzi, C. and Paoluzzi, A. Dimension-Independent BSP (2): Boundary-to-Interior Mapping. International Journal of Shape Modeling, 4 (1), 1998.
16. [BFPP93] Bernardini, F., Ferrucci, V., Paoluzzi, A. and Pascucci, V. A Product Operator on Cell Complexes - Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications.

Montreal, May, 43-52, J. Rossignac, J.Turner and G. Allen (Eds), ACM Press, 1993.

17. [PFP95] Pascucci, V., Ferrucci, V. and Paoluzzi, A. Dimension-Independent Convex-Cell Based HPC: Representation Scheme and Implementation Issues - Proceedings of the 3rd ACM/IEEE Symposium on Solid Modeling and Applications . Salt Lake City, May, 17-19, C. Hoffmann and J. Rossignac (Eds), ACM Press, 1995.
18. [PPV93] Paoluzzi, V. Pascucci and M. Vicentino. PLASM functional approach to Design: Rep resentation of geometry. Tn: U. Flemming and S. Van Wyk, (eds), CAAD Futures '93, Elsevier Science (1993).
19. [PPS95] Paoluzzi, A. Pascucci, V. and Sansoni, C., Prototype Shape Modeling with a Design Language, in The Global Design Studio, M. Tan and R. Teh (Eds.), Proc. of CAAD Futures 95, Singapore, Sept. 1995.
20. [PP95] Paoluzzi, A. and Pascucci, V., Building Design Programming with a Functional Language, 6th Int. Conf. on Computing in Civil and Building Engineering, Berlin, 12-15 July, AA Balkema Publ., Rotterdam, 1995.
21. [PPV94] Paoluzzi, V. Pascucci and M. Vicentino. PLASM: un Linguaggio Funzionale di Progettazione. Progetto Finalizzato Edilizia. Rapporto conclusivo. C.N.R., Palermo, 1994.
22. [PPV96] Paoluzzi, V. Pascucci and M. Vicentino. Generative, Constraint-based and Variational Modeling through Geometric Programming. Rome, 1996.
23. [P95] Paoluzzi A. Geometric Programming in Architecture, Int. Workshop on Research Directions in Architectural Computing, R. Junge (Ed.), Munich, 16-19 July, 1995.
24. [P96] Paoluzzi A., Generative Geometric Modeling in a Functional Environment (invited lecture), in Design and Implementation of Symbolic Computation Systems, J Calmet and C. Limongelli (Eds.), Lecture Notes in Computer Science, 1128, 79—97, Springer & Verlag, 1996.
25. [PDc96] Paoluzzi A. and Di Carlo A. ,Differential Operators in a Functional Design Language, in Int.

Conf. on Advanced Tools for Scientific Computing, Oslo, September 16-18, 1996.

26. [P03] Paoluzzi, A. Variable-free representation of manifolds via transfinite blending with a functional language, Tenth IMA Conf. on the Mathematics of Surfaces, University of Leeds, UK, 15th - 17th September 2003.
27. [BCP01] Bernardini, F., Cenciotti, G. and Paoluzzi, A. Transfinite Interpolation of Surfaces with Integral and Differential Constraints Using a Geometric Design Language. Proc. of Seventh IFIP Workshop on Geometric Modeling: Fundamentals and Applications, Kluever, 2001.
28. [PPS03] Paoluzzi, A., Pascucci, V. and Scorzelli, G. Progressive Framework for Dimension-Independent Solid Modeling, SIAM Conference on Geometric Design and Computing, Nov. 10-13, 2003, Seattle, Washington.

Last updated 10-Apr-2007 07:32:16 CEST