Computer-Aided Design and Applications

http://www.cadanda.com

Top-Down Archaeology with a Geometric Language

Dario Lucia[*], Francesco Martire[*], Alberto Paoluzzi[*] and Giorgio Scorzelli[*]

[*]Università degli Studi "Roma Tre", {lucia, martire, paoluzzi, scorzell}@dia.uniroma3.it

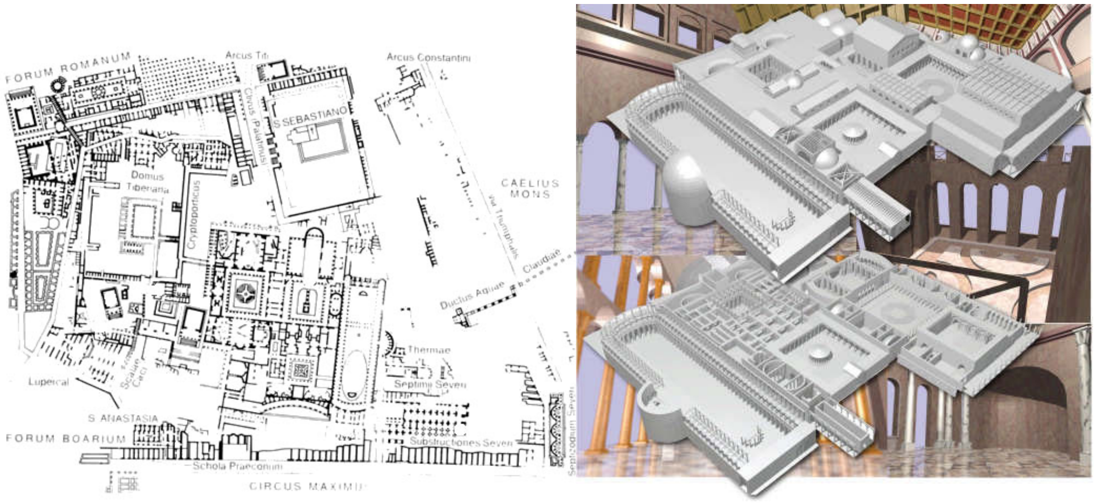


Fig. 1: (a) 2D map of the Palatino hill in Rome; (b) the 3D PLaSM reconstruction of the emperor's palace (Domus Flavia, Domus Augustana and Hippodrome of Domitian).

ABSTRACT

In this paper we discuss a fast reconstruction of an archaeological site *consisting of many different and connected parts*. In particular, we discuss the geometric reconstruction of the Domus Flavia and the Domus Augustana, both built by Emperor Domitian on the Palatine hill in Rome, including the adjacent Hippodromus. The Domus Flavia was the official part of the palace, while the Domus Augustana was the emperor's luxurious private residence. They are the most impressive ruins remaining today on the Palatine. The aim of this paper is to introduce the reader to the power of generative semantics, and to show how fast and easy is the generation of complex VR models if using a generative language. For this purpose, we capitalize on a novel parallel framework for high-performance solid and geometric modeling, that (a) compiles the generating expressions of the model into a dataflow network of concurrent threads, and (b) splits the model into fragments to be distributed to computational nodes and generated independently. We trust that both the language and its kernel are suitable for Cell-BE (Broadband Engine) implementation, that someone believes the reference architecture for advanced modeling, imaging and simulation of next years.

Keywords: VR systems and toolkits, archaeology applications, architectural design, CAD.
DOI: 10.3722/cadaps.2008.483-496

1. INTRODUCTION

The reconstruction of the shape of ancient monuments is normally performed bottom up (see e.g. [8]), by (i) decomposing the building into more or less elementary elements of the building fabric, (ii) elaborating this detailed shape by using some CAD system, (iii) attaching appropriate textures, often derived from actual photographs, and finally by (iv) assembling the various parts in world coordinates. A last exporting into some standard VR environment, usually VRML, concludes the process. This workflow requires both a great amount of human work and a strong expertise in reading and interpreting the architectural drawings and the other diagrams used by architects and archaeologists. Conversely, we discuss here a pretty speedy top-down development approach, that may allow the VR scientist to generate some preliminary models in a very fast way, to use them as the basis for discussion and feedback with archaeologists and art historians, and ready to easily import both a main revision of the overall structure and/or local adjustments, improvements and further detail. On the way to a plausible and consistent reconstruction, a simulation of different solutions can be very helpful. A whole set of different hypothesis can be realized at every stage of the genesis of the model with only a small amount of extra work.

For the RAD (Rapid Application Development) of the Imperial Palace in Rome, to be used as a proof-of-concept for such top-down development, we use the geometric language PLaSM and shortly introduce its syntax, semantics and use modes. PLaSM, (the Programming LAnguage for Solid Modeling) is a design language, strongly influenced by FL (programming at Function Level), a novel approach to functional programming developed by the Functional Programming Group leaded by John Backus and John Williams at the IBM Research Division in Almaden in the first nineties [2],[3]. PLaSM is a geometric extension of FL, allowing for a powerful algebraic calculus with dimension-independent geometric objects that include polyhedral complexes and parametric polynomial and rational manifolds (curves, surfaces, curved solids, and higher-dimensional objects). In this environment for geometric computations, a complex shape is generated as an assembly of component shapes, highly dependent from each other, where (a) each part may result from computations with other parts, (b) a generating function is associated to each, (c) geometric expressions may appear as actual parameters of function calls.

This functional approach implements an *algebraic calculus with geometric shapes*, where a direct mapping is possible between values of the calculus and expressions of the language. This approach, based on dimension-independent representations with guaranteed geometric validity, leads to a versatile approach to Virtual Reality, supporting classes of objects with varying topology and shape, and also retains the good properties of functional programming. In particular, it allows writing program code which is clear, highly concise and self-documenting. The power of this approach comes from the use of a number of operators able to combine shapes, mainly parametric curves, to generate a great number of different geometries. Like CSG, it supports a property of closure, so that other operators can be applied to shapes generated by some expression in the language, resulting in new and more complex shapes.

In this approach a very complicated VR model will be described hierarchically and developed either top-down or bottom-up, and even by using some mixed strategy. Using a syntactically validated programming approach the model updating becomes easier in both cases: both when the changes concern some specific components and when they apply to the overall model organization. In either case it is sufficient to update some generative functions at suitable hierarchical levels. We show that program units that are evaluated more frequently are easily abstracted and elegantly formalized as library operators, and may be reused in other reconstruction projects.

Common languages used for virtual reality modeling are either markup data languages like VRML or X3D, or procedural languages, like Java3D or JavaScript, that may either embed object descriptions based on markup languages or being embedded within them. In both cases the emphasis of modeling is put on structured, hierarchical data languages that do not have the full power of a Turing-complete language. Conversely, we believe that the only way to produce scalable modeling of *complex systems* like critical infrastructures (say, e.g. government facilities or airports) or multi-scale biosystems, we need a specialized language engine like the TEX environment, that may support further code layers (e.g., LATEX and its packages, and its growing family of specialized binaries) aiming at separating contents from presentation, interaction and media. Therefore, this paper has a triple intention: first, it introduces the reader to the extensible semantics of PLaSM, that is reaching its maturity and shows a strong mix of descriptive power, code compactness and versatility; then it shortly discusses the novel data structures in the kernel of version 5; and finally provides an example of Virtual Reality modeling for the archaeological domain.

The design of PLaSM, the open-source Programming Language for Solid Modeling, started in 1992, long time before the internet-based access to virtual worlds was even imagined. The second optimized and extended version of the language interpreter, written in Common Lisp, and characterized by a unique multidimensional approach to geometric modeling [14], was available in 1994. The version 3, extended with animations, colours and cameras, and exporting to Open Inventor and VRML, came in 1999. The following version 4 was completely rewritten in Scheme and C++, with javascripted animations. The novel version 5, introduced here, is being deployed with a fast geometric engine and with rewriting optimizations in the interpreter, aiming to automatically exploit multi-core processing. The XGP (eXtreme Geometric Processing) engine is using a novel multidimensional *decompositive* representation, based on *progressive* BSP trees [15] and on the Hasse graph of the generated d-complex; at upper level is using HPC (Hierarchical Polyhedral Complex) structures. The language allows for powerful operations, including progressive Booleans, the Cartesian product of cell complexes, the extraction of their k-skeletons ($k = 1,\ldots,d$), and the Minkowski sum of a polyhedral complex with parallelotopes, allowing for sweeping and offset. Current extensions aim to encode the object topology as a sparse matrix, using a tensor representation of the chain complex underlying the cell decomposition [5].

The present-day skyrocketing production of virtual buildings and environments is being pushed by novel applications and social networks like *Google SketchUp* and *Second Life*. While both such frameworks provide software tools to interactively create, view, modify and share 3D models on the web, they also require significant user effort to produce non trivial models, and do not easily allow for code refactoring and reusing. A major advance with SketchUp was the recent introduction of a Ruby application programming interface (API) to extend its functionality. This interface allows users to create tools, menu items, and other macros, and even automated component generators. An analogous importing of virtual models into Second Life is allowed by *Blender* and its Python APIs. Various interpreted languages were developed for modelling and animation purposes, including *BAGS* at Brown [17], *Obliq-3D* at SRC [11], and *Alice* [4], that uses Python (interfacing Java) as its embedded interpreted language.
As well as the very influential *Open Inventor* toolkit [18] built on OpenGL, all such environments are hierarchical (after the ISO PHIGS in the eighties) and object-oriented for fast prototyping purposes. The main difference with our approach is that they *describe* procedurally the shape, whereas, conversely, PLaSM *computes* the geometry as the result of *evaluation of algebraic expressions*, as this paper aims to show. With respect to recent published work on building design using shape grammars [10],[19], the main difference is that our approach may be finalized to the *actual reconstruction* of a site and of its actual *interior shape*, even *meshed* at variable levels of detail, and not only to the visual simulation of the possible exterior building shells at the town-planning level. “CGA shape, the novel shape grammar for the procedural modeling of CG architecture, produces building shells with high visual quality and geometric detail. It produces extensive architectural models for computer games and movies, at low cost”. To tell this more clearly, the two main differences with the present approach are emphasized: (a) CGA shape is a *shape grammar*, hence able to generate *abstract* virtual buildings according to shape generating rules, providing a specified style; conversely PLaSM generates *concrete* virtual buildings, starting from actual engineering drawings, in this case the photogrammetry of the archaeologic site; (b) CGA shape produces a photorealistic view of the *building shells*, i.e. a *surface* representation of the *exterior envelope*; conversely, PLaSM produces a *solid model* with any desired degree of accuracy of the interior building, as well as of the exterior envelope. PLaSM is currently being used to modelling real-world critical infrastructures (say, airports or nuclear plants), in order to provide the integration layer for all the data collected by the various sensor systems, including video-surveillance, presence and RFID tracking, fire and smoke alarms, etc.

Further development of the PLaSM geometric kernel has recently introduced (in the new advanced kernel) the attachment of properties, not only predefined (like color-per-vertex or color-per-face or textures, etc), but also dynamically user-defined, to any geometric value. A new exporting to the new interchange file format for interactive 3D applications named *Collada* [1] that stands for COLLAborative Design Activity, originally created by Sony as the official format for the PlayStation 3. The streaming character of the language will fit well with the CUDA (Compute Unified Device Architecture) environment, the GPGPU technology that allows to code algorithms for execution on the GPU. A next version of the geometric kernel is planned to lay over a CUDA interface. Some preliminary experiments were already done.

In Section 2 the very basic elements of the PLaSM language syntax and semantics are introduced. In Section 3 we shortly describe the dimension-independent representation schemes used by the language kernel, namely the BSP trees and the Hasse graph, and briefly introduce our next representation of topology, based on sparse matrices. In Section 4 the extensible character of the language is discussed by giving several simple implementations of useful extensions. In Section 5 the application of top-down modeling for archaeology is discussed with reference to the reconstruction of the Flavian palaces on the Palatine hill. In the Conclusion section some development directions are outlined.

2. LANGUAGE SYNTAX AND SEMANTICS

According to the FL semantics, an arbitrary PLaSM script can be written by using only three programming constructs:

- **application** of a function to the actual value of input *parameters*, elements of the function domain, producing an output *value* in the function codomain;
- **composition** of two or more functions, producing the pipelined execution of their reversed sequence (see Fig. 2);
- **construction** of a vector function, allowing for the parallel execution of its component functions.

The model of a PLaSM computation is a directed graph $G=(N,A)$ where the set $N=N_p \cup N_d$ of nodes is partitioned in two disjoint subsets of *programs* and *data*, where a data object may denote either a single value or a sequence of values.

The set A of directed arcs is partitioned into three disjoint subsets, $A_a \subset N_d \times N_p$ denoted as *application*, $A_c \subset N_p \times N_p$ denoted as *composition*, and $A_e \subset N_p \times N$, denoted as *evaluation*, respectively. There is no semantic difference between data and programs. A datum in a given computation can be a program in another computation. The nodes in N_d may be visually represented as rounded boxes, those in N_p as squared boxes (see Fig. 2).

Primitive objects are characters, numbers, truth values and *polyhedral complexes*. A polyhedron is a disjoint union of polytopes, i.e. of bounded convex sets. *Expressions* are either primitive objects, functions, applications or sequences.

The application expression $\exp r1 : \exp r2$ applies the function resulting from the evaluation of $\exp r1$ on the argument resulting from the evaluation of $\exp r2$. It may be useful to remember that: (a) binary functions can be also *applied* in infix form, as shown below; (b) application has the *greatest precedence* with respect to other operators, and that (c) application is *left-associative*.

$$+ :< 1,3 > \equiv 1+3 \equiv 4 \tag{0.1}$$

Composition is pipelining According to the mathematical definition of function composition: $(f \circ g)(x) \equiv f(g(x))$ requiring that function f applies to the value resulting from the g application to the x value, our main paradigm, represented in Fig. 2, states that *a function composition is a computational pipeline* working in the reverse order of the functions to be composed.

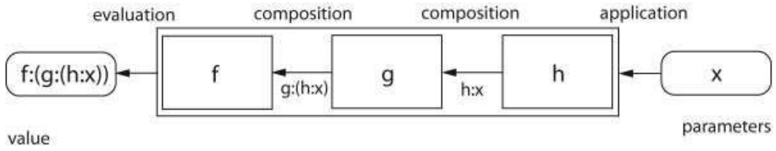


Fig. 2: A visual representation of the expression $(f \sim g \sim h) : x \equiv (f \sim g) : (h : x) \equiv f : (g : (h : x))$.

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

$$CONS :< f_1, f_2, \ldots, f_n >: x \equiv [f_1, f_2, \ldots, f_n] : x \equiv < f_1 : x, f_2 : x, \ldots, f_n : x > \tag{0.2}$$

Partial functions A higher-order curried function of the kind: $f : A \to B \to C \equiv f : A \to (B \to C)$ can be defined, using formal parameters, as in Listing (2.3),

$$\begin{aligned} & DEF\ f(a :: isA)(b :: isB) = body\_\exp r \\ & WHERE \\ & \qquad local\_env \\ & END; \end{aligned} \tag{0.3}$$

Listing (0.3): Params a and b optional; eval of $body\_\exp r$ returns its value; $local\_env$ hidden outside f.

where isA and isB are (either pre- or user-defined, respectively) predicates used at run-time to test the set-membership of arguments. The function f is applied to actual arguments as $f : x : y \equiv (f : x) : y$, and returns a value $c \in C$. Let notice that the value generated by the expression $f : x$ is a *partial function*, and remember that application is left-associative.

3. GEOMETRY ENGINE ARCHITECTURE

We briefly introduce the dimension-independent geometric structures in the kernel of PLaSM 5, where *progressive BSP trees* are used for streaming evaluation of geometric expressions [15]. *Hasse graphs* are used to maintain a complete representation of the model topology. A novel bi-diagonal block-matrix representation of the *chain complex* of the model, called *Hasse matrix* [5], should soon support both the geometric design and the physical simulation.

Progressive BSP trees (PBSP), supporting progressive Boolean operations, are introduced in the V.5 geometric kernel, as a progressive, adaptive and multi-resolution solid representation. Three types of labels are used for leafs: IN (full cells); OUT (empty cells) and FUZZY (*undecided* cells). A concept of *frontier*, defined as a tree-node subset giving a space partition into convex cells, is needed for this purpose. A frontier is any collection of PBSP nodes such that their cells are a partition of the embedding Euclidean space. A (progressive) *refinement*, or *forward computation*, proceeds by splitting some fuzzy cell in the current frontier. A (progressive) cell *join*, or *backward computation*, proceeds by gluing a pair of sibling nodes, and returning their parent node. An adaptive local refinement or collapse is always guaranteed by forward or backward moves of the frontier.

Hasse graphs A cell complex K is a collection of compacts subsets of E^d, called *cells*, such that: (a) if $c \in K$, then every *face* of c is in K; (b) the intersection of any two cells is either empty or a face of both. A *d-polytope* is a solid, convex and bounded subset of E^d. A *polytopal d-complex*, or *d-mesh*, is a cell complex of *d*-polytopes and their *k*-faces $(0 \leq k \leq d)$. A complete representation of a *d*-mesh is given by the *Hasse graph* of the *cover* relation of cells, whose nodes are the members of the complex K, partially ordered by containment, and where there is an arc from node x to node y if: (a) $x \subset y$ and (b) there is no z such that $x \subset z \subset y$. Hasse graphs are used in the language kernel as a complete representation of the K topology, that cannot be handled efficiently by BSP trees.

Hasse matrices The Hasse representation of (co)chain complexes [5] is being currently ported to Plasm, with the aim of simplifying and making algebraic the interface between domain representation and field computations, providing an adaptive multigrid approach to simulation. As a matter of fact, the (co)chain-complex formalism and the Hasse-matrix representation generalize in a natural and straightforward way to physical modelling. Chains assign measures to cells, measures that may be tuned to represent the physical properties of interest (mass, charge, conductivity, stifness, and so on). Cochains, on the other side, may be used to represent all physical quantities associated to cells via integration with respect to a measure.

4. DEFINING NEW PRIMITIVES

The PLaSM language is extensible by design. Even the most common graphics primitives (say, polyline and triangle-stripe) may be natively defined in PLaSM, and novel geometric operations can be easily added to the language by putting them in a library. The current libraries contain about 600 functions. We show the implementation of some architectural primitives of growing complexity. For a full discussion of the constructs used, the reader is referred to the book [13]. Notice that both basic operations and library functions are displayed in red.

4.1 Columns

Our examples of specialized generative primitives start with the columna function, that assemblies a parametric column in pseudo-Corinthian style. The *formal parameters*, whose scope covers the function body, i.e. the generating expression on right-hand side of the function head as well as the local environment delimited by the $WHERE$, END keyword pair, stand respectively for:

- dm is the circumference diameter at the column basis;
- h is the column height;
- h_base is the height of the column base.

The column is assembled by gathering the parts of the object, and by putting each part on the top of the previous ones. In particular, it is composed of a vertical cylndr that is wider at the bottom, two geometric toruses, torus_bot at the bottom, and torus_top at the top of the cylinder, respectively, by two parallelepiped bases base and base_top, and by the capital, generated by rotated union $(+)$ of two squared baskets. $Truncone$ and $Torus$ are the PLaSM primitives for generation of a truncated cone and a torus surface, respectively. The composition with the $Join$ operator transforms their output from a surface to its solid convex hull. Notice, with respect to the sequence of TOP operations in the function body, that they are left-associative.

$$
\begin{aligned}
& DEF\, columna(dm, h, h\_base :: isReal) = base\, TOP\, torus\_bot\, TOP\, cylndr\, TOP \\
& \qquad\qquad\qquad\qquad\qquad\qquad torus\_top\, TOP\, capital\, TOP\, base\_top \\
& WHERE \\
& \quad cylndr = (JOIN \sim TRUNCONE :< dm/2, 0.8*dm/2, h >) : 24, \\
& \quad torus\_bot = (JOIN \sim TORUS :< dm/12, d/2 >) :< 8, 24 >, \\
& \quad torus\_top = (JOIN \sim TORUS :< 0.8*(dm/12), 0.8*(dm/2) >) :< 8, 24 >, \\
& \quad base = (T :< 1,2 >:< 7*dm/-12, 7*dm/-12 >\sim CUBIOD) :< 7*dm/6, 7*dm/6, h\_base >, \\
& \quad base\_top = (T :< 1,2 >:< 7*dm/-12, 7*dm/-12 >\sim CUBIOD) : \\
& \qquad\qquad\qquad\qquad < 7*dm/6, 7*dm/6, dm/6 >, \\
& \quad capital = (JOIN \sim TRUNCONE :< 0.8*dm/2, 1.2*dm/2, h/8 >) : 4 + \\
& \qquad (R :< 1,2 >: (PI/4) \sim JOIN \sim TRUNCONE :< 0.8*dm/2, 1.2*dm/2, h/8 >) : 4 \\
& END;
\end{aligned} \tag{0.4}
$$

Listing (0.4): Definition of a column.

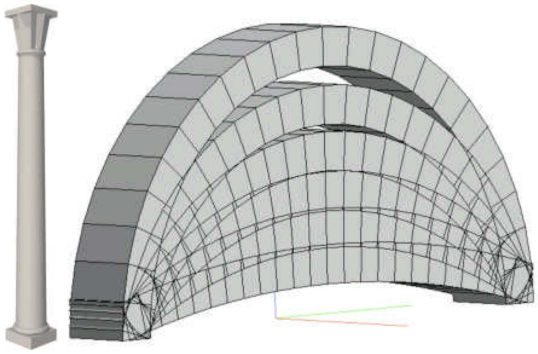


Fig. 3: (a) Model generated by instancing the *columna* function; (b) Arch values generated by the geometric expression: $(STRUCT \sim AA : (Arch :< 3, 0.2, 0.4 >)) : (AA \sim C : * : (PI/6)) : (1..6)$.

4.2 Archs and Domes

The MAP operator is the constructor of curved parametric geometry. Its semantics can be described as MAP : coordfuns : domain, where domain $\subset \mathrm{E}^d$, with $d = 1$ for curves, $d = 2$ for surfaces, $d = 3$ for volume maps, and so on (higher-dimensional manifolds). The coordfuns parameter is a sequence $< x_i >$ of *coordinate functions* x_i : domain $\to \mathbb{R}, 1 \le i \le n$, where $n \ge d$ is the dimension of the embedding space E^n of the generated d -manifold.

A valued property of the language PLaSM is that it allows for *transfinite* implementation of curved parametric geometry, where the function basis may be combined with *vectors of coordinate functions* used as *control* points.
An example of the expressive power of the geometric language is shown by the Arch and ArchSurface functions in Listings (4.3) and (4.2), that may generate both *round* (i.e. semicircular) and *segmental* arcs, with any opening angle. So, the ArchSurface function was first defined, that generates the transfinite blending (linear Bézier interpolation) of two "control curves": circumferences with same center and different radiuses rr and rr − w (for *width*), respectively. Such a function, when mapped on a 2D interval $[\alpha_1, \alpha_2] \times [0, w], \alpha_1 < \alpha_2$, returns a segment of surface arch with angle $\alpha_2 - \alpha_1$ and width w .

In order to generate a solid arch, the transfinite interpolation is repeated, using now two "control surfaces", i.e. two parallel ArchSurf2D translated each other by the vector (depth, 0, 0). The results of the expressions $(K:0\,AL$ ArchSurf2D$)$ and $(K:$ depth AL ArchSurf2D$)$ where AL stands for *append-left* so that $x\,AL < y,z > \equiv < x,y,z >$, are two 3D vectors of *bivariate* coordinate functions, that are then used as the two Bézier control "points" to generate a *three-variate* map, later applied to a proper 3D interval domain to generate the solid arc. The expression $K:x$ transforms a number x into a *constant* function with the same value. The formal parameters of the Arch function are: the arc length; the horizontal arc width; the arc depth; the angle, usually given in the ancient architecture as a multiple of $\pi/6$ (see Fig. 3(b)).

$$\begin{aligned}
&DEF\ ArchSurface(rr, w :: isReal) = Bezier : S2 :< Circle0, Circle1 > \\
&WHERE \\
&\qquad Circle0 =< K : rr * \cos \sim S1, K : rr * \sin \sim S1 >, \\
&\qquad Circle1 =< K : (rr - w) * \cos \sim S1, K : (rr - w) * \sin \sim S1 > \\
&END;
\end{aligned} \tag{0.5}$$

Listing (0.5): Definition of an arch surface blending function.

$$\begin{aligned}
&DEF\ Arch(length, w, depth :: isReal)(angle :: isReal) = (T : 3 : (- : ceiling) \sim MAP : SolidMap) : domain3D \\
&WHERE \\
&\qquad radius = (length/2)/\cos : (angle/2), \\
&\qquad ceiling = MIN : 2 : (MAP : ArchSurf2D : domain2D), \\
&\qquad domain2D = (T : 1 : (angle/2) \sim intervals : (PI - angle)) : 16 * q : 1, \\
&\qquad domain3D = domain2D * q : 1, \\
&\qquad SolidMap = Bezier : S3 :< Surf3D_0, Surf3D_1 >, \\
&\qquad ArchSurf2D = ArchSurface :< radius, w >, \\
&\qquad Surf3D_0 = K : 0\,AL\,ArchSurf2D, \\
&\qquad Surf3D_1 = K : depth\,AL\,ArchSurf2D \\
&END;
\end{aligned} \tag{0.6}$$

Listing (0.6): Definition of an arch.

Another similar example is given by the function Dome, that generates a dome whose basis equates a regular polygon with a given number of sides. The generative technique is the same used for the Arch function, i.e. transfinite blending between two translated halfspheres, or better, between their two-variate *generative maps*. The formal parameters are: the number n of the sides of the basis; the lateral length (diameter); the width of the solid; the solid angle, in order to produce segmental domes, as for segmental arches.

$$\begin{aligned}
&DEF\ HalfSphere(r :: isRealPos) =< fx, fy, fz > \\
&WHERE \\
&\qquad fx = K : r * - \sim SIN \sim S2 * COS \sim S1, \\
&\qquad fy = K : r * COS \sim S1 * COS \sim S2, \\
&\qquad fz = K : r * SIN \sim S1 \\
&END;
\end{aligned} \tag{4.4}$$

Listing **Error! Reference source not found.**: Definition of a half sphere blending function.

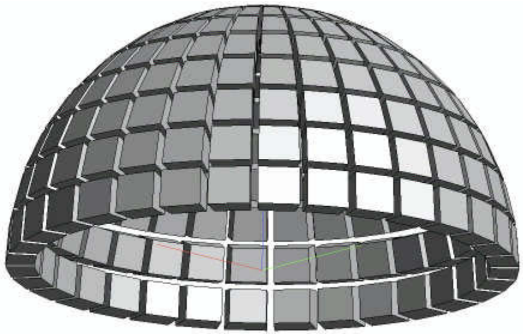


Fig. 4: Dome instance view with exploded cells. The reader should remember that the PLaSM internal representation is decompositive, so that a cellular decomposition of the model is natively available.

$$
\begin{aligned}
& DEF\ Dome(n :: isNat)(length, w, angle :: isReal) = (T:3:(-:ceiling) \sim MAP:SolidMap):domain3D \\
& WHERE \\
& \qquad radius = length/(2*\cos:angle), \\
& \qquad ceiling = MIN:3:dome1, \\
& \qquad SolidMap = Bezier:S3:<Surf3D\_0, Surf3D\_1>, \\
& \qquad Surf3D\_0 = HalfSphere:radius, \\
& \qquad Surf3D\_1 = HalfSphere:(radius - w), \\
& \qquad domain2D = (T:1:(angle) \sim intervals:(PI - angle)):12*intervals:(2*PI):n, \\
& \qquad domain3D = domain2D * q:1 \\
& END;
\end{aligned}
\tag{0.7}
$$

Listing (0.7): Definition of a dome generative function.

4.3 Truss

A *truss* is a structure comprising one or more triangular units constructed with straight slender members whose ends are connected at joints. A plane truss is one where all the members and joints lie within a 2-dimensional plane. The <u>*Truss*</u> function in Listing (4.6) defines a space truss of given <u>*length*</u> and height <u>*h*</u>.

The solid truss is produced from the wire-frame model by applying an operation $offset:<\underline{x}, \underline{y}, \underline{z}>$, where $\underline{x}$, $\underline{y}$, $\underline{z}$ respectively denote the 3 dimensions of the cuboid that slides on the wire-frame to produce the offset members of the instanced <u>*truss*</u>. Notice that 2D <u>*verts*</u> in the expression $(MKPOL):<\underline{verts}, \underline{cells}, \underline{pols}>$ produce a normalized wire frame in 2D, that is embedded into the $y=0$ subspace by the function $MAP:[S1, K:0, S2]$, scaled to the actual dimensions by the affine transformation $S:<1,3>:<\underline{length}/12, \underline{h}/4>$, and finally made solid by the function $offset:<\underline{x}, \underline{y}, \underline{z}>$.

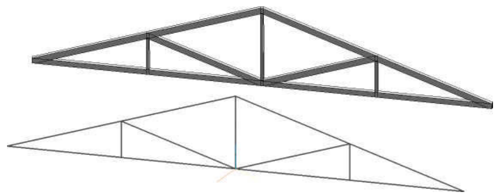


Fig. 5: Wire frame and offset truss instances.

$$
\begin{aligned}
& DEF\ truss(length, h :: isReal)(x, y, z :: isReal) = (offset :< x, y, z > \sim S :< 1,3 >:< length / 12, h / 4 > \sim \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad MAP : [S1, K : 0, S2] \sim MKPOL) :< verts, cells, pols > \\
& WHERE \\
& \qquad verts = << -6,0 >, < -3,0 >, < -3,2 >, < 0,0 >, < 0,4 >, < 3,0 >, < 3,2 >, < 6,0 >>, \\
& \qquad cells = << 1,2 >, < 1,3 >, < 2,3 >, < 2,4 >, < 3,4 >, < 3,5 >, < 4,5 >, \\
& \qquad\qquad\qquad < 4,6 >, < 4,7 >, < 5,7 >, < 6,7 >, < 6,8 >, < 7,8 >>, \\
& \qquad pols = aa : list : (1..13) \\
& END;
\end{aligned}
\tag{0.8}
$$

Listing (0.8): Definition of a truss generative function.

A truss instance is shown in Fig. 5, where both the reference wire frame model and the offset model are given. Notice that they contain 13 members (*pols*), each one defined as the convex hull of two vertices (*cells*), and 8 joints (pairs of coordinates in *verts*).

4.4 Roof

The function *RoofAngled*, shown in Listing (4.7), generates a roof segment of trapezoidal shape, with a given angle on the extreme joint.

The function *Roof* conversely produces a roof with rectangle shape, i.e. a two-dimensional array of tiles. The meaning of the formal parameters of the *RoofAngled* is: *len_x* length of roof; *len_y* width of roof; *len_z* z-displacement of the highest side; *alpha* joint angle.

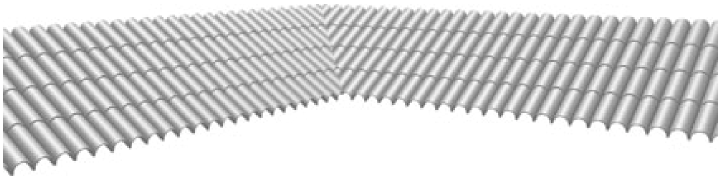


Fig. 6: Angled roof with half-cylindrical tiles.

$$
\begin{aligned}
& DEF\ Tile(n\_x :: isIntPos; \dim\_x, \dim\_y, \dim\_z :: isRealPos) = ...; \\
& \\
& DEF\ Roof(len\_x, len\_y, len\_z :: isRealPos) = ...; \\
& \\
& DEF\ RoffAngled(len\_x, len\_y, len\_z, alpha :: isRealPos) = (STRUCT \sim AA : - \sim DISTR \sim \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad [SPLITCELLS, K : cubo]) : tiledRoof \\
& WHERE \\
& \qquad tiledRoof = Roof :< len\_x, len\_y, len\_z >, \\
& \qquad cubo = (T : 1 : (SIZE : 1 : tiledRoof - (SIZE : 2 : tiledRoof / \cos : alpha * \sin : alpha)) \sim \\
& \qquad\quad R :< 1,2 >: (- : alpha) \sim cuboid) : \\
& \qquad\quad < SIZE : 2 : tiledRoof / \cos : alpha, SIZE : 2 : tiledRoof / \cos : alpha, SIZE : 2 : tiledRoof > \\
& END;
\end{aligned}
\tag{0.9}
$$

Listing (0.9): Definition of a tile, a row of tiles and a roof generative function with assigned angle of join.

4.5 Groin Vault

The function *GroinVault*, given in Listing (4.8), generates a squared *groin vault*, also known as a *double barrel vault* or *cross vault*. It is a vault produced by the intersection at right angles of two barrel vaults. The real parameters of the macro are: ***length*** of the vault side; ***w*** width of walls; opening ***angle*** of the vault (usually a multiple of $pi/6$). The coding of the new macro largely exploits the several symmetries that may be found in a groin vault. In particular, the PLaSM expressions $(STRUCT \sim [id, S:1:-1]):$ ***vault4th*** and $(STRUCT \sim [id, S:2:-1]):$ ***vaultHalf*** produce a double instance of the arguments ***vault4th*** and ***vaultHalf***, respectively reflected against the $x=0$ and the $y=0$ planes. Analogously, the ***vault4th*** value is generated by assembling the ***vault8th*** (i.e. the geometry contained in a $\frac{1}{8}$ of the reference frame) and its copy reflected against the plane $x=y$. For this purpose is used the mapping produced by the expression $MAP:[S2,S1,S3]$, that just swaps the first and second coordinate of all vertices of its geometric argument. For reader's convenience, we remember that the ***ArchSurface*** is shown in Listing (0.5).

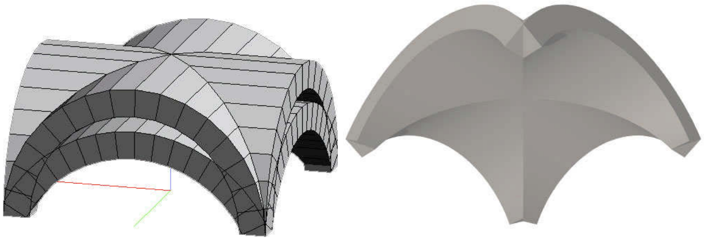


Fig. 7: Groin vaults of different opening angle.

$$
\begin{aligned}
&DEF\ GroinVault(length, w, angle :: isReal) = (T:3:(-:ceiling) \sim STRUCT \sim [id, S:2:-1,]) : vaultHalf\\
&WHERE\\
&\qquad radius = length/(2 * \cos : angle),\\
&\qquad ceiling = MIN:3:Vault\_8th,\\
&\qquad solidMap = Bezier:S3:< surf\_0, surf\_1 >,\\
&\qquad surf\_0 = K:(length/2)\ AL\ archSurf2D,\\
&\qquad surf\_1 = [S1,S1,S3]:archSurf2D,\\
&\qquad archSurf2D = archSurface:< radius, w >,\\
&\qquad vault8th = MAP:solidMap:domain3D,\\
&\qquad domain3D = (T:1:(angle) \sim intervals:(PI/2 - angle)):8*q:1*q:1,\\
&\qquad vault4th = (STRUCT \sim [id, MAP:[S2,S1,S3]]):vault8th,\\
&\qquad vaultHalf = (STRUCT \sim [id, S:1:-1]):vault4th\\
&END;
\end{aligned}
\tag{0.10}
$$

Listing (0.10): Definition of a generative function of groin vaults with assigned length, angle of join and width.

5. RECONSTRUCTION OF THE FLAVIAN PALACE

We discuss in this section the geometric reconstruction of the *Domus Flavia* and the *Domus Augustana*, both built by the Flavian dynasty (Vespasian, Titus and Domitian) on the Palatine hill in Rome, including the adjacent *Hippodromus*. The Domus Flavia was the official part of the palace, while the Domus Augustana was the emperor's luxurious private residence. They are the most impressive ruins remaining today on the Palatine.

The Flavian Palace, that was built during the reign of the Flavian emperors, and extended and modified by several emperors, is spread across the Palatine Hill and looks out over the Circus Maximus (see Fig. 8(a)). Immediately adjacent to the palace of Severus is the Hippodrome of Domitian. It can be better described as a Greek Stadium, that is, a venue for foot races. While it is certain that during the Severian period it was used for sporting events, it was most likely originally built as a garden shaped like a stadium. The reconstruction project started with searching for documentation from “Sopraintendenza Archeologica di Roma”, and with the study of some recent reconstruction, including the one of the ”Accademia Germanica di Roma” and related studies [6],[7],[20].

The first step concerned the analysis of the digital records (in TIFF format) of the stereo-photogrammetric maps of the archeological site (see Fig. 9(a)) using vectorial drawing tools like Adobe’s Illustrator™ so starting a top-down interpretation process paired with the production of interpretative drawings, being accompanied with zooming and naming of every zone of the site (see Fig. 9(b) and Fig. 9(c)), to be used as denotations of the PLaSM macros modeling each and every room and open space of the site.
The traced outlines of spaces where exported from Illustrator™ to SVG (Scalable Vector Format) file format, the vector graphics open standard for web applications, and hence imported into PLaSM polylines and polygons, by way of a simple text manipulation using Grep expressions, i.e. standard UNIX filters that search and replace through a set of files for an arbitrary text pattern, specified through a regular expression. Then a coordinate transformation from integer (pixel) units to standard metric unit was applied, including a change of origin and reflection of the y axis, so defining a right-handed Cartesian coordinate system with origin set in one of the most visually remarkable points of the map, on the south-east angle of the *Peristylium*. A first top-level symbolic model of the whole site was assembled in this stage. The next step consisted in assigning heights and z measures to each zone, using information taken from photogrammetries, photographs, archeological and architectural drawings. Both 2.5D and 3D volumetric models were produced in this phase, see e.g. Fig. 12, to be later used as containment boxes of model parts developed in local coordinates. From this point on, only local coordinates were employed to specify the detailed architecture of each single zone. Either affine 3D transformations (specified by 4 affinely independent points, if the containment box is regular) or trilinear maps (specified by the 8 extreme vertices of the containment volumes, if irregular) were used to map the parts from local to global coordinates.
The whole top-down reconstruction process can be summarized as follows:

1. development of a library of parametric architectural primitives, including: semicircular and segmental arches; barrel and groin vaults with various opening angles; architraves and colonnades; columns and savastæ; wooden frames and trusses; exedræ; circular, elliptical, octahedral and squared domes, as well halfdomes; paneled ceilings, etc.
2. analysis of the archeological site and the available geometric information, including stereo-photogrammetric maps, and ancient relief drawings;
3. tracing out, zooming and labeling of macro zones and single spaces;
4. importing of geometric information into the PLaSM language through simple definitions whose body is either a polyline or a polygon;
5. introduction of z local information of both floors and ceilings, producing the containment volumes of the spaces;
6. separate development of microzones in local coordinates, and assembling into macrozones;
7. global assembly via either local to global affine transformations or three-linear maps defined by 8 corresponding points.

The development process was pretty quick, and required no more that 1.5 man-month, including the development of the library of architectural elements. The progress of next archeological projects will be faster and faster while the system will be used and the know-how is gathered.

Fig. 8: (a) View, seen from the Circus Maximum, of the Imperial Palace on the Palatine Hill; (b) an archeological drawing of the ruins site, with the Stadium on the right, the Domus Flavia on the bottom and the Domus Augustana on the top.

Fig. 9: Top-down site analysis: (a) aerophotogrammetric plan of the archeological site; (b) selection and naming of palace zones; (c) selection and naming of single spaces in each zone.

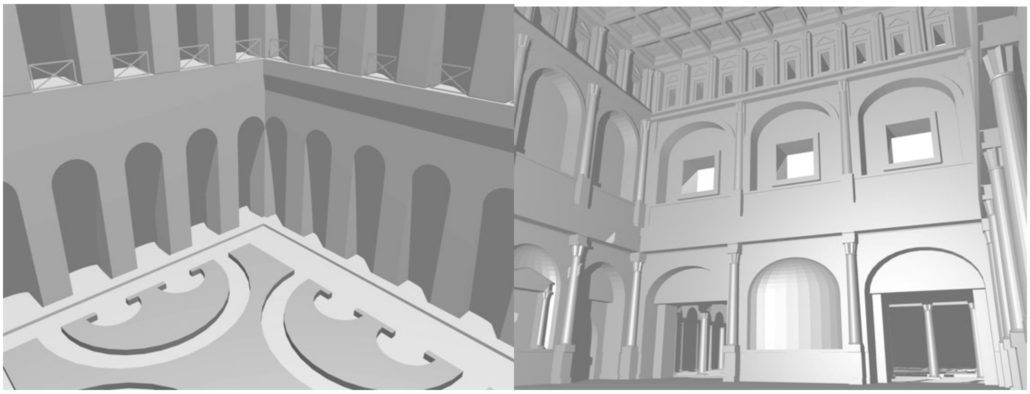
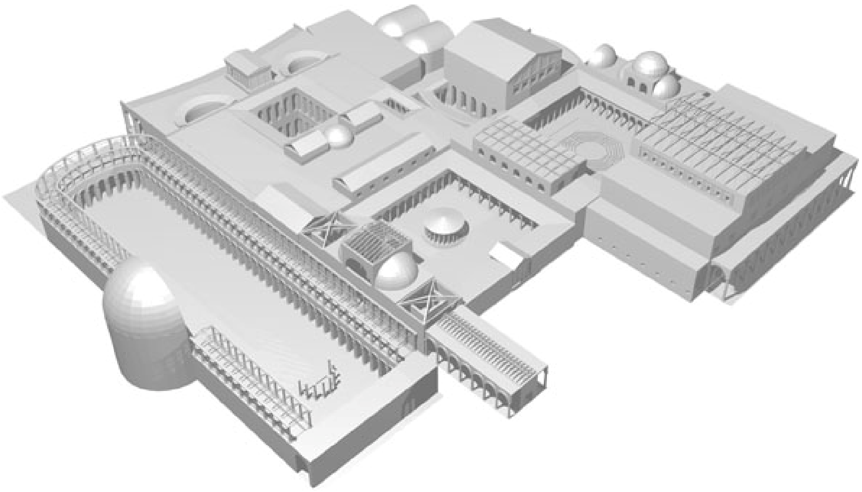


Fig. 10: (a) A view of the Inferior Perystilium in the reconstructed palace model (see the yellow area in Fig. 9(c).); (b) an internal view of the Aula Regia (see the green area on top of Figure 9(b)).



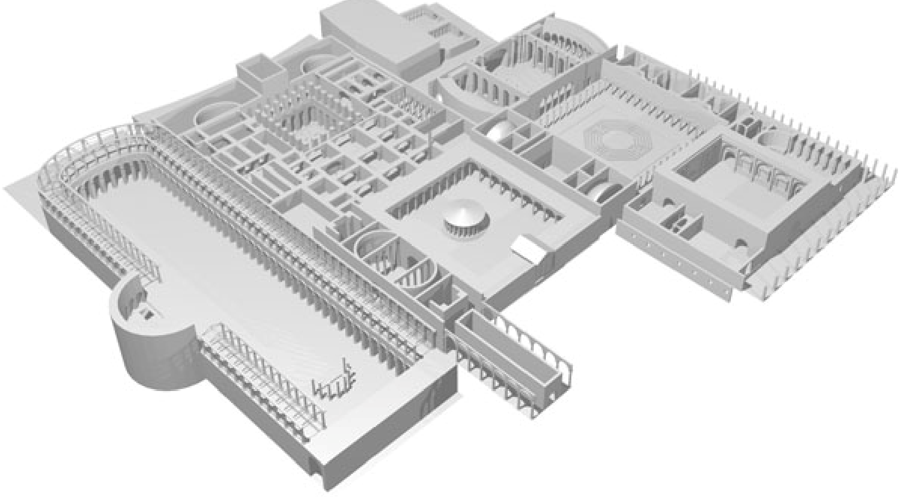


Fig. 11: Two top-views of the reconstruction of the site. (a) the whole model from the above; (b) same view with the model cut at the roof level, to have a picture of the interior reconstruction.

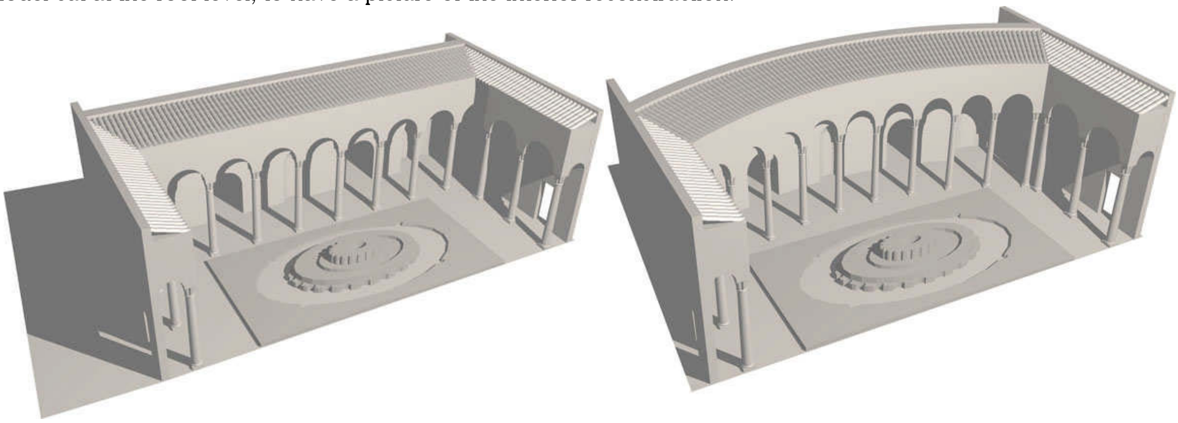


Fig. 12: (a) First development of Nymphæum; (b) Final value of Nymphæum after application of a volume map.

6. CONCLUSION

We shortly described here the syntax, semantics and use of PLaSM 5, a functional language allowing for a powerful geometric calculus as well as for progressive model generation and visualization via data parallelism and pipelined streaming.

The PLaSM language is leaving its infancy, and looking around for complex applications and people speaking it fluently. Novel virtual reality and simulation frameworks are currently being developed with encouraging perspectives, including modeling and visualization of critical infrastructures and multi-scale modeling of biosystems. Why learning PLaSM? Our answer is: to get first-class descriptive power, exploit next generation hardware, generate models by coding few lines, test new geometric algorithms in minutes, and finally explore, like a fascinating new world, the isomorphism between the algebra of shapes and the algebra of PLaSM programs.

7. ACKNOWLEDGEMENTS

This project was partially supported by a grant from AlmavivA-CNR to the Department of Informatica e Automazione. The authors would like to thank Alessandra Raffone for kind encouragement and support. Great thanks to present and past PLaSM developers.

8. REFERENCES

[1] Arnaud, R.; Barnes, M.: Collada: sailing the gulf of 3D digital content creation, A.K. Peters LTD, 2006.
[2] Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Commun. ACM, 21(8), 1978, 613–641, Turing Award Lecture.
[3] Backus, J.; Williams, J.-H.; Wimmers, E.-L.: An introduction to the programming language FL, In Research topics in functional programming, Addison-Wesley Longman Publ., Boston, MA, USA, 1990, 219–247.
[4] Conway, M.; Pausch, R.; Gossweiler, R.; Burnette, T.: Alice: a rapid prototyping system for building virtual environments, In CHI '94, New York, NY, USA, 1994, 295–296, ACM Press.
[5] DiCarlo, A.; Milicchio, F.; Paoluzzi, A.; Shapiro, V.: Solid and physical modeling with chain complexes, In ACM Solid and Physical Modeling Symposium, ACM Press, Beijing, China, 2007, ACM SPM 2007.
[6] Hoffmann, A.; Wulf, U.: Vorbericht zur bauhistorischen Dokumentation der sogenannten Domus Severiana auf dem Palatin in Rom, Römische Mitteilungen, 2002, 279–298.
[7] Hoffmann, A.; Wulf, U.: Die Kaiserpaläste auf dem Palatin in Rom. Das Zentrum der römischen Welt und seine Bauten, Zaberns Bildbände zur Archäologie, 2004.
[8] Koob, M.: Synagogen in Deutschland. Eine virtuelle Rekonstruktion, Birkhäuser-Verlag, 2004.
[9] Lewis, R.; Séquin, C.-H.: Generation of 3D building models from 2D architectural plans, Computer-Aided Design, 30(10), 1998, 765–779.
[10] Müller, P.; Wonka, P.; Haegler, S.; Ulmer, A.; Gool, L.-V.: Procedural modeling of buildings, In SIGGRAPH '06, pages, New York, NY, USA, 2006, 614–623, ACM Press.
[11] Najork, M.-A.; Brown, M.-H.: Obliq-3d: A high-level, fast-turnaround 3d animation system, IEEE Transactions on Visualization and Computer Graphics, 1(2), 1995, 175–193.
[12] Naylor, B.; Amanatides, J.; Thibault, W.: Merging BSP trees yields polyhedral set operations, In SIGGRAPH '90, New York, NY, USA, 1990, 115–124, ACM Press.
[13] Paoluzzi, A.: Geometric Programming for Computer Aided Design, John Wiley & Sons, Chichester, England, 2003.
[14] Paoluzzi, A.; Pascucci, V.; Vicentino, M.: Geometric programming: a programming approach to geometric design, ACM Trans. Graph., 14(3), 1995, 266–306.
[15] Scorzelli, G.; Paoluzzi, A.; Pascucci, V.: Parallel solid modeling using BSP dataflow, Journal of Computational Geometry and Applications, 2007, to appear.
[16] So, C.; Baciu, G.; Sun, H.: Reconstruction of 3D virtual buildings from 2D architectural floor plans, In VRST '98, Proc. of the ACM symposium on Virtual reality software and technology, New York, NY, USA, 1998, 17–23, ACM Press.
[17] Strauss, P.-S.: BAGS: The Brown Animation Generation System, Technical Report CS-88-22, Brown University, Dept. of Computer Science, Providence, RI, 1988.
[18] Wernecke, J.: The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.
[19] Wonka, P.; Wimmer, M.; Sillion, F.; Ribarsky, W.: Instant architecture, In SIGGRAPH '03, New York, NY, USA, 2003, 669–677, ACM Press.
[20] Wulf, U.: Die Kaiserpaläste auf dem Palatin in Rom. Von den bescheidenen Anfängen unter Augustus zum urbanistischen Zentrum eines Weltreiches, Nürnberger Blätter zur Archäologie, 19, 2003, 121–136.