

Part I

Programming and Geometry

Geometric Programming for Computer-Aided Design. Alberto Paoluzzi, Valerio Pascucci, Michele Vicentino, Claudio Baldazzi and Simone Portuesi
Copyright © 2003 John Wiley & Sons, Ltd. ISBN: 0-471-89942-9

1

Introduction to FL and PLaSM

A *statement* is a programming language construct that is evaluated only for its effect. Examples include assignment, input/output statements, and control statements. Programs in most languages are composed primarily of statements; such languages are said to be *statement oriented*.

Programming language constructs that are evaluated to obtain values are called *expressions*. Arithmetic expressions are the most common example. Expressions may occur as parts of statements, as in the right-hand side of an assignment statement. Expressions that are evaluated solely for their value, and not for any other effects of the computation, are said to be *functional*.

Some programming languages, such as Scheme, are *expression oriented*; their programs are constructed of definitions and expressions; there are no statements.

(D. P. Friedman, M. Wand, and C. T. Haynes
Essentials of Programming Languages
The MIT Press and McGraw-Hill, 1992)

The *design language* PLaSM, which this book aims to describe, is a geometry-oriented extension of a subset of FL, an advanced language for programming at *Function Level*, developed by the Functional Programming Group of IBM Research Division at Almaden [BWW90, BWW⁺89]. The design language PLaSM, whose name stands for “Programming LAnguage for Symbolic Modeling”, was developed [PPV95] by the CAD Group at the University “La Sapienza” and then supported and further developed at “Roma Tre” University in Rome. Such language is strongly influenced by FL; actually, it can be considered a geometry-oriented extension of a FL subset, with only a few small syntactical differences. In the present chapter a short outline of the FL approach to functional programming is given, together with an introduction to PLaSM and to its geometric operators. The chapter is aimed at discussing the language

syntax and at getting started with a working system. In the last part of the chapter we introduce the first examples of *geometric programming*. The main goal of the chapter is to give the flavor of language style and expressive power. Therefore, many concepts are introduced informally here and defined carefully in later chapters.

1.1 Introduction to symbolic design programming

The FL language, on the line traced by the Backus' Turing lecture [Bac78], introduces an algebra over programs, where a set of algebraic identities between functional expressions is established. Such an algebraic approach to programming allows, among several other interesting features, formal reasoning about computer programs. Furthermore, programs are easily combined, so that new programs are obtained from simpler ones in a easy and elegant way. Also, it is possible to find simpler equivalent programs, both at the design and at compiling stages. Great advantages are thus obtained in the style and efficiency of program prototyping.

More generally, it is well known that functional programming enjoys several good properties:

1. The set of syntax rules of a functional language is very small.
2. Each rule is very simple.
3. The program code is terse and clear.
4. The meaning of a program is well understood, since there is no state.
5. Functions may be used both as programs and as data.
6. Programs are easily connected by concatenation and nesting.

The PLaSM language was designed upon the main assumption that a functional computing environment is *the* natural environment for geometric computations and generation of geometric models of shape. In fact, a complex geometric shape is often constituted by an assembly of components, which are highly dependent on each other. In particular:

1. Components may result from computations invoking other components.
2. Parameterized generating functions may be associated with each component.
3. Geometric expressions are the best candidates to produce actual parameter values when generating assembly components.

Our design language, strongly inspired from FL, can therefore evaluate *geometric expressions*, that are expressions whose value is a polyhedral complex, i.e. a set of polyhedral point sets. It is also able to combine functions to produce higher-level functions in the FL style. Beyond the adopted approach to programming at *Function Level*, which allows computing with functions as well as with numbers, we note the unique design choice of dealing only with a *dimension-independent* implementation of geometric data structures and algorithms. The first feature results in a very natural approach to parametric geometry. The second feature, coupled with the “combinatorial engine” of FL, gives the language an amazing descriptive power in computing with geometry.

1.1.1 Computational model

In this section a first introduction to our approach to symbolic design programming is given. A more detailed discussion of language syntax and semantics is left to the following sections and chapters.

Programs are functions Generally speaking, a program is a *function*. When applied to some input *argument*, a program produces some output *value*. Two programs are usually connected by using functional composition, so that the output of the first program is used as input to the second program.

Program composition and application The composition of PLaSM programs behaves exactly as the standard composition of mathematical functions. For example, the application of the compound mathematical function $f \circ g$ to the x argument

$$(f \circ g)(x) \equiv f(g(x))$$

means that the function g is first applied to x and that the function f is then applied to the value $g(x)$. The PLaSM notation for the previous expressions will be

$$(f \sim g) : x \equiv f : (g : x)$$

where \sim stands for function *composition* and where $g:x$ stands for *application* of the function g to the argument x .

Naming objects In PLaSM, a name can be assigned to every value generated by the language, by using a DEF construct, either with or without explicit parameters. In both cases the so-called *body* of the definition, i.e. the expression which follows the definition *head*, at the right hand of the “ $=$ ” symbol, will describe the computational process which generates the value produced by the computation. The parameters which it implicitly/explicitly depends on may be embedded in such a definition.

For example we may have

```
DEF object = (Fun3 ~ Fun2 ~ Fun1):parameters;
```

The computational process which produces the *object* value can be thought as the computational pipeline shown in Figure 1.1.

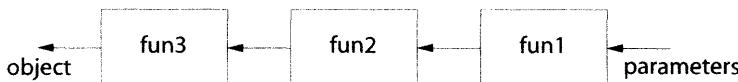


Figure 1.1 Example of computational pipeline

In the previous example the dependence of the model upon the parameters is implicit. In order to modify the generated object value it is necessary (a) to change the source code in either the body or the local environment of its generating function; (b) to compile the new definition; and (c) to evaluate again the object identifier.

Parametrized objects A *parametric* geometric model can be defined, and easily combined with other such models, by using a generating function with formal *parameters*. Such kind of function may be instantiated with different actual *arguments*, thus obtaining different output values. For example, we may have

```
DEF object (params::IsSeq) = (Fun3 ~ Fun2 ~ Fun1):params;
DEF obj1 = object:< p1, p2, ..., pn >;
DEF obj2 = object:< q1, q2, ..., qn >;
```

It is interesting to note that the generating function of a geometric model may accept parameters of *any* type, including other geometric objects.

1.2 Getting started with PLaSM

When taking the first steps with a new computer language, it is useful to carefully introduce the operations to start. In particular, in this section we discuss step-by-step how to download the PLaSM design environment from the web, its installation on the desktop machine, and the first computational experiences with it.

1.2.1 Installing the language

The integrated PLaSM “design environment” consists at least of a language *interpreter*, and may contain a source *editor*, a local or remote language *server*, and the preferred web *browser* enriched with one or more graphics plug-ins. PLaSM is available for all the *Windows* versions, all the brands of *GNU/Linux*, and the Apple’s *Mac OS X* operating system.

Software download

The current PLaSM interpreter is written in Scheme and C++, by using a first-class multi-platform Scheme implementation called PLT Scheme.¹ The first task for the user is to download the PLaSM interpreter and, possibly, other useful softwares from the web sites they reside on.

1. Interpreter

The PLaSM interpreter is located at the web address

<http://www.plasm.net/download/>

The typical user may like to get the binary executables for the preferred computational environment. Conversely, the advanced user might prefer to get the sources and to recompile and build the interpreter.

2. Editor

An integrated language editor is not strictly required, since a standard text editor would be sufficient, but it may be very useful. The specialized PLaSM

¹ The common page of the PLT Group (Programming Language Team Group), currently disseminated in several universities, is <http://www plt-scheme.org/>.

editor *Xplode*, which stands for “*Is a PLasm Open Design Environment*”, will provide for syntax coloring, tab completion, menus of available libraries and functions, quick documentation and direct evaluation of every sub-expression. A standard version is already integrated in the interpreter package.²

3. *Browser plug-ins*

The PLaSM environment does not currently offer an integrated viewer for graphics data. Conversely, it allows the user to export the geometric objects generated by the language into some largely diffused web standard formats, including VRML (Virtual Reality Modeling Language) for 3D graphics, as well as SVG (Scalable Vector Graphics) and Flash for 2D graphics. The reader may find large collections of VRML resources on the web.³ The browser plug-ins for .svg (SVG) and .swf (Flash) files can be downloaded from the *Adobe*⁴, and the *Macromedia*⁵ web sites, respectively.

Software installation

A minimal PLaSM system is constituted by a language client connected to a remote server and by a web browser with a VRML viewer installed. A full PLaSM environment contains a local language server, an integrated editor and more plug-ins for supported graphics formats.

The following instructions are for all *Windows* environments. For other platforms follow the installation guidelines given on the language site.

1. *Language server*

The installation on the local machine is very simple: double-click on the file **plasm.exe**⁶ and answer the installer requests.

2. *Language client and editor*

The language client and integrated editor *Xplode* are already installed automatically in the previous step. Launch *Start* → *Programs* → *Plasm* → *Xplode* to start working. If some problems arise, the editor will ask to browse within the disk and to show where the server is located.

3. *Plug-ins*

Follow the installation instructions given at the web sites of the graphics plug-ins used, normally to be installed within a web browser.

² The PLaSM editor can be also downloaded separately, for each of the supported platforms.

³ <http://www.web3d.org/vrml/vrml.htm>

⁴ <http://www.adobe.com/svg/viewer/install/>

⁵ <http://www.macromedia.com/software/flash/download/>

⁶ In *Mac OS X* and *Linux*, the installation package is called **plasm.pkg** and **plasm.rpm**, respectively.

1.2.2 Using the language

A typical PLaSM session consists in reading/writing/editing definitions, evaluating definitions and/or expressions, exporting geometric values to external files, saving/restoring geometries to/from XML files, and in visualizing the contents of graphic files.

Getting started

In order to check if the language environment has been installed correctly, try generating a red cube and displaying it within your web browser. To do this:

1. Launch the *Xplode* editor from either the *Start → Programs → Plasm → Xplode* menu (on *Windows*) or from the *Applications* folder (on *MacOS X*).
2. Write on the superior editor window the following code:

```
DEF mycube = CUBOID:<1,1,1> COLOR RED;
mycube;
```

then either launch the menu item *PLaSM → Evaluate Buffer* or hit <**ctrl**>B (on *Windows*) or <**cmmnd**>B (on *MacOS X*).

3. The listener should write, on the inferior window, the message:

```
=====
mycube DEFINED
=====

PolComplex < 3 , 3 >
$ < < 'RGBcolor' , < 1 , 0 , 0 > > >
```

that acknowledges the correct definition of the `mycube` symbol and tell the user about the type of the expression evaluated on the last input line.

4. Select the `mycube` symbol using the mouse.
5. Either launch the menu item *PLaSM → Vrml Export* or hit <**ctrl**>M (on *Windows*) or <**cmmnd**>M (on *MacOS X*). Answer *OK* to the dialog window asking for confirmation on the symbol/expression to evaluate and export.
6. Insert, in the exporting dialog window the filename `mycube.wrl` and select the directory where to export the generated file. You should own the writing permissions on such directory.
7. Load the `mycube.wrl` file within your browser, where you must have previously installed and configured a VRML plug-in.
8. Enjoy rotating and scaling your first geometric model!

The remainder of this section is quite technical, and can be avoided at a first reading by a non-computer science-oriented reader. That kind of reader can go directly to Section 1.3. He or she will go back when willing to understand what is going on within the language design environment and exporter.

Interacting with the language

First of all, the reader must understand that every PLaSM source program is a *string*, i.e. a sequence of characters enclosed between (double) quotes, that must be processed

by the language interpreter.

The PLASM interpreter was previously written in Common Lisp, and is currently implemented in MzScheme (“Miss Scheme”), a first-class implementation [FFFK98] of Scheme, which is a simple, powerful and beautiful Lisp dialect [SF97, FH92].

Evaluating expressions The processing of every PLASM program requires the evaluation of a Scheme expression. For example, the evaluation of the PLASM expression

```
(SQRT:3 + 10) / (3.5 * COS:(PI/4))
```

where SQRT:3 stands for $\sqrt{3}$ and COS:(PI/4) stands for $\cos \frac{\pi}{4}$, as well as the evaluation of every other PLASM program, requires the processing of a source string, in this case

```
"(SQRT:3 + 10) / (3.5 * COS:(PI/4))"
```

The translation of such a source string into some machine-executable code is done by asking the language interpreter to evaluate a Scheme form (plasm " ... "). Here, we must evaluate the Scheme expression:

```
(plasm "(SQRT:3 + 10) / (3.5 * COS:(PI/4))")
```

The wrapping of a PLASM expression as a string within a Scheme expression is not needed when an integrated PLASM editor is available. Only in that case, can the user write some PLASM code and directly evaluate every expression or sub-expression in it, simply by:

1. Selecting the text portion he or she wants to evaluate.
2. Executing the menu item *PLASM → Evaluate expression*, or by pressing the corresponding keyboard shortcut.

Clearly the completion of the PLASM expression as one of the two equivalent Scheme expressions

```
(plasm " ... ")
(eval-plasm " ... ")
```

will in this case be automatically performed by the editor.⁷

Scripting A source program, or *script*, is constituted by one or more PLASM definitions and expressions. A script is normally stored in a file with suffix .psm, that stands for “plasm”.

A script is loaded within the run-time environment as the result of the application of the load operator on the PLASM string that contains the name, and possibly the path, of the script file. For example, the script contained within the file example.psm of the subdirectory path of the current directory is loaded as result of the evaluation of

```
(plasm " LOAD:'path/example.psm' ")
```

Notice that a PLASM string, and in particular a file name, is a sequence of characters enclosed between *single quotes*.

⁷ The two forms are needed for backward compatibility with former interpreter versions.

Several interpreter actions may be requested by the user, by just including the `(plasm " ... ")` expressions within a `(begin ...)` environment. For example, we may write:

```
(begin
  (plasm " LOAD:'script1.psm' ")
  (plasm " DEF fun1 (a::Isreal) = (SIN * COS):a ")
  (plasm " fun1:(PI/6) + fun1:(PI/4) ")
)
```

In order to be *loadable* as described above, a script file must contain a Scheme expression, i.e. it should contain either `(plasm " ... ")` or `(begin ...)`. Scheme expressions like the above one will be usually substituted by simpler equivalent expressions, which contain a *unique* source string. Therefore, in order to write a loadable file, the user must type within the file:

Script 1.2.1

```
(plasm "
  LOAD:'script1.psm'
  DEF fun1 (a::Isreal) = (SIN * COS):a ;
  fun1:(PI/6) + fun1:(PI/4)
")
```

Conversely, when using the integrated PLaSM editor, the previous Scheme expression is substituted by the *contents* of the source string, i.e. by a set of PLaSM definitions and/or expressions. So, within *Xplode*, the user must write and evaluate the following code:

Script 1.2.2

```
LOAD:'script1.psm' ;
DEF fun1 (a::Isreal) = (SIN * COS):a ;
fun1:(PI/6) + fun1:(PI/4)
```

In the remainder of the book we normally make use of this simpler approach. Only occasionally, some PLaSM expressions will be embedded in their ground Scheme forms.

The reader should notice that every pair of adjacent definitions and/or expressions — within the same source string — is separated by a semicolon. A final semicolon to terminate the last expression or definition is optional. It may be useful to insert such a final semicolon when the program script is still under development, in order to avoid the compilation error generated when two subsequent expressions are not separated by a semicolon.

Listener Every time the user requires the evaluation of some PLaSM code, the interpreter generates an internal value for each of the processed definitions and expressions, and at the same time produces some log on the screen of the action, type or value generated. If the specialized *Xplode* editor is available, such a screen

echo is sent into a separate window, called the *listener*; otherwise, the log messages are sent to the window of the language server.

Libraries A predefined set of function definitions, when recognized to have some general utility, may be defined as a *library*. Libraries are .psm files stored in the special subdirectory `plasm/psmlib` and are loaded by using either the Scheme form

```
(plasm " LOADLIB:'libname' ")
```

in the PLaSM interpreter, or the PLaSM expression

```
LOADLIB:'libname';
```

in the integrated editor. In both cases the library name is given without file path and suffix. The remarkable difference between libraries and standard script files concerns the symbols defined in libraries, which — after the library loading — become *protected* by the language environment, and cannot be changed nor redefined by the user. This design choice is supposed to help avoiding naming conflicts, which might produce unpredictable mistakes. Currently, all libraries are loaded when starting the interpreter. The user may easily change this behavior, by either preventing the loading of some libraries, or by loading them as non protected at set-up, or by loading some libraries only on request during a work session.

Exporting graphics

The PLaSM environment does not currently contain an integrated viewer of graphics data, but may export geometric objects as data files to various well-known graphics formats, as summarized below. In general, PLaSM is committed to support graphics web standards.

VRML The *Virtual Reality Modeling Language* (VRML) is the ISO standard for 3D graphics on the *World-Wide Web*.⁸ In particular, the geometric value of the PLaSM object named `out` can be exported to the file `out.wrl`, using the current VRML standard coding, as the effect of evaluating the statement

```
VRML: out: 'out.wrl';
```

Flash The .swf file format by *Macromedia's Flash* is the *de facto* industrial standard for 2D vector graphics animation and streaming on the World-Wide Web. A .swf file with the contents of the `out` object, reversed into a drawing area wide 150 pixels, may be produced as:

```
... some PLaSM code ...
DEF out = ...

flash: out: 150: 'out.swf' ;
```

⁸ New updated versions of VRML are currently under development by the W3C consortium, and are called VRML2000 and x3D, which stands for “VRML coded in XML”.

SVG The *Scalable Vector Graphics* (SVG) is the ISO draft proposal developed by w3C for 2D vector graphics on the web. The exporting into the `out.svg` file of the geometric value of the 2D PLaSM object named `out`, assuming a drawing area with 15 centimeters of width, is obtained by

```
svg: out: 15: 'out.svg';
```

Saving/restoring geometry

A very useful feature of the current PLaSM version is the ability to save and restore geometric objects into/from external files by using a XML coding. The XML (Extensible Markup Language), can be considered the best method known today for putting structured data in a text file. It is a subset of the text processing international standard SGML (Standard Generalized Markup Language) (ISO 8879), specialized for use on the World-Wide Web.

Thus, large and computing intensive models can be generated only once, stored in external files and directly restored in memory in subsequent work sessions with PLaSM. Even more, the XML coding can be very useful when storing geometric objects in a database or when exchanging geometries across a computer network, within a collaborative and spatially distributed design environment.

The syntax to save/restore the geometric value associated to the `object` symbol is

```
DEF object = ... ;
save: object:'path/object.xml';
```

in the saving session, whereas in the restoring session the saved `object` must be defined as:

```
DEF object = open:'path/object.xml';
```

1.3 Programming at Function Level

PLaSM can be considered a geometry-oriented extension of a subset of the FL language [BWW90, BWW⁺89], which is a pure functional language based on combinatorial logic. In particular, the FL language makes use of both pre-defined and user-defined *combinators*, i.e. higher-order functions which are applied to functions to produce new functions. The small but very significant FL subset which is used as the base environment of PLaSM is summarized in this section.

Notice that here and in the remainder of this book the infix symbol \equiv is normally used to tell the reader that the *expression* on its left side evaluates to the *value* on its right side. Sometimes this symbol is also used to denote an equivalence between syntactical forms.

1.3.1 Elements of FL syntax

Primitive FL *objects* are characters, numbers and truth values. Primitive objects, functions, applications and sequences are *expressions*. *Sequences* are expressions separated by commas and contained within a pair of angle brackets:

`<5, fun>`

An *application* expression `exp1:exp2` applies the *function* resulting from the evaluation of `exp1` on the *argument* resulting from the evaluation of `exp2`. Notice that binary functions can also be used in infix form:

`1 + 3 ≡ +:<1,3> ≡ 4`

Application associates to left, i.e. a sequence of repeated applications is evaluated from left to right. Notice that this is only possible if all the applications, but possibly the last one, generate a new function to be applied to the next argument:

`f:g:h ≡ (f:g):h`

Application binds stronger than composition, i.e. applications are evaluated first before compositions, as is shown in the following example:

`f:g ~ h ≡ (f:g) ~ h`

1.3.2 Combining forms and functions

The function level approach to programming of FL emphasizes the definition of new functions by combining existing functions in various ways. The result of this approach is a programming style based on function-valued expressions. Some more important FL *combining forms* and functions follow.

Construction The combining form `CONS` allows application of a sequence of functions to an argument producing the sequence of applications:

`CONS:<f1, ..., fn>:x ≡ [f1, ..., fn]:x ≡ <f1:x, ..., fn:x>`

A CONSed sequence of functions is a sort of *vector function*, that can be composed with other functions and that can be applied to data. E.g. `cons:<+, ->`, written also `[+, -]`, when applied to the argument `<3, 2>` returns the sequence of applications

`[+, -]:<3,2> ≡ <+:<3,2>, -:<3,2>> ≡ <5,1>`

Apply-to-all The combining form `AA` has a symmetric effect, i.e. it applies a function to a sequence of arguments giving a sequence of applications

`AA:f:<x1, ..., xn> ≡ <f:x1, ..., f:xn>`

For example, we may apply the `SIN` function to all the elements of a list of numeric expressions:

```
AA:SIN:< 0, PI/3, PI/6, PI/2 >
≡ < SIN:0, SIN:(PI/3), SIN:(PI/6), SIN:(PI/2) >
≡ < 0 , 0.8660254037844382 , 0.4999999999999956 , 1.0 >;
```

The reader should notice that numeric computations often introduce round-off and approximation errors. Just remember that π is an irrational number and cannot be represented exactly by using finite precision arithmetic. Also, functions like `SIN` are computed by using some truncated series expansion.

Identity The function ID returns its argument unchanged

$$\text{ID}:x \equiv x$$

In other words, the application of the identity function to *any* argument, gives back the same argument:

$$\begin{aligned}\text{ID}:0.5 &\equiv 0.5 \\ \text{ID}:\text{SIN} &\equiv \text{SIN} \\ \text{ID}:\text{SIN}:0 &\equiv \text{SIN}:0 \equiv 0 \\ \text{ID}:'\text{out.wrl}' &\equiv '\text{out.wrl}'\end{aligned}$$

Constant The combining form K is evaluated as follows, for whatever x_2 :

$$K:x_1:x_2 \equiv x_1;$$

In other words, the first application returns a constant function of value x_1 , i.e. such that when applied to *any* argument x_2 , *always* returns x_1 . Some concrete examples follow:

$$\begin{aligned}K:0.5 &\equiv \text{Anonymous-Function} \\ K:0.5:10 &\equiv 0.5 \\ K:0.5:100 &\equiv 0.5 \\ K:0.5:\text{SIN} &\equiv 0.5\end{aligned}$$

Composition The binary composition of functions, denoted by the symbol “~”, is defined in the standard mathematical way:

$$(f \sim g):x \equiv f:(g:x)$$

n-ary composition of functions is also allowed:

$$\text{COMP}:< f, g, h >:x \equiv (f \sim g \sim h):x \equiv f:(g:(h:x))$$

In this case we have, where PI, COS and ACOS are the PLaSM denotations for π , cos and arccos, respectively:

$$\begin{aligned}(\text{ACOS} \sim \text{COS}):PI &\equiv \text{ACOS}:(\text{COS}:PI) \equiv \text{ACOS}:-1 \equiv 3.141592653589793 \\ (\text{COS} \sim \text{ACOS}):-1 &\equiv \text{COS}:(\text{ACOS}:-1) \equiv \text{COS}:3.141592653589793 \equiv -1 \\ (\text{ACOS} \sim \text{COS} \sim \text{ACOS}):-1 &\equiv (\text{ACOS}:(\text{COS}:(\text{ACOS}:-1))) \equiv 3.141592653589793\end{aligned}$$

Conditional The conditional form IF:< p, f, g > is evaluated as follows:

$$\begin{aligned}\text{IF}:< p, f, g >:x &\equiv f:x \quad \text{if } p:x \equiv \text{TRUE} \\ &\equiv g:x \quad \text{if } p:x \equiv \text{FALSE}\end{aligned}$$

Notice that both the predicate⁹ p, as well as f and g, to be alternatively executed depending on the truth value of the expression p:x, must be all *functions*. E.g., we have:

$$\begin{aligned}\text{IF}:< \text{IsIntPos}, \text{K:True}, \text{K:False} >:1000 &\equiv \text{True} \\ \text{IF}:< \text{IsIntPos}, \text{K:True}, \text{K:False} >:-1000 &\equiv \text{False}\end{aligned}$$

where IsIntPos is a predefined predicate that returns True when applied to some positive integer.

⁹ A *predicate* is a function $p : \text{Dom} \rightarrow \{ \text{True}, \text{False} \}$, where Dom is any set. Both True and False are called *truth values*, and also *Boolean values*.

Insert Right/Left The combining forms `INSR` and `INSL` allow the user to apply a *binary* function f , with signature¹⁰ $f : D \times D \rightarrow D$, on a sequence of arguments of *any* length n . Notice that in the right-hand expressions below, f is applied to a *pair* of arguments:

```
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 $\text{INSR}:f:<x> \equiv \text{INSL}:f:<x> \equiv x$.

An interesting example of use of the `INSL` combinator is given below, where the function `bigger : Num × Num → Num` is defined, being the syntax of definitions explained in detail in the next section. Notice that `IsNum` is a predicate used to check at run-time if the arguments of function application are of the correct type. The `bigger` function returns the one of its *two* arguments with maximum value; the `biggest` function does the same from a list of arguments of *arbitrary length*:

```
DEF bigger (a,b::IsNum) = IF:< GT:a, K:b, K:a >:b;
DEF biggest = INSL:bigger;

bigger:<-10, 0> ≡ 0
biggest:<-10, 0, -100, 4, 22, -3, 88, 11 > ≡ 88
```

Catenate The `CAT` function 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>
```

A pair of concrete examples of how the `CAT` function is used follows. The second one is quite interesting: it gives a *filter* function used to select the non-negative elements of a number sequence:

```
CAT:<<1,2,3>,<11,12,13>,<21,22,23>> ≡ <1,2,3,11,12,13,21,22,23>
(CAT ~ AA:(IF:< LT:0, K:<>, [ID] >)):< -101, 23, -37.02, 0.1, 84 >
≡ CAT:<<>,<23>,<>,<0.1>,<84>>
≡ <23, 0.1, 84>
```

It may be very useful to *abstract* a *filter* function with respect to a generic predicate and to a generic argument `sequence`, by giving a function *definition*:

Script 1.3.1 (Filter function)

```
DEF filter (predicate::IsFun) (sequence::IsSeq) =
  (CAT ~ AA:(IF:< predicate, K:<>, [ID] >)):sequence;
```

Two examples of application of the `filter` function to actual arguments follow. Notice that the two applications respectively return the positive and the negative elements of the input sequence. Remember that a sequence of applications is *left-associative*:

¹⁰ The *signature* of a function f from a *domain A* to a *codomain B* is the ordered pair of sets (A, B) . It is normally associated to f by writing $f : A \rightarrow B$.

```

filter:(LE:0):< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ (filter:(LE:0)):< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ Anonymous-Function:< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ <23, 0.1, 84>

filter:(GE:0):< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ (filter:(GE:0)):< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ Anonymous-Function:< -101, 23, 0, -37.02, 0.1, 84 >
  ≡ <-101, -37.02>

```

Distribute Right/Left The functions DISTR and DISTL are defined as:

```

DISTR:<<a,b,c>, x> ≡ <<a,x>, <b,x>, <c,x>>
DISTL:<x, <a,b,c>> ≡ <<x,a>, <x,b>, <x,c>>

```

They accordingly transform a *pair*, constituted by an arbitrary expression *x* and by an arbitrary sequence, into a *sequence of pairs*.

Let us give an example of use. The Euler number *e* is defined as the sum of a series of numbers. In particular:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots$$

We compute an *approximation* of *e*, named **euler**, as the sum of the first 21 terms of the series above. The definition of the factorial function **fact** is given in Script 2.1.6. Notice that the **+** operator may be applied to a sequence of numeric arguments and remember that $0! = 1$.

```

DEF euler = (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:(0..20) >

euler ≡ (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:(0..20) >
  ≡ (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:< 0, 1, 2, ..., 8, 9 > >
  ≡ (+ ~ AA:/ ~ DISTL):< 1.0, < fact:0, fact:1, ..., fact:8, fact:9 > >
  ≡ (+ ~ AA:/ ~ DISTL):< 1.0, < 1, 1, 2, 6, ..., 40320, 362880 > >
  ≡ (+ ~ AA:/): (DISTL:< 1.0, < 1, 1, 2, 6, ..., 40320, 362880 > >)
  ≡ (+ : (AA:/: < <1.0, 1>, <1.0, 1>, ..., <1.0, 40320>, <1.0, 362880> >))
  ≡ + : < 1.0/1, 1.0/1, 1.0/2, 1.0/6, ..., 1.0/40320, 1/362880 >
  ≡ 2.7182815255731922

```

Above we have seen our first important example of FL computation as a sequence of expression transformations using the defining rules of the combinators. The order of transformations is induced by the parenthesis included into an expression. The sub-expressions nested more deeply are transformed first.

A simpler and more elegant implementation of the Euler number is given in Script 1.3.2, where C is the currifier combinator discussed in Section 1.4.3.

Script 1.3.2 (Euler number)

```
DEF euler = (+ ~ AA:(C:/:1.0 ~ fact)):(0..20)
```

Example 1.3.1 (Conditional operator)

As we have seen, the conditional form `IF:<p,f,g>:data` has the following semantic: “*IF* the predicate `p` applied to `data` is true, *THEN* apply `f` to `data`; *ELSE* apply `g` to `data`”. This construct is very useful when it is necessary to apply different actions to input data depending on the value of some predicate evaluated on them, and is probably more “natural” than the conditional statements available in other languages.

From a syntactical viewpoint, notice that the `IF` operator is a higher-order function that *must* be applied to a *triplet of functions* in order to return a function which is in turn applied to the input data.

Such a syntax and semantics of the `IF` operator can be demonstrated by the following code, where a string is generated depending on the truth value of a simple predicate. The reader should notice that the result of evaluating both the expressions `K:'True'` and `K:'False'` is a (constant) function.

```
IF:< EQ, K:'True', K:'False' >:<10, 20> ≡ 'False'  
IF:< EQ, K:'True', K:'False' >:<20, 20> ≡ 'True'
```

1.4 Basics of PLaSM programming

The PLaSM language was designed to introduce a well-founded programming approach to *generative modeling*, where geometric objects are generated by invoking the generating functions with *geometric expressions* as actual parameters. This is achieved by allowing for a sort of *geometric calculus* over embedded polyhedra. For this purpose the language contains in its kernel a *dimension-independent* approach to the representation of geometric data structures and algorithms.

The programming approach to geometric design enforced by PLaSM makes it possible to manage a sort of extended *variational geometry* [LG82], where classes of objects with varying topology and shape are parametrized by some language function and handled and combined as a whole. The language can be considered a *geometry-oriented extension* of a quite small subset of the functional language FL, where the *validity* of geometry is always syntactically guaranteed. In other words it is guaranteed that any well-formed language expression which generates a polyhedrally-typed data object always corresponds to some valid internal data structure.

Such a functional and dimension-independent approach to geometric design achieves a *representation domain* broader than usually done by standard solid modelers, so that points, wire-frames, surfaces, solids and even higher dimensional manifolds can be suitably combined or blended altogether.

1.4.1 Expressions

The syntax of PLaSM is very similar to the syntax of FL (see Section 1.3.1). For the reader experienced in FL we may say that the differences mainly concern the meaning of few symbols (composition and constant operators) and the lack of pattern matching in the definition of type predicates. In particular, the PLaSM language:

1. uses a case-insensitive alphabet;
2. allows for overloading of some (pre-defined) operators;
3. evaluates expressions which return a polyhedral complex as value;

4. produces higher-level functions in the FL style;
5. allows for partially specified (curried) functions.
6. introduces two new combining forms **AC** (*apply-in-composition*) and **AS** (*apply-in-sequence*) to use a function with a number of specification parameters greater than the number used in its definition.

Unlike FL, the PLaSM language does allow the use of identifiers for any language object, including numbers, strings and polyhedral complexes, but does not provide pattern matching to identify the sub-components of a data structure. The current PLaSM implementation also provides free nesting of local functional environments.

1.4.2 User-defined functions

User-defined functions as recognized by PLaSM are either *global* or *local*. Global functions are also called top-level functions.

Global functions A global function definition always contains a *function head* separated from the *function body* by the = character. The body is optionally followed by a *local environment*, which contains one or more local function definitions separated by commas and enclosed between a pair of **WHERE** and **END** keywords. Global functions are allowed to contain formal parameters in the head. Formal parameters are specified together with a predicate which is used to perform dynamic type checking at run-time.

Global function definitions are separated by ";" (semicolon) punctuation.

Local functions The visibility of local functions is restricted to the scope of a global function. No formal parameters are allowed in a local function definition. In such a case an identifier is associated by equality to an expression. The value resulting from the expression evaluation is returned by invoking the name. Such a value may be any PLaSM object, including numbers, functions, sequences, polyhedral complexes, etc.

Local function definitions are separated by "," (comma) punctuation.

A top-level function template in the general case is the following:

```
DEF global (p1::IsType1; p2::IsType2) (q1,q2::IsType3) = Body
WHERE
    local1 = body1,
    local2 = body2
    WHERE local21 = body21 END
END;
```

where **DEF**, **WHERE** and **END** are reserved keywords; **global** is the name of the top-level function; **Body** is any PLaSM expression which is used to compute the function values; **p1**, **p2** and **q1**, **q2** are formal parameters; **IsType1**, **IsType2** and **IsType3** are predicates used for dynamic type checking of actual values of parameters at function invocation time; **local1** and **local2** are the names of two local functions; **body1** and **body2** are any PLaSM expression used for computing their values.

Notice that local function definitions are separated by commas. Formal parameters to be checked by the same predicate are separated by commas. The triplet

“parameters”, “::”, “IsType“, where IsType is any PLaSM predicate, is called *parameter sublist*. A *parameter list* contain one or more parameter sublists separated by semicolons and enclosed in round brackets.

A global function head may contain zero, one or more parameter lists. Correspondingly, the function is invoked with zero, one or more applications to *actual arguments*. The presence of more than one parameter list is allowed to permit specification of *partial functions*.

Function definition It is useful to look at some different global function definitions, given for the sake of simplicity without any local functional environment.

```
DEF f (a::type1) = bodyF;
DEF g (a1,...,an::type2) = bodyG;
DEF h (a1::type1)(a2::type2) = bodyH;
DEF l (a1::type1; a2,a3::type2) = bodyL;
DEF m = bodyM;
```

Function invocation The functions previously defined are correspondingly invoked with one, more or zero application to actual arguments, depending on the number of parameter lists in their definition:

```
f:x ;
g:<y1, ... , yn > ;
h:x1:y2 ;
l:<x1, y2,y3 > ;
m ;
```

Example 1.4.1 (Function definition and invocation)

As simple and useful examples of function definitions, we give in Script 1.4.1 two predicates IsEven and IsOdd that return True when applied to some even or odd argument, respectively, and False otherwise. Examples of function application using the CONS combinator are given in the last two rows of the script.

Script 1.4.1 (IsEven and IsOdd predicates)

```
DEF IsEven (n::IsInt) = IsInt:(n / 2);
DEF IsOdd = NOT ~ IsEven;

[IsEven, IsOdd]: 100 ≡ < True , False >
[IsEven, IsOdd]: 101 ≡ < False , True >
```

1.4.3 Built-in functions

Several non-geometric functions and combining forms are predefined in PLaSM, and are introduced in this section. Various simple examples are also given. The reader is strongly encouraged to explore the language by trying similar examples, possibly with the insertion of small changes.

Type predicates A *predicate* is a function which returns either **True** or **False** when applied to actual arguments. We conventionally start the name of most predicates with the “Is” substring. A quite large set of predefined predicates to be used for dynamic type checking is available. For example:

```
IsInt, IsIntPos, IsReal, IsRealPos, IsSeq, IsFun, IsPol, IsChar, IsString
```

return **True** when applied to integers, positive integers, reals, positive reals, sequences, functions, polyhedra, characters and strings, respectively.

```
IsInt: 1000 ≡ True
IsIntPos: -123 ≡ False
IsReal: 345.28 ≡ True
IsReal: 3.4528E+2 ≡ True
IsRealPos: 3.4528E-2 ≡ True
IsSeq: < 345.28, -123, SQRT, 'aaaaa' > ≡ True
IsFun: ID ≡ True
IsPol: (CUBOID:<1,1,1>) ≡ True
IsChar: 'a' ≡ True
IsString: 'Alberto' ≡ True
```

A second-order function **IsSeqOf** must be applied to a predicate and then to a sequence of objects or expressions. For example:

```
IsSeqOf: IsInt: < 10, 2, (12 - 8)/2, 99 + 100, 0 > ≡ True
IsSeqOf: IsInt: < 10, 2, (11 - 8)/2, 99 + 100, 0 > ≡ False
```

The following equivalence holds:

```
IsSeqOf: predicate ≡ AND ~ AA: predicate
```

Logical functions and constants The truth values *True* and *False* are respectively denoted as **TRUE** and **FALSE**. According to the **Lisp** approach, the empty sequence **<>** may be also used for *False*. The available logical operators are the standard ones:

AND, OR, NOT

Some examples of logical expressions follows:

```
AND: < True, True, True, True > ≡ True
AND: < True, True, False > ≡ False
OR: < True, False, True > ≡ True
OR: < False, False, False, False > ≡ False
NOT: False ≡ True
AND: < True OR NOT:True, NOT:False > ≡ True
```

Comparison functions A standard set of comparison predicates between numeric expressions is given, where

GT, GE, LT, LE, EQ

respectively stand for “greater than”, “greater or equal”, “less than”, “less or equal”. “equal”.

All such operators compare two arguments, but there is a significant difference in their use. The first four are second-order predicates. EQ is conversely a standard binary predicate. Hence they must be used as follows:

GT:1:2 ≡ True	to read: "is 'greater than 1' 2 ?"
GE:1:2 ≡ True	
LE:1:2 ≡ False	
LE:2:(20 / 10) ≡ True	
EQ:< 1, 5 - 3 > ≡ False	
EQ:< 6 / 3, 2 > ≡ True	

Comparison functions are often used together with logical operators to construct logical expressions. Syntactically equivalent forms of same logical expressions or functions follow:

```
GE:0:0.5 AND LE:1:0.5 ≡ True
(GE:0 AND LE:1): 0.5 ≡ True
(GE:0 AND LE:1): -0.5 ≡ False
(AND ~ [GE:0, LE:1]): 0.5 ≡ True
(NOT ~ AND ~ [GE:0, LE:1]): 0.5 ≡ False
(IsIntPos AND GT:10): 5 ≡ False
(AND ~ [IsIntPos, GT:10]): 50.001 ≡ False
(AND ~ [IsIntPos, GT:10]): 50.0 ≡ True
```

Mathematical functions The standard mathematical functions on numbers are also available. For example

+, -, *, /, **, SIN, COS, TAN, ATAN, LN, EXP, SQRT, ...

denote respectively the algebraic operators, the power raising, the trigonometric functions sin, cos, tan, the arc associated with a given tangent, the logarithm in basis e , the e^x function, the square root, etc. The symbol PI is the PLaSM denotation of the π number. The ** operator denotes power raising. In other words, the x ** y expression stands for x^y . The reader should notice that every arithmetic operator, except **, are n -ary, i.e. may be applied to non-empty sequences of arguments. Some examples of use follow. Notice also their *overloading* as algebraic operator between functions.

```
3 + 4 ≡ 7
+:< 1,2,3,4 > ≡ 10
-:< 1,2,3,4 > ≡ -8
*:< 1,2,3,4 > ≡ 24
/:< 2,2,3,4 > ≡ 1/12
PI ≡ 3.14159265358979
SIN:PI ≡ 0.0
COS:PI ≡ -1.0
ACOS:-1 ≡ 3.141592653589793
SQRT:(9 * 9) ≡ 9
(9 * 9) ** (1/2) ≡ 9.0
(9 * 9) ** (-1/2) ≡ 0.1111111111111111
1 / 0.1111111111111111 ≡ 9.0
COS + SIN ≡ Anonymous-Function
(COS + SIN):PI ≡ -1.0
```

Sequence functions Several operations on sequence are built in. In particular:

AL, AR, FIRST, LAST, TAIL, CAT, SEL, LEN, LIST

stand respectively for append left, append right, return the first element, return the last element, return the subsequence from the second to the last element, catenate a sequence of sequences, and select a specified element of a sequence. LEN returns the integer length of the input sequence. LIST encloses its input into a pair of angle brackets.

```

AL:< 99,< 1,2,3,4,5 > > ≡ < 99,1,2,3,4,5 >
99 AL < 1,2,3,4,5 > ≡ < 99,1,2,3,4,5 >
AR:< < 1,2,3,4,5 >,99 > ≡ < 1,2,3,4,5,99 >
< 1,2,3,4,5 > AR 99 ≡ < 1,2,3,4,5,99 >
CAT:<< 101,102 >,< 901,902,903 >> ≡ < 101,102,901,902,903 >
SEL:4:< 101,102,901,902,903 > ≡ 902
(SEL:4 ~ CAT):<< 101,102 >,< 901,902,903 >> ≡ 902
(SEL:4 ~ CAT):<< ID,K >,< COS,SIN,TAN >> ≡ Anonymous-Function
((SEL:4 ~ CAT):<< ID,K >,< COS,SIN,TAN >>):(PI/2) ≡ 1.0
LIST:10 ≡ [ID]:10 ≡ < 10 >
```

String functions The *string*, i.e. an ordered set of characters used as an atomic value, is a primitive data type in PLaSM. A *char* (character) is a string with only one character. Some predefined functions on chars and strings are

ISCHAR, ISSTRING, ORD, CHAR, CHARSEQ, STRING, LT, LE, EQ, GE, GT

whose use is summarized by the following examples. They are further discussed in Section 2.1.4.

```

ISCHAR:'p' ≡ True
ISSTRING:'abcdefghijklm' ≡ True
ORD:'p' ≡ 112
CHAR:112 ≡ 'p'
CHARSEQ:'Homer' ≡ < 'H','o','m','e','r' >
STRING:< 'H','o','m','e','r' > ≡ 'Homer'
LT:'a':'b' ≡ False
GT:'a':'b' ≡ True
```

Sequence generators The FROMTO operator generates integer sequences between two given extreme integers. It is also denoted by an infix double period as an equivalent syntactical form:

2..5 ≡ FROMTO:<2,5> ≡ <2,3,4,5>

Similarly, the unary INTSTO (*Integers to*) operator generates integer sequences starting from 1:

INTSTO:5 ≡ <1,2,3,4,5>

Some further examples are:

```

CAT:< 20..22, 40..42, 60..62 > ≡ < 20,21,22, 40,41,42, 60,61,62 >
(AA:SEL:(5..8)):(CAT:< 20..22, 40..42, 60..62 >) ≡ < 41,42, 60,61 >
(INTSTO ~ LEN ~ CAT):< 20..22, 40..42, 60..62 > ≡ < 1,2,3,4,5,6,7,8,9 >
(STRING ~ AA:CHAR):(65..70) ≡ 'ABCDEF'
```

Repeated instancing The repetition operator `#` allows for instancing n times any given expression:

```
#:3:expr ≡ < expr,expr,expr >
```

Similarly, the catenation of repeated sequences `##` repeats and catenates a given sequence:

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

Selector functions The selector functions, as introduced by FL, are used to respectively select the first, second, or n -th element of a sequence:

```
S1, S2, S3, ..., Sn
```

`SEL` is a primitive (second-order) 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.
```

Extended application Two new useful combining forms `AC`, `AS` are introduced, which respectively stand for *apply-in-composition* and for *apply-in-sequence*, and are defined as follows:

```
AC:f:<x1, ..., xn> ≡ f:x1~...~f:xn
AS:f:<x1, ..., xn> ≡ [f:x1, ..., f:xn]
```

They have several interesting geometric uses. When combined with `AC` or `AS`, the `SEL` operator is modified as follows:

```
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>>
```

Higher-order functions

A function with n parameter lists is called a function of n -th *order*. Such a function, when applied to actual arguments for the first parameter list, returns a function of order $(n - 1)$. This one, when further applied to actual arguments for the second parameter list, returns a function of order $(n - 2)$, and so on. Finally, when all the parameter lists are *bound* to actual arguments, the function returns the value generated by the evaluation of its body. Functions of order higher than one are called *higher-order* functions. The functions returned from the application of higher-order functions to some (ordered) *subset* of their parameter lists are called *partial* functions.

Example 1.4.2 (Partial functions)

The primitive predicates `LT` (Less Than) and `GT` (Greater Than) are defined as second-order functions. Therefore, to test if $a < b$, the user may write one of the two expressions:

```
LT:b:a
GT:a:b
```

In both cases, the first application returns a partial function, i.e. `LT:b` and `GT:a`, to read “Is Less Than b ?” and “Is Greater Than a ?”, respectively. The second application (clearly on the second argument) conversely returns a truth value, i.e. either `True` or `False`.

The possibility of producing partial functions may be very useful. For example, in order to count the number of positive numbers in a sequence of real numbers, we might write:

```
DEF CountPositive = + ~ AA:(IF:< GT:0, K:1, K:0 >);
CountPositive:< 6, -22, 0, 4.25, 999, -33, 0 > ≡ 3
```

Curried functions

An higher-order function g is said to be *curried*, from the name of logician Haskell B. Curry, who introduced this technique, when the following equivalence with a function f exists:

$$f(x_1, x_2, \dots, x_n) \equiv g(x_1)(x_2) \cdots (x_n).$$

In this case g is called a curried version of f , which can be used also when only a subset of f arguments is available. If we look at the *signatures* of the two functions, i.e. to their domain and range sets, then we may write:

$$\begin{aligned} f &: \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_n \rightarrow \mathcal{B} \\ g &: \mathcal{A}_1 \rightarrow \mathcal{A}_2 \rightarrow \cdots \rightarrow \mathcal{A}_n \rightarrow \mathcal{B} \end{aligned}$$

The FL language gives a predefined combinator `C` to currying any function with arguments:

```
f:< x1, x2, ..., xn > ≡ C:f:x1:x2: ... :xn
```

Example 1.4.3 (Curried equality)

The comparison predicate `EQ` is normally used as a binary predicate, i.e. is normally applied to a pair of arguments.¹¹ It can sometimes be very useful to have a partial function to test the equality of a variable argument with a fixed one. This test could be done, e.g., by using the curried function `C:EQ`.

For example, we may write:

```
DEF CountZero = + ~ AA:(IF:< C:EQ:0, K:1, K:0 >);
CountZero:< 6, -22, 0, 4.25, 999, -33, 0 > ≡ 2
```

¹¹ Actually, it can be applied to any n -tuple of arguments, with $n \geq 1$.

1.5 Geometric operators

In this section we introduce the built-in geometric operators. Therefore, here we enumerate the predefined operations available in the language, by giving some very simple examples. In the next chapter some more interesting programs are discussed, to introduce the reader to the language flavour and methods.

1.5.1 Pre-defined geometric operators

The PLaSM language is characterized by a small number of pre-defined geometric operators that allow to easily implement several geometric operations useful in generative modeling. Most geometric operators are *dimension-independent*, i.e. can be applied to 2D objects as well as to 3D objects, and even to higher dimensional objects.

Our geometric operations are *closed* in the space of polyhedral complexes. This implies that sub-expressions which contain operators, arguments and parentheses can be combined into more complex expressions, thus giving a well-founded generative *geometric calculus*.

In this section such operators are briefly introduced, together with some simple examples of use. In later sections and chapters they are all described in much greater detail.

Elementary shape constructors

A small set of geometric primitives is provided, including

CUBOID and SIMPLEX

which respectively allow for generating d -dimensional cuboids, i.e. 1D segments, 2D rectangles, 3D parallelepipeds, 4D hyper-parallelepipeds, etc., and d -dimensional simplices, i.e. 1D segments, 2D triangles, 3D tetrahedra, etc.

CUBOID primitive The CUBOID primitive generates d -dimensional intervals of intrinsic dimension given by the number of actual parameters, which ordinarily correspond to the sizes of sides. Such intervals always have a vertex on the origin and edges parallel to the axes of the reference frame. For example:

```
CUBOID:5 ≡ 1D segment of length 5
CUBOID:<5,10> ≡ 2D rectangle of area 5 × 10
CUBOID:<5,10,5> ≡ 3D parallelepiped of volume 5 × 10 × 5
CUBOID:<1,1,1,1> ≡ 4D hypercuboid of volume 1
```

SIMPLEX primitive The SIMPLEX primitive generates simplices of different intrinsic dimension. For instance:

```
SIMPLEX:5 ≡ 1D segment of length 5;
SIMPLEX:<1,1> ≡ 2D standard triangle of area  $\frac{1}{2!}$ ;
SIMPLEX:<1,a,1> ≡ 3D standard tetrahedron of volume  $\frac{a}{3!}$ ;
SIMPLEX:<1,a,b,1> ≡ 4D simplex of volume  $\frac{ab}{4!}$ .
```

Polyhedral complex handlers

Polyhedral complexes are *the* geometric objects used by the language and discussed in Section 4.6. Each cell of such a complex is a polyhedron, which can be further subdivided into convex cells. Some primitives that work on polyhedral complexes are the following:

MKPOL, UKPOL, JOIN, EMBED

The **MKPOL** constructor is the most powerful geometric primitive, which provides a very high geometric covering to the language. This constructor allows for easily implementing *polylines*, *polymarkers*, *triangle stripes*, *quadrilateral meshes*, various kinds of *solid meshes*, and so on. The **UKPOL** primitive is conversely used to unpack the internal data structure and return an external representation of a polyhedral complex. The **JOIN** function is applied to a sequence of polyhedral complexes, embedded in the same space \mathbb{E}^d , and returns their convex hull.¹² The **EMBED** operator is used to give a higher number of coordinates to its argument, i.e. to *embed* a polyhedron or a complex into some Euclidean space of higher dimension. For example **EMBED:1:polygon** is used to embed a 2D **polygon** in the $z = 0$ subspace of the 3D space, by adding 1 more coordinate (with 0 value) to its points.

MKPOL primitive The "Make polyhedron" constructor **MKPOL** generates polyhedral complexes of any dimension. It is the basic geometry constructor in PLaSM. The **MKPOL** operator is a mapping from triples of sequences to polyhedral complexes:

MKPOL:< verts, cells, polys >

where **verts** is a sequence of points in the same space \mathbb{E}^d ; **cells** is a sequence of convex cells (given as sequences of point indices); **polys** is a sequence of polyhedra (given as sequences of cell indices). Each cell is defined as the convex hull of its vertices, each 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 1.5.1 (Polyhedral complex)

Two slightly different models of a 2D L-shaped polygon are given as *Polyhedral complexes* in Script 1.5.1, by using the **MKPOL** primitive. At this point the sequence of polygon *vertices* is associated with the symbol **verts**, then two convex cells are given as sequences of indices of vertices, and associated with the **cells** symbol. Finally, a set of polyhedral cells, each one defined as a sequence of cell indices, is associated with the **polys** symbol.

Notice that **L_shape1** is defined by using two polyhedral cells, each one associated with a convex cell. Conversely, **L_shape2** contains only one polyhedral cell, defined by two convex cells, that meet along a common face. The important difference between the two definitions can be observed by looking at Figures 1.2b and 1.2c. An explanation of this behavior requires concepts on *Polytopal subdivisions* discussed in Section 4.6. The skeletons of a polyhedral complex are introduced on page 28.

¹² The minimum convex set that contains them. Alternate definitions of this concept are given in Section 4.2.

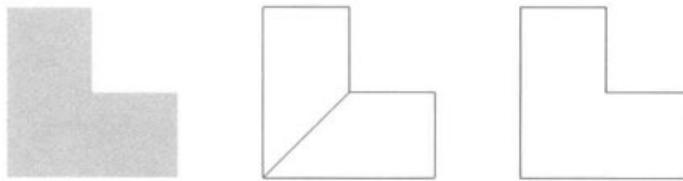


Figure 1.2 (a) L-shaped *Polyhedral complex* produced by the MKPOL primitive
 (b) 1-skeleton of the definition with 2 polyhedral cells (c) 1-skeleton of the definition
 with 1 polyhedral cell

Script 1.5.1

```
DEF L_shape1 = MKPOL:< verts, cells, polys >
WHERE
  verts = <<0,0>,<2,0>,<2,1>,<1,1>,<1,2>,<0,2>>,
  cells = <<1,2,3,4>,<4,5,6,1>>,
  polys = <<1>,<2>>
END;

DEF L_shape2 = MKPOL:< verts, cells, polys >
WHERE
  verts = <<0,0>,<2,0>,<2,1>,<1,1>,<1,2>,<0,2>>,
  cells = <<1,2,3,4>,<4,5,6,1>>,
  polys = <<1,2>>
END;
```

UKPOL primitive As we already said, the UKPOL primitive is exactly the inverse function of the MKPOL constructor. So it can be applied to any geometric object and returns the triplet of its vertices, convex cells and polyhedral cells. A simple example is given in Script 1.5.2, where a parallelepiped with sides of measure 1, 2 and 3, respectively, is *unpacked* as a triplet of vertices, cells and polyhedra.

Script 1.5.2 (Unpacked parallelepiped)

```
UKPOL:(CUBOID:<1,2,3>) ≡ <
  < < 0.0,2.0,3.0 >,< 1.0,2.0,3.0 >,< 0.0,0.0,3.0 >,< 1.0,0.0,3.0 >,
  < 0.0,2.0,0.0 >,< 1.0,2.0,0.0 >,< 0.0,0.0,0.0 >,< 1.0,0.0,0.0 > >,
  < < 1,2,3,4,5,6,7,8 > >,
  < < 1 > > >
```

Affine transformations

Affine transformations, i.e. the elementary translation, rotation, scaling, reflexions and shearing, as well the general bijective transformations of an Euclidean space, are fundamental tools in graphics and modeling. PLaSM provides both specific operators

T, S, R, H, MAT

for dimension-independent elementary transformations and a **MAT** operator which returns the transformation associated to any given non singular matrix.

PLaSM gives as pre-definite some elementary affine operators, including

```
T:coords:parameters ≡ translation
R:coords:parameters ≡ rotation
S:coords:parameters ≡ scaling
```

where **coords** denotes the coordinates affected by the transformation, and **parameters** denotes its parameters.

For efficiency reasons, a special form for parameter specification has been used for some frequently found elementary geometric functions. In particular, 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, **T:1** denotes translation on the first coordinate direction, and **T:1:2.5** means translation of 2.5 units on the first coordinate. For a translation on more than one coordinate we can write, e.g., **T:<1, 3>:<2.5, 6/1.5>**. When evaluated, this functional expression returns the *translation tensor* that translates the geometric objects it is applied to, by 2.5 units on the first coordinate direction (i.e. on the *x* axis) and by $6/1.5 \equiv 4.0$ units on the second coordinate direction (i.e. on the *y* axis). Note that several equivalences hold for such *functions*. For instance:

$$\mathbf{T}:\langle 1, 3 \rangle : \langle 2.5, 6/1.5 \rangle \equiv \mathbf{T}:1:2.5 \sim \mathbf{T}:3:(6/1.5)$$

Since they are functions, affine transformations can be composed and applied to polyhedra. In such a case they are equivalent to the application of a **STRUCT** operator, described in the following subsection. Below we show how to generate a unit cube with a vertex on the origin, move its basis center to the origin and rotate it by 45 degrees about *z*.

```
(R:<1,2>:(PI/4) ~ T:<1,2>:<-0.5,-0.5>): (CUBOID:<1,1,1>
≡ STRUCT:< R:<1,2>:(PI/4), T:<1,2>:<-0.5,-0.5>, CUBOID:<1,1,1> >
≡ STRUCT:< Anonymous-Function, Anonymous-Function, PolComplex<3,3> >
≡ PolComplex<3,3>
```

Hierarchical assemblies

According to modern high-level graphics systems, e.g. to *PHIGS*, *Inventor* or *VRML*, PLaSM provides an easy tool to build hierarchical assemblies, where each component or sub-assembly is defined in its local coordinate frame. This task is performed by the

STRUCT

operator. This function can only be applied to sequences containing polyhedral complexes, affine transformations and other invocations of the **STRUCT** function.

STRUCT primitive The constructor **STRUCT** is used to generate hierarchical assemblies of objects defined in local coordinates. It is applied to sequences of

polyhedra, affine transformations and **STRUCT** invocations. 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 into the coordinates of the root of the network. Such world coordinates coincide with those of the first polyhedral complex in the structure argument sequence. Its semantics, discussed in Chapter 8, is very close to the one of ISO PHIGS graphics standard.

Example 1.5.2 (Table assembly)

In Script 1.5.3 two alternate equivalent definitions of the **Table** model displayed in Figure 1.3 are given. Both **Table** and **Table1** definitions use local coordinates for the assembly components (four **Leg** instances and one **Plane** instance). The reader is advised to analyze the differences between the two definitions, and to make some hybrid attempts, while looking at the obtained results.

Script 1.5.3 (Table model)

```
DEF Leg = CUBOID:< 0.1,0.1,0.7 >;
DEF Plane = CUBOID:< 1, 1, 0.2 >;

DEF Table = STRUCT:<
    Leg, T:1:0.9, Leg, T:2:0.9, Leg, T:1:-0.9, Leg,
    T:<2,3>:<-0.9,0.7>, Plane >;

DEF Table1 = STRUCT:<
    Leg, T:1:0.9:Leg, T:<1,2>:<0.9,0.9>:Leg, T:2:0.9:Leg,
    T:3:0.7:Plane >;

VRML: Table:'out.wrl';
```

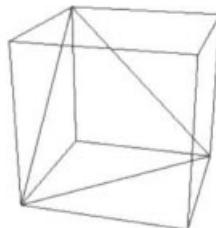
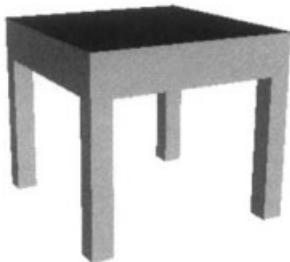


Figure 1.3 (a) **Table** model (b) 1-skeletons of the unit 3D cube and the unit 3-simplex

Skeletons of a complex

The cells of a polyhedral complex have different dimensions. In particular, the boundary of a d -dimensional cell, or d -cell, contains cells of dimension between 0 and $(d - 1)$. PLaSM provides a set of extractor operators

`@0, @1, @2, @3, ...`

of the d -dimensional skeletons of a complex, i.e. of the sub-complex of cells of dimension less or equal to a given integer.

SKELETONS primitives In PLaSM are called *skeleton extractors* operators, or r -skeletons, the mappings

$$@r : \mathcal{P}^{d,n} \rightarrow \mathcal{P}^{r,n}, \quad 0 < r < d,$$

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

Example 1.5.3 (1-skeleton of 3-cube and 3-simplex)

The 1-skeletons of both the 3D cube and the tetrahedron (3-simplex) are extracted and assembled together in Script 1.5.4 and displayed in Figure 1.3b. The reader should notice here the infix use of the STRUCT aggregation operator, and is asked to discover in Figure 1.3b the (common) position of the origin of the local reference frames of the cube and the tetrahedron.

Script 1.5.4 (1-skeleton)

```
DEF out = @1:(CUBOID:<1,1,1> STRUCT SIMPLEX:3);
VRML: out:'out.wrl';
```

Constructor of 1D polyhedra

A constructor of 1D polyhedra is extensively used in PLaSM programs. It is denoted as

`QUOTE`

and transforms non-empty sequences of non-zero reals into 1D polyhedra. In particular, positive numbers are transformed into solid segments, whereas negative numbers are used to translate the solid ones along the 1D line. Consecutive numbers along the input sequence correspond to adjacent segments along the output polyhedral complex.

QUOTE primitive The `QUOTE` operator 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 this operator is the set of sequences of non-zero reals:

$$\text{QUOTE} : (\mathbb{R} - \{0\})^* \rightarrow \mathcal{P}^{1,1}$$

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

Example 1.5.4 (User-defined operator)

Let us introduce in Script 1.5.5 our first user-defined geometric operator. This one will be called Q and defined as follows:

Script 1.5.5 (Q operator)

```
DEF Q = QUOTE ~ IF:< IsSeq, ID, [ID] >
```

which is the PLaSM composition of the simpler functions QUOTE and IF:< IsSeq, ID, [ID] >. The reader should notice that such functions are applied in reverse order to the input. In particular, when Q is applied to some number x, first the

$\text{IF}:\langle \text{IsSeq}, \text{ID}, [\text{ID}] \rangle : x$

expression is evaluated. The resulting value is then passed to the QUOTE function.

According to the semantics of the IF conditional construct, the predicate $\text{IsSeq}:x$ is tested to check if x is either a sequence or not. Since the predicate “*Is x a sequence?*” is false, the expression $[\text{ID}]:x$ is evaluated and $\langle x \rangle$ is returned. Such a sequence is passed to the QUOTE operator, and further transformed into a 1D line segment of x size.

Also, any sequence of non-zero numbers will be transformed by the Q function into a *complex* of segments on the real line. For example, the evaluation of the expression

$Q:\langle 5, -2, 5, -2, 5 \rangle$

will return the complex of 1D segments shown in Figure 1.4. Most of figures in this chapter are generated by grabbing the screen during the browsing of a VRML file generated by the PLaSM interpreter.

Figure 1.4 The complex of segments generated by evaluating the expression

$Q:\langle 5, -2, 5, -2, 5 \rangle$

The application of Q to a negative number or to a sequence of negative numbers will produce a language *exception*, i.e. the capture of a run-time error.

Cartesian product

An important operator on polyhedral complexes of any dimension is their Cartesian product, denoted in PLaSM by the standard product symbol

*

According to the standard FL syntax, such operation symbol can be used either infix in an argument pair or as prefix to a sequence of arguments. From both a formal and an algorithmic viewpoint, it is a special case of the “generalized product” introduced in [BFPP93].

PRODUCT primitive The Product operator is defined as a mapping from pairs of polyhedra to polyhedra, where the cells in the polyhedral output are generated by pair-wise Cartesian products of the cells in the polyhedral input pair. Algorithms for dimension-independent “generalized” product of both polyhedra and polyhedral complexes are given in [BFP93] and [PFP96], respectively. A detailed discussion of the meaning and algorithmic implementation of this important operator is deferred until Chapter 14. Anyway, it is so easy and natural to use, that we will make great use of it through the whole book.

Example 1.5.5 (*d*-dimensional (multiple) intervals)

The geometric objects generated by the following expressions, which contain the Cartesian product and the user-defined Q operation discussed above, are shown in Figure 1.5.

```

Q:<10,-10,10> ≡ PolComplex{1,1}
Q:<10,-10,10> * Q:<10,-10,10> ≡ PolComplex{2,2}
Q:<10,-10,10> * Q:<10,-10,10> * Q:3 ≡ PolComplex{3,3}

```

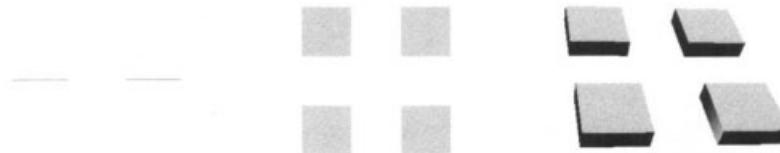


Figure 1.5 1-, 2- and 3-dimensional complexes generated by Q and * operators

Example 1.5.6 (Building facade)

An example is given in Script 1.5.6, and shown in Figure 1.6a, where a 2D complex is generated by Cartesian product of 1D complexes produced by the QUOTE operator. Several other examples of the QUOTE operator are given in Section 1.6.2.

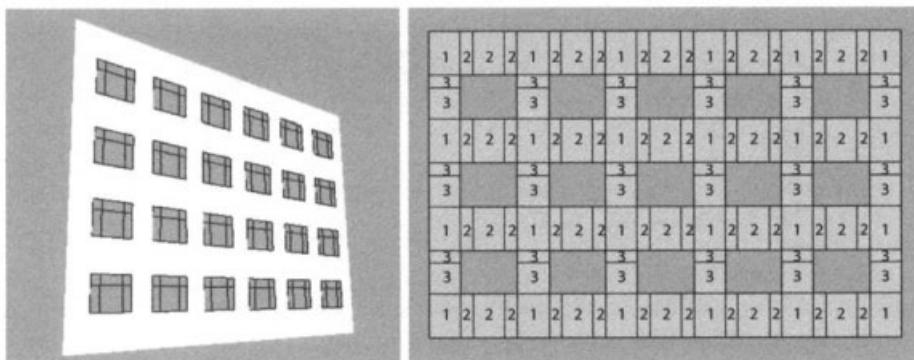


Figure 1.6 (a) the 2D complex generated as facade:<6,4> by combining QUOTE, product and 1-skeleton operators (b) the generating scheme of facade panels

The facade generating function works by assembling three 2D Cartesian products of alternating 1D complexes, produced by `Q:xRithm`, `Q:xVoid` and by `Q:yRithm`, `Q:yVoid`, respectively. In particular, the `xRithm` sequence contains the numeric series used in the *x* direction; analogously `yRithm` for the *y* direction. Conversely, `xVoid` and `yVoid` host the series with opposite signs of elements. So, the first three Cartesian products in the `STRUCT` sequence produce a sort of checkboard covering that follows the scheme given in Figure 1.6b.

Script 1.5.6 (Building facade)

```
DEF facade (n,m::IsIntPos) = STRUCT:<
    Q:xRithm * Q:yRithm,
    Q:xVoid * Q:yRithm,
    Q:xRithm * Q:yVoid ,
    @1:(Q:xVoid * Q:yVoid) >
WHERE
    xRithm = ##:n:<5,-2,-5,-2> AR 5,
    yRithm = ##:m:<7,-5,-2> AR 7 ,
    xVoid = AA:-:xRithm,
    yVoid = AA:-:yRithm
END;
```

Intersection of extrusions

Another special case of the generalized product of polyhedra [BFPP93] gives an operation where the arguments are first properly embedded into coordinate subspaces of the space of the result, then are subject to indefinite orthogonal extrusions, and are finally pair-wise intersected. Such operation is denoted (in the 3D case) as

`&&:<<i0,i1,i2>,<j0,j1,j2>>`

where both `<i0,i1,i2>` and `<j0,j1,j2>` are permutations of the sequence `<0,1,2>`. Such permutations are needed to specify how to embed the arguments into the coordinate subspaces of the result. The `&&` operation, combined with the 2-skeleton extractor `@2`, is the perfect solution to automatically generate a 3D model of a building starting from its 2D plans and sections.

Example 1.5.7 (Intersection of extrusions operator)

In this example a building `plan` is defined as a 2D assembly, and it is used both to generate a (2,3)-dimensional `House`, shown in Figure 1.7a, and a (2,3)-dimensional apartment `Block`, shown in Figure 1.7b. To generate the house a 2D `section` is also defined, which is combined with `plan` by using the infix operator

`<1,2,0> && <1,0,2>`

To generate the block a Cartesian product with the 1D polyhedron `Q:<3,3,3>` is conversely sufficient.

A solid offset version of the 1-skeleton of the `House` object is shown in Figure 1.8.

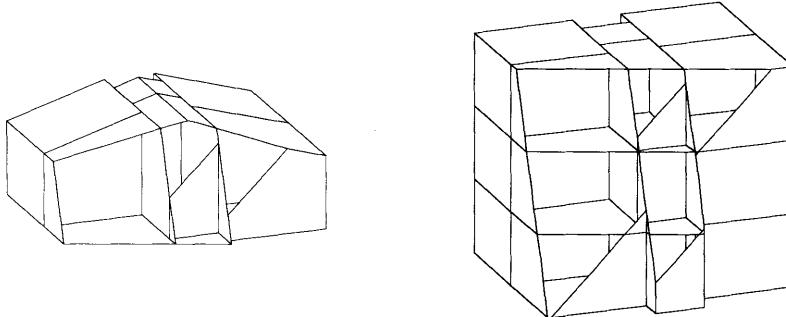


Figure 1.7 Polyhedral product of a 2D plan times a (1,1)-complex: (a) clipped view of the House (b) clipped view of the Block

Script 1.5.7

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

Parametric mapping

The **MAP** operator applies a vector-valued function to the vertices of a suitable cell decomposition¹³ of a polyhedron embedded in the mapping domain, thus returning a curved or deformed instance of it without changing its topology. The **MAP** operator is used with a double application

MAP:VectorFunction:Domain

where **VectorFunction** is a **CONSed** sequence of coordinate functions and **Domain** is any polyhedral complex. Such operator is used to generate curves, surfaces, splines and in general to generate any kind of parametric geometry, including transfinite approaches like Coons-Gordon's surfaces or meshes. It is also used in PLaSM for simpler tasks, like the generation of an arc of circumference.

MAP primitive 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 d 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**.

¹³ Actually, a so-called *simplicial complex*, discussed in Section 4.5.

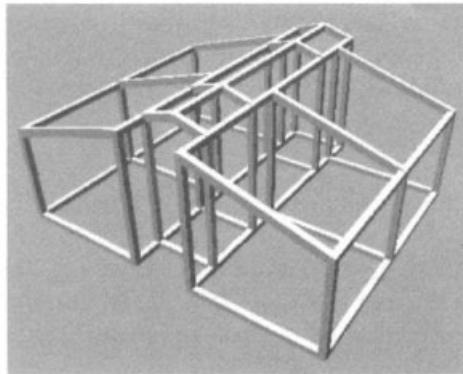


Figure 1.8 Solid frame generated by the expression `(Offset:<0.2,0.2,0.2> ~ @1) :House`. The `Offset` function is discussed in Section 14.6.3

Example 1.5.8 (Graph of the SIN function)

A piecewise linear approximation with 32 segments of graph of the sin function in the interval $[0, 2\pi]$, where PI is the PLaSM denotation for π , is generated in Script 1.5.8 and shown in Figure 1.9. Notice the mandatory use of the FL selector S1, which is used to select the needed coordinate of vertices of domain decomposition. Vertices are in fact represented as sequences of coordinates.

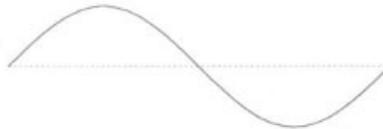


Figure 1.9 Polyhedral approximation of MAP of the [ID, SIN] function on the $[0, 2\pi]$ domain

Script 1.5.8

```
DEF sinFun = [ID, SIN] ~ S1;
DEF domain (n::IsIntPos) = (QUOTE ~ #:n):(2*PI/n);

MAP: sinFun: (domain:32);
```

Regularized Booleans

Set-theoretic operations like *union*, *intersection* and *difference* of solid models, also called Boolean operations, are very useful in a geometric modeling system. Actually, a modified definition of such operations is normally used (see p. 585), where the resulting sets are *regularized*, i.e. are generated with removal of dangling parts of lower dimensions.

In the geometric calculus allowed by PLaSM, regularized Boolean operators can be currently applied only to pairs of polyhedrally valued expressions of the same dimension, i.e. which are both 2D, or 3D, and so on. Also, a further requirement

concern the *solidity* of both arguments. We say that a polyhedral complex is *solid* when its *intrinsic dimension* is equal to the dimension of the space that contains it, that we call the *embedding dimension*. For example, a polygon is solid in 2D but not in 3D. The operation symbols

`+, &, ^, -`

respectively denote union, intersection, symmetric difference (XOR) and difference of polyhedral complexes.

According to the standard approaches in solid modeling, the built-in Boolean operators in PLaSM are regularized.¹⁴ Conversely, PLaSM currently includes the first dimension-independent implementation of Boolean operations.

Example 1.5.9 (Boolean operations between cubes)

In Script 1.5.9 a composite geometric value is produced, that contains the union, intersection, xor and difference of two rotated unit cubes `a` and `b`. It is displayed in Figure 1.10. As the reader will notice, at least after reading Section 13.1.1, the results of the xor and difference operations are *non-manifold* and *unconnected*, respectively.

Script 1.5.9 (Boolean example)

```
DEF a = T:<1,2>:<-0.5,-0.5>:(CUBOID:<1,1,1>);
DEF b = R:<1,2>:(PI/4):a;

STRUCT:< a + b, T:1:2, a & b, T:1:2, a ^ b, T:1:2, a - b >;
```

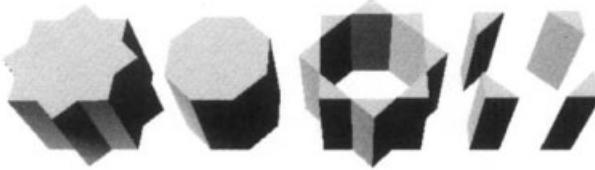


Figure 1.10 From left to right: (a) union (b) intersection (d) XOR and (e) difference of two rotated cubes

Relative positioning operators

To help the user in positioning a polyhedral complex with respect to another, PLaSM provides the functions

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

where `ALIGN` is truly dimension-independent, whereas `TOP, BOTTOM, LEFT, RIGHT` are some shortcuts usable only in relative positioning of 3D objects. Several examples are

¹⁴ Anyway, a non-regularized implementation of Booleans is under study, and will be enclosed in some future release of the language.

given in the next chapter.

An alternative (and much easier) definition of the **Table** model already discussed in Script 1.5.3 is given in Script 1.5.10. The new definition uses a suitable combination of the **Q**, ***** and **TOP** operators. In particular, the **Legs** object is produced as Cartesian product of three sets of intervals along the *x*, *y* and *z* coordinate directions. The **Plane** object is generated analogously. The **Table** value is the assembly produced by the binary **TOP** operator described above. The **Chair** object is produced by suitably scaling the **Table**.

Script 1.5.10 (Table model (2))

```
DEF legs = Q:< 0.1,-0.8,0.1 > * Q:< 0.1,-0.8,0.1 > * Q:0.7;
DEF plane = Q:1 * Q:1 * Q:0.2;
DEF table = Legs TOP Plane;
DEF chair = S:<1,2,3>:<0.4,0.4,0.5>:Table;

DEF assembly = chair RIGHT table RIGHT chair UP chair DOWN chair
VRML:assembly:'out.wrl'
```

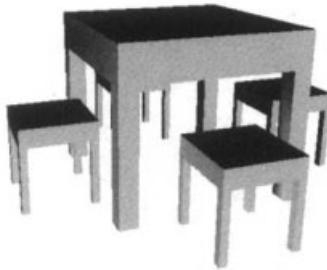


Figure 1.11 assembly using positioning operators

Note The more interesting line of code is the **assembly** definition. It is written by sequencing several infix instances of binary positioning operators. The language syntax states that all operators, when used infix, are left-associative. This rule implies that the above expression is actually evaluated as follows:

```
((chair RIGHT table) RIGHT chair) UP chair) DOWN chair
```

Therefore, the **table** is put **RIGHT** the first **chair** instance; the second **chair** instance is put **RIGHT** the previous *sub-assembly*, the third **chair** instance is put **UP** the previous *sub-assembly*, and so on.

Note A further remark upon the code given in Script 1.5.10 may be interesting. By changing some arbitrary parameters in the definition of some assembly component, the whole assembly construction continues to work!

Sizing functions

A set of pre-defined functions is also provided, including

SIZE, BOX, MAX, MIN, MED

where **SIZE:<i₁, ..., i_n>:pol** returns the sequence of sizes of the projection of the polyhedral argument along the specified coordinate directions. The expression **BOX:<i₁, ..., i_n>:pol** returns the containment box of the **pol** projection into the subspace of the specified coordinates. Conversely, **MAX:<i₁, ..., i_n>:pol** return the maximum values achieved on **pol** by the same coordinates. The behavior of **MIN** and **MED** is similar.

An example of use of the **BOX** primitive is given in Figure 1.12. that shows the geometric result of the following computation:

```
(STRUCT ~ [ID, BOX:<1,2>]):assembly
  ≡ STRUCT:([ID, BOX:<1,2>]:assembly)
  ≡ STRUCT:([ID, Anonymous-Function]:assembly)
  ≡ STRUCT:< ID:assembly, Anonymous-Function:assembly >
  ≡ STRUCT:< assembly, PolComplex<2,3> >
  ≡ PolComplex<3,3>
```

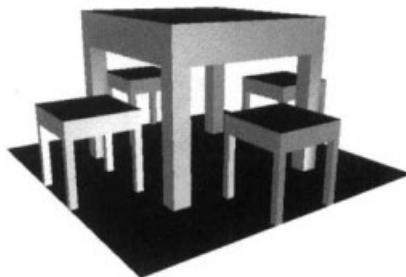


Figure 1.12 Geometric value of the expression **(STRUCT ~ [ID, BOX:<1,2>]):assembly**

Parametric vs implicit geometry

Some further explanation should be addressed to the **PLaSM** representation and treatment of *curved* objects (curves, surfaces, solids, and so on). They are always represented in a *parametric* form, as the mapping of an appropriate set of coordinate functions over a suitable cell decomposition of some polyhedral domain, as discussed in following sections of this book.

Mostly it is possible to delay the evaluation of such mapping as late as possible, i.e. until the exporting of the geometry to some external viewer, as well as linking it with appropriate rendering properties (like color or normal shading) to recover a curved appearance of the exported object. Other times such delayed mapping evaluation is not possible, as when performing Boolean operations between curved objects, so that the user is in this case obliged to work with some faceted approximation of the Boolean result.

The impossibility of dealing with *implicit* representations of curved geometry is probably the weakest aspect of the PLaSM approach. Conversely, the language gives exact (i.e. symbolic) and very high level support to the parametric representation, which is the “natural” representation in differential geometry and mechanics.

A quite detailed discussion of the possible representations of “curved” geometry is given in Section 11.1.

1.6 Examples

In this section some non-trivial examples of the PLaSM programming approach to geometric design are given. The aim is both to transmit the flavour of the language and to allow the reader to build some geometric models of increasing complexity.

1.6.1 First programs

First, we go to generate a simple 2D Flash artwork, shown in Figure 1.14. For this purpose we start by defining a `circle` generating function. Then we construct an assembly of 3D cubes, and make some simple variations to the PLaSM code, thus generating the pictures given in Figure 1.15, produced by grabbing a portion of the screen during the browsing of the exported VRML files.

In the sequel, the pre-defined PLaSM functions are written by using all capital letters. The user-defined functions will be normally written by capitalizing only the first letter of each word included in their names, like e.g. in `MyFirstFunction`.

Circle

A piecewise linear approximation of the circle of variable radius and centered at the origin is discussed here. For this purpose we need a parametric representation. *parametric representation* of a plane curve is a *function* $D \rightarrow \mathbb{E}^2$, with $D \subset \mathbb{R}$, such that for each value of the parameter u in the curve *domain* $D \subset \mathbb{R}$, a point $(x(u), y(u))$ of the curve is generated. The parametric equations of the 2D circle boundary with r radius and centered at the origin are:

$$x(u) = r \cos u, \quad y(u) = r \sin u, \quad u \in [0, 2\pi].$$

The generating functions of a polygonal approximation with n sides of the `Circumference` and `Circle` are given in Script 1.6.1. We notice that:

1. the parameter interval $[0, 2\pi]$ is decomposed into n subintervals of equal size $2\pi/n$, by the expression `Intervals:(2*pi):n`;
2. the 1D polyhedral approximation of circle boundary so generated is curved by the `mapping:r` function;
3. the 2D circle is generated as a mapping of two real parameters from the 2D domain `Intervals:(2*PI):n * Intervals:r:m`, where `*` is a Cartesian product of point sets.

A detailed discussion of the geometric approach used is deferred to later sections, and in particular to Chapter 5.

Script 1.6.1 (Circle)

```

DEF mapping (r::IsRealPos) = [K:r * COS, K:r * SIN] ~ S1;
DEF Intervals (a::IsRealPos)(n::IsIntPos) = (QUOTE ~ #::n):(a/n);
DEF Circumference (r::IsRealPos)(n::IsIntPos) = MAP:(mapping:r):(domain:n);
DEF Circle (r::IsReal)(n,m::IsIntPos) = MAP:
    <S2 * COS ~ S1, S2 * SIN ~ S1>:(Intervals:(2*PI):n * Intervals:r:m);

Circumference:1:48 RIGHT circle:1:<48,1>

```

The geometric value generated by last expression is shown in Figure 1.13. Several other examples of PLaSM scripts that make use of the **MAP** operator can be found in subsequent chapters.

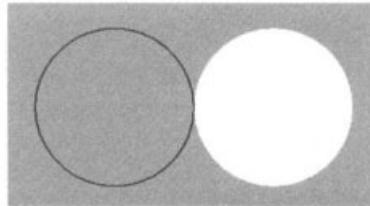


Figure 1.13 Assembly of circumference and circle of unit radiiuses

Flash 2D artwork

The objects **c1**, **c2** and **c3** given in Script 1.6.2 are generated by applying the **circle** operator to actual arguments, i.e. to the radius values 1, 0.7 and 0.4, and to the number 48 of sides of the approximating polygon. **ACOLOR** is a pre-defined operator contained in the **flash** library; expressions like **rgbacolor:<0,1,0,1>** produce an object with value in the **RGB α** color space used by **flash**, where α stands for *transparency*. Later in this chapter we will learn that **STRUCT** is the operator frequently used to aggregate simpler objects into a composite geometric object.

Script 1.6.2 (Italy's Football Cup)

```

DEF c1 = circle:1:<48,1> ACOLOR rgbacolor:<0,1,0,1>;
DEF c2 = circle:0.7:<48,1> ACOLOR rgbacolor:<1,1,1,1>;
DEF c3 = circle:0.4:<48,1> ACOLOR rgbacolor:<1,0,0,1>;

DEF out = STRUCT:< c1, c2, c3 >;
flash:out:100:'path/out.swf';

```

As the reader already experienced with graphics knows, **Flash** may only manage 2D graphics and animations. The last expression of Script 1.6.2 has the effect of exporting the value of **out** object to the file '**out.swf**' using a graphics *canvas* wide 100 pixels. The reader may check the result by just opening the file with the preferred web browser.

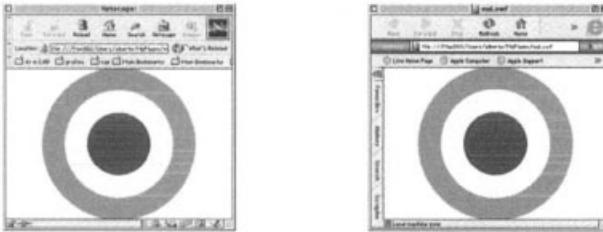


Figure 1.14 The symbol of the Italy's Football Cup, exported as Flash file by the PLaSM script 1.6.3

VRML assembly of 3D cubes

A simple assembly of colored cubes is produced in Script 1.6.3, and exported as VRML file 'out.wrl'. In this case the unit cube object is produced by the CUBOID primitive applied to the sequence of object lengths in the e_i directions ($1 \leq i \leq 3$) of the reference axes. Such object is then *instanced* three times, associated to predefined color constants CYAN, MAGENTA and YELLOW. The assembly composite object contains the result of the repeated (infix) application of the binary operator TOP to the cube1, cube2 and cube3 arguments. Conversely, the basis object is defined as the 2D square of side length 3, embedded in the coordinate subspace $z = 0$ of 3D space. Finally, the object named out contains the geometric value exported to the VRML file and visualized in Figure 1.6.3.

Script 1.6.3 (Assembly of colored cubes)

```
(plasm "
    DEF cube = CUBOID:<1,1,1>;
    DEF cube1 = cube COLOR CYAN;
    DEF cube2 = cube COLOR MAGENTA;
    DEF cube3 = cube COLOR YELLOW;
    DEF basis = (EMBED:1 ~ CUBOID):<3,3>;
    DEF assembly = cube1 TOP cube2 TOP cube3;

    DEF out = basis TOP assembly;
")
```

Interaction with the design environment

We continue now by introducing small variations to the code of Script 1.6.3. The PLaSM *design environment* allows the user to build complex models incrementally, by introducing new objects and definitions, and by re-defining operators and/or objects already defined or evaluated. The design environment will automatically take care of tracing the user actions, and will maintain continuously updated and consistent the internal geometric data base.

In the remainder of this book we do not further display the embedding Scheme forms of type (plasm " ... "). As the reader already knows, they are not necessary when using a PLaSM specialized *editor*.

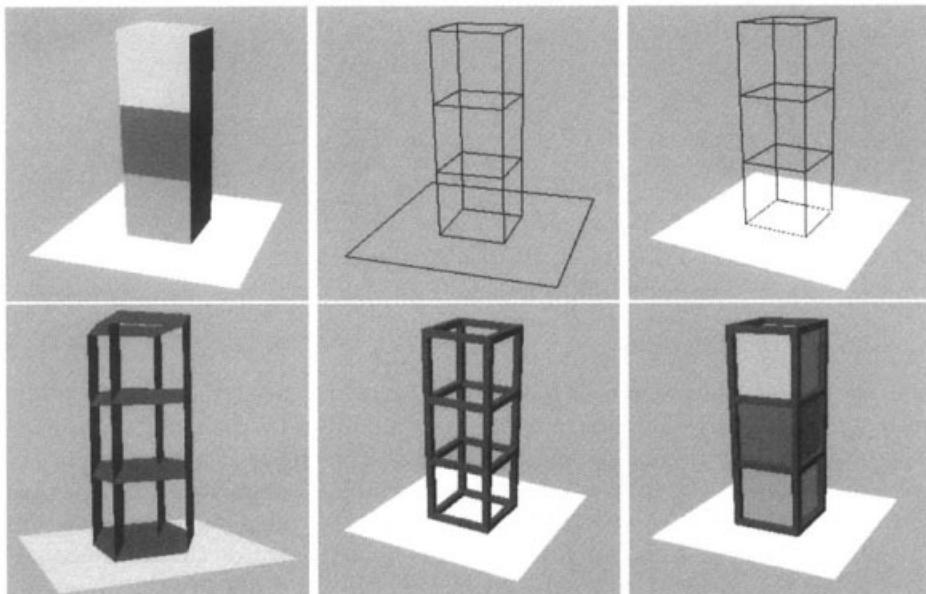


Figure 1.15 Small variations of the source Script 1.6.3

- a. First, we export the geometric content of the `out` object to the external VRML file '`out.wrl`'. The result is shown in Figure 1.15a. The pictures are ordered from top to bottom, and from left to right.

```
VRML:out:'out.wrl';
```

- b. Then we export the *1-skeleton* of the `out` object, i.e. the set of boundary edges of all the *polyhedral cells* it is composed of. Such concepts will be clearly detailed in the rest of the book. The result is shown in Figure 1.15b. We are used to export to the same output file, because in this way only one click on the Refresh (*MS Explorer*) or the Reload (*Netscape Navigator*) button is sufficient to display the result.

```
VRML:(@1:out):'out.wrl';
```

- c. Here we return to give a more solid aspect to the `out` object, and redefine it as compound by the 2D **basis** and by the @1-skeleton of the **assembly** object. Then we export the new `out` value, displayed in Figure 1.15c.

```
DEF out = basis TOP @1:assembly;
VRML:out:'out.wrl';
```

- d. We use the **SWEET** operator to transform every segment of the 1-skeleton of the **assembly** object into the polygon generated by translation with vector $\langle 0.2, 0.2, 0.2 \rangle$. Clearly enough, the result of the evaluation of the expression **SWEET**: $\langle 0.2, 0.2, 0.2 \rangle$ is a *function*, in order to be composed with the @1 function. Notice that only the sub-expression on the right-hand of **TOP** operator is affected by the binary operator **COLOR**. This fact is obtained by proper use of parenthesis. The resulting geometric value is shown in Figure 1.15d.

```
DEF out = basis TOP ((SWEET:< 0.2, 0.2, 0.2 > ~ @1):assembly COLOR RED);
```

```
VRML:out:'out.wrl';
```

- e. A better *solidification* of the 1-skeleton of **assembly** is provided by the functional value of the **OFFSET:< 0.1, 0.1, 0.1 >** expression. The result is given in Figure 1.15e.

```
DEF out = basis TOP ((OFFSET:< 0.1, 0.1, 0.1 > ~ @1):assembly COLOR RED);
VRML:out:'out.wrl';
```

- f. Finally, the **assembly** object is aggregated with the “solidified” instance of its 1-skeleton. The geometric result is displayed in Figure 1.15f.

```
DEF out = basis TOP
((STRUCT ~ [ID, OFFSET:< 0.1, 0.1, 0.1 > ~ @1]):assembly COLOR RED);
VRML:out:'out.wrl';
```

In order to see the generated geometric model, it is required that either your web *browser* is already equipped with a VRML *plug-in*, or your machine contains a separate VRML *viewer*. Therefore, let point your browser to the **out.wrl** file on your machine, open it and look at the result.

At this point, we strongly hope that both the novice and the experienced user have appreciated how easy is to produce quite complex models with PLaSM. So, don't wait: try PLaSM and enjoy!

1.6.2 Further examples

A sequence of integer numbers is generated by evaluting some PLaSM expression like the following one:

```
5 .. 10
```

which produces as the output of its evaluation the sequence:

```
<5, 6, 7, 8, 9, 10>
```

The **InsertSpace** function of Script 1.6.4 allows the introduction of a negative number **x** between any pair of consecutive numbers of the input sequence **seq**.

Script 1.6.4

```
DEF InsertSpace (seq::IsSeqOf:IsReal) (x::IsRealNeg) =
(CAT ~ TRANS):< seq, #:(LEN:seq):x >

DEF Xdims = InsertSpace:(2..10):-2;
DEF Ydims = InsertSpace:(3..8):-2;
```

Notice that it results, respectively:

```
Xdims ≡ <2,-2,3,-2,4,-2,5,-2,6,-2,7,-2,8,-2,9,-2,10,-2>
Ydims ≡ <3,-2,4,-2,5,-2,6,-2,7,-2,8,-2>
```

The previous sequences of number are so transformed into 1D polyhedra by the **Q** operator of Script 1.5.5, and multiplied by Cartesian product to give the 2D and 3D models shown in Figures 1.16a and 1.16b, respectively:

```
Q:Xdims * Q:Ydims;
Q:Xdims * Q:Ydims * Q:<4,-20,4>;
```

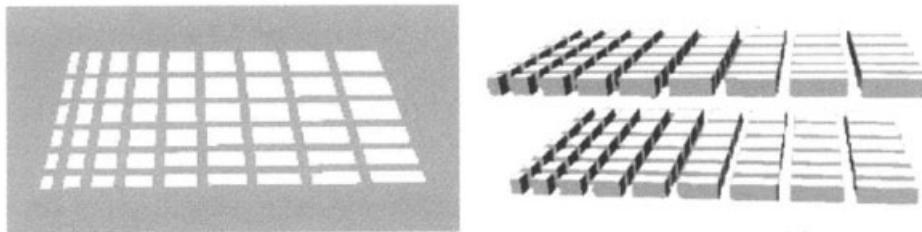


Figure 1.16 (a) Cartesian product of two 1D polyhedral complexes (b) Cartesian product of three 1D polyhedral complexes

A user-defined **Extrude** function is given in Script 1.6.5 to extrude with height **h** the **n**-th polyhedral cell of the polyhedral complex **p**. Such a result is simply obtained by Cartesian product of the **n**-th cell with the 1D segment **Q:h**. The **n**-th cell of the **p** polyhedral complex is extracted by first unpacking (**UKPOL**) the **p**'s internal data structure, then by **SELecting** the appropriate data components and finally by reconstructing (**MKPOL**) the internal representation of the complex.

Script 1.6.5

```
DEF Extrude (n::IsIntPos; p::IsPol; h::IsRealPos) =
  (MKPOL ~ [S1,S2,LIST~SEL:n~S3] ~ UKPOL):p * Q:h

DEF plan2D = Q:Xdims * Q:Ydims;
DEF building1 = Extrude:<10,plan2D, 12>;
DEF building2 = Extrude:<25,plan2D, 20>;

STRUCT:< EMBED:1:plan2D, building1, building2 >;
```

A 2D complex **plan2D** is so defined, together with two 3D polyhedra **building1** and **building2**, which are obtained by extruding with heights 12 and 20, respectively, the 10-th and 25-th polyhedral cells of **plan2D**. The model in Figure 1.17 is finally generated by evaluating the last line of Script 1.6.5, where **plan2D** (a) is **EMBEDded** in 3D by adding 1 more coordinate, and (b) is aggregated into a 3D **STRUCTure** together with the polyhedra **building1** and **building2**.

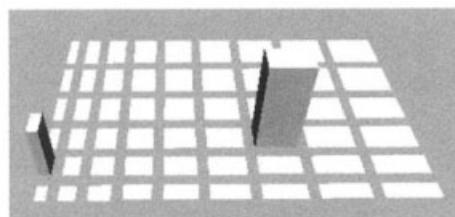


Figure 1.17 2D complex with selective extrusion of two cells

1.6.3 Virtual Manhattan

A more realistic Manhattan-like 2D complex of polyhedra is defined in Script 1.6.6 by using the MKPOL basic geometric constructor and by explicitly giving vertices, convex cells and polyhedral cells of the object to be generated. The resulting 2D polyhedral complex is shown in Figure 1.18.

Script 1.6.6

```
DEF Manhattan2D = MKPOL:< verts, cells, polys >
WHERE
  verts = < <0,0>,<3,0>,<5,0>,<7,0>,<8,0>,<9.5,1>,<10,1.5>,<0,3>,<3,3>,<5,3>,
    <7,3>,<8,3>,<9.5,3>,<0,4>,<3,4>,<5,4>,<9.5,4>,<12,4>,<9.5,5>,<10,5>,
    <12,5>,<0,6>,<3,6>,<5,6>,<0,7>,<3,7>,<5,7>,<9.5,7>,<12,7>,<9.5,8>,
    <12,8>,<0,9>,<3,9>,<5,9>,<8,9>,<9,9>,<12,9>,<0,10>,<3,10>,<5,10>,
    <8,10>,<9,10>,<9.5,10>,<10,10>,<12,10>,<6,11>,<7,11>,<0,12>,<3,12>,<9,12>,
    <9.5,12>,<0,13>,<3,13>,<6,13>,<7,13>,<9,13>,<9.5,13>,<0,14>,<3,14>,<5,14>,
    <8,14>,<9,14>,<9.5,14>,<10,14>,<12,14>,<0,15>,<3,15>,<5,15>,<8,15>,<0,16>,
    <6,16>,<7,16>,<9,17>,<9.5,17>,<10,17>,<12,17>,<6,18>,<7,18>,<9,18>,<9.5,18>,
    <10,18>,<12,18>,<2,19>,<3,19>,<5,19>,<8,19>,<9,19>,<9.5,19>,<10,19>,<12,19>,
    <5,20>,<12,20>,<7,22>,<10,22>,<9,6>,<12,6>,<9,15>,<9.5,15>,<10,15>,<12,15> >,
  cells = < <1,2,9,8>,<3,4,11,10>,<5,6,13,12>,<14,15,23,22>,<16,17,19,24>,
    <7,18,21,20>,<25,26,33,32>,<27,29,28,35,34>,<95,96,29,28>,<30,31,37,36>,
    <38,39,49,48>,<40,41,47,46>,<41,61,55,47>,<55,61,60,54>,<54,60,40,46>,
    <42,43,51,50>,<44,45,65,64>,<52,53,59,58>,<56,57,63,62>,<66,67,84,83,70>,
    <68,69,72,71>,<69,86,78,72>,<78,86,85,77>,<71,77,85,68>,<97,98,74,73>,
    <99,100,76,75>,<79,80,88,87>,<81,82,90,89>,<91,92,94,93> >,
  polys = AA:LIST:(1..29)
END;
```

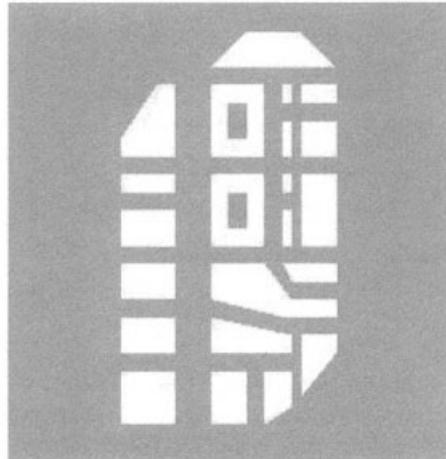


Figure 1.18 Polyhedral complex generated by evaluating the object `Manhattan2D`

The `cells` of the `Manhattan2D` complex are finally extruded to produce a 3D virtual environment. For this purpose the `MultExtrude` function is given, which accepts as input a polyhedral complex `p` and a sequence `h` of real numbers. The multiple extrusion

is thus performed by the Cartesian product of each cell of p with one of elements of the h sequence, properly transformed into a 1D segment.

Two sequences `GrowingH` and `ManhattanH` of heights are also given in Script 1.6.7. The two geometric assemblies of Figure 1.19 are finally generated by respectively evaluating the two PLaSM expressions:

```
MultExtrude:Manhattan2D:GrowingH;
MultExtrude:Manhattan2D:ManhattanH;
```

Script 1.6.7

```
DEF MultExtrude (p::IsPol) (h::IsSeqOf:IsReal) =
  (STRUCT ~ AA:* ~ TRANS):< ThePolSeq, AA:Q:h >
WHERE
  ThePolSeq = (REVERSE ~ AA:MKPOL ~ AA:CAT ~ DISTR):
    < DISTL:< MyVerts, MyCells >, <<<1>>> >,
  MyVerts = (S1 ~ UKPOL):P,
  MyCells = (AA:LIST ~ S2 ~ UKPOL):P
END;

DEF GrowingH = 1..29;
DEF ManhattanH = <1,3,1,11,1,0.2,1,0.1,1,8,15,1,
  1,1,1,8,1,15,8,1,2,2,2,2,5,9,1,1,1>;
```

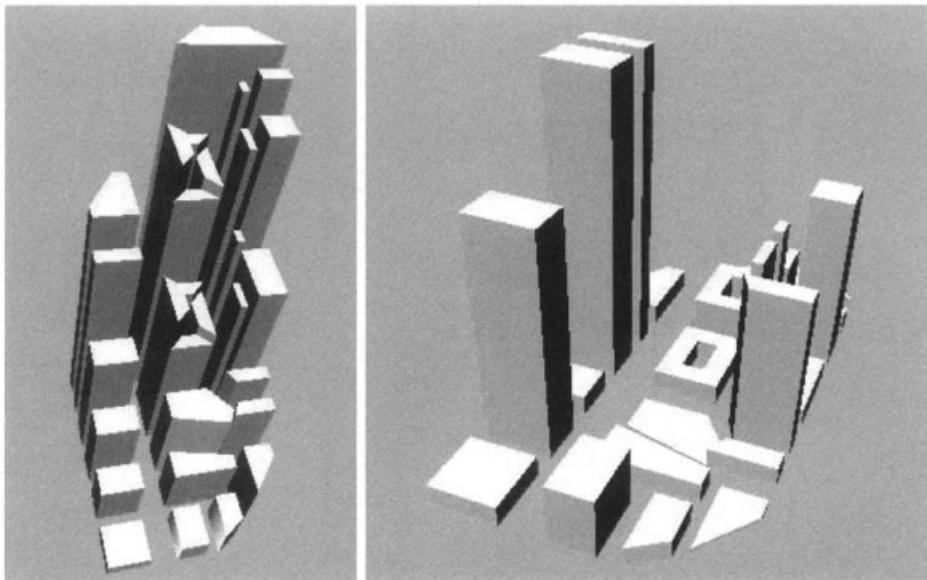


Figure 1.19 (a) 3D complex generated from `Manhattan2D` with increasing heights of ordered cells (b) A more realistic model of our simplified virtual Manhattan

1.6.4 Virtual skyscraper

At this point we want to generate the geometric model of a skyscraper, to be put into our virtual Manhattan. For this purpose we start to generate a possible external envelope.

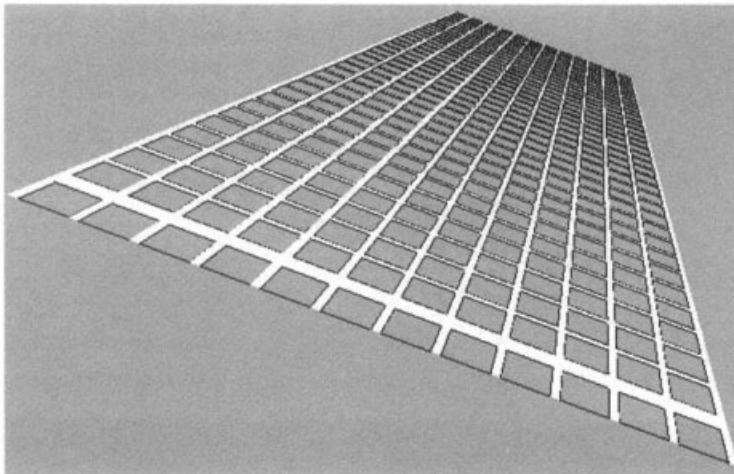


Figure 1.20 2D complex to be used as facade of a virtual skyscraper

The 2D facade with 12 bays and 36 floors of Figure 1.20 is generated by Script 1.6.8, and in particular by evaluating the expression

```
BuildingFront:<1,0.5,0.2>:<12,36>;
```

where a function **BuildingFront** is used with three real parameters **a**, **b**, **c** to set the geometric rythm of the facade and with two integer parameters **n**, **m** to set the numbers of bays and floors, respectively. The infix function **AR** (**AL**) is used to “append right” (“append left”) one more object to a sequence.

Script 1.6.8

```
DEF BuildingFront (a,b,c::IsRealPos)(n,m::IsIntPos) = STRUCT:<
  Q:xFull * Q:yFull,
  Q:xVoid * Q:yFull,
  Q:xFull * Q:yVoid,
  @1:(Q:xVoid * Q:yVoid) >
WHERE
  xFull = ##:n:< c,-:a > AR c,
  yFull = -:a AL (b AL ##:m:< -:a,c > AR b),
  xVoid = AA:-:xFull,
  yVoid = AA:-:yFull
END;
```

Geometric models of a parametric skyscraper with **m** floors and **n1** × **n2** boundary bays are generated by Script 1.6.9. In particular, the model with 36 floors and 12 × 6

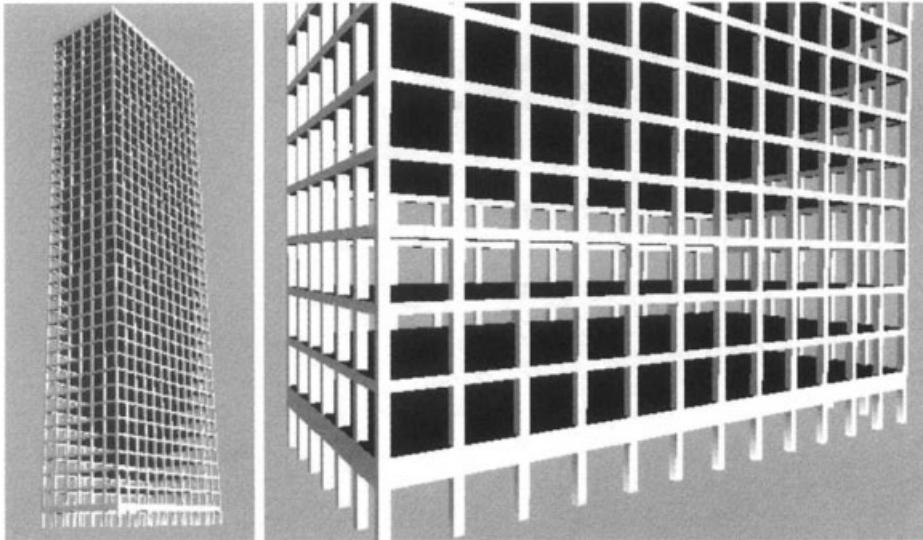


Figure 1.21 (a) Virtual skyscraper (b) Particular of the external envelope

boundary bays shown in Figure 1.21 is obtained by evaluating the PLaSM expression

```
Skyscraper:<1,0.5,0.2>:<12,6,36>;
```

Notice that such a model includes also the floor surfaces, generated by the 6-th source line of the **Skyscraper** function.

Script 1.6.9

```
DEF Skyscraper (a,b,c::IsReal)(n1,n2,m::IsIntPos) = STRUCT:<
    front1,
    front2,
    T:3:(-:side2):front1,
    T:1:(side1 - 0.2):front2,
    T:3:(-:side2):(Q:side1 * (@0 ~ Q):(-:(a+b) AL #:m:(a+c)) * Q:side2) >
WHERE
    front1 = BuildingFront:<1,0.5,0.2>:<n1,m> * Q:0.2,
    front2 = R:<1,3>:(PI/-2):(BuildingFront:<1,0.5,0.2>:<n2,m> * Q:0.2),
    side2 = SIZE:3:front2,
    side1 = SIZE:1:front1
END;
```

1.6.5 Roof of S. Stefano Rotondo

A different example of generation of a quite complex geometric assembly is given in Script 1.6.10, where the PLaSM code necessary to the generation of the trellis of the central roof of the old basilica of S. Stefano Rotondo in Rome is shown. Such a complex trellis is generated by first modeling the 2D wire-frame of the single beam, then by properly embedding it in 3D, then by automatically transforming it into a

solid model using an `OffSet` with parameters `<1, 0.5, 1>`. The dimension-independent `OffSet` function, which implements the *Minkowski sum* of a polyhedral d -complex with a n -dimensional interval, is described in Section 14.6.3. The rotational assembly of n instances of either a wire-frame or a solid element is finally performed by the function `Radial`.

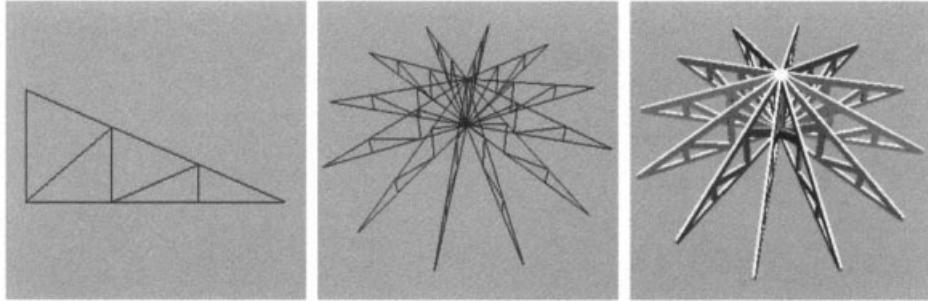


Figure 1.22 (a) 2D model generated by `Trellis2D:<12,23>` (b) 3D model generated by `Radial:12:Trellis3D` (c) Solid model generated by `Radial:12:SolidTrellis`

Script 1.6.10

```
DEF Trellis2D (H,L::IsRealPos) = MKPOL:
  <<<0,0>,<L/3,0>,<2*L/3,0>,<L,0>,<2*L/3,H/3>,<L/3,2*H/3>,<0,H>>,
  <<1,2>,<2,3>,<3,4>,<4,5>,<5,6>,<6,7>,<7,1>,<1,6>,<6,2>,<2,5>,
  <5,3>>,<1..11> > ;

DEF Trellis3D = (R:<2,3>:(PI/2) ~ EMBED:1 ~ Trellis2D):<12,23>;
DEF SolidTrellis = OffSet:<1,0.5,1>:Trellis3D;
DEF Radial (n::IsIntPos)(obj::IsPol) =
  (STRUCT ~ ##:n):< obj, R:<1,2>:(2*PI/n) >;

VRML: (Radial:12:Trellis3D):'out.wrl';
VRML: (Radial:12: SolidTrellis):'out.wrl';
```

The whole basilica of S. Stefano Rotondo in Rome has been implemented in PLaSM with few pages of source code. Two views of the resulting VRML model, to be experimented on the web with any browser for virtual reality, are given in Figures 1.23 and 1.24.

1.7 Annotated references

The interested reader is referred to the Backus' Turing lecture [Bac78] for motivation of programming at function level, to [Wil82, BWW90] for a more introductory treatment of the topic, and to [BWW⁺89] for a complete description of FL syntax and semantics.

A discussion on program optimization using program transformation in the context of a programming language designed to facilitate writing high level programs that are

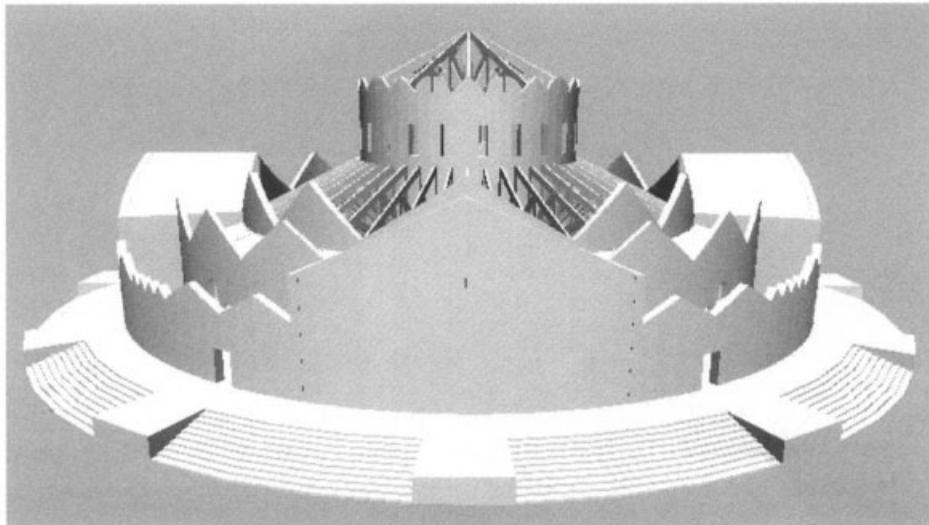


Figure 1.23 External view of basilica of S. Stefano Rotondo in Rome

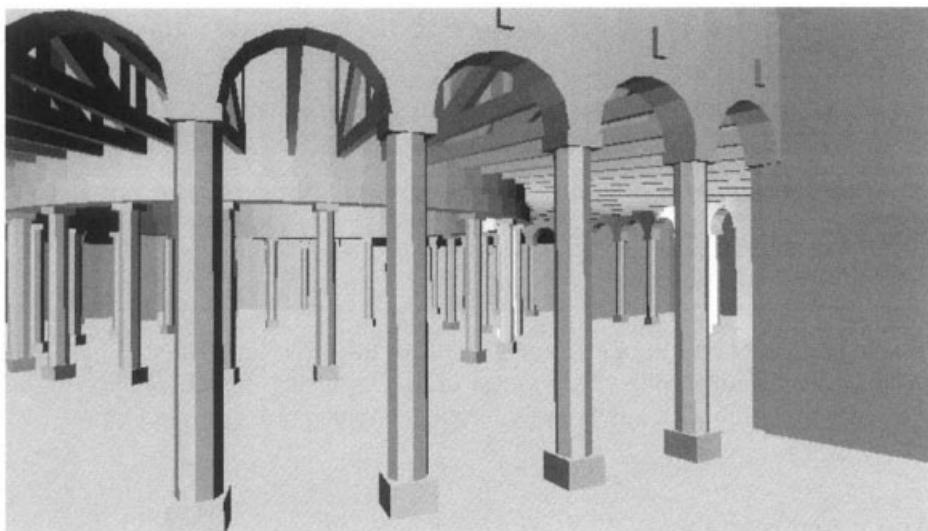


Figure 1.24 Internal view of one of the cross-chapels

clear and concise can be found in [WW91]. A very interesting approach to 2D graphics with functional programming can be found in [LZ88, ZLL⁺88] where a functional system equivalent to *Postscript* is described.

The PLaSM language design was introduced in [PS92] and later presented and discussed on [PPV95]. The generalized product operation has been introduced in [BFPP93] on polyhedra defined as complexes of convex cells, and discussed in its generalization to hierarchical polyhedral complexes in [PFP96]. The design of the HPC data structure is also discussed in [PFP96]. The dimension-independent approach to geometric data structures and algorithms extensively adopted in the language has been introduced in [PBCF93].