Visual Comput (2005) 21: 329–339
DOI 10.1007/s00371-005-0289-z

ORIGINAL ARTICLE

Jean-Eudes Marvie
Julien Perret
Kadi Bouatouch (✉)

The FL-system: a functional L-system for procedural geometric modeling

Published online: 24 May 2005

IRISA/INRIA Rennes,
Campus Universtaire de Beaulieu,
Avenue du Général Leclerc,
35042 Rennes Cedex, France
(jemarvie,juperret,kadi)@irisa.fr

Abstract In this paper, we present an FL-system, an extension of an L-system that allows us to generate any kind of object hierarchy and mesh on the fly. This has been made possible thanks to a modification of the classical L-system rewriting mechanism that produces a string of symbols interpreted afterwards. In our system, terminal symbols are not characters, but functions that can be executed at any step of the rewriting process. Thanks to this extension, our system allows the instantiation of generic objects during the course of the rewriting process as well as their initialization. Therefore, we are able to simulate all of the existing solutions proposed by classical L-systems, but we are also able to generate VRML97 scene graphs and geometry on the fly, since VRML97 nodes are handled as generic objects. As an example, we will show in the second part of this paper how to use our extension to describe building styles that are utilized to generate large sets of different building models. We also present some models of urban features (street lamps, etc.) and plants modeled and generated using FL-systems.

Keywords L-systems · Grammars · Object modeling · Real-time rendering

1 Introduction

We are interested in generating very complex city models; that is, city models containing many different styles of buildings and many buildings for each style. We also want to add some vegetation as well as urban features (street lamps, etc.) in the streets. As these models may be very large, we seek a description method that would encode 3D databases at a low memory-storage cost and that would be able to reconstruct the models on the fly.

Given these constraints, the use of L-systems is natural. Indeed, thanks to their amplification role [15], L-systems allow the generation of complex models using a small set of input parameters that are the parameters of the axiom (see Fig. 1). Although the size of the file used to encode a grammar used to describe a model can be quite large, the grammar can be used to describe a single building style that is used to generate a large set of different building models matching the given style (see Fig. 1). Furthermore, using L-systems naturally allows the description of plants, since the L-systems were primarily designed for this purpose. Thus, the use of L-systems is of interest for meeting all of our objectives. Finally, the use of L-systems is highly scalable and portable, since the L-systems consist of scripts that can be parsed and rewritten on the fly.

By contrast, the turtle paradigm proposed by L-systems is not really convenient to describe models of buildings, or more generally, 3D models or object hierarchies. Therefore, we have extended different parts of the L-system language to be able to use generic objects as rule parameters. As we will see in this paper, these generic objects are generated by functions that replace classical terminal symbols. Our extended L-system is thus called a functional L-system (FL-system). We have also introduced the possibility of controlling the parallel

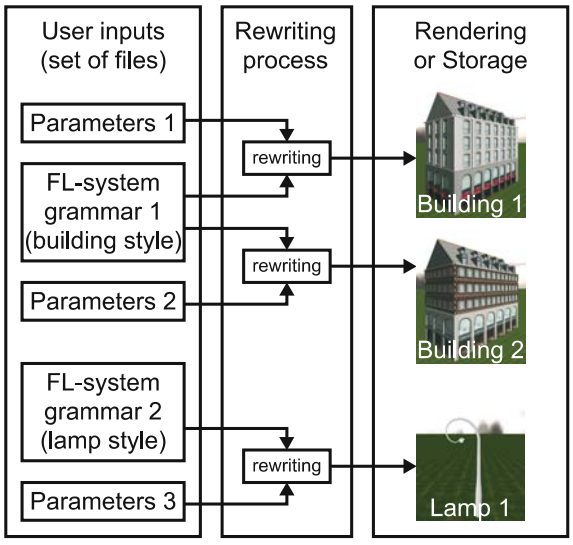

Fig. 1. Generation of geometric models using FL-system grammars. The output of each rewriting is a geometric model that can be stored in a new file or directly used for rendering

rewriting process. Thanks to these two major contributions, our grammars are now able to generate any kind of VRML97 [8] scene graph and mesh on the fly. Thus, we are capable of describing grammars for the generation of building models, urban features, as well as vegetation. The process of generating geometric models using FL-system grammars is illustrated by Fig. 1.

2 Related work

Procedural techniques have been studied for many years in order to provide an efficient alternative to exhaustive representation of complex geometric objects. Such techniques propose the drastic reduction of the size of geometric representations thanks to the use of procedures (functions or rules) for geometric model construction. Rewriting systems such as L-systems [7], shape grammars [4, 16], or instant architecture [18] can be considered as procedural techniques.

L-systems [7], proposed by Lindenmayer as a basis for a theory of biological development, are parallel string rewriting systems. Most L-systems use a Logo-style turtle [1] as a geometric interpretation of strings. The basic idea is to define a state of the turtle as a set of attributes such as its position and orientation in the Cartesian coordinate system, its color, its linewidth, etc. Various applications of L-systems using the turtle interpretation have led to the generation of fractals and to the realistic modeling of plants and networks of streets. Nevertheless, the turtle interpretation seems quite inappropriate for particular models, especially for buildings. This particular point will be discussed later. From the generation of fractals and the modeling of plants [2, 12–14, 17] to street modeling [11], L-system formalisms have quickly evolved. The rules of such systems can be parametric, stochastic, or conditional. These powerful formalisms increase the accuracy of realistic plant models through the simulation of the growth process. However, the lack of high-level parameters and the use of a turtle in the geometric interpretation of strings make current L-systems inappropriate for modeling buildings. Moreover, in L-systems, the turtle interpretation is made once the rewriting process is over: the generation of geometry is made as a postprocess.

Hart [3, 5] describes the object instancing paradigm, a procedural modeling technique allowing efficient representations of objects presenting redundancy. It is possible to convert a turtle-based L-system into an instancing hierarchy. The turtle controls are replaced by geometric primitives and affine transformations. L-system productions are converted into an instancing hierarchy.

Used in the generation of architectural models, shape grammars [16] allow the definition of various styles through the definition of composition rules. However, the derivation process (the application of rules) is not automatic since the rules are mostly chosen by the user.

Instant architecture [18] presents a mechanism for the automatic generation of architectural models based on shape grammars. Unlike previously presented formalisms, in which each model is defined by a grammar, this framework is based on a large database of grammar rules defining a variety of designs. This database contains two types of grammars: a split grammar, which derives shapes, and a control grammar, which sets out the shapes spatially. Both grammar rules are selected by an attribute-matching system, which also attributes the grammar parameters. While this system offers a powerful tool for the creation of building models from different sizes of input data, model generation remains quite long. Moreover, it does not allow the definition of any geometric models and is only appropriate for the modeling of buildings.

For a good comparison of L-systems and Chomsky grammars, one can refer to Prusinkiewicz [14]. Briefly, L-system rewriting rules refer to an evolution of a component over time, whereas Chomsky grammar rules specify a decomposition over space. Moreover, the L-system rewriting process is controlled by the number of derivation steps, which predefines the terminal age of the system, while Chomsky grammar decomposition is limited by atomic terminal symbols.

Since the motivation of our approach is to provide a generic framework for procedural modeling, we develop a hybrid derivation process combining both iterative and parallel applications of rules. We also redefine the interpretation of terminal symbols by using functions instead of strings or shapes. Moreover, with the manipulation of object references as parameters, such functions

can be interpreted in many different ways according to the user's choices. Finally, we present a *for* expression offering a simple and effective way to generate sets of objects. Our approach allows for the definition of complex structures such as VRML97 scene graphs and the use of object instances.

3 Overview

This paper proposes our extended L-system-based language, a formalism that is well-suited to procedural geometric modeling. As an extension of L-systems, we introduce the generalization of rule parameters to generic object references to allow the manipulation of scene graphs and complex geometric structures. The system is, thanks to these mechanisms, strictly independent of the underlying data structures that are created and modified by terminal symbols corresponding to functions. In our system, functions are embedded as dedicated extensions. Finally, a new type of grouping rules dedicated to the description of *sets*, as well as a parallelism control scheme, are defined in order to increase the expressiveness of the language.

Thus, in order to specialize the system to geometric modeling, we developed the following two extensions: the first one, which is an algebraic extension, enables geometric transformations such as translation or rotation on vertex lists. The second, which is a VRML97 extension, allows the creation of VRML97 nodes and the modification of fields in order to generate scene graphs and geometric objects automatically.

In this way, we use the whole system for the definition of buildings by introducing the notion of building style. A building style is defined as a set of rules, which, taking into account different parameters, give birth to a set of different buildings. From now on, the set of rules defined by the user will be called the *grammar*. Finally, the generative use of our system is also illustrated by the modeling of simple plants and urban features such as street lamps.

The rest of this article is structured as follows: Sect. 4 defines our rewriting system and its specificities. Then, Sect. 5 describes our extensions dedicated to the creation of 3D models, while Sect. 6 presents an application of the system to the generation of buildings. Finally, Sect. 7 illustrates and discusses our results, and is followed by our conclusions in Sect. 8.

4 FL-system

In this section, we present context-free L-systems (also known as OL-systems) upon which our FL-system was constructed. Then we describe the major functionalities we added to the existing formalism.

4.1 Context-free L-systems

A stochastic and parametric context-free L-system can be defined as an ordered 5-tuple $G_\pi =< V, \Sigma, \omega, P, \pi >$, where

- V is the *alphabet* of the system,
- Σ is the *set of formal parameters*,
- $\omega \in (V \times \mathrm{Re}^*)^+$ is a nonempty parametric word called the *axiom* (Re is the set of real numbers),
- $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \rightarrow (V \times E(\Sigma))^*$ is a finite *set of productions*,
- function $\pi : P \rightarrow (0, 1]$, called *probability distribution*, maps the set of productions to the set of *production probabilities*,
- $\mathcal{C}(\Sigma)$ denotes a *logical expression* with parameters from Σ,
- $E(\Sigma)$ is an *arithmetic expression* with parameters from the same set.

An FL-system basically has most of the capabilities of context-free L-systems, such as parameterized rules, conditions, and stochastic rule selection. Each of these rule types is illustrated in Fig. 2 using the formalism presented in the work of Prusinkiewicz et al. [13]. $n \in \Sigma$ is a parameter; $n = 0$ in $\mathcal{C}(\Sigma)$ is a conditional expression; and $p \in (0, 1]$ is a probability.

$$
\begin{array}{llll}
parameters & A(n) & \rightarrow & B(n)\ D \\
condition & B(n) \quad : n = 0 & \rightarrow & D \\
stochastic & D & \xrightarrow{p} & E \\
 & D & \xrightarrow{1\text{-}p} & F
\end{array}
$$

Fig. 2. A simple L-system example that illustrates parameterized, conditional, and stochastic rules

4.2 Object references as generic parameters

Since our goal is to describe the generation of complex structures, we define $\mathcal{F}(\Sigma)$ as the set of *functions* with parameters in Σ. In our system in particular, $\mathcal{F}(\Sigma)$ includes functions such as the creation and modification of lists, vectors, and matrices.

Additionally, we define $\mathcal{O}$ as the set of generic object references. A rule parameter p is thus defined over $\mathcal{P} = \mathrm{Re} \cup \mathcal{O} \cup \mathcal{S}$, where $\mathcal{S}$ is the set of strings and $\mathcal{P}$ the set of rule parameters. Parameters can therefore be real numbers, strings, or generic object references that can refer, for example, to vectors or higher-level objects such as VRML97 nodes, as will be seen in Sect. 5. Therefore, the type of the generic objects that are referred to by parameters do not need to be known by the rewriting system but only by the functions that will require them.

In the set of functions $\mathcal{F}(\Sigma)$, we can distinguish two types of functions. The first type includes the functions

that do not return a value. These functions, which replace the usual L-system terminal symbols, will be called *terminal functions*. They are used to generate or modify the content of a generic object whose reference is given as a parameter of the function. The other type includes the functions that return a value in $\mathcal{P}$. These functions, mainly used as *parameter functions*, are typically used to create the generic objects and return a reference to a created object. When these functions are used as rule parameters, they are first executed and their returned values are then used as the effective parameters of the corresponding rule during its rewriting.

Taking these two new definitions into account, we redefine the set of productions P and the axiom ω as follows:

- $\omega \in (V \times \mathcal{P}^*)^+$
- $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \rightarrow (V \times (E(\Sigma) \cup \mathcal{F}(\Sigma)))^*$

4.3 Functions as terminal elements

As said before, the terminal symbols of FL-systems are functions with which an execution scheme is associated. These terminal functions are typically used to generate or modify the content of objects generated by parameter functions using geometric data. Therefore, an important feature of FL-systems is the generation of geometry during the rewriting process. Whereas the turtle interpretation is made sequentially and as a postprocess [12], the execution of terminal functions, in our system takes place during the rewriting process. Actually, our rewriting system operates on a rewriting queue. At the beginning of the derivation process, this rewriting queue only contains the axiom of the system. As the axiom is rewritten, it is replaced by its successors in the queue. When terminal functions are found in this queue, they are applied immediately; that is to say, they are executed. The use of object references as parameters allows the functions to operate on objects provided by previously rewritten rules or those generated by parameter functions. More precisely, this allows a terminal function to generate geometry and to initialize a previously generated object with this new geometry at any step of the rewriting process.

4.4 Sets and iterations

Defining a set of n derivation branches can be done, in L-systems, using a simple recurrence as shown in the following example:

$$A(n) : n = 0 \rightarrow B$$
$$A(n) : n <> 0 \rightarrow A(n-1)B$$

However, this notation is not intuitive and hides the existence of the generated set of elements. Moreover, in our opinion, this is contrary to the parallel spirit of L-systems to define a set of elements this way. Finally, the derivation process follows a comb (Fig. 3a) whereas we would prefer a flat tree (Fig. 3b). In order to provide a true set definition, we add the well-known iterative notation:

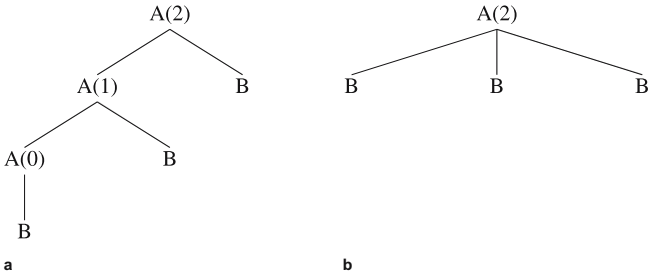

Fig. 3. Trees of generation with $n = 2$. (**a**) for the recurrence example. (**b**) for the iteration example

$$A(n) \rightarrow for(i = 0; i <= n; i++)\ B$$

This notation allows the definition of sets of derivation branches in a simple, clear, and efficient way. The generated structure is similar to the notion of group as defined by Leyton [6]. Note that even if this notation is basically iterative, it is used to describe an ordered set of symbols that are rewritten in parallel. That is to say, the *for* expression builds an ordered set of symbols with which associated rewriting rules can potentially be parameterized with the value of the iterator i. The rewriting of these rules starts only after all of the parameters are passed to the ordered child rules. This means that the sequential notation is used in a first step, followed by the parallel rewriting. Thus, a child rule parameterized with $i = 0$ knows that it is the first enumerated rule even if another rule parameterized with $i = 4$ is rewritten before it.

Let $\mathcal{R}(\Sigma)$ be a *repetition expression* (a *for* expression) with parameters in Σ and $\mathcal{R}(\Sigma) \times V$ denoting the repetition of symbol V. The set of productions P is thus redefined as follows:

$$P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \rightarrow (\mathcal{R}(\Sigma) \times (V \times (E(\Sigma) \cup \mathcal{F}(\Sigma)))^*)^*$$

4.5 Parallelism tuning

From the use of references and functions arises the constraint of synchronizing the creation of objects with functions operating on them. That is why we introduce the symbol *'!'* as a synchronization operator. This operator cuts the parallel derivation and delays the rewriting of the marked rules until the rewriting of the non-terminal symbols written on its left has been completed. The example depicted in Fig. 4 specifies that the derivation of rule C must be done after the complete derivation of rule B. Therefore, if B dynamically generates a structure, C can

$$
\begin{array}{ll}
A(n) \rightarrow & B(n)\ !C\ D \\
B(n) \rightarrow & for(i = 0; i < n; i++)\ E \\
C \rightarrow & f1 \\
D \rightarrow & E\ E \\
E \rightarrow & f2
\end{array}
$$

Fig. 4. The '!' symbol is used to control the parallel rewriting process. It ensures that *B* is completely rewritten before *C*, whereas *D* can be rewritten at any moment

use it. Note that in this example, $f1$ and $f2$ are terminal functions.

As our system does not operate on strings but on objects, we use a derivation queue to deal with the derivation process. Each parallel derivation consists of rewriting each rule contained in the queue. Terminal symbols, being functions, are executed, whereas non-terminal symbols, say rules, are rewritten into the next step of the queue. Figure 5 shows the successive steps of the derivation of the above system. Note that in this example, *D* is rewritten before *C* due to our parallel implementation, but it could also be rewritten after.

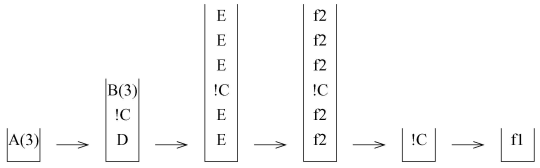

Fig. 5. Successive states of the derivation queue during the derivation of the system described in 4.5. Note that rule *C* is not rewritten until it comes to the top of the queue; it waits for the complete derivation of rule B

This synchronization allows a strictly parallel derivation process as well as a sequential or a hybrid one. As parameters can be references to data sets, this mechanism ensures that an object is created before being manipulated.

Let $\mathcal{S}_y$ be a synchronization expression (*'!'* or nothing); the set of productions P is finally redefined as follows:

$$
\begin{aligned}
P \subset\ & (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \\
& \rightarrow (\mathcal{S}_y \times \mathcal{R}(\Sigma) \times (\mathcal{S}_y \times V \times (E(\Sigma) \cup \mathcal{F}(\Sigma)))^*)^*
\end{aligned}
$$

4.6 Potential of the grammar

To summarize this section, we have presented a generic rewriting system offering many possibilities. Indeed, the functional characteristics of FL-systems together with the enhanced control of parallelism allow the development of new extensions. Finally, we introduced a better control of the derivation process through the *for* expression.

Furthermore, as a geometric model is generated during the rewriting process, it is possible to perform a lazy rewriting for the generation of levels of detail. The idea is that the rewriting process can be stopped after the generation of a level and restarted when its subsequent level is requested. For instance, this mechanism could be used to generate a small set of leaves for a tree and to add more leaves as the viewpoint gets closer to the model.

5 Application to 3D models

Now that our generic rewriting system is defined, we can extend it to the generation of 3D models. For this purpose, new functions will be introduced to perform on-the-fly algebraic computations, turtle simulation, as well as the generation and modification of VRML97 data structures.

5.1 Algebraic extension

We lay down basic algebraic functions adapted to geometric transformations, such as translation, rotation, and scaling. These terminal functions are defined in $\mathcal{F}(\Sigma)$ as

- *translate*(M, x, y, z),
- *rotate*(M, x, y, z, α),
- *scale*(M, s_x, s_y, s_z),

where M is a reference to a 4×4 homogeneous matrix to which we apply the transformation. Other terminal functions such as

- *multMatrix*(M, L), which apply the matrix M to the elements of list L and parameter functions, and
- *copyMatrix*(M), which returns a copy of matrix M,

are also introduced. Finally, basic mathematical functions such as the cosine and sine are also defined.

5.2 Turtle simulation

To replace the turtle, the user has the opportunity to insert a local Cartesian coordinate system as a parameter of the rules. This solution enables the simulation of the turtle interpretation using geometric objects and geometric transformations as turtle moves. The turtle position vector and its three perpendicular vectors, $\vec{H}$, $\vec{L}$, and $\vec{U}$, are replaced by a local coordinate system handled through a 4×4 homogeneous matrix parameter where $(\vec{H}, \vec{L}, \vec{U})$ is mapped to $(\vec{y}, \vec{z}, \vec{x})$. The module F, which moves the turtle forward, is replaced by a translation along the y-axis and the modules +, -, &, ^, /, and \, by rotations. Figure 6 shows the conversion from turtle interpretation to affine transformations.

Thus, when a terminal function generates a mesh, it can apply the current matrix to its set of vertices in order to place it in the global coordinate system. Such a function can also use the current matrix to initialize a transformation node inside a scene graph generated during the

$+(\alpha)$	$rotate(M,1,0,0,\alpha)$	$\hat{}(\alpha)$	$rotate(M,0,0,1,-\alpha)$
$-(\alpha)$	$rotate(M,1,0,0,-\alpha)$	$/(\alpha)$	$rotate(M,0,1,0,\alpha)$
$\&(\alpha)$	$rotate(M,0,0,1,\alpha)$	$\backslash(\alpha)$	$rotate(M,0,1,0,-\alpha)$

Fig. 6. The turtle rotations. L-system modules and associated rotations in $(\vec{x}, \vec{y}, \vec{z})$, where M is the current matrix

rewriting process. Finally, in order to replace the bracket notation that is used in L-systems to specify a branching in turtle moves, we use the *copyMatrix* terminal function to duplicate the local coordinate system in order to provide each successor rule with a copy. Therefore, each successor rule can operate on its own local coordinate system (that is to say, on its own current matrix). This naturally follows the construction of transformation hierarchies.

5.3 VRML97 extension

To create 3D models, we generate VRML97 [8] objects and scene graphs dynamically. As with traditional L-systems, which use terminal symbols to describe the turtle moves together with their geometric interpretation, we introduce two functions for the creation of VRML97 nodes and the initialization of their fields:

- *defVrmlNode*(*type*, *name*) which has two parameters: *type* is a VRML97 node name (IndexedFaceSet, Transform, etc.) and *name* its optional definition name (for handling the DEF/USE mechanism). This parameter function creates a node, then returns a value in Σ that is a formal parameter pointing to the created node.
- *setVrmlfield*(*node*, *fieldName*, *value*) which has three parameters: $node \in \Sigma$ is the formal parameter pointing to a VRML97 node, *fieldName* the name of the field which is modified, and *value* is its new value. This function is a terminal function.

These two simple terminals allow for the creation of complete VRML97 scene graphs. This extension can be implemented as follows: first, a node is created using the *defVrmlNode* parameter function. Then, a set of rules is applied recursively, creating and/or manipulating data sets representing the future content of the node. Finally, the fields of the node are assigned afterwards using the resulting data. For that purpose, one can mark the *setVrmlField* terminal function with the *!* operator. In this manner, the terminal function is executed only after all predecessors are rewritten; that is, only once the referred VRML97 node is created and the data sets are produced.

The example presented in Fig. 7 illustrates the use of this extension. In the notation we use to describe our grammars, '=' represents the symbol '→' and ( ) an empty list. When used in a parameter expression, the symbol '=' denotes the naming of the parameter. In this example, the grammar is composed of an axiom and two rules named *w*, *initIfs*, and *computeQuad*, respectively. These three symbols are non-terminals.

```
w = initIfs( defVrmlNode(IndexedFaceSet,MY_IFS),
             defVrmlNode(Coordinate));

initIfs( ifsNode, coordNode ) =
  computeQuad( vertices=(), indices=() )
  !setVrmlField( ifs, coordIndex, indices )
  !setVrmlField( coordNode, coord, vertices );
  setVrmlField( ifs, coord, coordNode );

computeQuad( vertices, indices ) =
  conc( vertices, ( 0 0 0, 1 0 0, 1 1 0, 0 1 0 ))
  conc( indices, ( 0 1 2 3 -1 ));
```

Fig. 7. A set of rules that generates and initializes an *IndexedFaceSet* node with a quad produced by the *computeQuad* rule

The axiom *w* derives the rule *initIfs*, which takes two parameters that are references to generic objects. These generic objects are VRML97 nodes that are created by the axiom using the parameter function called *defVrmlNode*. The first object is an `IndexedFaceSet` node and the second is a `Coordinate` node.

The rule named *initIfs* derives the rule named *computeQuad*, which takes two lists references as parameters. These two lists will be filled by the rule *computeQuad*; the first, called *vertices*, is filled with the vertices of the quad, and the second, called *indices*, is filled with the vertex indices that defines the quad. These parameters are named so that they can be given as parameters to two terminal functions, collectively called *setVrmlField*. These two terminal functions initialize the fields of the VRML97 nodes *ifsNode* and *coordNode* with the list of vertices and the list of indices, respectively. Note that these terminal functions are marked with the '!' character. This notation means that the rule *computeQuad* has to be rewritten before the following terminal functions use its result (i.e., the two lists) to initialize the VRML97 data structure. The third terminal function, named *setVrmlField*, finally sets the VRML97 field `coord` of the `IndexedFaceSet` node with the `Coordinate` node (it constructs the node hierarchy). Note that this last terminal function can be placed either before or after the *computeQuad* rule, since it only uses the node references previously produced by the axiom and does not rely on *computeQuad* results.

Finally, the rule called *computeQuad* derives two terminal functions. This last rule generates and stores the quad into the two lists called *vertices* and *indices* that are created by the *initIfs* rule and whose references are given as parameters. The two lists are constructed using the *conc* terminal function that concatenates two lists. For clarity, the quad example we present is very simple, but one can also add texture coordinates as well as material properties to the generated objects. Furthermore, one can use the algebraic extension to describe more complex meshes using parametric surfaces, for example.

5.4 On-the-fly rewriting

In our implementation, the module in charge of the rewriting of the grammars is embedded as a plug-in into a Magellan [10] application that is in charge of the visualization process (see Fig. 8). The grammars are described in VRML97 files as script nodes. When a VRML97 file containing a xcript node is opened, the grammar is analyzed and rewritten by a rewriting thread that runs in parallel to the Magellan rendering thread. Therefore, data can be produced on the fly and placed into the Magellan scene graph in parallel with rendering.

In order to add the generated models into the VRML97 scene graph of Magellan, the axiom is assigned a parameter that is a reference to a VRML97 `Group` node to which the grammar can add the nodes it generates. Thus, once the grammar is rewritten by the rewriting thread, the latter grammar can add the `Group` node that contains the model to the Magellan scene graph.

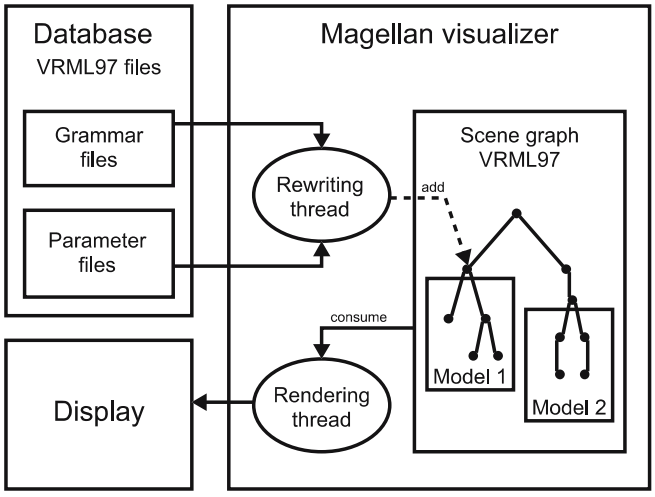

Fig. 8. On-the-fly rewriting process. The grammars are analyzed and rewritten by a rewriting thread that runs in parallel to the Magellan rendering thread

6 The case of buildings

Now that we have extended our system to the modeling of 3D models, we can use it for a more precise application. This section presents a study showing how to model building styles with FL-systems. The basic idea is that we can describe a certain set of buildings with the same grammar. Such a grammar generates not only a building, but a set of buildings matching the style described by the grammar (i.e., their building style). Thus, by changing the parameters of the axiom or by stochastic rule-selection, the same grammar can produce a wide variety of buildings of the same style.

6.1 A building style study

We will focus our study on the description of a particular building style starting from an existing building depicted in Fig. 9a. The associated grammar consists of two main components.

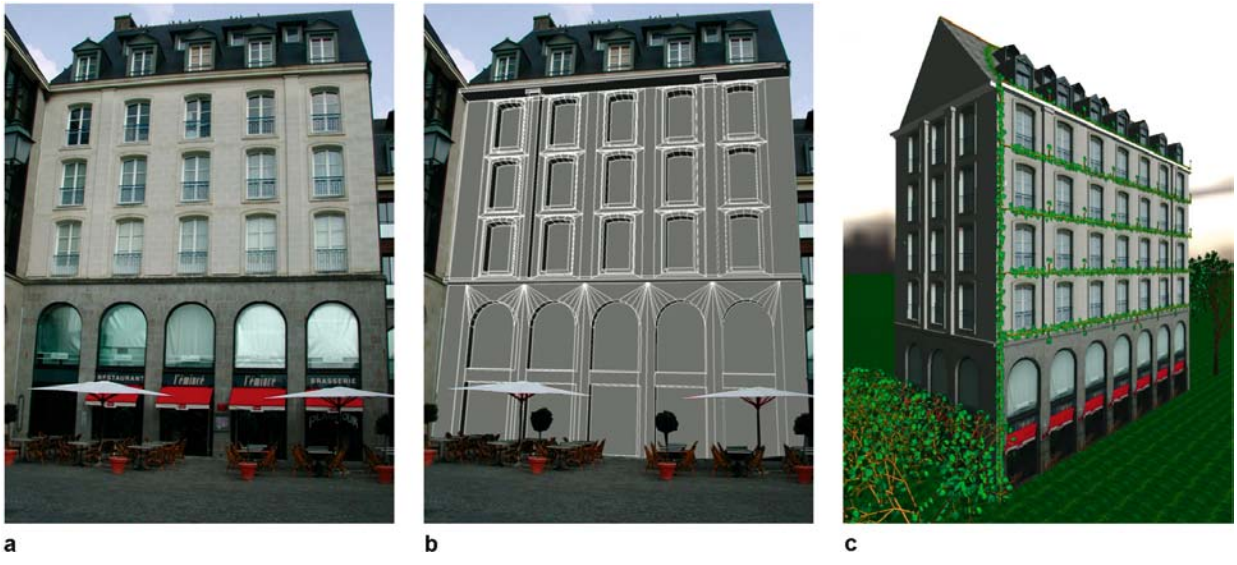

Fig. 9. (**a**) The left picture represents the real model. (**b**) The center picture shows the incrustation of the façade, reconstructed with our grammar, into the picture of the original model. (**c**) The right picture shows a different model generated using the complete building style with different axiom parameters. Note that the plant growing on the façade is generated using another grammar following the same grammar structure as the one used to generate the façade. Other plants are also generated using FL-systems

– Terminal elements, which are atomic architectural elements such as doors and windows. With regards to VRML, the terminal functions generate the geometry that is put into the `Shape` nodes used to describe the model.
– Productions are rules describing the non-terminal architectural elements. Such rules can be rules of decomposition (e.g., decomposition of a façade into a set of floors), growth rules (e.g., to obtain the volume of a building from its footprint), or both. With regards to VRML, these decompositions can be used to generate node hierarchies.

The choice of terminal elements is quite straightforward for the user. It determines the elements that are not decomposed. Terminal elements are, in our grammar, parameterized 3D meshes that describe windows, walls, and doors produced by the terminal functions. The non-terminals are rules of composition or decomposition of terminals as well as of other non-terminals. These rules decompose the space into subspaces and set them out spatially. Finally, the axiom parameters determine the following inputs: the building's footprint as a convex polygon (a list of 3D vertices), the number of floors of the building, and their heights.

According to these parameters, the footprint is used within the growth process to obtain the façades and the roof of the building. Each façade is decomposed into a set of floors composed of a wall, a set of windows and possibly a door. This modeling process, illustrated in Fig. 10, can be seen as a set of growth and decomposition steps.

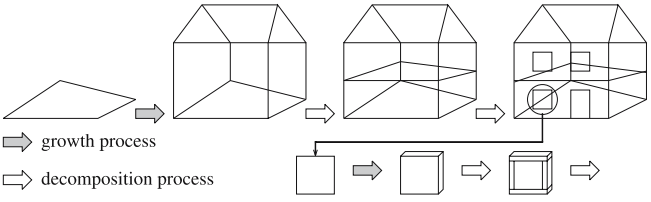

Fig. 10. An alternation of growths and decompositions in the modeling of a building

To extend the footprint of the building into façades, for each vertex of the footprint, we create a local coordinate system and compute the width of the façade as the distance between two consecutive vertices of the footprint. The height of the façade can be computed using the number of floors m and the floor heights h and H given as parameters of the axiom.

Having written a set of rules that compute these parameters, we then write a set of rules that compute the sizes of the different elements of each façade, as well as their placement. This set of rules is presented in Fig. 12. For clarity, we did not detail the generation of the geometry nor the generation or initialization of the VRML97 node hierarchy.

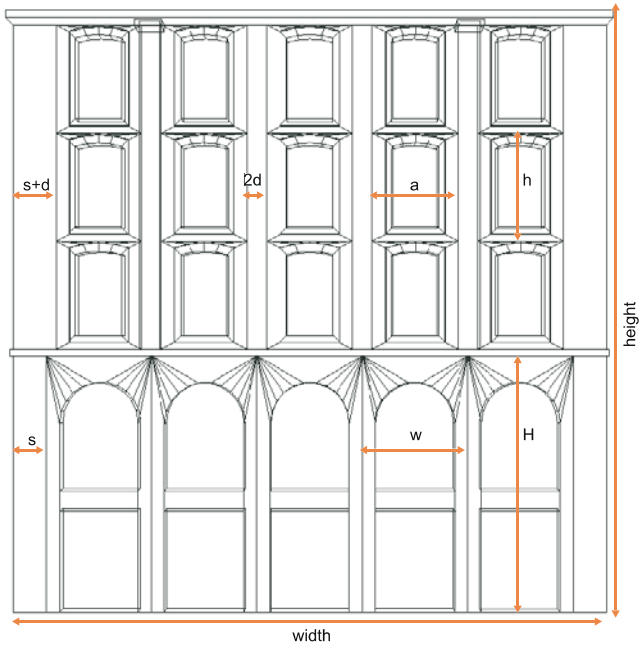

Fig. 11. Input parameters for the façade

The rule called *facade* produces the façades of Figs. 9b and 9c. In this façade style, the sizes of all of the terminal elements are fixed. The only parts that are stretchable are the wedgings of the façade. The parameters that control the structure of the façade are depicted in Fig. 11. The *facade* rule derives to the *layout* rule and provides the rule with parameters: m, the number of floors; $n = \mathit{floor}((\mathit{width} - 2 * \mathit{smin})/w)$, the number of fixed-width vertical sections (window and pier) that can be placed along the width of the façade; w, the width of a vertical column, H the height of the first floor; a, the width of a window; d, the half-width of column and pier; h, the height of other floors; and $s = (\mathit{width} - n * w)/2$, the difference between the wedging width and the part of the façade that is not large enough to add another window (s is the stretchable value).

The rule called *layout* derives two rules – *firstFloor*, which is used to place the elements of the first floor, and *nextFloors*, which places the elements of all of the other floors. Thus, *layout* performs a part of the decomposition of the façade.

Then, the rule *firstFloor* derives three rules – the two rules called *wedging* are used to produce the geometry of the left and right wedgings of the first floor, while the *for* expression is used to generate the repetition of arcades that compose the first floor. Therefore, *firstFloor* is also a decomposition rule, whereas *wedging* and *arcades*, which are not detailed here, are rules that produce the geometry of the terminal elements.

```
facade(width,m,w,H,a,d,h,smin...) =
    layout(m,n=floor((width-2*smin)/w),
           w,H,a,d,h,(width-n*w)/2,...)
    ...;

layout(m,n,w,H,a,d,h,s,...) =
    firstFloor(n,w,H,s,...)
    nextFloors(m-1,n,w,H,a,d,h,s,...)
    ...;

firstFloor(n,w,H,s,...) =
    wedging(0,H,s,...)
    for(i=0;i<n;i++) arcade(i,n,w,H,...)
    wedging(s+n*w,H,s,...);

nextFloors(m,n,w,H,a,d,h,s,...) =
    wedging(0,H,s+d,m*h,...)
    pilaster(s+d+a,H,2*d,m*h,...)
    for (i=0;i<m;i++)
        for(j=0;j<n;j++)
            window(s+d+j*w,H+i*h,...)
    for(j=2;j<n-1;j++)
        pier(s+j*w-d,H,2*d,m*h,...)
    pilaster(s+(n-1)*w-d,H,2*d,m*h,...)
    wedging(s+n*w-d,H,s+d,m*h,...);

...
```

Fig. 12. A part of the grammar that generates the façades presented in Figs. 9b and 9c

Finally, *nextFloors* derives six rules – four rules that are used to produce the geometry of wedgings and columns, and two *for* expressions that are respectively used to perform the placement of windows and piers.

6.2 Optimizations

A geometric model can be well-suited for different purposes, according to the way it is generated by a grammar. All of our grammars, dedicated to the generation of buildings, generate a set of levels of detail (LOD) that is used to accelerate the rendering process (Fig. 13). Our grammars always share a maximum of geometry between different LODs in order to reduce the memory (RAM) required to store the VRML97 model of the building. This instance-sharing is performed through the use of the VRML97 DEF/USE mechanism; for instance, if the geometry of the walls is the same for the second and third levels, it is then "DEFined" for the second level and "USEd" for the third level.

In addition to this general mechanism, we also write, for each building style, two versions of the grammar. The first version, which is designed to produce models that are optimized for real-time rendering, generates a geometric model where the geometry that composes the building is described using absolute coordinates. This prevents the graphics hardware from having to compute costly transformations. Furthermore, all elements using the same texture map are encoded into a single `Shape` node. For instance, in our building-style study, the geometry of all of the windows of the upper floors can be encoded into a single `Shape` node whose `Material` node describes the visual appearance and the texture map of these windows. This modeling structure significantly prevents the graphics hardware from swapping between different textures.

The second version is designed to produce compact models. Therefore, using this version, it is possible to generate a huge number of models requiring a smaller amount of RAM. Thus, this version generates a more complex VRML97 node hierarchy, each node of which places the geometry of its child `Shape` nodes, using `Transform` nodes. For instance, the geometry of a window that is generated only once into a `Shape` node using a local coordinate system is then instantiated multiple times. Each instance of the window *Shape* is then placed to compose the façade using transformation nodes. It is the description of the meshes that consumes memory; therefore, this model structure generates more compact models. On the other hand, the rendering process has to perform a more complex scene graph traversal and the graphics hardware has to compute costly geometric transformations to produce the final images. Furthermore, distributing the objects into different `Shape` nodes introduces more texture swaps.

To summarize, our grammars always generate LODs, which is, in our opinion, an important basic optimization. Furthermore, one of the two versions might be chosen according to the application targeted by the user. In any case, we are able to provide a grammar that fits one of the following constraints: speed and compactness.

7 Results

To experiment with FL-systems, we defined a small set of grammars representing buildings, as well as plants and trees. This set of files is very compact, since, for example, the set of 40 rules representing the model of Fig. 13 takes only 23KB.

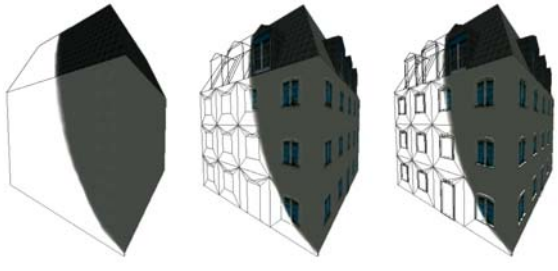

Fig. 13. LODs generated by a grammar

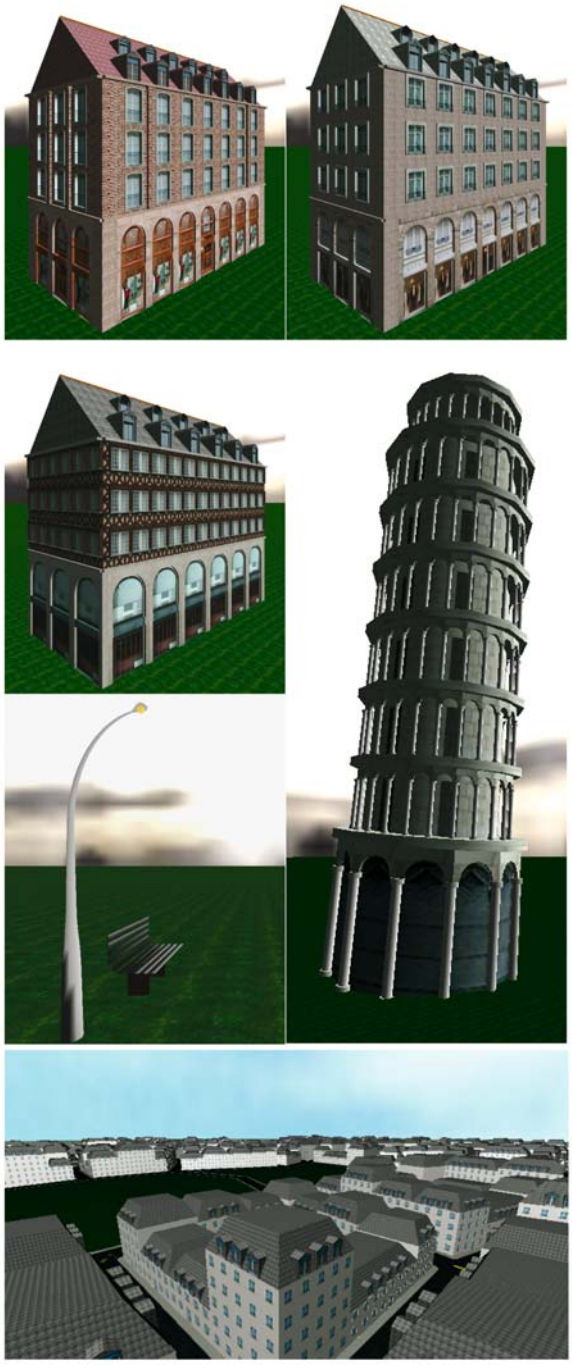

Fig. 14. Top: example of generated models using few rules and different normalized texture sets. Center left: example of generated urban features. The street lamp is generated using parametric surfaces. Bottom: bird's eye view of a city

We defined a few rules representing patterns of windows, walls, doors, and roofs. By combining these rules, we wrote different building grammars. Defining a set of texture names as an input parameter of the axioms allows the use of different sets of textures within the same grammar. This combination of models and texture sets allows the generation of a rich set of different buildings. Finally, using footprint and elevation parameters also produces different building models of a given style. Figure 14 shows different building models we generated using three grammars and six texture sets. The computation of each model took less than 20 ms on a 2.4 GHz Intel Pentium IV.

As our objective was to provide a generic language for procedural geometric modeling, we also tested the generation of plants. Figure Fig. 9c shows the output of two L-system grammars directly translated into our language. We used the examples given in [13](Figs. 1.25 and 2.6).

In [9, 10] we integrated procedural models in a street network that was generated automatically. For further details, one could refer to those papers, which show how to manage the interactive visualization and transmission of a very large city model over low-bandwidth networks. In this model, shown in Fig. 14, roads and buildings are procedurally generated using FL-systems. Furthermore, a new parameter is given to the axiom of the building grammar. This parameter contains the heights of the neighboring buildings. With a knowledge of the neighborhood, the grammar that generates the building can avoid the creation of unuseful geometry (e.g., shared walls) and aberrations (e.g., a window cut by the wall of a neighboring building).

8 Conclusion

We have proposed and developed three extensions to the classical L-system: an adaptive parallel rewriting mechanism, a set derivation, and the use of generic parameters together with terminal and parameter functions replacing strings. The first extension offers complete control of the derivation process, whereas the second allows the flattening of the derivation tree. The third extension offers easier manipulation of generated data structures, since interpretation of the grammar is performed during the rewriting process. Furthermore, the functional properties of FL-systems allow any kind of new user extensions.

The resulting formalism lies between L-systems and a Chomsky grammar. In fact, its derivation process combines the L-system terminal age and the Chomsky grammar terminal symbols. It allows the conditional rules not only to choose the rule to be applied, but also to stop the derivation process.

Moreover, this language has proved to be efficient for interactive generation of complex 3D models optimized for real-time rendering. It has been tested for the generation of buildings, plants, trees, and urban features.

Further research will focus on the lazy rewriting of our grammar and the generation of dynamic scenes. We are also studying the extraction of grammars from input images using vision-based methods and user inter-

action. Finally, we intend to illustrate the flexibility of our language through the use of other geometric languages (different from VRML) such as solid modeling languages.

References

1. Abelson, H, diSessa AA (1981) Turtle geometry. MIT Press, Cambridge
2. Bloomenthal J (1985) Modeling the mighty maple. In: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp 305–311
3. Ebert DS, Musgrave FK, Peachey D, Perlin K, Worley S (2003) Texturing and modeling: a procedural approach, 3rd edn. Kaufmann, San Francisco
4. Gips J (1974): Shape grammars and their uses. Dissertation, Stanford University
5. Hart JC (1992) The object instancing paradigm for linear fractal modeling. In: Proceedings of the conference on Graphics interface '92. Kaufmann, San Francisco, pp 224–231
6. Leyton M (2001) A generative theory of shape. Lecture notes in computer science, vol. 2154. Springer, Berlin Heidelberg New York
7. Lindenmayer A (1968) Mathematical models for cellular interactions in development, I & II. J Theor Biol 18:280–315
8. Marrin C, Carey R, Bell G (1997) A VRML specification, 1997. http://www.vrml.org/Sprecifications/VRML97
9. Marvie J-E, Perret J, Bouatouch K (2003) Remote interactive walkthrough of city models. In: Proceedings of Pacific Graphics, IEEE Computer Society, October 2003 2:389–393
10. Marvie J-E, Perret J, Bouatouch K (2003) Remote interactive walkthrough of city models using procedural geometry. Technical Report PI-1546, IRISA, July 2003. http://www.irisa.fr/bibli/publi/pi/2003/1546/1546.html
11. Parish YIH, Müller P (2001) Procedural modeling of cities. In: Proceedings of SIGGRAPH 2001, Los Angeles, CA, USA, August 2001, pp 301–308
12. Prusinkiewicz P, James M, Mech R (1994) Synthetic topiary. In: Proceedings of Computer Graphics and Interactive Techniques, pp 351–358
13. Prusinkiewicz P, Lindenmayer A, Hanan JS et al. (1990) The algorithmic beauty of plants. Springer, Berlin Heidelberg New York
14. Prusinkiewicz P, Mundermann L, Karwowski R, Lane B (2001) The use of positional information in the modeling of plants. In: Proceedings of Computer Graphics and Interactive Techniques, pp 289–300
15. Smith AR (1984) Plants, fractals, and formal languages. In: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp 1–10
16. Stiny G (1975) Pictorial and formal aspects of shape and shape grammars. Birkhauser, Basel
17. Van Haevre W, Bekaert P (2003) A simple but effective algorithm to model the competition of virtual plants for light and space. J WSCG 2003 11(3):464–471
18. Wonka P, Wimmer M, Sillion F, Ribarsky W (2003) Instant architecture. In: Proceedings of SIGGRAPH 2003, July 2003, pp 669–677

JEAN-EUDES MARVIE is a computer science engineer (INSA 2001). He was awarded a PhD in computer science in the field of computer graphics in 2004. His research interests are: real time rendering, large models visualization and generation, procedural modeling, distributed applications dedicated to interactive visualization, rendering on large displays and virtual reality. He is currently doing a postdoc fellowship, granted by the INRIA Futurs, at the LABRI (France).

JULIEN PERRET is currently a PhD student at the IRISA (a computer science research unit located in Rennes, France). He received a MSc degree from the University of Rennes 1 and a MEng degree from the National Institute of Applied Sciences in 2002. His research interests include urban and natural virtual environments modeling, procedural modeling techniques, architectural models, and evolving virtual environments.

KADI BOUATOUCH is an electronics and automatic systems engineer (ENSEM 1974). He was awarded a PhD in 1977 and a higher doctorate on computer science in the field of computer graphics in 1989. His research interests are: global illumination, lighting simulation and remote rendering for complex environments, real-time high fidelity rendering, parallel radiosity, virtual and augmented reality and computer vision. He applied his research work to infrared simulation and tunnel lighting. He is currently Professor at the university of Rennes 1 (France) and researcher at IRISA. He is member of Eurographics, ACM and IEEE. He is and was member of the program committees of several conferences and workshops, and referee for several Computer Graphics journals like: The Visual Computer, IEEE Computer Graphics and Applications, IEEE transaction on Visualization and Computer Graphics, IEEE transaction on image processing, ACM Transaction on Graphics, etc. He also acted as a referee for many conferences and workshops.