Design and implementation of the L+C modeling language

Radoslaw Karwowski and Przemyslaw Prusinkiewicz
Department of Computer Science
University of Calgary

Abstract

L-systems are parallel grammars that provide a theoretical foundation for a class of programs used in procedural image synthesis and simulation of plant development. In particular, the formalism of L-systems guides the construction of declarative languages for specifying input to these programs. We outline key factors that have motivated the development of L-system-based languages in the past, and introduce a new language, L+C, that addresses the shortcomings of its predecessors. We also describe the implementation of L+C, in which an existing language, C++, was extended with constructs specific to L-systems. This implementation methodology made it possible to develop a powerful modeling system in a relatively short period of time.

1. Background

L-systems were conceived as a rule-based formalism for reasoning on developing multicellular organisms that form linear or branching filaments [Lindenmayer, 1968]. Soon after their introduction, L-systems also began to be used as a foundation of visual modeling and simulation programs, and computer languages for specifying the models [Baker and Herman, 1972]. Subsequently they also found applications in the generation of fractals [Szilard and Quinton, 1979; Prusinkiewicz, 1986] and geometric modeling [Prusinkiewicz et al., 2003]. A common factor uniting these diverse applications is the treatment of structure and form as a result of development. A historical perspective of the L-system-based software and its practical applications is presented in [Prusinkiewicz, 1997].

According to the L-system approach, a developing structure is represented by a string of symbols over a predefined alphabet V. These symbols represent different components of the structure (e.g., points and lines of a geometric figure, cells of a bacterium, apices and internodes of a plant). The process of development is characterized in a declarative manner using a set of productions over the alphabet V. During the simulation of development, these productions are applied in parallel steps to all symbols of the string, thus capturing the development in discrete time slices.

Lindenmayer [1971] observed that L-system productions can be specified using standard notation of formal language theory. In the simplest, context-free case, productions have the form:

$$\textit{predecessor} \rightarrow \textit{successor}$$

where *predecessor* is a letter of alphabet V and *successor* is a (possibly empty) word over V. For example, the division of a cell A into cells B and C can be written as $A \rightarrow BC$. In the context-sensitive case, productions are often written as

$$lc < predecessor > rc \rightarrow successor\ ,$$

where symbols < and > separate the strict predecessor from the left context *lc* and the right context *rc* [Prusinkiewicz and Hanan,1989]. Both contexts are words over *V*. For example, the production pair:

$$Y < A > O \rightarrow LYS$$
$$O < A > Y \rightarrow SYL$$

describes asymmetric division of a mother cell *A* into a short daughter cell *S* and long daughter cell *L*, separated by a cell wall *Y*. The sequence of these cells in the filament is guided by the state of the walls that delimit the mother cell, which may be young (*Y*) or old (*O*). Obviously, a complete description of the filament's development would also require productions that characterize the growth of cells and walls over time.

Early L-system-based programming languages closely followed the above notation [Baker and Herman, 1982; Prusinkiewicz and Hanan, 1989]. The needs for expressing increasingly complex models led, however, to the addition of constructs found in other programming languages. A pivotal moment in this evolution was the introduction of parametric L-systems [Prusinkiewicz and Hanan, 1990; Hanan, 1992] and related constructs [Chien and Jurgensen, 1992], which associated numerical attributes to L-system symbols, similar to those found in attribute grammars [Knuth, 1968]. This created a need for calculating new parameter values (in the production successor) on the basis of old ones (found in the predecessor and its context). According to the original definition of parametric L-systems [Prusinkiewicz and Hanan, 1990; Hanan, 1992], these calculations were specified as arithmetic operations on the argument parameters, e.g.

$$A(x) < B(y) > C(z) \rightarrow D(x+y)\ E(y+z)\ .$$

In modeling practice, however, entire procedures soon became needed to calculate new parameter values. Recognizing this need, Hanan [1992] introduced the following syntax for L-system productions:

$$lc < predecessor > rc\ \{\alpha\} : cond\ \{\beta\} \rightarrow successor\ .$$

Here α and β are C-like compound statements, and *cond* is a logical expression that guards production application. A production is applied in stages. First, it is determined whether production predecessor *pred*, surrounded by the left context *lc* and the right context *rc*, matches the given symbol in the string. If this is the case, the compound statement α is executed, and condition *cond* is evaluated. If the result of this evaluation is non-zero ('true'), the second compound statement β is executed. On this basis, parameters values in the production successor are then determined, and the successor is inserted into the resulting string. For example, the following is a valid production:

$$A(x) < B(y) > C(z)\ \ \{r = x*x+ y*y+z*z;\} : r> 2\ \{t = x+y+z;\} \rightarrow D(t)\ E(2*t).$$

At the top level, an L-system with productions in the above form operates in a declarative fashion, by rewriting elements of a string according to their type, context, and the associated parameters. Within each production, however, calculations are performed sequentially, using constructs borrowed from an imperative language. This combination of paradigms suggests two strategies for translating L-system-based languages into a representation directly used by simulation programs [Prusinkiewicz and Hanan, 1992]:

- extend the formal notation for productions with constructs borrowed from an imperative language, or
- extend an existing imperative language with constructs inherent in L-systems.

The modeling program cpfg [Hanan, 1992] and its modeling language [Prusinkiewicz et al., 2000] are representative of the first approach. The interpreter of the cpfg language was constructed following the standard steps of lexical analysis, parsing, and object code generation. Nevertheless, in spite of well-developed methodology for translator construction (e.g. [Aho et al, 1986]), construction of a compiler for a comprehensive language is a large task. Consequently, the cpfg language only includes a limited subset of C-like statements; for example, it does not support user-definable functions and typed parameters associated with the modules. As a result, while simple L-system models can be expressed using cpfg language in an elegant, compact manner, specification and maintenance of larger models becomes difficult.

An alternative approach, first suggested in [Prusinkiewicz and Hannan 1992], is to create an L-system-based programming environment by extending an existing language with support (classes, libraries) specific to L-systems. Using this approach, Hammel [1996] implemented differential L-systems [Prusinkiewicz et al., 1993] in SIMULA, and Erstad [2002] implemented an L-system-based programming environment in LISP. Both implementations preserve the syntax of the underlying languages (SIMULA and LISP). In contrast, Karwowski [2002] implemented the L-system-based programming language L+C by extending the syntax of C++ [Sievanen]. We describe here the design and implementation of this language.

2. The L+C modeling language

The key new elements introduced in the L+C modeling language are:

- typed module parameters, including all primitive and compound data types (structures) supported by C++
- productions with multiple successors
- extension of the notion of context-sensitivity with the ‘new context’ constructs, which speed up information transfer across simulated structures.

In addition, by virtue of being based on the C++ language, L+C has the full expressive power of C++. In particular, user-defined functions are supported as in C++.

At the top level, an L+C program is a set of declarations for:

- Structures and classes,
- Global variables,
- Functions,
- Modules,
- The axiom,
- The derivation length,
- Productions,
- Decomposition rules,
- Interpretation rules,
- Control statements.

The declarations of structures, classes, variables and functions have exactly the same syntax and meaning as in C++. The remaining declarations are specific to L+C, and are described below.

2.1. Module declarations

Modules are the elements of the L-system string. A module consist of an identifier (which must follow the C++ syntax [Stroustroup, 1991]) and an optional list of parameters. In L+C modules have to be declared before they can be used. Declaration specifies the number and types of parameters that are associated with the given module type using the following syntax:

`module` *identifier* (*parameter-list*$_{opt}$);

Examples of valid module declarations are:

```
module A(); // module A with no parameters
module N(float); // module N with one parameter of type float
module Metamer(int, MetamerData); // module Metamer with a
                       // parameter of type int and
                       // a user-defined type MetamerData
```

2.2. Axiom declaration

The axiom declaration specifies the initial L-system string using the following syntax:

`axiom`: *parametric-string*;

where the *parametric-string* must be non-empty. Assuming that the modules have been declared as in Section 2.1, and `s_init` is a structure of type `MetamerData`, the following is a valid axiom declaration:

```
axiom: Metamer(1,s_init) N(0.25) A();
```

2.3. Derivation length specification

Derivation length is the number of derivation steps for the simulation. It is specified using the syntax:

`derivation length`: *integer-expression*;

2.4. Specification of productions

The syntax of productions is a combination of the formal L-system notation and the C++ syntax for function definition. In general, it has the syntax:

predecessor:
{
    production body
}

The predecessor has one of the following forms:

new-left-context << *left-context* < *strict-predecessor* > *right-context* :
left-context < *strict-predecessor* > *right-context* >> *new-right context*:

The strict predecessor specifies the part of the string being rewritten by the production. It can be a single module, as assumed in the usual definition of L-systems, or a string of several modules, as defined for pseudo-L-systems [Prusinkiewicz, 1986]. The optional left and right contexts are strings of modules that need to be in the neighborhood of the strict predecessor in order for the production to apply. The new contexts specify the modules that must be present in the neighborhood of the production successor, in the string being derived. This information is easily available if the string is being rewritten in a particular direction: from left to right in the case of new left context, and from right to left in the case of new right context (Figure 1). In theory, two-sided new context could also be defined, but its implementation is more difficult and, therefore, it is not supported by L+C.

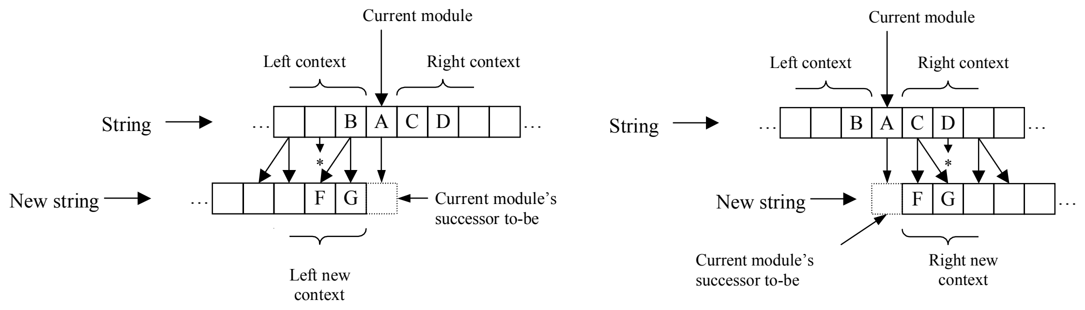

Figure 1. Context of L-system productions. Left new context is available if the successor string is built left-to-right (left figure). Right new context is available if the successor string is built right to left (right figure).

The parameters that appear in the production predecessor are formal parameters. All the formal parameters of every module in a production predecessor must be listed, even if they are not used in the production body. An example of a valid production predecessor that uses the modules declared in Section 2.1 is:

```
Metamer(i_l, d_l) N(w) < Metamer(i, d) > A()
```

Formal parameters have types determined by the declarations of the respective modules. They are bound to the actual parameters in the string during production application [Prusinkiewicz and Hanan, 1990]. The scope of the formal parameters is the same as the scope of formal parameters in C++ functions.

The production body is a compound statement that may contain any code allowed inside a C++ function. In addition, the production body may include one or more `produce` statements, which specify possible successors of the production. The `produce` statement has the syntax:

`produce` *parameteric-string*$_{opt}$;

where *parameteric-string* is defined as in the axiom (Section 2.2). Each `produce` statement is implicitly followed by a `return` statement. Thus, if several produce statements are present in the production body, the first statement executed terminates the production application. Typically, the choice of alternative successors is controlled by C++ conditional statements.

2.5. Decomposition rules

As defined by Lindenmayer [1968], L-systems operate in discrete derivation steps. Each step consists of a (conceptually) parallel application of suitable productions to all symbols in the predecessor string. This parallelism is intended to capture progression of time by a given interval, the same for all components of the modeled structure. Thus, for example, the L-system production $A \rightarrow BC$ expresses the idea "module *A* develops into modules *B* and *C* over a given time interval." In practice, it is also often necessary to express the idea that a given module is a compound module, consisting of several elements. A logical analysis of the notions "develops over time" and "consists of" was presented by Woodeger [1937]. Prusinkiewicz et al. [2000, 2001] showed that, in a grammar setting, these notions correspond to L-system productions and Chomsky context-free productions, respectively. In L+C, Chomsky productions are called decomposition rules. They are specified using the same syntax as context-free L-system productions, and are identified using the keyword `decomposition`, as in the following example:

```
decomposition:
Metamer(i, d) : { produce Internode(i, d) Leaf(d) Bud();}
```

This production characterizes a `Metamer` as a compound module consisting of an `Internode`, a `Leaf`, and a `Bud`. Obviously, all modules must have been declared earlier in the L+C program.

The integration of decomposition rules into the L-system framework affects the way in which a derivation step is performed [Prusinkiewicz et al., 2000]. In L+C, decomposition rules are applied recursively, after the definition of the initial string by the `axiom` statement (Section 2.2) and after each step of standard L-system production applications (Section 2.4).

2.6. Interpretation rules

Structures generated with L-systems may be visualized by assigning a graphical interpretation to a predefined set of modules [Szilard and Quinton, 1979; Prusinkiewicz, 1986, Prusinkiewicz et al., 2003]. For example, in L+C, a predefined module `F(float)` draws a line of a given length in the current direction (as defined in the turtle geometry [Abelson and diSessa, 1982]); `Line2D (point2D, point2D)` draws a line between two given points, and `SetColor(int)` assigns a color to geometric primitives. From the user perspective, however, it is often more convenient to express the model in terms of modules inherent in the modeling domain (e.g., apices, internodes, and leaves in the case of plant models) rather than directly in terms of modules with a geometric interpretation (e.g., points, lines, and polygons). In order to separate these conceptual and visual aspects of model specification, Kurth [1994] introduced the notion of interpretation rules. Interpretation rules are similar to decomposition rules in that they are context-free Chomsky productions, and are applied recursively, after each derivations step (specifically, after the decomposition rules have been applied). In contrast to decomposition rules, however, interpretation rules do not affect the outcome of the following derivation steps. Instead, they are applied "on the side", producing modules that are passed to the graphical part of the modeling program, and discarded once they have been interpreted (Figure 2).

$$\begin{tikzcd}
\omega \arrow[r, Rightarrow, "G^*"] & \mu_0 \arrow[r, Rightarrow, "L"] \arrow[d, Rightarrow, "I^*"] & \mu'_1 \arrow[r, Rightarrow, "G^*"] & \mu_1 \arrow[r, Rightarrow, "L"] \arrow[d, Rightarrow, "I^*"] & \mu'_2 \arrow[r, Rightarrow, "G^*"] & \dots \\
& \nu_0 & & \nu_1 & & \dots
\end{tikzcd}$$

Figure 2. Generation of a developmental sequence using an L-system with decomposition and interpretation rules. Beginning with the axiom ω, the progressions of strings μ_1, μ_2, μ_3,… results from the interleaved application of decomposition rules G and L-system derivation steps L. The interpretation rules I map strings μ_i into strings ν_i , which are interpreted graphically.

In L+C, interpretation rules are identified using the keyword `interpretation`, as in the following example:

```
interpretation:
Internode(i, d) : { produce SetColor(1) F(d.length); }
```

The above production specifies that module `Internode` will be represented graphically as a straight line, (`F`) drawn using color with index `1`. The line length is specified by field `length` in data structure `d`.

2.7. Control statements

Control statements were introduced by Hanan [1992] (see also Prusinkiewicz et al., 2000]) to specify procedures that are executed at specific points during an L-system-based derivation. In L+C, they are specified using the syntax:

```
Start|StartEach|EndEach|End:
{
        compound statement
}
```

The control statements are executed as follows:

- `Start` is executed at the beginning of the program,
- `StartEach` is executed before every derivation step,
- `EndEach` is executed after every derivation step,
- `End` is executed after the last derivation step.

Any code that is allowed inside a C++ function can be specified as the *compound statement*. Typical uses of the control statements include initialization of global variables, opening and closing of I/O streams, and reporting of simulation statistics after each simulation step.

2.8. Example

A sample L+C program that generates a branching structure is presented below:

```
#include <lpfgall.h>
#include <math.h>

const int Delay = 1;
const float BranchingAngle = 45.0;
const float LengthGrowthRate = 1.33;

derivation length: 17;
```

```
 9
10  struct InternodeData
11  { float length, area; };
12
13  module A(int,float);
14  module Metamer(float);
15  module Internode(InternodeData);
16
17  Start: { Backward(); }
18  ignore: Right;
19
20  axiom: A(0,BranchingAngle);
21
22  A(t,angle) :
23  {
24    if (t<0) // young apex
25      produce A(t+1,angle);
26    else     // mature apex
27      produce Metamer(angle) A(0,-angle);
28  }
29
30  Internode(id) >> SB() Internode(id2) EB() Internode(id3) :
31  {
32    id.area = id2.area + id3.area;
33    id.length *= LengthGrowthRate;
34    produce Internode(id);
35  }
36
37  Internode(id) >> Internode(idr) :
38  {
39    id.area = idr.area;
40    id.length *= LengthGrowthRate;
41    produce Internode(id);
42  }
43
44  Internode(id) >> A(t,angle):
45  {
46    id.length *= LengthGrowthRate;
47    produce Internode(id);
48  }
49
50  decomposition:
51  Metamer(angle) :
52  {
53    InternodeData id = {1, 1};
54    produce
55        Internode(id)
56        SB() Right(angle) A(-Delay,angle) EB()
57        Internode(id);
58  }
59
60  interpretation:
61  Internode(id) :
62  {
63    produce SetColor(2) SetWidth(pow(id.area,.5)) F(id.length);
64  }
```

The modeled structure consists of three types of modules, which are given biologicaly meaningful names `A`, `Metamer`, and `Internode` (lines 13-15). The process of string derivation is performed backward (from right to left) as indicated in the Start statement (line 17). In the process of context matching module Right (used to specify the branching angle in line 56) is ignored (line 18). The initial structure defined by the axiom is a single apex. Its parameters characterize the developmental stage and the branching angle of the next branch that will be produced by this apex. According to the first production (lines 22-28), an immature apex will grow older, and a mature apex will produce a metamer, over the time interval associated with a derivation step. The decomposition rule (lines 51-58) specifies that the metamer consists of two internode segments, and a lateral branch delimited by the language-predefined modules `SB()` (start branch) and `EB()` (end branch). The branch initially consists of a lateral apex, placed at a given `angle` with respect to its supporting internode. The development of internodes is described by the three productions in lines 30 to 48. These productions specifies that an internode will grow in length by factor `LengthGrowthRate` per derivation step. They also determine the cross-section area of each internode as the sum of the cross-sections of internodes supported by it. Specifically, the new context construct is used to accumulate the cross-section of branches when moving from the apices toward the base of the structure. Finally, the interpretation rule (lines 61-64) specifies that each internode will be visualized as a line of length and width determined by the internode parameters. The structure generated by this L-system is shown in Figure 3.

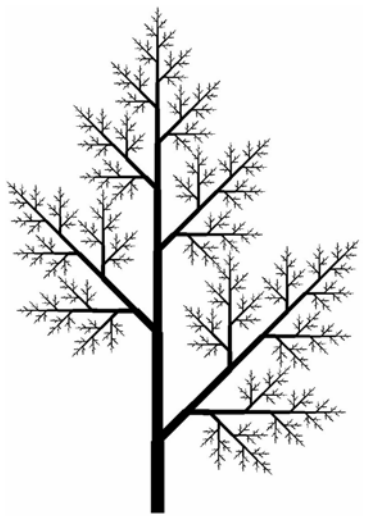

Figure 3. Example of a structure generated by the sample L-system.

3. Implementation of the L+C translator

The main difference between L+C and C++ is not at the level of syntax, but at the level of the programming paradigm: L+C is a declarative language, whereas C++ is an imperative language. Furthermore, L+C programs operate in a specific topological space [Giavitto and Michel, 2001, 2002] of a linear or branching string, whereas C++ does not presuppose any such space. Despite these differences, most of the L+C grammar is the C++ grammar. Given that, the process of compiling and executing an L+C program consists of translating some specific L+C constructs into C++, while leaving other constructs are left intact. This leads to the modeling system design shown in Figure 4.

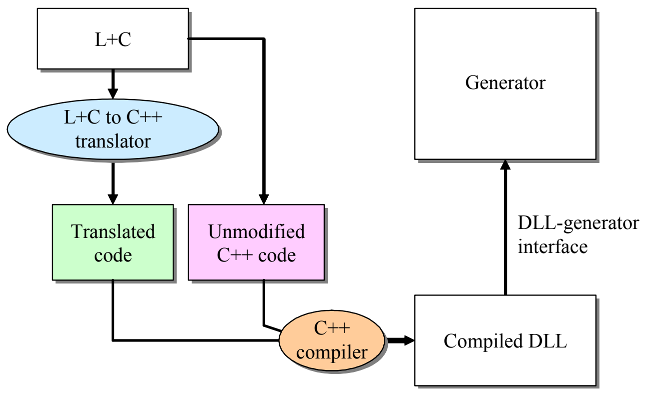

Figure 4. Components of our modeling system

Based on this design, the translator divides the input L+C code into two categories: the constructs specific to L+C are translated into C++ code, while the remaining C++ code is passed verbatim to the compiler. The resulting C++ code is then compiled using a standard C++ compiled as a DLL (dynamic link library). The actual execution of the L+C program is performed by a fixed component of a modeling program, called the generator (Figure 4). For the user's convenience, modified L+C program can be translated, compiled and ran without a need for restarting the modeling program. Consequently, the L-system string derivation is performed based only on the information that can be provided by the DLL at run-time (since the generator is a fixed component and is not recompiled for every L+C program).

The DLL includes the interfacing information that makes the generator and the compiled L+C program communicate. We present this interface from the perspective of string derivation by the generator. The core of the generator is the `Execute()` function:

```
void Execute()
{
   Start();
   Axiom();
   DecomposeString();
   for (int i=0; i<DerivationLength(); ++i)
   {
      StartEach();
      Derive();
      DecomposeString();
      EndEach();
   }
   End();
}
```

where the functions written in boldface are defined in the process of translating the L+C program to C++ as follows:

- `Start()`, `StartEach()`, `EndEach()` and `End()` execute the compound statements specified in the corresponding L+C control statements (Section 2.7);
- `Axiom()` creates the initial L-system string (Section 2.2),
- `DerivationLength()` returns the value specified in the L-system `derivation length` statement (Section 2.3).

The translation of the L+C control statements into C++ functions is straightforward. For example, the L+C `Start` statement is translated as follows:

Original code

```
Start:
{
  …
}
```

Translated code

```
void Start()
{
  …
}
```

Analogous substitutions are made for the other L+C control statements. To process the `derivation length` statement, the translator replaces the L+C keyword with a C++ function prototype:

Original code

```
derivation length: 3;
```

Translated code

```
int DerivationLength() { return 3; }
```

In order to present the translation of productions, let us consider the following L+C production as an example:

```
module A(data, float);
module B(int, float);

A(dl, xl) < B(n, a) :
{
   if (a>xl)
      produce B(n+1, xl);
   else
      produce B(n-1, xl);
}
```

Elements of the production typical for L+C are highlighted in boldface. The process of translation is based on the fact that productions are similar to functions in imperative programming languages. The similarities can be summarized into the following:

- A production is a piece of code to be executed,
- Its input is its predecessor and optionally, parameters of the predecessor's modules, and
- Its output is the successor.

The differences between productions and functions are as follows:

- L-system programs do not call productions explicitly. The general mechanism of matching productions determines which production should be applied and when.
- Productions do not return a value in the traditional sense. Instead, their output modifies the contents of the L-system string.

The first step in translating a production into a C++ function is to declare a function prototype, using the types declared in the relevant modules. For example, the following substitution is made:

Original code: Translated code:

```
A(dl, xl) < B(n, a)      void P1(data dl, float xl, int n, float a)
```

Another element in the production code that needs to be translated is the produce statement. The code resulting from the translation of this statement must add the successor to the new string, and terminate the production. In our example, the produce statement is translated into code similar to this:

Original code: Translated code:

```
produce B(n+1, x);       {App(B_id); App(n+1); App(x); return;}
```

It should be noted that the translation process, as so far described, does not retain all the necessary information. In particular, the modules in the strict predecessor and the context information are not present in the generated code. It is then necessary to add information that bridges the generator and the translated L+C code. However, as this is of a purely technical concern of program implementation, the additional code is not further discussed here.

4. Conclusions

We have described a modeling language L+C, which incorporates C++ into the framework of L-systems. We have also implemented a modeling system that uses L+C programs as input. To implement the L+C translator, we have introduced a methodology based on the separation of the constructs specific to L-systems from the C++ code. This methodology made it possible for a single person to implement the L+C translator in one month. The L-system-specific code is translated into C++ and combined with the C++ code taken verbatim from the L+C programs. The resulting code is translated into a DLL module using a standard C++ compiler. This module is linked with the generator that executes the L-systems. In practice, the DLL module is small in size compared to the generator and the graphical interpreter associated with it. Consequently, the DLL module compiles and links fast (of the order of one second on the current Windows and Linux workstations), which allows for interactive manipulation and modification of the models. The increased expressiveness of L+C, compared to the previous L-system based languages, makes it possible to create models of a relatively greater complexity. L+C is currently being used to model aspects of plant genetics, physiology, and biomechanics.

References

Abelson, H. and diSessa, A. [1982]: *Turtle geometry*. M.I.T. Press, Cambridge.

Aho 1986: Aho, A., Sethi, R. and Ullman, J. [1986], *Compilers: Principles, techniques and tools*. Addison-Wesley, Reading.

Baker R. and Herman G. T. [1970]: Simulation of organisms using a developmental model, parts I and II. *International Journal of Bio-Medical Computing* **3**, pp. 201-215 and 251-267.

Chien, T. and Jurgensen, H. [1992]: Parameterized L systems for modelling: Potential and limitations. In: G. Rozenberg and A. Salomaa (Eds.): *Lindenmayer systems: Im-*

pacts on theoretical computer science, computer graphics, and developmental biology. Springer, Berlin, pp. 213—229.

Erstad, K. [2002]: *L-systems, twining plants, Lisp*. M. Sc. thesis, University of Bergen.

Giavitto, J.-L. and Michel, O. [2001]: *MGS: A programming language for the transformation of topological collections*. Research Report, 61-2001, CNRS – Universite d'Evry Val d'Esonne.

Giavitto, J.-L. and Michel, O. [2002]: Data structures as topological spaces. Proceedings of the 3rd International Conference on Unconventional Models of Computation UMC02, *Lecture Notes in Computer Science* **2509**, pp. 137-150.

Hammel, M. [1996]: *Differential L-systems and their application to the simulation and visualization of plant development*. Ph. D. thesis, University of Calgary.

Hanan, J. [1992]: *Parametric L-systems*. Ph. D. thesis, University of Regina.

Karwowski, R. [2002]: *Improving the process of plant modeling: The L+C modeling language*. Ph. D. thesis, University of Calgary.

Knuth, D. [1968]: Semantics of context-free languages. *Mathematical Systems Theory* **2**, pp. 191-220.

Kurth, W. [1994]: *Growth grammar interpreter (GROGRA 2.4): A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modeling. Introduction and reference manual*. Forschungszentrum Waldokosysteme der Universitat Gottingen.

Lindenmayer, A. [1968]: Mathematical models for cellular interaction in development. *Journal of Theoretical Biology* **18**, pp. 280-315.

Lindenmayer, A. [1971]: Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology* **30**, pp. 455-494

Prusinkiewicz, P. [1986]: Graphical applications of L-systems. Proceedings of Graphics Interface '86 – Vision Interface '86, pp. 247-253.

Prusinkiewicz, P. and Hanan, J. [1989]: *Lindenmayer systems, fractals and plants*. Lecture Notes in Biomathematics **79**, Springer, Berlin.

Prusinkiewicz, P. and Hanan, J. [1990]: Visualization of botanical structures and processes using parametric L-systems. In: D. Thalmann (Ed.), *Scientific visualization and graphics simulation*, J. Wiley & Sons, Chichester, pp. 183-201.

Prusinkiewicz, P. and Hanan, J. [1992]: L-systems: From formalism to programming languag*e*s. In: G. Rozenberg and A. Salomaa (Eds.), *Lindenmayer systems: Impacts on theoretical computer science, computer graphics and developmental biology*. Springer, Berlin, pp. 193-211.

P. Prusinkiewicz, P., Hammel, M. and Mjolsness, E. [1993]: Animation of plant development. Proceedings of SIGGRAPH 93, pp. 351-360.

Prusinkiewicz, P. [1997]: A look at the visual modeling of plants using {{L}-systems. In R. Hofestadt and T. Lengauer and M. Loffler and D. Schomburg (Eds.): Bioinformatics. *Lecture Notes in Computer Science* **1278**, Springer, Berlin, pp.11-29.

Prusinkiewicz, P., Hanan, J., and Mech, R. [2000]: An L-system-based plant modeling language. In M. Nagl, A. Schuerr and M. Muench (Eds.): Applications of graph transformation with industrial relevance. *Lecture Notes in Computer Science* **1779**, Springer, Berlin, pp. 395-410.

Prusinkiewicz, P., Muendermann, L., Karwowski, R. and Lane, B. [2001]: The use of positional information in the modeling of plants. Proceedings of SIGGRAPH 2001, pp. 289-300.

Prusinkiewicz, P., Samavati, F., Smith, C. and Karwowski, R. [2003]: L-system description of subdivision curves. To appear in the *International Journal of Shape Modeling*.

Sievanen R., Perttunen J., Prusinkiewicz P., Karwowski R., Modeling language L, unpublished report.

Stroustroup, B. [1991]: *The C++ Programming Language*, Addison-Wesley, Reading.

Szilard, A. and Quinton, R. [1979]: An interpretation for D0L systems by computer graphics. *The Science Terrapin* **4**, pp. 8-13.

Woodger J. [1937]: *The axiomatic method in biology*, University Press, Cambridge.