Specification of morphological models with L-systems and relational growth grammars

Winfried Kurth

Brandenburg University of Technology at Cottbus, Chair for Practical Informatics / Graphics Systems,
P.O.Box 10 13 44, 03013 Cottbus, Germany, wk@informatik.tu-cottbus.de

Abstract
Among the techniques for the creation of photorealistic virtual organisms, particularly plants, and in scientific models of vegetation structure, rule-based specifications (formal grammars) play a prominent role. Lindenmayer systems (L-systems) are the most widespread formalism of this sort, but certain types of graph grammars, combined with standard object-oriented programming, offer even more possibilities to specify rule-driven developments of 3-dimensional arrangements, morphology of virtual organisms and underlying processes like, e.g., metabolic reactions. Examples of grammar rules and the virtual geometrical structures generated from them, all realizable with the open-source software GroIMP (www.grogra.de), are shown. This grammar-based approach is often not directly used for the specification of a picture as a pattern of graphical elements in a plane, but for virtual 3-dimensional scenes, which are then rendered visible using standard techniques of geometry-based computer graphics.

1. Introduction: Rule-based modelling of development

The programme for a computer-based simulation of a process is often specified by writing down the elementary steps of calculation in a prescribed order, which is to be applied when they are executed by the machine. This order can include the use of conditional branching and loops. Furthermore, in this classical programming style, commands have usually the meaning that the state of the machine – manifested e.g. in the values of some memory cells – is changed in a predefined manner. This programming paradigm is called "imperative" or "von Neumann programming", and can be very useful in technical calculations or for simulations in physics.
However, when living organisms and the development of their morphological structure are to be modelled, another sort of programming seems to be more natural. E.g., let us consider a growing tree: All parts of the organism coexist, and the young shoots of the tree grow all in parallel, often according to the same pattern. An intuitive way to specify this behaviour is to list a number of rules for growth of single buds and shoots (or whatever organs are considered as the basic constituents), and to let the computer apply them in parallel to all tree organs, wherever they are applicable. When the growth flush of the next year is to be simulated, the application of these rules is to be iterated. Here, the order in which the rules of growth are written down is not important: The computer is expected to pick those rules which are applicable in a given situation and to use them, regardless of their position in a list. This "rule-based" programming paradigm is well known in other branches of information science: *Grammars* of natural languages and of programming languages are used in a similar manner, with the aim to deduce all correctly-formed sentences. Another example is the programming language PROLOG, where logical rules are applied to generate automatic proofs of statements. In all these cases, some structure – a botanical tree / a sentence / a logical formula – is transformed or *rewritten* by the application of rules. The systems of rules, or grammars, are therefore also called "rewriting systems". Rule-based programming can be a more intuitive way to specify models of natural phenomena, because we do not need to bother about a specific order of execution of commands. The rules work at a higher level of abstraction.

The biologist Aristid Lindenmayer invented in 1968 a special sort of grammar, later called L-system, to describe the growth of arrangements of plant cells [Lindenmayer 1968]. At that time, the notion of formal grammar, developed by Noam Chomsky for natural languages, was already known. However, in a Chomsky grammar, normally only one rule is applied in each deduction step. In contrast, L-systems work in a parallel manner, thus reflecting the parallelism of growth in plants: That means, in every timestep all constituents of the virtual plant where some rule is applicable are transformed according to that rule. (If there are some objects on which no rule can be applied, it is assumed that these objects are just resting: They remain unchanged.)
Later on, Lindenmayer's formalism, which is basically a string-rewriting mechanism, was extended.[1] A command language for a geometrical interpretation of strings was introduced to give a precise definition of the morphological meaning of the structures obtained from L-system application. We will briefly introduce this "Turtle Geometry" in Chapter 2. In Chapter 3, L-systems will be exactly defined, and we will see some simple examples. Several extensions of the original concept were used to solve various problems in the modelling of plant growth and architecture; some of these extensions will be explained and demonstrated in Chapter 4. An important generalization, which is currently still in the focus of research, is introduced in Chapter 5: "Relational Growth Grammars" (RGG), a variant of graph rewriting systems. These grammars overcome some of the limitations of L-systems and can be used to connect different levels of the organization of plants in a unifying model framework: Genetic processes influencing metabolism, metabolic reaction networks influencing macroscopic growth and morphogenesis. Simulation models based on this sort of grammar representation can not only produce even more realistic images of plants and plant communities, but will also aid the biologists in checking hypotheses and designing new experiments. A discussion of possible future trends in modelling morphological phenomena and of the relation of the rule-based programming paradigm to picture morphology will close the article.

2. Turtle geometry

To establish a connection between the language of *character strings* and the language of *geometrical forms*, a simple alphabet of commands, each with a geometrical meaning, is defined. Using these commands, we build programmes in a strictly imperative manner, which are interpreted by a virtual drawing device, called the "turtle" [Abelson & diSessa 1982]. The turtle is equipped with a simple memory containing information about the length *s* of the next line to be drawn, its thickness *d*, its colour *c*, the turtle's current position on the plane, its current direction of moving, etc. Among the possible commands are:

M0	move forward by length *s* (without drawing)
F0	move forward and draw simultaneously a line of length *s*
M(*a***)**	move forward by length *a* (without drawing); the explicitely specified number *a* overrides the turtle's inherent *s*
F(*a***)**	move forward and draw simultaneously a line of length *a*
L(*a***)**	overwrite *s* by the value *a*
D(*a***)**	overwrite *d* by the value *a*
P(*a***)**	overwrite *c* by the value *a* (interpreted as a colour index)
RU(*a***)**	rotate clockwise by the angle *a* (around the "up" axis, which is perpendicular to the plane where the turtle is moving)
Sphere(*a***)**	produce a filled circle with radius *a* around the current position without moving

[1] see [Prusinkiewicz & Lindenmayer 1990] for references and historical remarks.

The zero in **M0** and **F0** means that there is no explicit argument; instead, the memorized "state variable" *s* of the turtle is used. Strings composed of these commands can be used to specify structures made of consecutive lines with changing length, thickness and visibility. Each such string describes a *static* geometrical structure. E.g., the string

```
L(100) D(3) RU(-90) F(50) RU(90) M0 RU(90) D(10) F0 F0
D(3) RU(90) F0 F0 RU(90) F(150) RU(90) F(140) RU(90)
M(30) F(30) M(30) F(30) RU(120) M0 Sphere(15)
```

describes the structure in Figure 1.

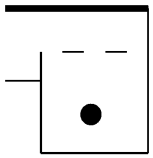


Fig. 1: The result of a simple turtle command sequence (see text).

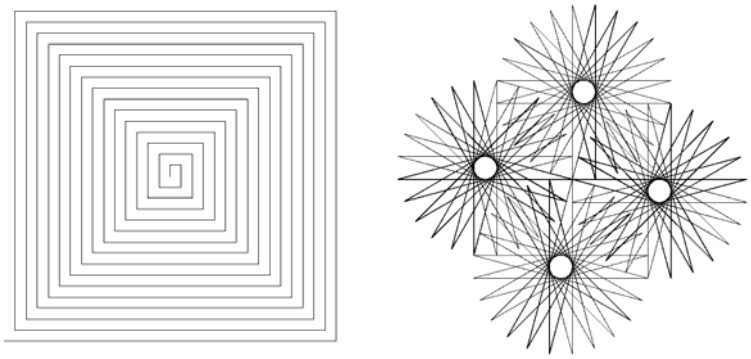
As in other imperative programming languages, *loops* can be used to abbreviate iterated parts of the string: **for (i:(1:***n***)) (** *X* **)** generates *n* replications of the string *X*. Hence, the turtle command programme

```
L(100) for (i:(1:30))
         ( for (j:(1:i)) ( F0 ) RU(90)
           for (j:(1:i)) ( F0 ) RU(90) )
```

generates the spiral in Figure 2a, and

```
L(100) for ((1:20))
         ( for ((1:36)) ( F0 RU(165) F0 RU(165) ) RU(270) )
```

generates the pattern in Figure 2b.



(a) (b)

Fig. 2: (a) A spiral specified by a simple iterative turtle programme,
(b) the result of another iterative turtle programme (after [Goel & Rozehnal 1991]).

To overcome the restriction to strictly linear forms, the possibility of branching is introduced by the special turtle commands "**[**" and "**]**": When the turtle encounters "**[**", its current state (including the values of *s*, *d*, *c* etc.) is stored on a stack. The following string can be seen as a branch which ends when "**]**" is encountered: Then the stored state is taken from the stack and replaces the turtle state which was obtained during the drawing of the branch. This means that

the turtle "jumps back" to its old position and resumes its operation as if the construction of the branch since "**[**" would not have taken place. Figure 3 shows the turtle interpretation of the string

F(50) [RU(60) P(4) F(20)] RU(-30) F(50) :

After the vertical segment of length 50, the smaller, red branch to the right (coloured according to the command **P(4)**) is constructed. After the closed bracket, the turtle resumes its old position and follows the commands **RU(-30) F(50)** to draw the upper-left part of the structure.

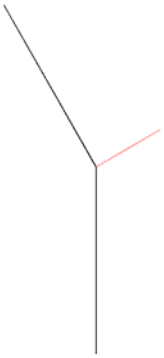


Fig. 3: A branched structure (see text).

The turtle can also be guided to draw structures in three dimensions. For this purpose, two further rotation commands are introduced: **RL(***a***)** and **RH(***a***)**, which rotate the turtle around an axis pointing (initially) to the left, resp. around its current head direction. (See the tutorial included in the GroIMP software, freely available under **www.grogra.de**, for further details about turtle commands.)

3. L-systems

Lindenmayer systems (L-systems) are *parallel rewriting systems on strings*. Mathematically, a "pure" L-system (without geometrical interpretation) consists of 3 components: an alphabet Σ which contains the basic symbols which are to be used to build strings, a start string called "Axiom", and a finite set of rules, each of which having the form

symbol **==>** *string of symbols*;

and the symbols are taken from Σ here. In a deterministic L-system, the left-hand side (l.h.s.) of each rule must be different from that of all other rules. An *application step* of the L-system to a given string *s* consists of the simultaneous replacement of all symbols in *s* occurring as a l.h.s. of a rule by their corresponding right-hand side (r.h.s.), whereas symbols which cannot be replaced with the help of a rule remain unchanged. By starting with the start string of the L-system and iteratively performing one application step to the result of the preceding one, we obtain the *developmental sequence* of strings generated by an L-system:

$$Axiom \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$$

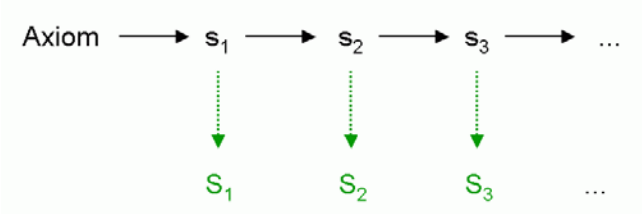
For example, let us consider the L-system with the alphabet Σ = { **A**; **B** }, *Axiom* = **A**, and with the two rules

A ==> B
B ==> AB.

The resulting developmental sequence is

A → **B** → **AB** → **BAB** → **ABBAB** → **BABABBAB** → ...

Following Lindenmayer's original intentions, **A** and **B** can be interpreted as two different cell types of filamentous organisms (e.g., algae). The rules say that a cell of type **A** can grow into a cell of type **B**, and a type **B** cell can divide into two cells of type **A** and **B**, respectively. The developmental sequence then reflects the growth of the filament of cells in discrete time steps. (Note that the number of cells generated in this sequence grows according to the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ..., where each number is the sum of its two predecessors.)
To produce more interesting structures from L-systems than just linear filaments of cells, Alvy Ray Smith [Smith 1984] and later Prusinkiewicz and Lindenmayer [1990] added turtle geometry as a fourth component to Σ, *Axiom* and the rule set. Turtle geometry serves as a geometrical interpretation, i.e. as a means to associate with each string (particularly with each s_i from the developmental sequence above) a geometrical structure S_i in 2- or 3-dimensional space. This is accomplished by letting the alphabet Σ contain the set T of all turtle commands. The strings s_i obtained from the L-system are then separately interpreted by the turtle, i.e. they are scanned from left to right, and the geometrical structure S_i is constructed by following the occurring commands. Symbols from Σ which are not in T are simply ignored by the turtle. Hence we have the following scheme of interpreted L-system application:



Here, the dotted green arrows stand for the turtle interpretation process.
The first example (after [Prusinkiewicz & Hanan 1989, p. 25]) will demonstrate this mechanism: Let the rules of our L-system be

```
Axiom ==> L(100) F0   and
F0    ==> F0 [ RU(25.7) F0 ] F0 [ RU(-25.7) F0 ] F0 .
```

Figure 4 shows the resulting structures S_1, S_2, S_3 and S_4.

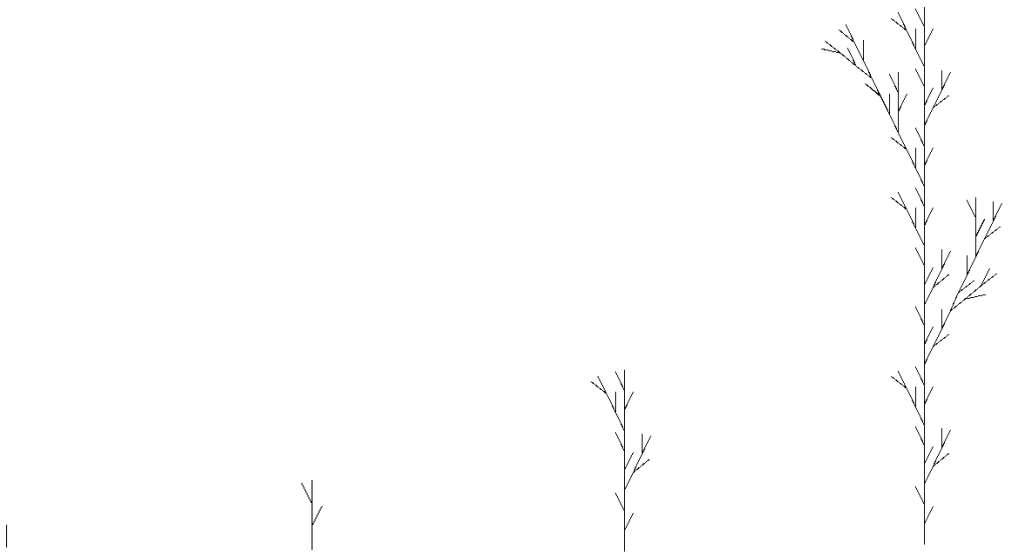


Fig. 4: A developmental sequence of branching structures in the plane, generated by a simple L-system (see text).

The next two examples use L-systems to generate plane-filling curves. Both make use of the possibility, given in the programming language XL [Kniemeyer 2007], to let symbols (in this context called "modules") inherit properties from other symbols. Such an inheritance from **A** to **B** is expressed in the form

```
module B extends A;
```

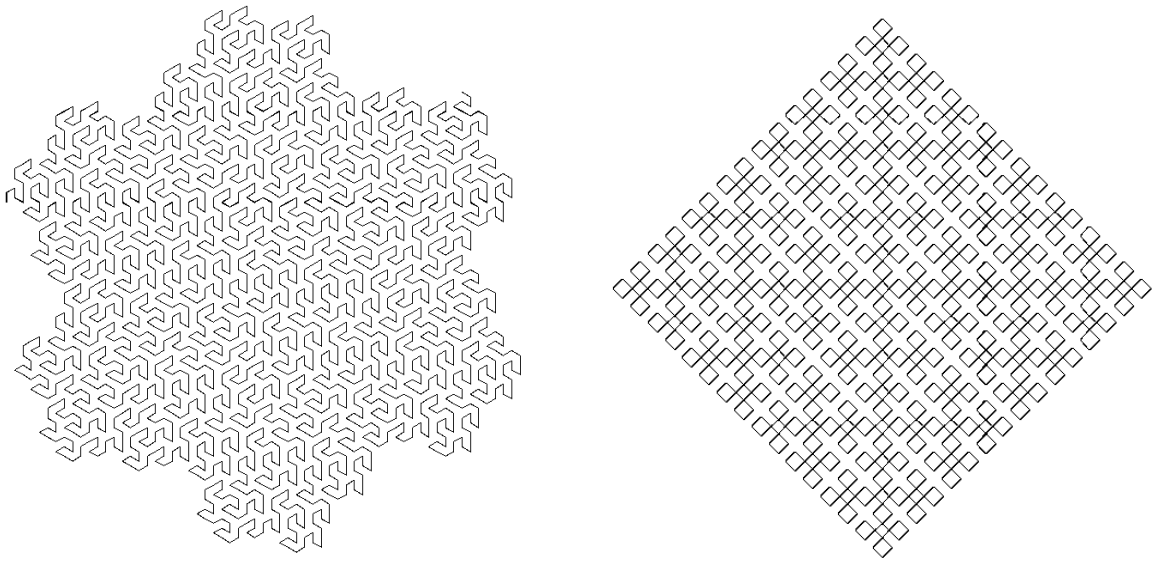
and this is a formalism typical for object-oriented programming. Its purpose in the following examples is simply the abbreviation of commands.
A so-called *hexagonal Gosper curve* is derived from

```
module A extends F0;
module B extends F0;
module C extends RU(60);
module D extends RU(-60);
Axiom ==> L(100) A;
A ==> A C B C C B D A D D A A D B C;
B ==> D A C B B C C B C A D D A D B;
```

with the result after 4 steps shown in Figure 5a (after [Prusinkiewicz & Hanan 1989, p. 19]), and the second curve resembles a traditional Indian *kolam* pattern (see [Ascher 2003]), called "Anklets of Krishna" (after [Prusinkiewicz & Hanan 1989, p. 73]), and is derived from

```
module R extends RU(-45);
module A extends F(10);
Axiom ==> L(100) R X R A R X;
X ==> X F0 X R A R X F0 X;
```

see Figure 5b.



(a) (b)

Fig. 5: Two plane-filling curves obtained from L-systems, see text.

4. Extensions of the L-system concept

4.1 Stochastic L-systems

Geometrical structures produced by the simple forms of L-systems which we have presented so far show a high degree of regularity. In real-world patterns, however, we have often a

variability and "noise", producing deviations from strict regularity. A first attempt to reflect this "noise" in a model is the inclusion of randomness. The computer can generate pseudo-random numbers, appearing as if they do not follow any predictable pattern, and this form of irregularity can be introduced in rewriting systems – either by directly using pseudo-random numbers as parameters (e.g., of **L** or **RU** commands) or by making rule application depend on some "oracle" driven by pseudo-random numbers. For example, let us consider the deterministic L-system

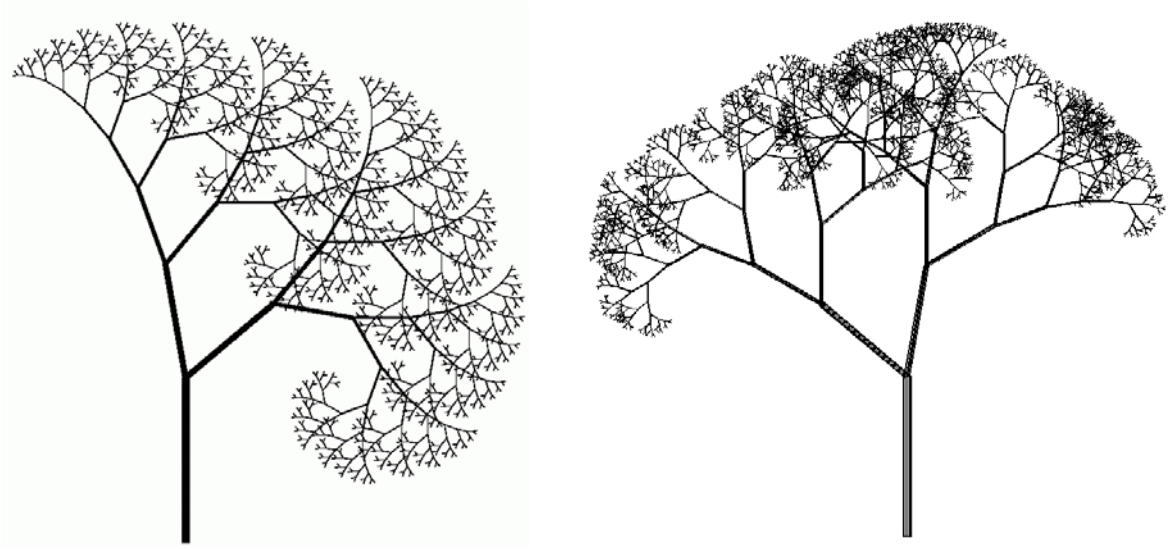
```
float c = 0.7;
Axiom ==> L(100) D(5) A;
A ==> F0 LMul(c) DMul(c) [ RU(50) A ] [ RU(-10) A ].
```

(Here, "**float**" declares a floating-point variable **c** which gets the value 0.7 and is used in the second rule; "**LMul(c)**" multiplies the current length *s* of the turtle steps with this number, and "**DMul(c)**" analogously for current thickness *d*.) The tree-like structure produced by this L-system looks very regular (Fig. 6a).

If we exchange the second rule by

```
A ==> F0 LMul(c) DMul(c)
      if (probabiliy(0.5)) ( [ RU(50) A ] [ RU(-10) A ] )
      else ( [ RU(-50) A ] [ RU(10) A ] );
```

the orientation of the two branches, specified by the "**RU**" commands, is switched (or not) in an arbitrary manner in each new bifurcation of the tree. Each of the two orientations is chosen with equal probability 0.5, as if the outcome would depend on coin-tossing, and the resulting structure has already a somewhat more natural look (Fig. 6b).



(a) (b)

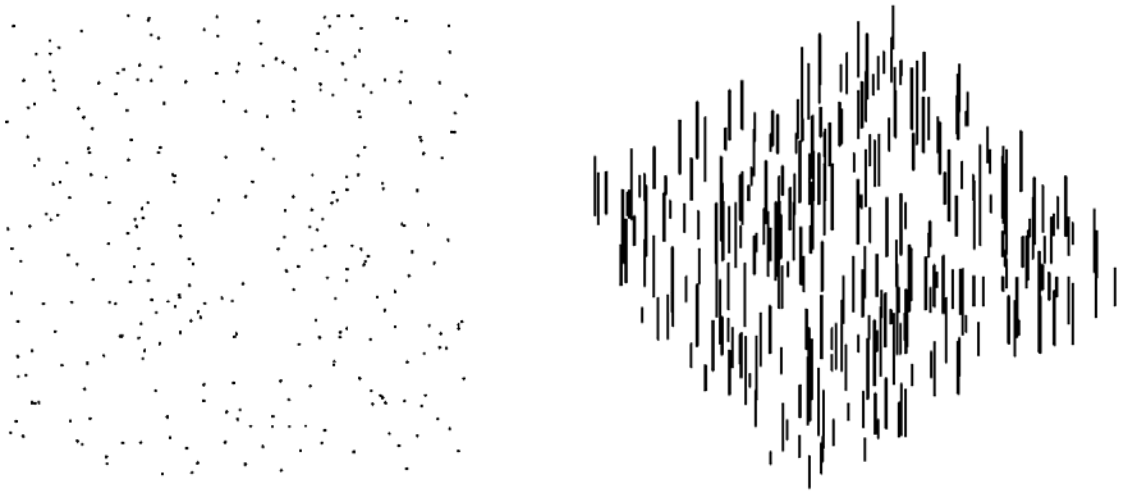
Fig. 6: Tree-like structures generated from an L-system.
(a) Deterministic, (b) stochastic version.

Of course, it would be possible to increase the irregularity even further, e.g. by replacing the constant c above by "**random(0.3, 1)**", a function call which gives back pseudo-random numbers with uniform distribution between 0.3 and 1. Using the same formalism, it is also easily possible to simulate random walks in the plane or in space (e.g., Brownian motion in

physics), or to generate more-or-less-controlled random distributions of small objects in an area – what is called a "point process" in geostatistics.
A very simple example is given by the following L-system, consisting of only one rule:

```
Axiom ==> D(0.5) for ((1:300))
          ( [ Translate(random(0, 100), random(0, 100), 0)
              F(random(5, 30)) ] );
```

which generates 300 vertical lines with random lengths between 5 and 30 units at random positions on a 100 × 100 square field (Fig. 7). Here, the command "**Translate**" works like "**M**", but the direction of the translation is given in absolute coordinates (*x*, *y*, *z*), not as a multiple of the current turtle head vector.



(a) (b)

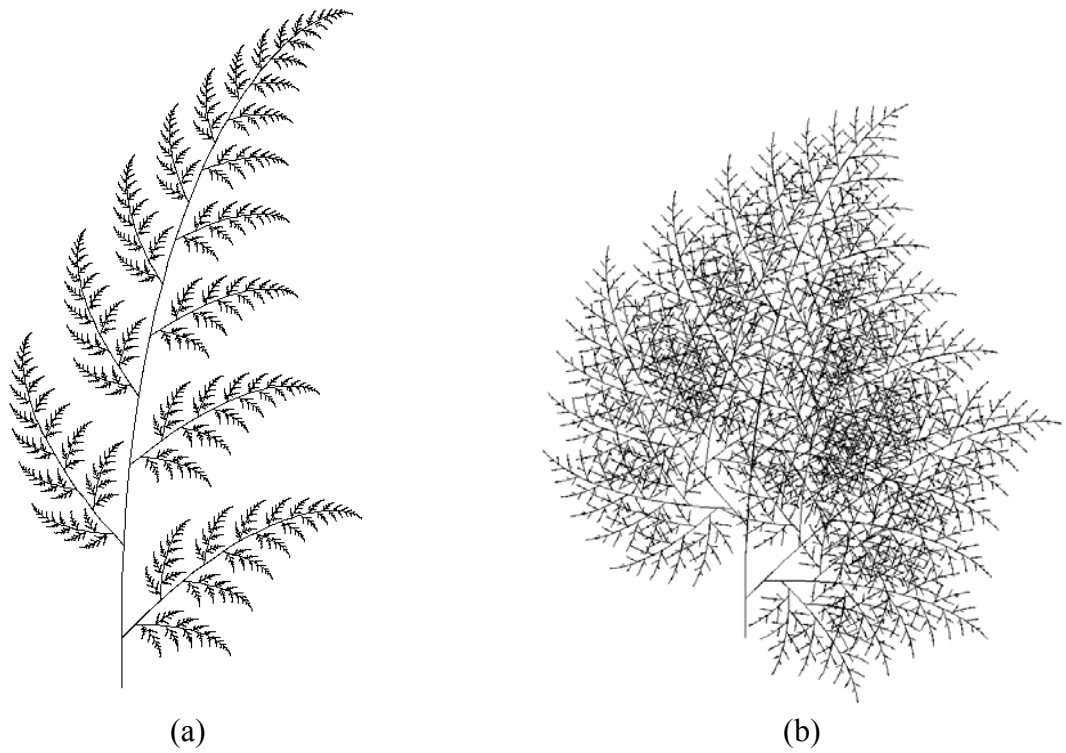
Fig. 7: A random pattern of vertical lines on a quadratic area. (a) View from above, (b) slanted view.

4.2 Parametric L-systems

We have already used parameters with numerical values in turtle commands like **L**, **LMul**, **D** or **F** in the examples above. If we permit the use of such parameters in connection with other symbols, too, the capacity of our rewriting mechanism to perform calculations of all kinds is greatly enhanced. For example, in the next L-system, which produces a fractal structure resembling a fern leaf (Fig. 8a), we use two integer parameters *t* and *k* for the symbol **A**. The symbol **A** stands for something like a bud here, and the first parameter, *t*, is a time delay: *t* is counted down, and a certain number of steps (here 6) must pass before a lateral branch starts growing. The second parameter, *k*, has only the values +1 or –1 and controls the orientation of the branch, similar to the tree example above, but not changing at random: *k* is systematically alternating between –1 and +1.

```
module A(int t, int k);
Axiom ==> L(100) A(0, 1);
A(t, k) ==>
         if (t > 0) ( A(t-1, k) )
         else
         ( F(1) [ RU(k*45) A(6, k) ] F(1) RU(3) A(0, -k) );
F(x) ==> F(1.15*x)
```

L-systems like this one naturally challenge the plant designer to explore their potential by playing around with parameters: E.g., if one reduces the initial delay in the branches from 6 to 2, branches will emerge earlier and a more compact form of the structure will result (Fig. 8b).

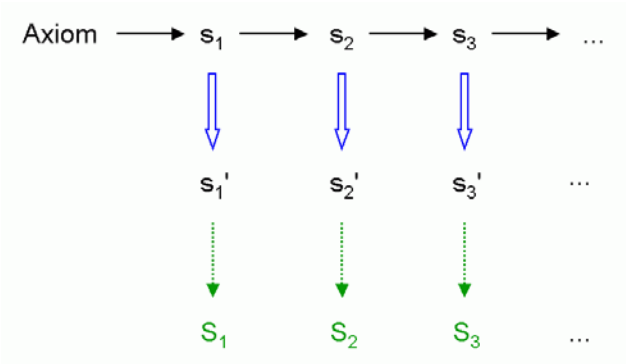


(a) (b)

Fig. 8: (a) Fern leaf produced by a parametric L-system (see text),
(b) variant with reduced delay parameter for branch emergence.

4.3 Interpretive rules

A very useful extension of the L-system formalism is an extra type of rules which are applied in a different manner: Whereas the "normal" L-system rules (also called *generative rules*) are iteratively applied to a string in order to obtain descriptions of new developmental stages, the so-called *interpretive rules* are applied only as a preprocessing for geometrical interpretation, and their application has no influence on the formation of the next developmental step:



In this diagram, the blue hollow vertical arrows represent the application of the interpretive rules, and dotted vertical arrows stand for the subsequent interpretation by the turtle. Particularly, the specification of graphical details of certain objects or organs, which are represented in the strings s_1, s_2, ... as a single symbol, can be given by an interpretive rule with this symbol as its left-hand side. (In the literature, interpretive rules were sometimes also

called "homomorphisms", but this is a misleading naming, because the usual, generative rules can mathematically also be seen as homomorphisms of a so-called free monoid; see, e.g., [Vitányi 1976].) For example, in the following L-system the symbol **A** is copied 8-fold and shifted in the plane by a generative rule which is iteratively applied, whereas the interpretive rule transforms this **A** into a quadratic box. Both types of rules have to be separated in different "blocks" named **run** and **interpret**, and a command "**applyInterpretation**" has to be given in order to apply the interpretive rules in the right moment:

```
public void run()
{
   [
   Axiom ==> A;
   A ==> Scale(0.3333) for (i:(-1:1))
                         for (j:(-1:1))
                           if ((i+1)*(j+1) != 1)
                             ( [ Translate(i, j, 0) A ] );
   ]
   applyInterpretation();
}
public void interpret()
   [
   A ==> Box;
   ]
```

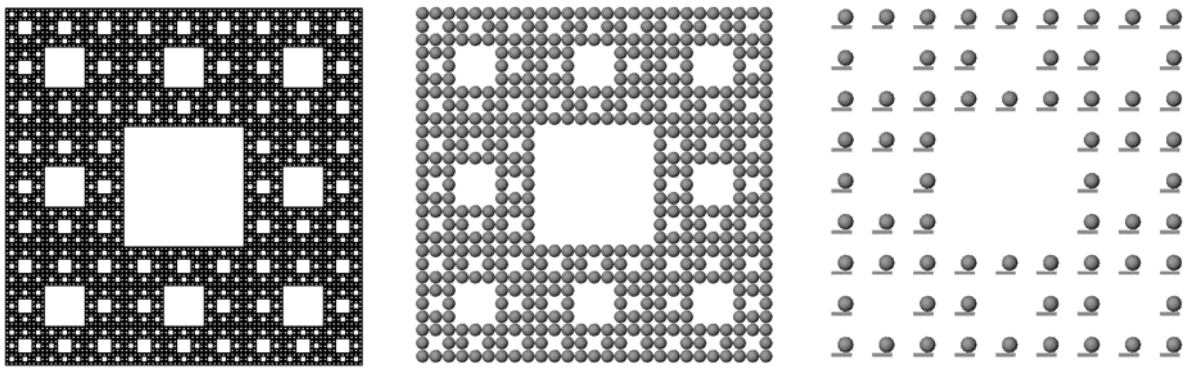
The resulting pattern after 5 steps, approximating a so-called Menger sponge fractal, is shown in Figure 9a. The "Scale" command enforces a shrinking in every developmental step, to compensate for the 3-fold length of the result of copying.
If we now replace the interpretive rule by

```
A ==> Sphere(0.5);
```

we get after 4 steps the result depicted in Fig. 9b. With the number of steps approaching infinity, the limit set will be the same fractal as in the first version. The same holds for the variant with

```
A ==> Box(0.1, 0.5, 0.1) Translate(0.1, 0.25, 0) Sphere(0.2);
```

which defines an arrangement of a flat box and a smaller sphere as initial configuration; the result after 3 steps is shown in Fig. 9c.



(a) (b) (c)

Fig. 9: Different approximations of the Menger sponge fractal,
obtained with different interpretive rules for the symbol **A** (see text).

The right-hand side of an interpretive rule must not necessarily contain a command generating a geometrical body, like **Box**, **Sphere** or **F** (the latter making a cylinder). The following example system, with an **RU** command on the r.h.s. of an interpretive rule, simulates a clock, with the correct ratio of revolvements of little and big hand (the hands modelled by **F** commands):

```
public void run()
{
   [
   Axiom ==> [ A(0, 0.5) D(0.7) F(60) ] A(0, 6) F(100);
   A(t, speed) ==> A(t+1, speed);
   ]
   applyInterpretation();
}
public void interpret()
   [
   A(t, speed) ==> RU(speed*t);
   ]
```

Interpretive rules considerably enhance the expressive possibilities of graphically-interpreted L-systems.

Using L-systems with the extensions introduced so far, it is already possible to create quite realistic-looking pictures of plants or twigs (Figs. 10, 11). Both models shown here are based on botanical observations and measurements and use only **F** commands for their geometrical elements, which are in fact arranged in a virtual 3-D space (shown is only a parallel projection to a plane).

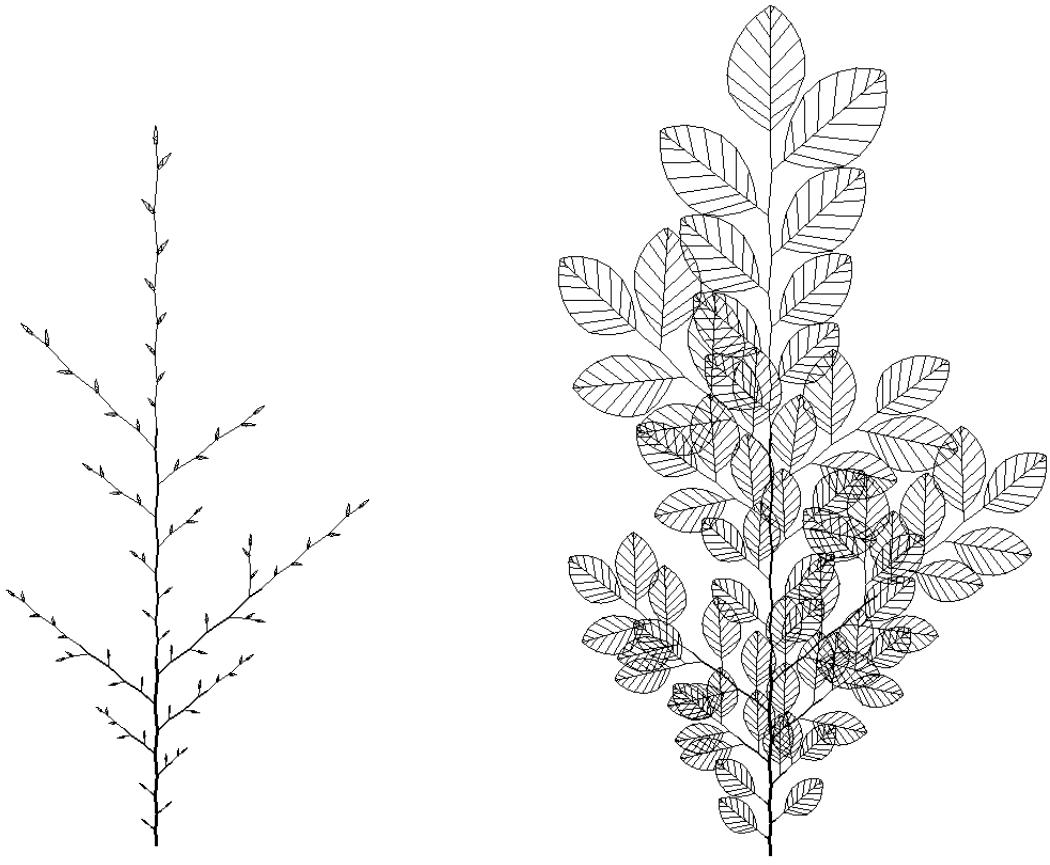


Fig. 10: Model of a beech twig (left: in winter, right: in summer with the buds grown to leaves) based on an L-system; from [Kurth 1999].

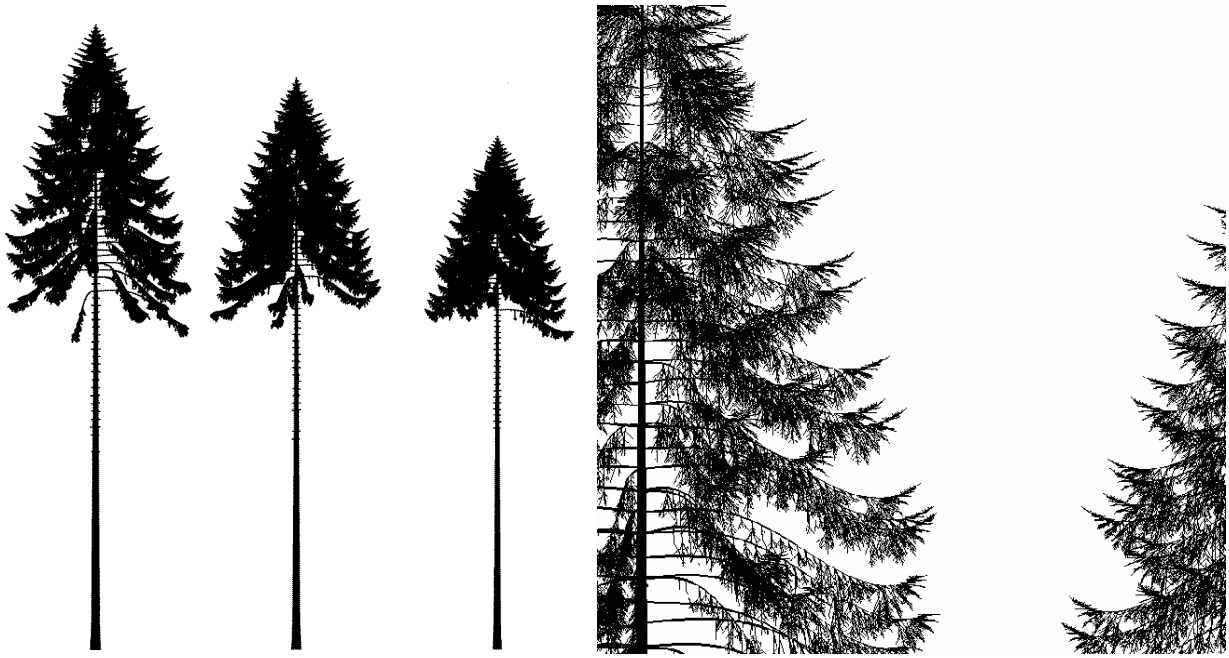


Fig. 11: L-system-based model of spruce (*Picea abies*) trees from the Solling mountains. Left picture: 3 representatives of tree classes (dominant, median, suppressed), right: zoom into two of the trees; from [Kurth 1999].

Although the trees from Figure 11 lack any surface details, colours or lighting and consist only of cylindrical elements, their patterns of branching are quite faithful to nature and allow their usage in simulation models of physical processes, e.g., water transport or distribution of sunlight in the canopy. Exemplarily, Figure 12 shows the resulting water potential profiles along selected branches in the crown of the virtual spruce tree shown on the left, when a flow simulation model based on differential equations is applied on the tree axes with their capacities and resistances [Früh & Kurth 1999].

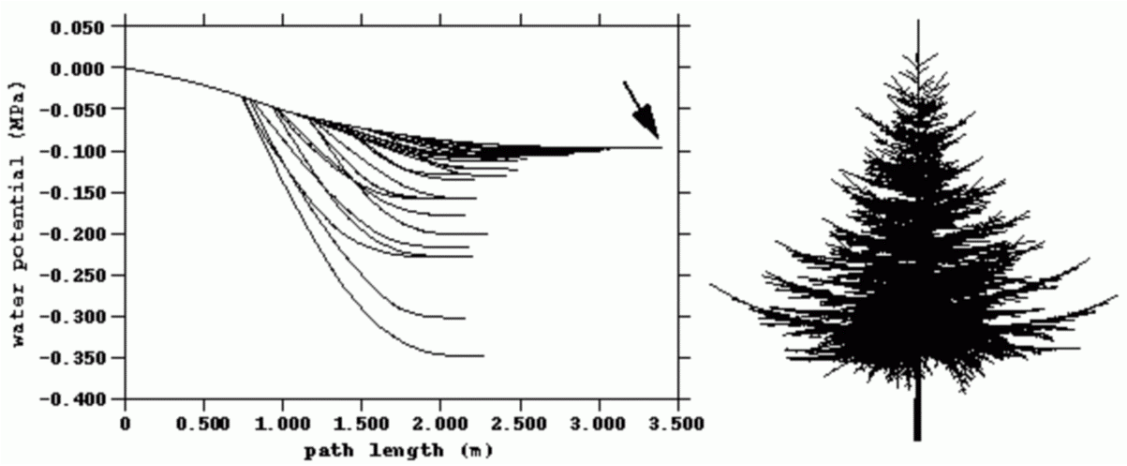


Fig. 12: Virtual water potential profiles (left side) along selected branches of the virtual spruce tree (right side), obtained with the software tools Grogra and Hydra (from [Früh & Kurth 1999]). Each line in the diagramme corresponds to a path from the tree base to a selected branch tip.

On the other hand, when the geometrical elements of the virtual plants are rendered using standard computer-graphics techniques, the trees can be copied and arranged in visualized virtual landscapes like in Figure 13. Here, an interface programme taking terrain data from a GIS (Geographical Information System) and an additional algorithm for the creation of realistic planting patterns of trees was used; see [Knauft 2000].

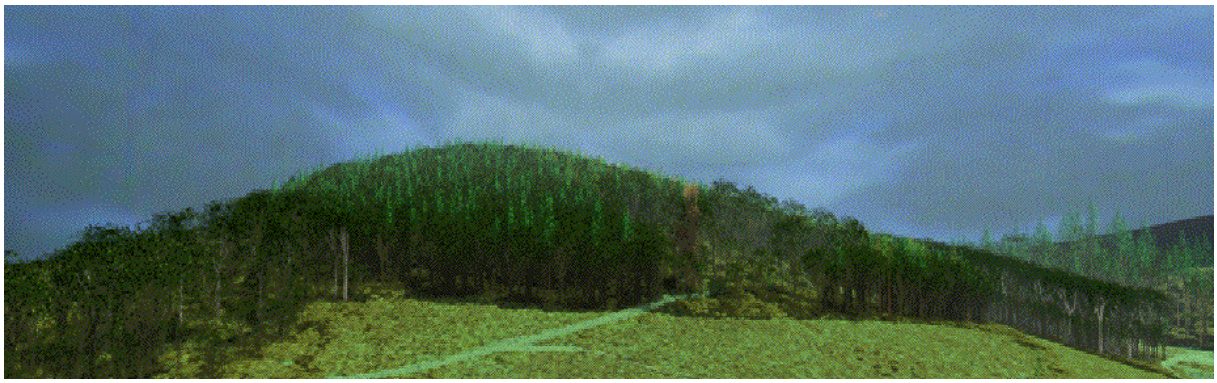


Fig. 13: Virtual Solling landscape, using rendered trees from L-systems and terrain data from a Geographical Information System. From [Knauft 2000].

4.4 Context-sensitivity

All the L-systems shown above allow only a flow of information from the predecessor (in a rule) of a symbol to the symbol itself ("lineage control"). However, in nature we have often the situation that growth or development of an organ is influenced by some information (signals, energy flow, substances) coming from other parts of the existing structure. If we assume that this information comes from the neighbourhood (in a topological sense) of the organ under consideration, it is possible to model such influences by *context-sensitive L-systems*: Applicability of a rule is restricted to the cases when a certain predefined context surrounds the symbol given on the left-hand side of the rule. This context is again specified by symbols, which must be present to the left or to the right of the given symbol in the string representation of the generated structure. (To be precise, we allow several neighbours to the right in the case of branching: The basic element of each branch emerging in *x* is considered as a neighbour of *x*. Furthermore, we permit the skipping of pairs of brackets [...] during checking the context conditions.) Using this formalism, the transport of a signal or of a substance through a growing or static structure can be simulated. Let us consider the following L-system:

```
1   module A(int age);
2   module B(super.length, super.color) extends F(length, 3, color);
3   Axiom ==> A(0);
4   A(t), (t < 5) ==> B(10, 2) A(t+1);
5   A(t), (t == 5) ==> B(10, 4);
6   B(s, 2) (* B(r, 4) *) ==> B(s, 4);
7   B(s, 4) ==> B(s, 3) [ RH(random(0, 360)) RU(30) F(30, 1, 14) ]
```

In line 2, **B** is defined to symbolise a cylinder of diameter 3 and of arbitrary length and colour. Symbol **A** has the meaning of a bud, which produces cylindric stem segments **B(10, 2)** of length 10 and colour 2 (green) while ageing (**A(t)** becomes **A(t+1)**) in line 4. When it reaches age 5, it is transformed in a red segment (**B(10, 4)**) and stops growing (there is no **A** on the right-hand side of the rule in line 5). The rule in line 6 is the context-sensitive one: It waits for a red segment (context **B(r, 4)**, enclosed in **(*** ... ***)**) to occur to the right (geometrically: above) a green segment. If this happens, the green segment is itself replaced by a red one (**B(s, 4)** on the right-hand side). The last rule tells us that a red segment is in the next step always transformed into a blue segment (**B(s, 3)**) with a long, thin yellow branch (**F(30, 1, 14)**) in random direction. The development of this simple structure in 12 steps,

with the red cylinders indicating the downward movement of the branch-inducing signal within the virtual plant, is traced in Figure 14.

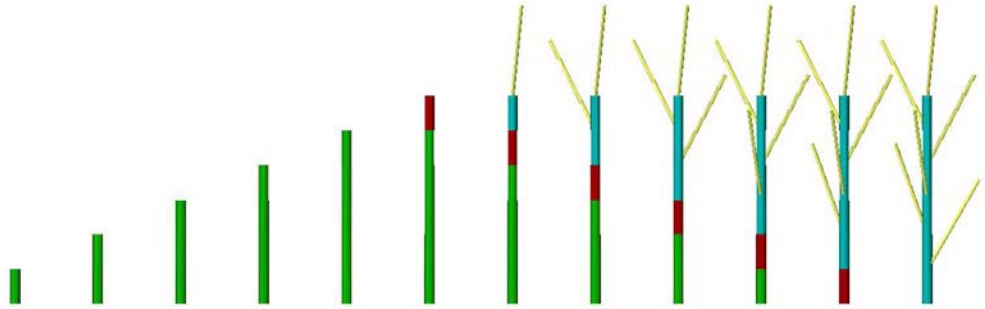


Fig. 14: Signal propagation modelled by a context-sensitive L-system (see text).

Our example was very simplistic, but the same formalism can be used to simulate realistic hormonal signals and induction of flowering in rendered virtual plants (Fig. 15).

Fig. 15: Simulation of flower development of the plant *Mycelis muralis*, obtained from a context-sensitive L-system. From [Prusinkiewicz & Lindenmayer 1990, p. 91].

4.5 Global sensitivity

Interaction in the real world does not only take place between objects which are immediate neighbours. E.g., in a tree, information can pass from a stem segment to a neighbouring

segment in the form of hormones or other substances (Fig. 16a), but also from segments which are far away, by shadowing (Fig. 16b).

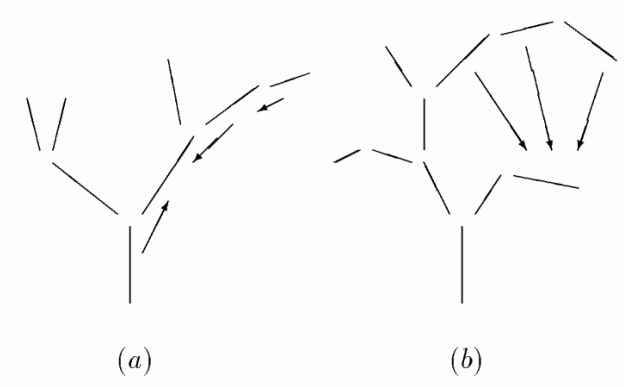


Fig. 16: Local (a) and global (b) interactions in a geometrical structure representing some organism. Far-reaching effects (b), like shadowing, cannot be modelled by context-sensitive L-systems.

Context-sensitive L-systems consider only a context in the sense of the string representation of the generated geometrical structure. This is not enough for modelling the behaviour of "globally sensitive" organs which, e.g., react to shadow and can be influenced by parts of the structure which are in a far distance. For this reason, Prusinkiewicz et al. [1994] introduced "environmentally-sensitive L-systems", which were later generalised by Měch & Prusinkiewicz [1996] under the name "open L-systems". Independently, Kurth [1994] introduced "sensitive growth grammars" (which are not identical with the "relational growth grammars" described below in this paper). Common to all these approaches is the possibility of communication between distant entities or "modules" by the use of special "communication modules" or "sensitive functions". Specific for the approach followed by Prusinkiewicz et al. is a strict conceptual separation between the simulated part (represented by strings) and its "environment" (with the created geometrical structure as a part thereof) is maintained. Both parts are differently modelled, and information exchange between the two simultaneously running simulations is mediated by special interfaces, the above-mentioned communication modules (Fig. 17).

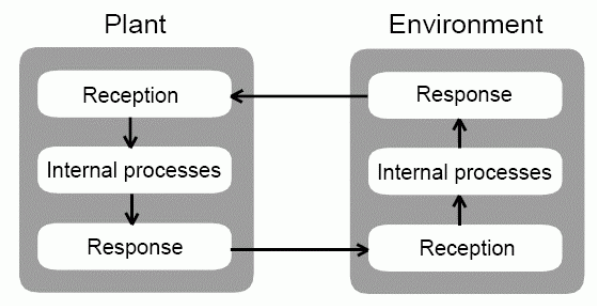


Fig. 17: Division between models of an organism and of its environment according to Měch & Prusinkiewicz [1996].

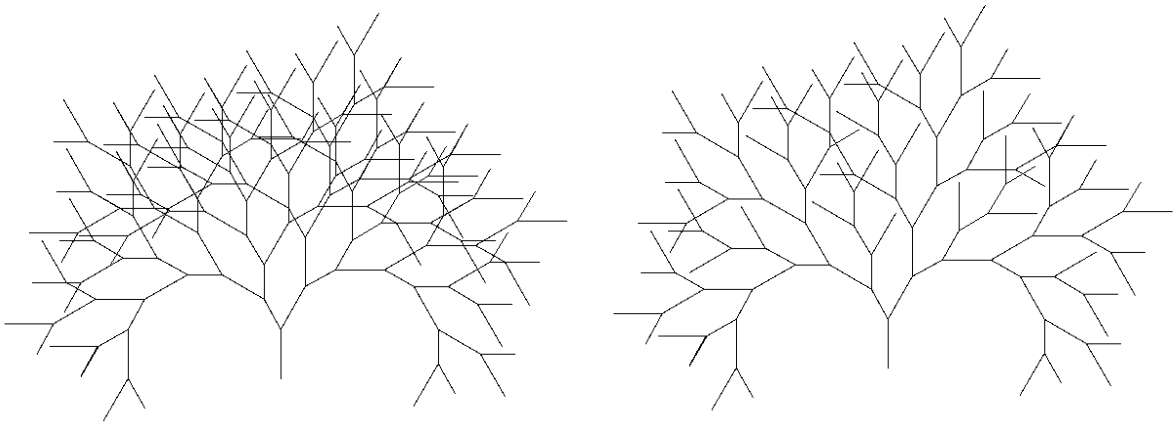
In contrast, we try in our approach to simulate organisms and their environment in a uniform manner and using the same language XL. We feel that the border between organism and environment is in many cases somehow artificial. E.g., the shadowing parts in Fig. 16 are at the same time parts of the virtual plant and of its virtual environment.
An example of a globally-sensitive L-system realised in our language XL is given below. It simulates "density-sensitive" buds which produce new shoots only if there is no other object

closer than 60 length units. (Notice that the "context condition" in purely geometrically defined and does not require that the potential obstacles are topological neighbours of the bud, i.e. that they are directly connected with it.) To make the structure not too symmetrical, two different shoot types **F(100)** and **F(70)**, the latter being shorter, are used. The bud is named **A** and carries the information about the length of the shoot which it will produce in the next step as its parameter:

```
module A(int s);
Axiom ==> F(100) [ RU(-30) A(70) ] RU(30) A(100);
a:A(s) ==> if ( forall(distance(a, (* F *)) > 60) )
           ( RH(180) F(s) [ RU(-30) A(70) ] RU(30) A(100) )
```

The first rule creates initially a long shoot with two buds, **A(70)** and **A(100)**, at its tip. In the second rule, the bud **A(s)** on the left-hand side is *labelled* by a name, **a**, to enable referencing on the right-hand side to this particular bud. In the "if"-condition on the right-hand side, we find a *query function*, "forall", which looks for all objects of type "**F**" (specified by "**(* F *)**") and checks their (euclidean) distance to bud **a**. Only if all these distances exceed 60 length units, the rule is applied and the bud is replaced by a new shoot (**F(s)**) with two buds at its end (last line). The search is done exhaustively in the whole created structure here.

If we omit the "if"-condition, the result of this L-system is just a binary tree with exponential growth, as shown in Figure 18a. With sensitivity in action, not all buds continue growing, and the resulting structure contains fewer branches and fewer crossings between them (Fig. 18b). Notice that not all crossings of branches are eliminated: The reason is that the emptiness of the geometrical neighbourhood of a bud is checked *before* all the new branches have grown. It can happen that closeness or even crossing occurs through simultaneous growth of two shoots whose buds were not close enough before, with the consequence that they did not stop to grow.



(a) (b)

Fig. 18: Simple tree with dichotomous branching after 7 developmental steps, generated by a grammar (a) without and (b) with a condition which incorporates global sensitivity (here: suppression of growth by close other objects); see text. (Adapted from [Kurth 1994].)

With similarly simple grammars, competition between several virtual plants for space and light can be simulated (Fig. 19; code not shown).

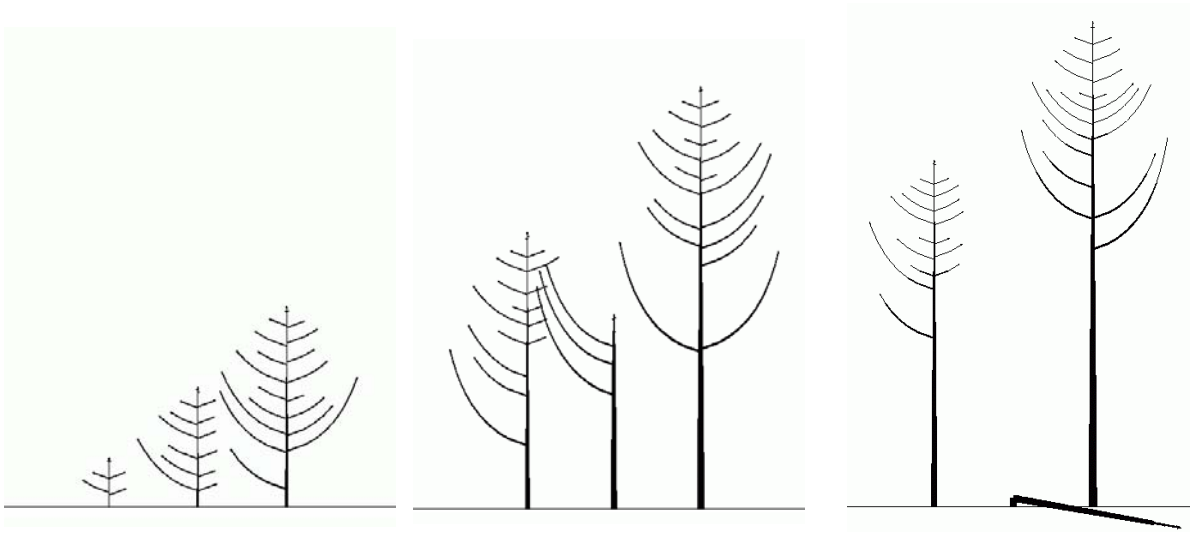
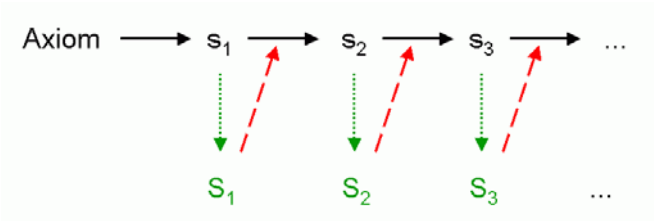


Fig. 19: Growth of three virtual trees competing for light,
modelled using a globally-sensitive grammar; from [Kurth 1999].

To condese the effect of global sensitivity again in a diagram, we find that the currently produced structure S_i can exert influence on the application of generative rules which rewrite the string s_i to s_{i+1} (red, broken arrows):



Of course, it is possible to combine this information flow with the use of interpretive rules (see above).

5. Relational growth grammars

L-systems have been very fruitful for theoretical investigations in formal language theory and for creating realistic-looking models of plants. However, even if all the above-presented extensions are included, they have some limitations:

- In interpreted L-systems (with turtle geometry and with brackets for branching), only two possible relations can be created between the simulated objects: **A** can be a *direct successor* of **B** or can be supported by **B** as a *branch*. In reality, many more sorts of relations between objects are possible and can be worth modelling.

- L-systems are not really an appropriate tool for the creation of truly 2-dimensional or even 3-dimensional arrangements, like tesselations in the plane or cellwork systems (e.g., in tissues). In fact, there exist formalisms like "map L-systems" and "cellwork L-systems" (see [Prusinkiewicz & Lindenmayer 1990]), but their definitions and usage are rather complicated. The reason is that the classical interpretation of bracketed strings by the turtle can only yield locally one-dimensional topologies which are homeomorphic to trees. Particularly, *cycles* and *networks* can be created only if additional tools or tricks are allowed.

- *Multiscaled modelling*, i.e. the simultaneous specification of some structure at several different levels of resolution, is not supported.
- For the biologists, it is a drawback that *genotype* and *phenotype* of an organism cannot be modelled in the same formal framework (although the DNA molecule has basically string structure).
- From the perspective of software development, L-systems as a programming language are a poor language; particularly, the *object-oriented* programming (OOP) style, which is today very commonly used by programmers, is not supported: The fundamental units of the formalism are only symbols (perhaps with some added numerical parameters), no objects in the sense of OOP. Particularly, no hierarchy of object classes, where specialised classes inherit properties from more general classes, can be defined in the classical L-system formalism.

These were reasons enough to design a new formalism, "relational growth grammars" (RGG), and a corresponding programming language, XL (eXtended L-systems language). An RGG is a rewriting system operating on *graphs* instead of strings – here, a graph is a structure consisting of nodes and arcs (also called "edges") connecting some of these nodes, and it can have cyclic substructures. We speak of "relational" grammars because we permit several types of edges (relations). This extension of the L-system concept addresses the first 4 points above [Kniemeyer et al. 2004]. The fifth point is addressed by permitting RGG rules as constructions in a programming language (XL), which is at the same time a true extension of the object-oriented language Java, and by permitting Java objects as nodes of the graphs which are rewritten. (A similar approach led to the language "L+C" [Karwowski & Prusinkiewicz 2003], which is an extension of C++ by L-system rules, but this language does not include graph transformations.) An exact mathematical definition of RGG and a precise language specification for XL will be given by Kniemeyer [2007].
The graphs which are rewritten by an XL programme can also be seen as generalisations of *scene graphs*, as they are known from 3-D modelling languages and tools like VRML, Java 3D or Maya. Particularly, their nodes can stand for geometrical objects and also for transformations of objects (like translation, rotation, scaling...). Indeed, we have already used this feature in our Menger sponge example above (see Fig. 9).
The general structure of an RGG rule is shown in Figure 20. An RGG is composed of such rules, which are usually applied to a given graph in parallel, like L-system rules.

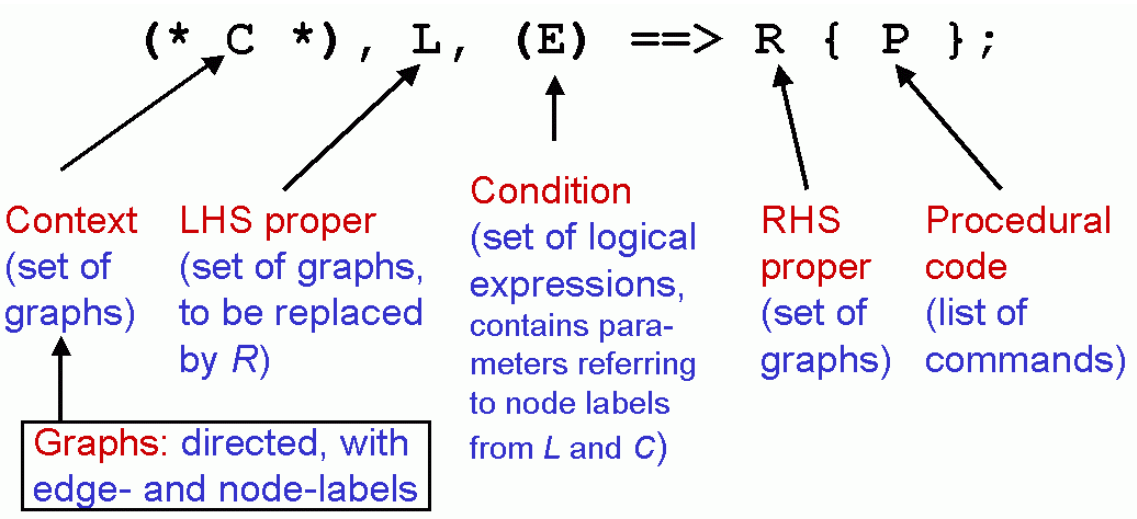


Fig. 20: Syntactic structure of an RGG rule. The essential effect of this rule is to replace *L* by *R* (and to execute *P*).

The application of a simple RGG rule to a given graph is demonstrated in Figure 21. Here, the upper part of the Figure describes the rule. There is no context *C*, no condition *E* and no procedural code *P* in this case. So, the left-hand sind, two nodes of classes **A** and **B** which are connected by a directed edge from **A** to **B**, has to be replaced by the right-hand side wherever it occurs. There are two sorts of edges (relations) in this example, which are visualised as solid and dotted arrows, respectively. The lower part of the Figure shows exemplarily an application of this rule: the red part on the left, encircled by a solid blue line, is identical with the left-hand side of the rule and is thus replaced by the corresponding right-hand side (result: lower right part of the Figure). Notice that the left-hand side of the rule does not match the part of the graph which is encircled by the broken blue line, because the edge connecting **A** with **B** is of the wrong sort there.

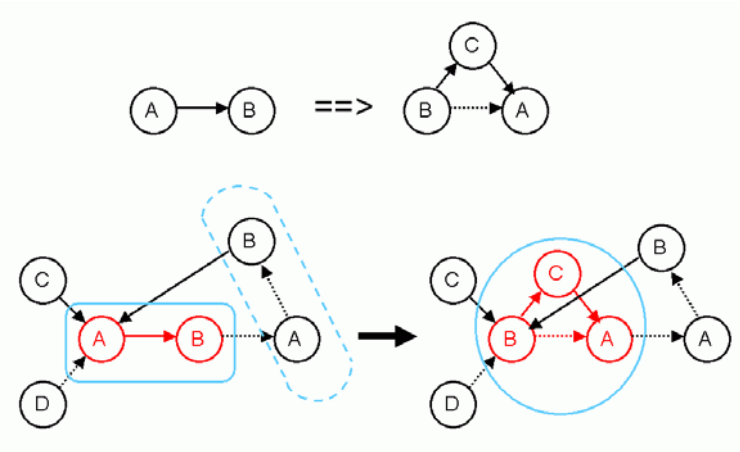


Fig. 21: Application of a relational growth grammar rule (upper part) to a graph (lower part).

Relational growth grammars are a special form of *graph grammars*. As for L-systems, there exists a well-developed theory about graph grammars [Rozenberg 1997]. L-systems can be subsumed as a special case, because strings can be represented as special graphs with a linear structure, with edges of a certain, fixed type "successor" between consecutive symbols. In XL, edges are generally written down in the form "`-`*edgelabel*`->`", where "*edgelabel*" specifies the type of the edge – but because the edge type "successor" is so often used, a simple blank symbol is allowed instead of "`-successor->`". This convention allows us to write down L-system rules in XL in a quite familiar manner – and in fact, all L-system examples shown above were directly taken from XL programmes. In order to make them readable by an XL compiler (like that in the software GroIMP, see below), one has only to enclose the rules (not the "module" declarations) in a surrounding construction of the form

```
public void run()
   [
   ...
   ]
```

(with the exception of the examples using interpretive rules, where a similar construction was already explicitly given). The reason is that RGG rules in XL can be organised in several blocks, in order to enable a better control of the order of rule application – thus making accessible the possibilities of so-called table L-systems [Rozenberg 1973].
However, the capacity of RGGs goes far beyond L-systems. A simple example for a graph transformation which cannot be expressed as an L-system rule occurs in genetics: In the context of sexual reproduction, there is the process of recombination of genetic information,

which takes place by so-called "crossing over" of two aligned DNA strings. The basic transformation which exchanges the bindings between the two DNA strings is shown in Figure 22.

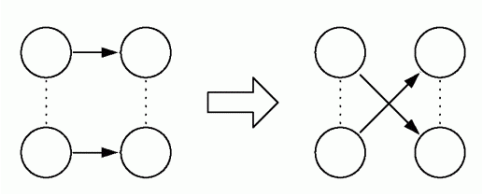


Fig. 22: Graphical representation of an RGG-rule for genetic crossing-over. Unbroken arrows stand for the successor relation in base sequences of DNA, dotted lines for alignment between two homologous DNA strings.

An XL representation of this rule is

```
a b, c d, (* a -align- c *) ==>> a d, c b;
```

and in fact we have used this rule together with one for mutation to simulate the evolution of artificial "biomorphs" [Kniemeyer et al. 2004].
In addition to the genetic level, it is also possible to represent biochemical reactions and metabolic reaction networks in the form of RGG rules. We will not go into details here (see [Buck-Sorlin et al. 2005]), but we show some of the visual results of models which have as a non-visible part also some metabolic and, in some cases, also genetic components: Figure 23 shows a flower and a mutant thereof, where some gene is deactivated – the effect of the "silent gene", namely, the lack of petals, is mediated by a reaction network (see [Kniemeyer et al. 2004]).

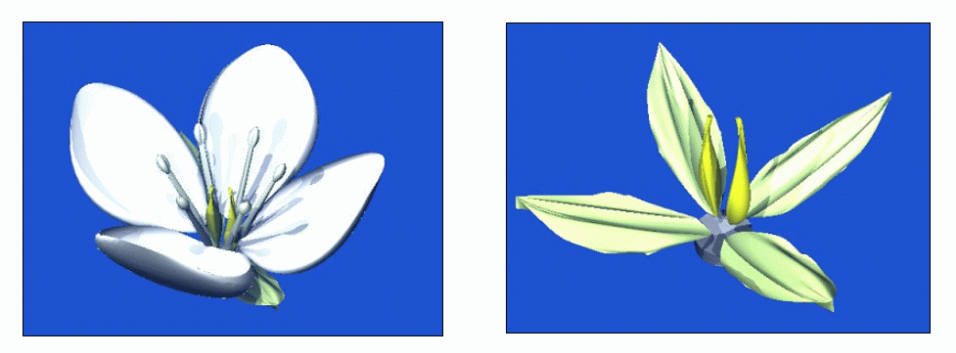


Fig. 23: Virtual "wild type" (left) and "mutant" (right) of a flowering plant, generated by an RGG which encodes also the causal genes and (a part of) the mediating transcription-factor reaction network (from [Kniemeyer et al. 2004], based on earlier work by [Kim 2001]).

Figure 24 shows two developmental steps of a virtuel rapeseed plant which is assimilating virtual carbon, depending on virtual light interception and nitrogen availability, and which allocates this carbon to its growing organs according to the (time-dependent) relations between source and sink strengths; see [Groer 2006]. Figure 25 shows a virtual barley plant (including the root system, which was not modelled in the other examples shown above) which depends in its growth not only on sunlight, but also on a reaction network producing a plant hormone (Gibberellic acid), like in real plants, and which can reproduce and mutate (Buck-Sorlin et al., partially unpublished work, see also [Buck-Sorlin et al. 2005]).

Fig. 24: Virtual rapeseed, generated by an RGG taking photosynthesis, nitrogen uptake and carbon allocation into account, all programmed in XL. From [Groer 2006].

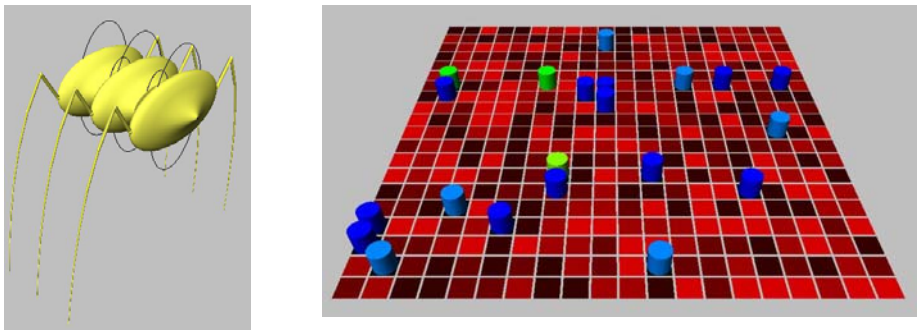
Fig. 25: Virtual barley plant with hormonal metabolism and genetic features, see text. From [Buck-Sorlin et al. 2006].

In the field of computergraphical modelling of plants, the traditional L-system approach has recently been challenged by the Xfrog software, developed by Deussen and Lintermann, see [Deussen 2003]: Here, graphs are interactively edited which define implicitly rules for the multiplication and arrangement of geometrical objects. Although this graph-controlled approach is not based on biological laws, it allows a quick interactive specification of complex vegetation models. However, the graphs used in Xfrog and the creation of geometrical structures based on them can exactly be reproduced in the language XL (if RGG

rules are complemented by a further type of rules, so-called instantiation rules) – see [Henke 2007]. The relations between Xfrog and our rule-based approach will be subject of a forthcoming article [Henke et al. 2007]. Figure 26 shows results of the simulation of Xfrog-defined structures in XL.

Fig. 26: Virtual plants generated using instantiation rules in XL which simulate the way how the Xfrog software [Deussen 2003] specifies virtual plants; from [Henke 2007].

But the use of RGGs is not restricted to plants. Figure 27 indicates other fields of application which are not yet completely explored.



(a) (b)

Fig. 27: RGG-based modelling beyond plants. (a) Insect-like animal (U. Bischof, unpublished). (b) Simulation of the agent-based "Sugarscape" model of an artificial society on a rectangular grid; from [Graeber 2006].

Relational growth grammars, embedded in XL programmes, can be read and executed by a software named *GroIMP* (Growth-grammar related Interactive Modelling Platform). This platform-independent software contains an XL compiler, a development tool (extended editor) for XL, a 3-D modeller and renderer (including a raytracer), a 2-D visualiser for the transformed graphs, windows for plotting functions, editing facilities for 3-D objects and for their attributes, tools for generating textures, networking facilities, a collection of RGG examples and a tutorial for the language XL. XL and GroIMP will be thoroughly documented in [Kniemeyer 2007]. The software is available by download under the GNU public licence (GPL), i.e. as an open-source tool; see **`http://www.grogra.de`**. A screenshot of the current GroIMP version is shown in Figure 28. All images of virtual structures in this paper were generated with GroIMP, with the following exceptions: Figs. 10–12 and 19 were created

with the GroIMP-forerunner *Grogra* [Kurth 1994], 13 is from [Knauft 2000], 15 is from [Prusinkiewicz & Lindenmayer 1990].

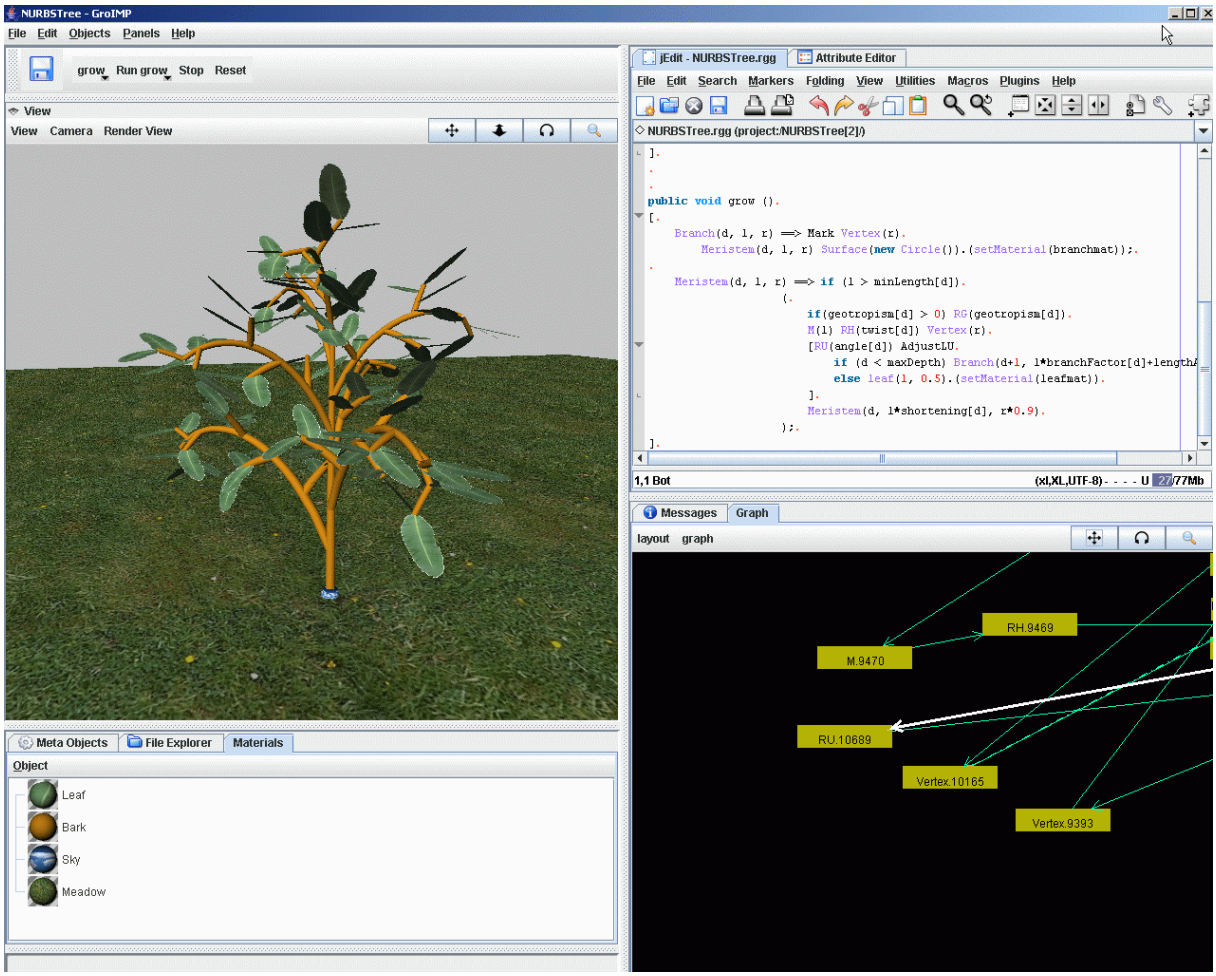


Fig. 28: Screenshot of the GroIMP software (download possible from www.grogra.de).

6. Discussion

The visual models of plants obtained with L-systems and relational growth grammars corroborate the assumption that these mathematical formalisms are appropriate tools to capture essential aspects of *morphological* structures in the world of *plants* – maybe analogously to the appropriateness of differential equations for modelling phenomena in physics. In fact, it is straightforward to formalise such morphological phenomena which are known by botanists under the notions of "acrotony", "neoformation", "sleeping buds", "reiteration", "apical control" etc. in the language of L-systems (see, e.g., [Kurth 1996]). If L-systems are extended to more flexible formalisms like relational growth grammars, this appropriateness can also be stated for plant models which connect several levels of spatial resolution and for *functional-structural models*, where the purely morphological layer is complemented by processes taking place "behind" the visual world or at smaller scales. It can be conjectured that models for the evolution of *network structures* could also profit from a formalisation in the frame of a rule-based language like RGGs. Early studies did already explore some non-biological applications of L-systems: Specification of planar tilings, music [Prusinkiewicz & Hanan 1989], ornaments, weave patterns, architecture, evaluation of mathematical ex-

pressions, robotics [Goel & Rozehnal 1991]. In a present students' course at the University of Technology at Cottbus, we just explore the usage of RGGs in architectural design.
Another possible field of applications for rule-based formalisms is chemistry. Chemical reactions have some similarity with grammar rules, but they usually take place in an unstructured "soup" consisting of a very large number of molecules – hence the linear ordering which we had in the L-system strings does not apply here, and it is doubtful if one can speak of morphological structures in this case, except when we restrict our focus to single, but complex molecules. Another feature of L-systems and RGGs which make them seem inappropriate for applications in chemistry and physics is their discretisation of time. Mathematical descriptions of classical dynamical systems make use of the concept of continuous time. This concept does also make sense when smooth animations of growth processes, of animal movement etc. are wanted. However, it has already been shown that "timed L-systems" can be defined which abandon the concept of fixed-length developmental steps in favour of continuous growth and event-driven application of rules [Prusinkiewicz & Lindenmayer 1990]. The incorporation of these modifications into more advanced formalisms like RGGs is still to be done, but will probably pose no great difficulties.
A probably even more urgent need for theoretical and practical research is revealed by the question how truly 2-D and 3-D structures like planar maps or spatial cellworks – in contrast to essentially 1-dimensional tree-like structures – and their growth and dynamics can be elegantly modelled using an appropriate grammar formalism. Until this challenge is not resolved by a really intuitive and compact calculus, we cannot say that true *picture morphology* can be satisfactorily modelled by known rule-based formalisms like L-systems. However, our experience from the creation of virtual plants and of some other interesting virtual patterns suggest that there are some features inherent to the rule-based programming paradigm which make it a promising candidate for playing a prominent role in a future theory of picture morphology.

Acknowledgements

Ole Kniemeyer designed and implemented the programming language XL and the GroIMP software; without these tools and without his help this study would not have been possible. Gerhard Buck-Sorlin, Reinhard Hemmerling, Michael Henke, Christian Groer, Bernd Graeber, Michael Tauer, Branislav Sloboda, Thomas Früh, Helge Dzierzon, Dirk Lanwert and Falk-Juri Knauft provided examples for applications, stimulated discussions and helped to improve certain aspects of the above-mentioned tools. Research was partially funded by the Deutsche Forschungsgemeinschaft and by the German Ministry of Research and Technology. All support is gratefully acknowledged.

References

Abelson, H.; diSessa, A. A. (1982): Turtle Geometry. Cambridge: MIT Press.

Ascher, Marcia (2003): Die kolam-Figuren Südindiens. *Spektrum der Wissenschaft*, 6/2003: 74-80.

Buck-Sorlin, Gerhard; Kniemeyer, Ole; Kurth, Winfried (2005): Barley morphology, genetics and hormonal regulation of internode elongation modelled by a Relational Growth Grammar. *New Phytologist*, 166 (3): 859-867.

Buck-Sorlin, Gerhard; Hemmerling, Reinhard; Burema, Benno; Kniemeyer, Ole; Kurth, Winfried (2006): Towards "Virtual Barley": Relational Growth Grammars as a universal framework for rule-based hierarchical modelling of morphogenesis, plant function, regulatory networks, and genetic processes. Waterman-Seminar, IPK Gatersleben, 15. 12. 2006 (unpublished ppt-presentation).

Deussen, Oliver (2003): Computergenerierte Pflanzen – Technik und Design digitaler Pflanzenwelten. Berlin: Springer.

Früh, Thomas; Kurth, Winfried (1999): The hydraulic system of trees: Theoretical framework and numerical simulation. *Journal of Theoretical Biology*, 201 (4): 251-270.

Goel, Narendra S.; Rozehnal, Ivan (1991): Some non-biological applications of L-systems. *International Journal of General Systems*, 18 (4): 321-405+color plates.

Graeber, Bernd (2006): Vergleich von Java und XL anhand des Agentenmodells "Sugarscape". Studienarbeit, BTU Cottbus, Lehrstuhl Praktische Informatik / Grafische Systeme.

Groer, Christian (2006): Dynamisches 3D-Modell der Rapspflanze (Brassica napus L.) zur Bestimmung optimaler Ertragskomponenten bei unterschiedlicher Stickstoffdüngung. Diploma thesis, BTU Cottbus (104 pp.).

Henke, Michael (2006): Entwurf und Implementation eines Baukastens zur 3D-Pflanzenmodellierung in GroIMP mittels Instanzierungsregeln. Diploma thesis, BTU Cottbus (156 pp.).

Henke, Michael; Kniemeyer, Ole; Kurth, Winfried (2007): Realisation and extension of the Xfrog approach for plant modelling in the graph-grammar based language XL. (in preparation)

Karwowski, R.; Prusinkiewicz, P. (2003): Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science*, 86 (2), 19 pp. http://algorithmicbotany.org/papers/l+c.tcs2003.pdf.

Kim, Jan T. (2001): A generic formalism for modelling regulatory networks in morphogenesis. In: Kelemen, J.; Sosík, P. (eds.), Advances in Artificial Life. *Lecture Notes in Artificial Intelligence*, 2159. Berlin: Springer, pp. 242-251.

Knauft, Falk-Juri (2000): Entwicklung von Methoden zur GIS-gestützten Visualisierung von Waldentwicklungsszenarien. Ph.D. thesis, Universität Göttingen, Fakultät für Forstwissenschaften und Waldökologie (126 pp.)

Kniemeyer, Ole; Buck-Sorlin, Gerhard; Kurth, Winfried (2004): A graph-grammar approach to Artificial Life. *Artificial Life*, 10 (4): 413-431.

Kniemeyer, Ole (2007): Design and implementation of a graph-grammar based language for functional-structural plant modelling. Ph.D. thesis, BTU Cottbus (forthcoming).

Kurth, Winfried (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 modelling. Introduction and Reference Manual. *Berichte des Forschungszentrums Waldökosysteme*, B, 38, Göttingen (192 pp.).

Kurth, Winfried (1996): Elemente einer Regelsprache zur dreidimensionalen Modellierung des Triebwachstums von Laubbäumen. In: Hempel, G. (ed.), Deutscher Verband Forstlicher Forschungsanstalten, Sektion Forstliche Biometrie und Informatik, 8. Tagung, Tharandt / Grillenburg, 25.-28. 9. 1995; Ljubljana: Biotechnische Fakultät, pp. 174-187.

Kurth, Winfried (1999): Die Simulation der Baumarchitektur mit Wachstumsgrammatiken. Berlin: Wissenschaftlicher Verlag Berlin (327 pp.).

Lindenmayer, Aristid (1968): Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18: 280-315.

Měch, Radomír; Prusinkiewicz, Przemyslaw (1996): Visual models of plants interacting with their environment. *Computer Graphics Proceedings*, *Annual Conference Series*, 1996; SIGGRAPH 96, New Orleans, August 1996: 397-410.

Prusinkiewicz, Przemyslaw; Hanan, James (1989): Lindenmayer Systems, Fractals, and Plants. *Lecture Notes in Biomathematics*, 79, New York: Springer (120 pp.).

Prusinkiewicz, Przemyslaw; Lindenmayer, Aristid (1990): The Algorithmic Beauty of Plants. New York: Springer. http://algorithmicbotany.org/papers/abop/abop.pdf.

Prusinkiewicz, Przemyslaw; James, Mark; Měch, Radomír (1994): Synthetic topiary. *Computer Graphics Proceedings*, *Annual Conference Series*, 1994; SIGGRAPH 94, Orlando, 24.-29. 7. 1994: 351-358.

Rozenberg, Grzegorz (1973): T0L systems and languages. *Information and Control*, 23: 357-381.

Rozenberg, Grzegorz (1997): Handbook of Graph Grammars and Computing by Graph Transformations. Vol. 1: Foundations. Singapore: World Scientific.

Vitanyi, Paul M. B. (1976): Digraphs associated with D0L systems. In: Lindenmayer, A.; Rozenberg, G. (eds.): Automata, Languages, Development. Amsterdam: North-Holland, pp. 335-346.