Computing and Informatics, Vol. 36, 2017, 33–54, doi: 10.4149/cai_2017_1_33

REALIZATION AND EXTENSION OF THE XFROG APPROACH FOR PLANT MODELLING IN THE GRAPH-GRAMMAR BASED LANGUAGE XL

Michael Henke

Department Ecoinformatics, Biometrics and Forest Growth
Büsgenweg 4
University of Göttingen
37077 Göttingen, Germany
e-mail: `mhenke@uni-goettingen.de`

Ole Kniemeyer

Antonianger 27
31061 Alfeld (Leine), Germany
e-mail: `ole.k@arcor.de`

Winfried Kurth

Department Ecoinformatics, Biometrics and Forest Growth
Büsgenweg 4
University of Göttingen
37077 Göttingen, Germany
e-mail: `wk@informatik.uni-goettingen.de`

Abstract. Two well-known approaches for modelling virtual vegetation are grammar-based methods (L-systems) and the Xfrog method, which is based on graph transformations expanding "multiplier" nodes. We show that both approaches can be unified in the framework of "relational growth grammars", a variant of parallel graph grammars. We demonstrate this possibility and the synergistic benefits of the combination of both methods at simple plant models which were processed using our open-source software GroIMP.

Keywords: Plant modelling, object instancing, graph grammar, Xfrog, XL, GroIMP

Mathematics Subject Classification 2010: 68-U05

1 INTRODUCTION

Modelling the detailed structures of plants with a custom interactive 3D modeller is very time-consuming. Several algorithmic solutions have been implemented using general-purpose programming languages to construct realistic vegetation structures automatically. Beyond these ad-hoc solutions, there are two approaches offering a more generic framework for the specification of the architecture of individual plants: the string-based formalism of Lindenmayer systems, realized, e.g., in the software LStudio [18], and the graph-based interactive approach proposed by Lintermann and Deussen [16] and realized in the software Xfrog [7].

Other approaches for modelling trees, like the one introduced by Pirk [19], use skeleton-based geometries extracted from images or laser scanners to generate 3D structures. The produced dynamic models can react on environmental influences, a feature which was in the past only possible when using growth models such as L-systems. On the other hand the simplicity of the models allows a creation on the fly so that they can be used in real-time scenarios such as games or simulations.

Ijiri [11] presented a sketch-based technique, a combination of rule-based and image-based techniques on procedurally created trees. Stroke inputs are used in L-systems to control the overall model appearance and the depth of recursion.

These approaches are mainly focused on fast production of realistically looking images of plants. They work with interactive design tools and simplified structures and do not claim to be botanically correct in any case. Modelling plant functions like transport processes is not considered.

Here we present a combination of the object instancing approach, as implemented in Xfrog, and rule-based modelling. Our modelling system consists of three components: Relational Growth Grammars (RGG) as formal basis, the programming language XL (eXtended L-system language) enabling an easy use of RGG and at the same time extending the well-known object-oriented language Java, and the software GroIMP (Growth-grammar related Interactive Modelling Platform), providing interactive facilities, rendering, and a full-scale development environment for XL.

2 STATE OF THE ART

Lindenmayer systems (L-systems for short) are systems of replacement rules operating on strings. Developed in the context of formal grammar theory, they can be used to specify the growth of vegetative structures according to botanical rules, which has been demonstrated in various papers and books, the most prominent by Prusinkiewicz and Lindenmayer [20]. To this purpose, the strings generated by the grammar mechanism have to be traversed from left to right and must undergo an interpretation by turtle geometry (see [20] for details).

Sequential graph grammars have previously been used in various fields of application, most often in software engineering, but also in pattern recognition and image analysis. Graph grammars with parallel mode of operation, as in our case, were theoretically investigated in the seventies (for references see [12]), but then got out of focus for a while, and their use to generate 3D scene graphs is new. Besides L-systems with interpretation [12] and structural factorization [21] the Xfrog multiplier nodes are another concept for object instancing that can be used within GroIMP. With the combination of these concepts we provide new ways for fast prototyping and model development.

The Xfrog approach, on the other hand, allows to interactively edit a graph made up of component prototypes, the so-called p-graph (see [3] for details). Among the nodes of this graph there are not only shape nodes representing graphical primitives, but also various sorts of *multipliers* which have the semantics of copying the structures encoded by their descendant nodes and placing them in specified positions. E.g., a "Wreath" node generates a circular arrangement of copies. The p-graph is then expanded to a tree, the so-called i-tree, having instances of the prototypes as nodes, and this tree is then traversed, similar to a scene graph [6], in order to build the geometrical model of the plant (Figure 1, cf. [3]). The interactive access to the p-graph requires a medium level of abstraction and allows a quick feedback from the resulting rendered model to the editing process, thus enabling a quite intuitive working. The portfolio of components (node types) is, however, restricted, and there is no natural way to simulate processes of growth and development in this framework – or even to include biologically-inspired process-based models (e.g., of plant hormonal effects controlling flowering), which is relatively easy in L-systems (cf. [20]). The current version of Xfrog is implemented as plugin for the 3D computer graphics softwares Cinema 4D and Maya.

3 METHODS

In the following, an introduction of Relational Growth Grammars (RGG) is given and the application of two sorts of RGG rules, generative and instantiation rules, will briefly be explained. After that, a short introduction of the interactive modelling platform GroIMP is given.

3.1 Relational Growth Grammars

While L-systems are widely used, they have still some drawbacks – not so much concerning their theoretical power, but with respect to transparency and simple use when complex, multi-level plant models including functional aspects and/or

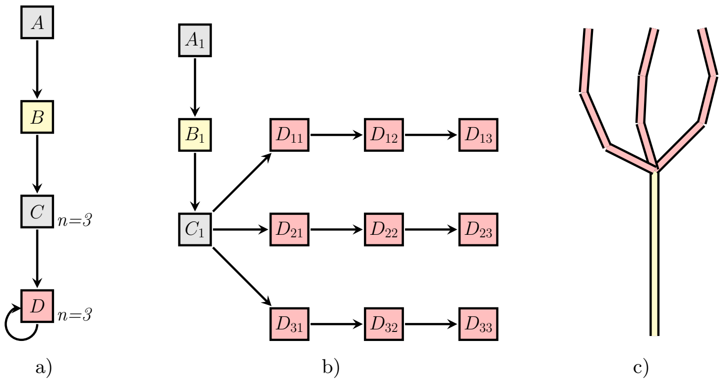


Figure 1. The Xfrog workflow. In this example the nodes B and D have a geometric interpretation while the node C is a replicator without a geometry. The node C replicates its subgraph three times. In this case the subgraph consists only of node D. The node D replicates itself also three times. a) i-tree, b) p-graph, c) one possible geometrical interpretation; colouring is in the corresponding graph colours.

genetic control are required. One critical feature is: L-systems operate basically on strings, which have to be translated into 3D-structures (representing plants or plant communities), the latter being the actual objects of modelling (see Figure 2 a)).

We use the concept of RGG, a graph grammar formalism, and its implementation in the language XL (eXtended L-Systems) to overcome this and other drawbacks. In XL, nodes are objects in the sense of object-oriented programming, they generalize the symbols in classical L-system strings and can be associated with Java classes. Edges can represent arbitrary, user-defined relations, they generalize the sequential order of symbols in strings. Hence the extra description level of strings can be dispensed of in the rewriting process (Figure 2 b)); we use strings only for writing down the rules.

Advantages:

- **Complex relationships** such as genotype-phenotype relations can now be represented with the same simplicity as a topological neighbourhood in classical L-systems,
- the same holds for **multiscale** plant descriptions [17],
- arbitrary sorts of **context** can easily be defined,
- the representation of **networks**, including feed-back loops, is possible in our formalism in an intuitive way (as graphs),
- the interface between rule-based model description and **procedural modelling** becomes more elegant by incorporating Java classes (as nodes) and scripts in the rules,

- **strings, trees** and **multisets** are **subcases** of our graph data structure, thus our RGG have at least the same descriptive power as the rewriting systems operating on these restricted structures.

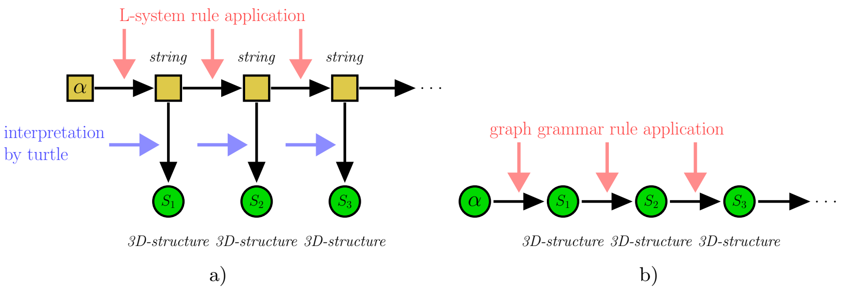


Figure 2. Functioning of a) a classical L-system, compared with b) a relational growth grammar. α is the start symbol (axiom). The developmental steps of a plant or plant community are represented by 3D-structures $S_1, S_2, \ldots$

Details about RGG, XL and GroIMP have been described in a thesis [12] and are used in the field of biological modelling of plants [2, 9, 23]. RGGs are parallel rewriting systems operating on typed attributed graphs (instead of strings) and form thus a variant of graph grammars. Their exact definition can be given in terms of algebraic graph-grammar theory [13] and captures the "dynamic component" which is inherent in L-systems but lacking in the Xfrog approach. For instance, an RGG rule given in XL in the form

```
Axiom ==> Cylinder(2,3) Cylinder(20,1) Sphere(3);
```

will replace a default initial node called `Axiom` by a graph consisting of three successive nodes (connected by successor edges – the blanks delimiting the components of the right-hand side of the rule are used to construct edges of type "successor"), and these nodes are interpreted as parts of a scene graph, namely as a cylinder with length 2 and radius 3, on top a cylinder with length 20 and radius 1, on top a sphere with radius 3. All three nodes can again be replaced by other nodes if there are corresponding additional rules. Furthermore, auxiliary nodes like `A` or `B(1)` without geometrical meaning are allowed, similar to L-systems. Edges of other types than "successor" can be specified using the notation "`-edgetype->`".

3.2 Generative Rules within XL

The "normal" type of rules used in an XL program is a generative RGG rule, a straightforward generalization of an L-system rule. In the example given above in the Introduction, one node of type `Axiom` is replaced by a new subgraph, consisting of two nodes of type `Cylinder` and one of type `Sphere`. These node types are predefined as Java classes for the scene graph of the modelling platform GroIMP; as Java classes they have encapsulated attributes, amongst them `radius` and (for `Cylinder`) `length`, and their appearance as part of the right-hand side of a rule is analogous to a constructor invocation in Java. Node types can also be defined by the user, and they can inherit from other node types. Besides conventional Java class declarations, a shorthand notation called module declaration is possible: for instance,

```
module A(int i, super.radius) extends Sphere(radius);
```

defines a new user-defined extension of the `Sphere` node class inheriting the `radius` parameter but with an additional parameter `i` with integer values as well as the geometrical shape of the `Sphere` node class.

In XL it is possible to insert imperative commands in right-hand sides of rules; thus an alternative, but equivalent rule to the example given in the Introduction would be

```
Axiom ==> Cylinder(2,3) Cylinder(20,1)
          s:Sphere { s.setRadius(3); };
```

where the created `Sphere` node is labelled `s` and the assignment of its `radius` attribute is done *a posteriori*. (It is also possible to have more than one node on the *left-hand side* of an RGG rule and thus to replace non-trivial subgraphs, but we will not use this possibility in the following.) Transformation nodes like `Translate(x,y,z)` or `Scale(u)`, as known from scene graphs, are also defined. With rules of this sort, classical L-systems as well as many of their extensions published in the plant-modelling literature can be emulated. However, what is still missing is the possibility to copy whole subgraphs, as it is required during the transformation of the Xfrog p-graph to the i-tree. This can be done in a generative XL rule like that shown above by introducing a user-defined node type (here called `Replicator`) and invoking the `cloneSubgraph` method provided by GroIMP. To connect the subgraph which is to be replicated to the replicator, we use an extra edge type called `multiply`. The method `getFirst` yields the subgraph beginning with the first node accessible via this edge from the replicator. Figure 3 shows the respective graphs where the initial state consists only of a default node (`Root`) and the initial node `Axiom`.

```
module Replicator;

public void run() [
      Axiom ==>
            Cylinder(2,3) Replicator -multiply->
            Cylinder(20,1) Sphere(3);

      r:Replicator ==>
            [cloneSubgraph(r.getFirst(multiply))]
            Translate(10, 0, 0)
            [cloneSubgraph(r.getFirst(multiply))];
]
```

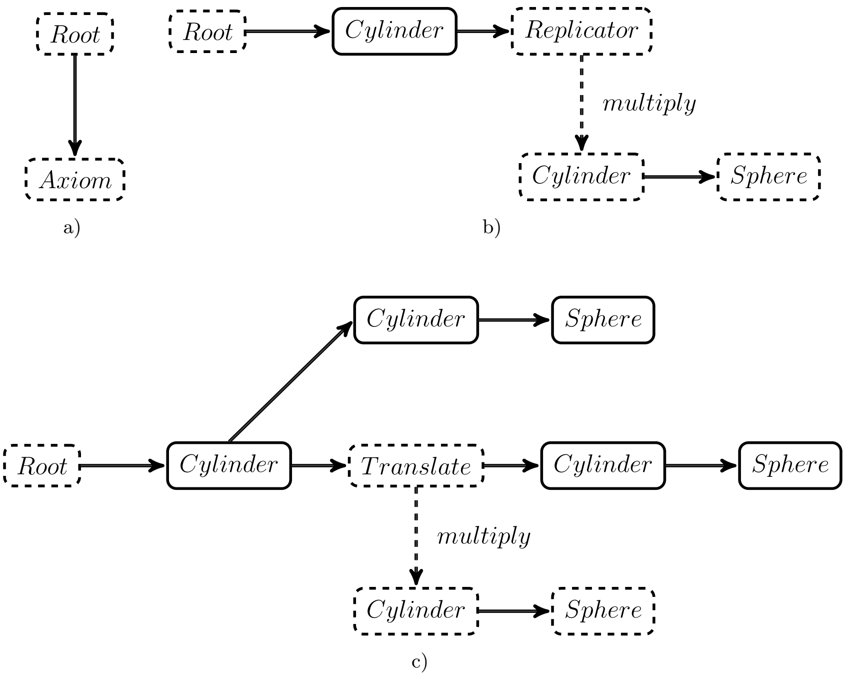


Figure 3. Graphs produced by the "Replicator" grammar from the text, a) for the initial state, b) after a single time step and c) after two time steps. Dashed nodes are not visible in the 3D view of the structure.

The visible result of this small XL program (Figure 4), consisting of two RGG rules which are first applied to the default start node `Axiom`, is after the first time step only the first cylinder (with length 2 and radius 3), because the `Replicator` node has no visual interpretation and the `multiply` edge is not traversed during geometrical interpretation of the graph (see Figure 4 a)). In the second time step, however, the replicator is replaced via application of the second rule by two copies of the subgraph consisting of the long cylinder and the sphere, which are separated (because of the `Translate` node) by 10 units in x direction, see Figure 4 b).

3.3 Instantiation Rules

By using relational growth grammars as described above, we can construct a scene graph consisting of nodes for geometric primitives and further nodes which may describe non-geometric states of the underlying (botanical) model. However, there are cases where it is advantageous to assign a set of primitives to a single node of the graph. For example, a single entity of the model may need several primitives for its 3D representation, and then it would be cumbersome, a waste of memory

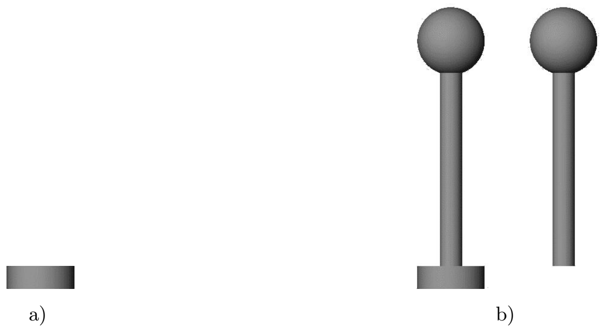


Figure 4. Graphical result of the "Replicator" grammar from the text, a) after a single time step and b) after two time steps

and a violation of the principle of separation of concerns if one has to include these primitives in the same graph as the entities of the model.

Therefore, the language XL defines *instantiation rules* which can be assigned to node classes and which may expand a single node to a set of primitives when they are invoked by GroIMP as part of the 3D visualization of the node. Although these rules resemble generative rules in syntax, they do not modify the graph and are only activated during visualization. A formal definition is given in [21]. For an instantiation rule, we have to use a module declaration and add the nodes which shall be used for visualization after an arrow symbol as in

```
module Stem(float len) ==> Cylinder(len,1);
```

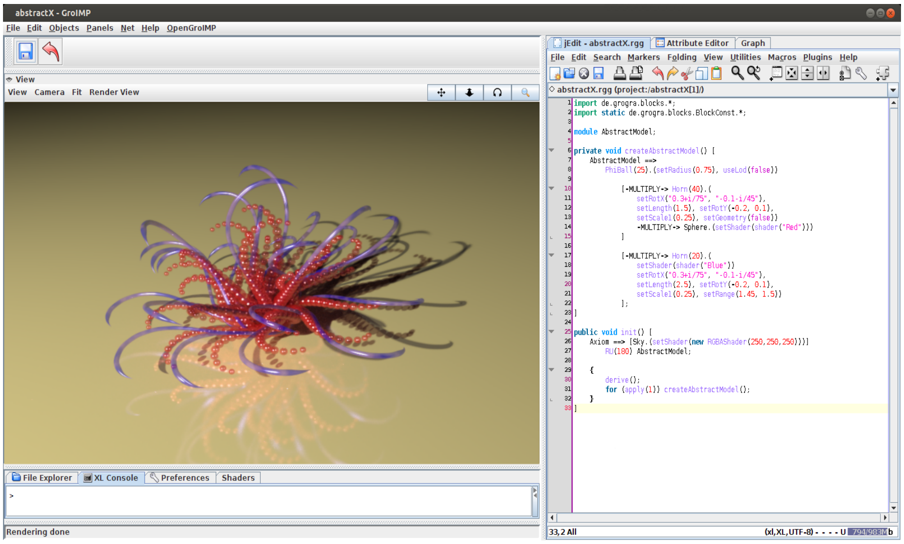
which represents a stem as a cylinder without the requirement that the class `Stem` inherits from `Cylinder`. The right-hand side may also contain references to other parts of the graph so that instantiation rules provide a simple means to specify object instancing, i.e. multiple occurrences of the same 3D structure at different locations. This can be used for the replicator example from above: if we use for the `Replicator` node an instantiation rule instead of the simple generative rule, we can dispose of the `cloneSubgraph` invocation.

```
module Replicator ==>
        [ getFirst(multiply) ]
        Translate(10, 0, 0)
        [ getFirst(multiply) ];

public void run() [
        Axiom ==>
                Cylinder(2,3) Replicator -multiply->
                Cylinder(20,1) Sphere(3);
]
```

The graphical result of this XL program is the same as above in Figure 4 b).

3.4 The Software GroIMP

The open-source 3D modelling platform GroIMP [8] has been developed together with the formalism of relational growth grammars and the language XL to have an integrated environment for rule-based 3D modelling. GroIMP provides a rich set of 3D node classes including simple ones like spheres, boxes and cylinders, but also NURBS surfaces, height fields and CSG operations. GroIMP maintains a current graph which is interpreted as a scene graph for visualization and may be transformed by rules specified in the language XL. The user may select a node in the 3D visualization and inspect or modify its attributes. Depending on the underlying rules of the model, interactive modifications by the user like the removal of branches of a plant may influence the further development of the structure. GroIMP contains an OpenGL visualization and an integrated raytracer with the option to use path tracing. Besides being a 3D modelling platform, GroIMP also contains a source code editor and an XL compiler to facilitate rule-based modelling with the language XL. Figure 5 shows a screenshot of the GroIMP modelling platform.


abstractX - GroIMP
File Edit Objects Panels Net Help OpenGroIMP
View
View Camera Fit Render View
File Explorer
XL Console
Preferences
Shaders
Rendering done
jEdit - abstractX.rgg
Attribute Editor
Graph
File Edit Search Markers Folding View Utilities Macros Plugins Help
abstractX.rgg (project:/abstractX[1]/)
33,2 All
(xl,XL,UTF-8) - - - - U

```
import de.grogra.blocks.*;
import static de.grogra.blocks.BlockConst.*;

module AbstractModel;

private void createAbstractModel() [
    AbstractModel ==>
        PhiBall(25).(setRadius(0.75), useLod(false))

            [-MULTIPLY-> Horn(40).(
                setRotX("0.3+i/75", "-0.1-i/45"),
                setLength(1.5), setRotY(-0.2, 0.1),
                setScale1(0.25), setGeometry(false))
                -MULTIPLY-> Sphere.(setShader(shader("Red")))
            ]

            [-MULTIPLY-> Horn(20).(
                setShader(shader("Blue"))
                setRotX("0.3+i/75", "-0.1-i/45"),
                setLength(2.5), setRotY(-0.2, 0.1),
                setScale1(0.25), setRange(1.45, 1.5))
            ];
]

public void init() [
    Axiom ==> [Sky.(setShader(new RGBAShader(250,250,250)))]
        RU(180) AbstractModel;

    {
        derive();
        for (apply(1)) createAbstractModel();
    }
]
```

Figure 5. Screenshot of the GroIMP software displaying an “alien plant” in the 3D-view. The editor window on the right hand side displays the corresponding model code.

4 APPLICATIONS

This section will demonstrate the emulation of Xfrog’s multiplier components in the language XL, but also some extensions of the Xfrog functionality which result in a natural way from the embedding in the new framework. A discussion of the advantages, possible applications and future extensions of our modelling approach will close the paper.

4.1 Simple Models

At the example of the `Wreath` component of Xfrog we want to show how to put into practice the Xfrog components in the modelling language XL. The `Wreath` component multiplies its child structure in a ring around a centre point. The radius of the ring where the instances are generated as well as their number can be controlled by the user. In our implementation of a wreath instantiation module we will use the following four attributes:

`number`	number of generated instances
`rX`	radius in x direction
`rY`	radius in y direction
`scale`	a uniform scaling factor

These attributes are included in line 1 of the following code specifying an instantiation rule. In the body of the subsequent loop we have two parts: one Java part (lines 3–10) with some calculations and one instancing part (lines 11–14) which produces the geometry. Like in the replicator example in the Introduction, the method `getFirst` (line 13) returns the first node which is attached to the current `Wreath` node by a `multiply` edge. This node represents the root of the subgraph to be multiplied.

```
 1 module Wreath (int number, float rX, float rY, float scale)
 2     extends Point ==> {
 3         float delta = (float)(2 * PI / number);
 4     }
 5 for (int i = 0; i < number; i++) (
 6     { // java part
 7         float w = i * delta;
 8         float x = rX * (float) Math.cos(w);
 9         float y = rY * (float) Math.sin(w);
10     }
11     [ // graph part
12         Translate(x, y, 0) Scale(scale)
13         getFirst(multiply)
14     ]
15 );
```

The following XL code produces an elliptic distribution of 15 cone nodes as shown in Figure 6, using the `Wreath` class defined above. The elliptic shape depends on the different radii for x and y axis, here 6 for x and 4 for y. The scaling factor in this example is for all instances one.

```
Axiom ==> Wreath(15,6,4,1) -multiply-> Cone;
```

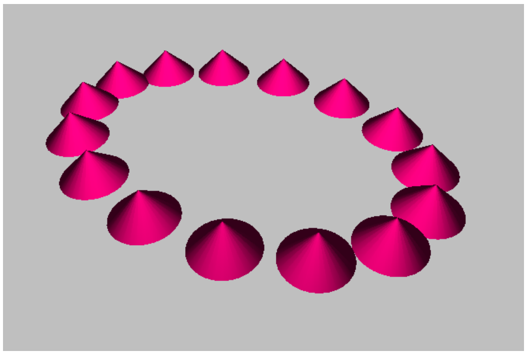


Figure 6. Graphical result of the application of the `Wreath` class to a cone, demonstrating the possibility to model an Xfrog component using the language XL.

In [10] the main Xfrog components were made available for GroIMP: `Tree`, `Horn`, `Hydra`, `Wreath` and `PhiBall`. As extensions there are also some new components: `Variation`, `BlockScale`, `BlockColor` and `Arrange`.

Because the user does not need to care about the technical realization of these multiplier nodes, they were predefined and collected in a "3D-construction-set package" (3D-CS) which is integrated as a library in the modelling platform GroIMP. They are called instantiation components or blocks. The following instantiation components are available in GroIMP:

Tree: Basic component for trees, creates the geometry of a stem and multiplies subsequent components as branches. Parameters are the distribution of branches, their scale, angle etc.

Horn: A component that places other components on a user-defined curve. It is used for stems, twigs, etc.

Hydra: Multiplies subsequent components on a curve with any direction relative to the direction of the parent component.

Wreath: The functionality is integrated in the Hydra component (as in Xfrog v.4.0.).

PhiBall: Multiplier that distributes all connected structures on a section of an ellipsoid according to the golden angle.

Arrange: Main component for arranging large numbers of instances on an area according to user-defined terrain data.

Variation: Allows in combination with any multiplier to vary the generated instances in a sequential, spread, exceptional or random way.

BlockScale: Scaling component, enables scaling depending on internal variables.

BlockColor: Enables colouring depending on internal variables.

With this set of instantiation components it is possible to model a huge number of not only organic structures. Such models are specified by two parts. The first part is a graph whose nodes are instantiation components and graphical primitives and the second part is a set of attributes for each component. The graph describes the physical structure of the model. For example, a tree consists of a stem with some levels of branches, and on the last level the leaves are located. The attribute set determines the properties of a component, e.g., for the tree, the number of leaves that a branch generates.

4.2 The Arrange Component

With great amounts of objects to be distributed on a terrain in a realistic manner, only in the rarest cases an individual positioning can be done manually. Hence efficient procedures for modelling of whole populations must be found. [4] already introduced a system built around a pipeline of tools that address this task. However, it was never implemented as Xfrog component. The `Arrange` component offers a wide range of possibilities to specify distributions on a terrain. They can be separated into four classes:

geometric arrangement generates a strict geometric arrangement, e.g., following lines or circles.

probability arrangement arranges the objects according to probability distributions (Poisson or normal distribution).

halftoning arranges the objects according to halftoning methods allowing for a user-defined density field [22, 15].

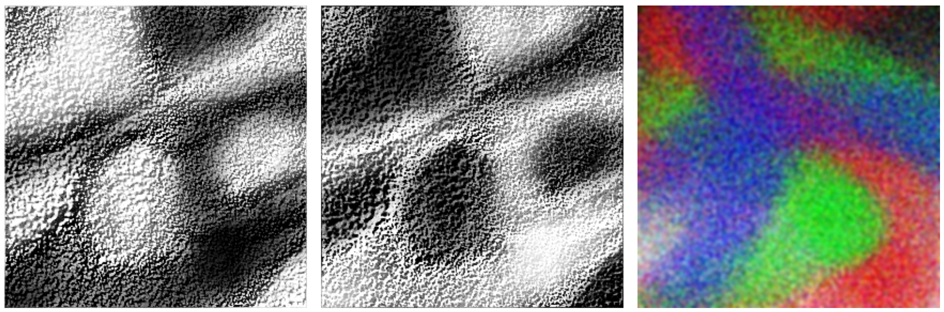
additional operations collection of operations like tilings or iterative methods like Voronoi-Lloyd [5].

In addition to arrangements of objects on a plane, the user can define terrain data and location parameters for the whole area to be filled. The information is available in every multiplied structure and can easily be changed just by exchanging an underlying image file.

The following small example generates an `Arrange` field with 50 uniformly distributed `Horn` instances.

```
Axiom ==>
          Arrange(50) -multiply->
          BlockScale("0.01+n1", "0.01+n1", "0.5")
          BlockColor("10", "n2*256", "10")
          Horn(5).(setLength("0.05+n3*0.2"));
```

Scaling, colouring and length of instances depend on user-defined location parameters, which are given by an image (see Figure 7). Each channel of this image, usually interpreted as RGB colour specification, can be accessed by using one of the variables `n1`, `n2` or `n3`, and so they can easily be fed into a model. In the above example, the colour of the instances depends on the variable `n2` which is used in the `BlockColor` component to set the green channel multiplied by 256.



a) height field b) density field c) location parameter field

Figure 7. User-defined location parameters for an `Arrange` component are defined by image files

The attributes as well as the inclusion of the location parameter field can be inspected and modified in an attribute editor. A part of the attribute editor for the arrange component is shown in Figure 8. In the lower part one can see how the location parameter field is included.

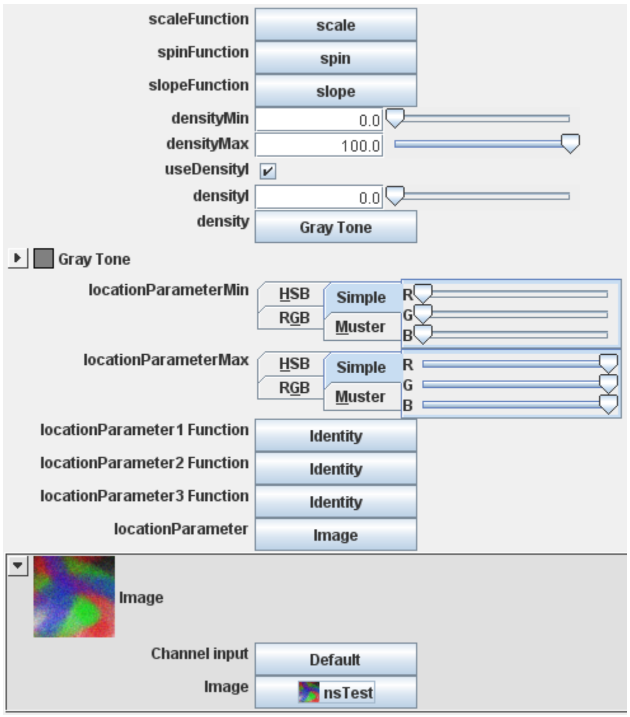
Starting from the initial state shown in Figure 9 a) we can now utilize the user-defined fields shown in Figure 7. Figures 9 b), 9 c) and 9 d) show the individual applications of user-defined fields. Figure 9 e) reflects the joint application of all three fields.

Besides the already mentioned attributes of the `Arrange` component, the possibility of scaling, rotation as well as random modification of positions of the produced instances is implemented.

4.3 Combination with Grammar-Based Models

Modelling with the 3D-construction set permits a fast and easy way to produce attractive models. However, because instantiation rules are a purely structural concept, modelling of functionality with them is not possible. Functional-structural models of plants, including dynamics of growth, can be obtained by a combination of our 3D-CS with generative RGG rules. So it is possible to instantiate complex plant organs like blossoms and use them in generated structures. They do not have to be generated using a complex derivation process. On the other hand, structures described by RGG-based models can be multiplied and/or positioned by instantiation components.

The first example does not use generative rules at all, but only a conventional control structure for iteration. The XL code "for (int i :(1: n)) (X)" generates `n` replications of X, where X can be any object or graph structure. Thus the following



scaleFunction scale
spinFunction spin
slopeFunction slope
densityMin 0.0
densityMax 100.0
useDensityI ✓
densityI 0.0
density Gray Tone
Gray Tone
locationParameterMin HSB RGB Simple Muster R G B
locationParameterMax HSB RGB Simple Muster R G B
locationParameter1 Function Identity
locationParameter2 Function Identity
locationParameter3 Function Identity
locationParameter Image
Image
Channel input Default
Image nsTest

Figure 8. Panel for the attributes of the `Arrange` component

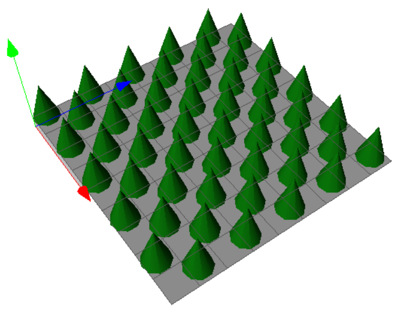
code section generates 15 `Horn` instances and arranges them at random positions on a 10×10 square field (Figure 10).

```
Axiom ==> for (int i:(1:15)) (
  [
          Translate(random(0, 10), random(0, 10), 0)
          Horn().(setLength(0.01+i/30))
  ]
);
```

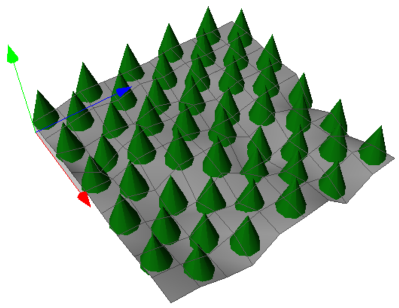
Additionally the length of the generated instance depends on its iteration index.

The next example demonstrates how a blossom produced by instantiation can be used as a module in an RGG generated structure. The branching structure is derived from these generative rules:

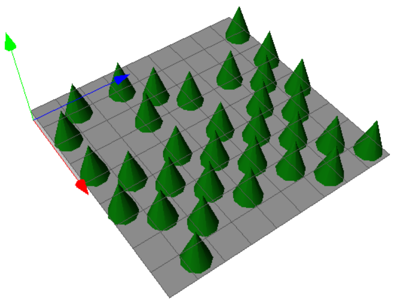
```
Axiom ==> Shaft(3.5, 0);

Shaft(x, a) ==>
         D(x/5) F(x)
          [
```



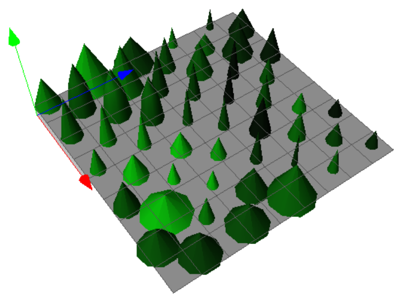
a) initial base



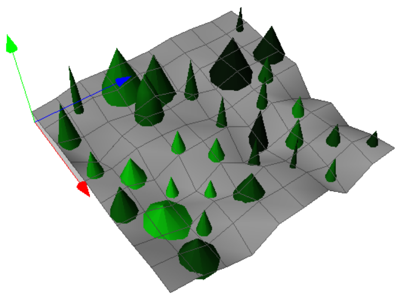
b) height field



c) density field



d) location parameter field



e) result, all fields applied

Figure 9. Example of a configuration of parameterised objects generated with the `Arrange` component and employing the user-defined fields from Figure 7. Figures a)–e) show variants where different parameter fields are switched on or off.

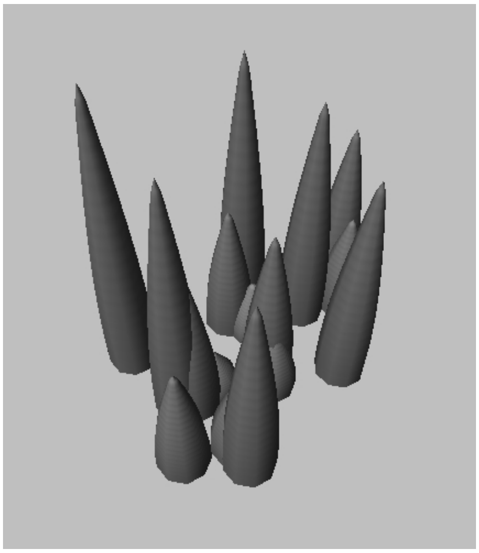


Figure 10. Graphical result of the simple "for" loop example: 15 randomized arranged `Horn` instances with increasing length

```
            RH(a) RU(45) LeafA(x) Branch(0.3*x, 20)
          ]
          Shaft(x*0.9, a+137.5);

Branch(x, age) ==>
          D(0.6*x) F(x) RU(-3) Branch(x*0.9, age+5);

LeafA(x) ==> LeafA(x*1.12);
```

The rule for `Shaft` generates the shaft and arranges the branches around it. There we use two parameters `x` and `a`. `x` is the actual length and `a` the actual branching angle. The next two rules for `Branch` and `LeafA` define instantiation modules. `LeafA` just generates the green leaf and lets it grow by a factor of 1.12 per step. The `Branch` module instantiates a branch-like structure:

```
module Branch(float x, int age) ==> {
          makeGraph ==>
            rootS:Scale(age/50)
            PhiBall(age/3).(setRadius(0.5),
              setFan1(0.2), setScale(1.0, 0), useLod(false)
            ) -multiply-> Rotate(0, age, 90) Petal();
}
rootS;
```

Starting with a `Scale` node a `PhiBall` multiplies a `Rotate` node followed by a `Petal`. The important parameter to control the blossom is the age. It controls the scaling factor, the number of petals as well as their opening angle. A petal itself generates a bent NURBS surface with a petal texture (code not shown). The result of the model is shown in Figure 11.

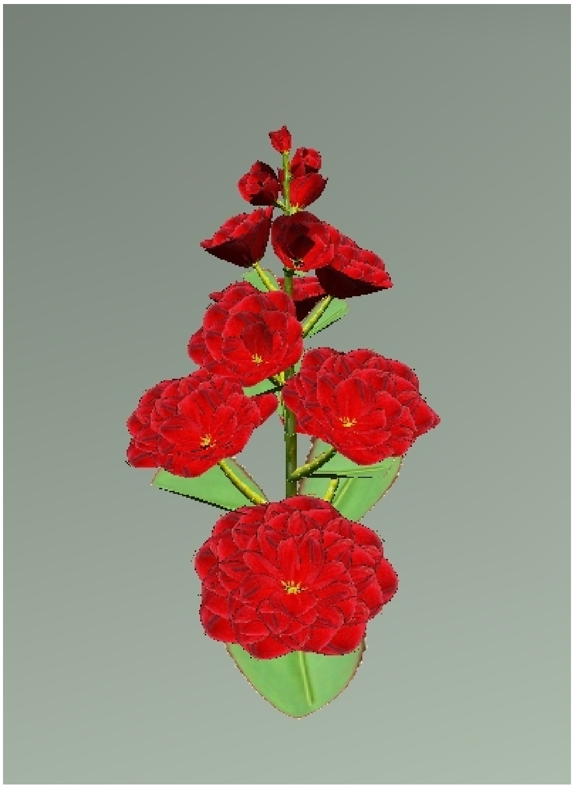


Figure 11. Blossoms generated by instantiation rules

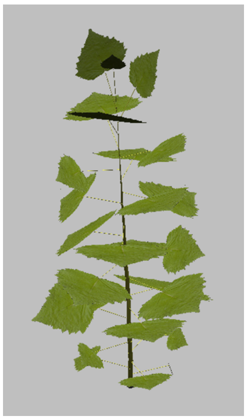
The last example demonstrates the reverse method, an instantiation component, in this case an `Arrange` component, multiplies a generated structure. Here we used a biological model of a young unbranched poplar taken from [1]. This model includes methods for biosynthesis and transportation of phytohormones as well as process-based calculations of photosynthesis depending on light interception. The instantiation is quite easy: In a first initial step five different trees will be instantiated and saved in an array. To arrange them, all to do is to place an `Arrange` component before the tree model. In XL code, this could look like

```
Axiom ==> Arrange.(
                setWidth(2, 2),
            setArrangeMethod(new AdditionalArrange(30))
        ) -multiply-> PoplarTree;
```

With `setArrangeMethod` we define the arrange method. The class `Additional-Arrange` includes some additional methods available in the arrangement process, where the default method is "DartThrowing", a random based distribution. The constructor argument 30 sets the number of generated instances. The module `PoplarTree` finally returns one of the predefined trees depending on the location parameters of the current position.

As a modification of the output of the original poplar model, the growth potential of the initial meristem of each individual is made dependent on its height over ground level, which can be determined by the location parameters of the `Arrange` component. Thus, small trees appear in lower regions and larger trees in the higher parts.

Figure 12 b) shows the result: 30 poplars arranged by an `Arrange` component.


a) original single tree model

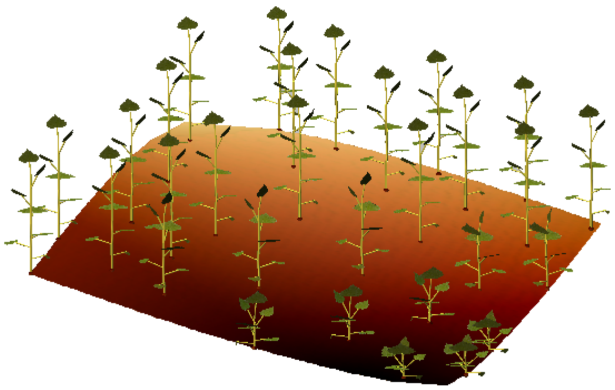

b) arranged trees; using the new `Arrange` component

Figure 12. Model of young unbranched poplar taken from [1]

5 CONCLUSION

Modelling with instantiation components is a powerful technique to get results quickly. Although this modelling technique has no botanical basis as it does not capture the growth process, the results are visually satisfying. It is hard to translate data taken form nature to the parameters of the component. A relation between reality and model can only be subsequently produced by measurements at the archetype and comparison with the model. In the rarest cases this gives botanically correct models. The degree of lifelikeness lies completely in the responsibility of the modeller.

The generated graphical results are nevertheless more than only nice pictures. With them, new possibilities arise for example in the visualisation of ecological data. In landscape planning, the results of interventions in ecological systems can be represented using the `Arrange` component. Decision makers are thus put into the situation to move around in a planned virtual landscape and then to consider the alternatives.

Further fields where the potential applications are promising are architecture, driving and flight simulators, films as well as games.

Using the combination of instantiation with generative relational growth grammars, some of the restrictions of pure instantiation-based modelling can be compensated. It is now possible to model significant causal aspects of processes of growth, their control being realized by a botanically-tested growth grammar. In this framework it is, e.g., possible to represent biochemical reactions and metabolic reaction networks; see [14, 2] for details. These highly-detailed modelling approaches can now easily be combined with the instantiation-based Xfrog approach for specifying virtual plants.

As further extension, an interactive graphical rule editor would be one next step to develop. This would free the user from the necessity to specify the rules and instantiation components by writing XL code.

Acknowledgements

Research was partially funded by the German Research Foundation (DFG) under the project identifier Ku 847/6-1. All support is gratefully acknowledged.

REFERENCES

[1] Buck-Sorlin, G. H.—Kniemeyer, O.—Kurth, W.: A Model of Poplar (Populus Sp.) Physiology and Morphology Based on Relational Growth Grammars. In: Deutsch, A., Parra, R. B., de Boer, R., Diekmann, O., Jagers, P., Kisdi, E., Kretzschmar, M., Lansky, P., Metz, H. (Eds.): Mathematical Modeling of Biological Systems, Volume II, Modeling and Simulation in Science, Engineering and Technology. Birkhäuser Boston, 2008, pp. 313–322.

[2] Buck-Sorlin, G. H.—Kniemeyer, O.—Kurth, W.: Barley Morphology, Genetics and Hormonal Regulation of Internode Elongation Modelled by a Relational Growth Grammar. New Phytologist, Vol. 166, 2005, No. 3, pp. 859–867, doi: 10.1111/j.1469-8137.2005.01324.x.

[3] Deussen, O.: Computergenerierte Pflanzen. Technik und Design Digitaler Pflanzenwelten, Springer, Berlin, 2003 (in German), doi: 10.1007/978-3-642-55822-1.

[4] Deussen, O.—Hanrahan, P.—Lintermann, B.—Měch, R.—Pharr, M.—Prusinkiewicz, P.: Realistic Modeling and Rendering of Plant Ecosystems. Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98), ACM, New York, NY, USA, 1988, pp. 275–286.

[5] Du, Q.—Gunzburger, M.: Grid Generation and Optimization Based on Centroidal Voronoi Tessellations. Applied Mathematics and Computation, Vol. 133, 2002, No. 2-3, pp. 591–607, doi: 10.1016/s0096-3003(01)00260-0.

[6] Foley, J. D.—van Dam, A.—Feiner, S. K.—Hughes, J. F.: Computer Graphics. Principles and Practice. Addison-Wesley, Reading, Massachusetts, 1997.

[7] Greenworks: Xfrog Modelling Software. Web Site. Available on: `http://xfrog.com`, 2017.

[8] GroIMP Developer Group: Web Site. Available on: `http://www.grogra.de`, 2017.

[9] Hemmerling, R.—Kniemeyer, O.—Lanwert, D.—Kurth, W.—Buck-Sorlin, G. H.: The Rule-Based Language XL and the Modelling Environment GroIMP Illustrated with Simulated Tree Competition. Functional Plant Biology, Vol. 35, 2008, pp. 739–750, doi: 10.1071/fp08052.

[10] Henke, M.: Design and Implementation of a Modular System for 3D Visualization of Plants Using GroIMP Instantiation Rules – Entwurf und Implementierung eines Baukastens zur 3D-Pflanzenmodellierung in GroIMP. Diploma thesis (in German), BTU Cottbus, 2006. Available on: `http://www.uni-forst.gwdg.de/~wkurth/cb/html/henke_dipl_final.pdf`.

[11] Ijiri, T.—Owada, S.—Igarashi, T.: The Sketch L-System: Global Control of Tree Modeling Using Free-Form Strokes. Smart Graphics, 2006, pp. 138–146, doi: 10.1007/11795018_13.

[12] Kniemeyer, O.: Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling. Ph.D. thesis, BTU Cottbus, 2008. Available on: `http://opus.kobv.de/btu/volltexte/2009/593/`.

[13] Kniemeyer, O.—Barczik, G.—Hemmerling, R.—Kurth, W.: Relational Growth Grammars – A Parallel Graph Transformation Approach with Applications in Biology and Architecture. In: Schürr, A., Nagl, M., Zündorf, A. (Eds.): Applications of Graph Transformations with Industrial Relevance. Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 5088, 2008, pp. 152–167, doi: 10.1007/978-3-540-89020-1_12.

[14] Kurth, W.: Specification of Morphological Models with L-Systems and Relational Growth Grammars. Computational Visualistics and Picture Morphology (Themenheft zu IMAGE 5), Vol. 5, 2007, No. 1, pp. 50–79.

[15] Lau, D.—Arce, R. G.: Modern Digital Halftoning. Marcel Dekker, Inc., New York, Basel, 2001.

[16] Lintermann, B.—Deussen, O.: A Modelling Method and User Interface for Creating Plants. Computer Graphics Forum, Vol. 17, 1998, No. 1, pp. 73–82, doi: 10.1111/1467-8659.00216.

[17] Ong, Y.—Streit, K.—Henke, M.—Kurth, W.: An Approach to Multiscale Modelling with Graph Grammars. Annals of Botany, Vol. 114, 2014, No. 4, pp. 813–827.

[18] Prusinkiewicz, P.—Karwowski, R.—Měch, R.—Hanan, J.: L-Studio/cpfg: A Software System for Modeling Plants. In: Nagl, M., Schürr, A., Münch, M. (Eds.): Applications of Graph Transformations with Industrial Relevance. Springer Berlin

Heidelberg, Lecture Notes in Computer Science, Vol. 1779, 2000, pp. 457–464, doi: 10.1007/3-540-45104-8_38.

[19] Pirk, S.—Stava, O.—Kratt, J.—Said, M. A. M.—Neubert, B.—Měch, R.—Benes, B.—Deussen, O.: Plastic Trees: Interactive Self-Adapting Botanical Tree Models. ACM Transactions on Graphics, Vol. 31, 2012, No. 4, pp. 50:1–50:10.

[20] Prusinkiewicz, P.—Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer, New York, 1990, doi: 10.1007/978-1-4613-8476-2.

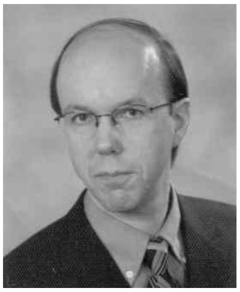
[21] Smoleňová, K.—Kurth, W.—Cournède, P.-H.: Parallel Graph Grammars with Instantiation Rules Allow Efficient Structural Factorization of Virtual Vegetation. Electronic Communications of the EASST, Vol. 61, 2013, p. 17.

[22] Ulichney, R.: Digital Halftoning. MIT Press, Cambridge, 1987.

[23] Xu, L.—Henke, M.—Zhu, J.—Kurth, W.—Buck-Sorlin, G. H.: A Functional-Structural Model of Rice Linking Quantitative Genetic Information with Morphological Development and Physiological Processes. Annals of Botany, Vol. 107, 2011, pp. 817–828, doi: 10.1093/aob/mcq264.

Michael Henke Diploma degree in computer science from the Brandenburg University of Technology Cottbus – 2007 Ph.D. student at the University of Göttingen, Department Ecoinformatics, Biometrics and Forest Growth – 2009–2010 Visiting scholar at Zhejiang University, Hangzhou, China – 2013 Researcher at French National Institute for Agricultural Research – 2014–2016 Researcher at Wageningen UR, The Netherlands – Assistant Lecturer at Brandenburg University of Technology Cottbus and University of Göttingen – GroIMP developer – research fields: functional-structural plant modelling, simulation.

Ole Kniemeyer 1997–2002 Studies of physics and mathematics at the University of Göttingen – 2002 Diploma degree in theoretical physics at the University of Göttingen – 2002–2006 Ph.D. student at the University of Technology in Cottbus – 2006–2008 Visiting researcher at the University of Göttingen, Institute of Forest Biometry and Informatics – 2008 Ph.D. in computer science at the University of Technology in Cottbus – since 2008 software developer at MAXON Computer GmbH.



Winfried Kurth Diploma degree in mathematics and Ph.D. in theoretical computer science at the University of Technology in Clausthal – subsequently junior researcher at the Universities of Göttingen and Bayreuth – 1992 research internship at CIRAD, Montpellier – venia legendi in forest biometrics and computer science obtained in 1999 at the University of Göttingen – 2000–2001 Heisenberg scholarship – 2001–2008 Professor for practical computer science/graphics systems at the University of Technology in Cottbus – since 2008 Professor for computer graphics and ecological informatics at the University of Göttingen – research fields: rule-based languages, representation of 3-d data, functional-structural plant models, simulation.