GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

A parametric functional-structural tree model based on graph rewriting

Tim Oberländer

Institute of Computer Science

Bachelor's Thesis
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

03. June 2020

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

Telefon +49 (551) 39-172000
fax +49 (551) 39-14403
Letter office@informatik.uni-goettingen.de
Mundus `www.informatik.uni-goettingen.de`

First Supervisor: Prof. Dr. Kurth
Second Supervisor: Prof. Dr. Damm

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 02. June 2020

Abstract

There are multiple approaches to simulate trees for different reasons. Most of these approaches are designed for specific species and can only be customized by changing the program code of the simulation itself. Therefore, a deeper knowledge of a programming language and computer science is often needed.

To find a way to simulate trees without this knowledge, the following work is using functional structural plant modeling to create a universal tree model which can be customized by parameters. The structural part is mostly based on the concept of tree architecture. The functional part is based on different theories: while the diameter is calculated by a pipe model, the length increase can be calculated either by statistical functions or by the use of a carbon balance model. This carbon balance model is based on a carbon tax extended concept of the autonomy of branches. The carbon tax is designed to simulate the needs of the trunk and the older branches.

All calculations are separated into three levels, tree level, shoot level and bud level. These levels can be found in the code as well. Each simulation step is handling a tree, shoot by shoot, and a shoot, bud by bud. This project is implemented on GroIMP a platform supporting a graph rewriting based generalization of L-systems.

The final model is capable of simulating European monopodial trees, based on 23 parameters. These parameters include several lambda functions. These lambda functions give the opportunity to create specific formulas for each species. Moreover, the module can simulate behavior such as light competition, tree forks and proventitious shoots.

This way a model is created that can be customized to simulate all European monopodial trees with minimal programming skills.

Contents

1 Introduction

Trees have fascinated humankind for a long time, as a source of timber, as part of ecosystems or as decoration. In all these cases it is interesting to understand and simulate how trees grow and spread. Historically, it was hard to grasp the complexity and diversity of trees, yet in the last century more and more models emerged [1]. With the rise of computer simulations, ways to visualize these models, such as L-Systems or functional structural tree modeling (FSPM), became possible [2] [3].

One of the botanic theories is crown architecture, which is based on the book "Tropical Trees and Forests - An Architectural Analysis" [4], in which tropical trees are categorized in 23 models. The analysis used visual observations of tropical trees to categorize these trees based on their stature and growth behavior. These categories are distinguished by for instance, the number of trunks, the position of the branches or the growth direction of the branches.

A computer science example for tree simulation is GroIMP (Growth Grammar-related Interactive Modelling Platform), a software, created by Ole Kniemeyer and Winfried Kurth. GroIMP uses parallel graph rewriting as a generalization of L-Systems, to simulate complex biological organisms or biotopes. Beside this core, GroIMP implements a list of functions for biological analyses and modeling. An example would be the reaction to global influences such as light and competition with other plants [5].

Nevertheless, most approaches to simulate plants still need an advanced knowledge in a programming language and in computer science. That is the point where this project aims to start. The main goal is to create a framework for a functional structural tree model that can be customized by parameters, without the need of programming. Beside the customisability and the realistic simulation, there are other aspects that must be considered. Especially the usability and the performance are important factors. These aspects are sometimes in conflict. A large amount of parameters can have a negative influence on the usability while it may be necessary for customisability. Moreover, realistic appearance and strong performance can be difficult to combine. This project tries to find a balance between these four aspects in order to create a framework for simulating multiple trees.

The following work is separated in five different parts. The first two sections are about the main theoretical aspects, GroIMP and the biological models. Based on these theoretical aspects the model will be designed and implemented in the fourth section. The parameterization, the modules and the used functions will be explained here. In the following section the modules will be used with different parameters to showcase the module based on the different theoretical concepts. The evaluation will be based on the aforementioned aspects and include a small outlook on how the project could be continued.

2 Notation

2.1 Shoot structure

This project separates branches in annual shoots. An annual shoot represents the part of a branch grown in one year. As illustrated in figure 1 each shoot is separated into inter-nodes, which are connected by nodes. A shoot has a bud at the top, which is called apical. Moreover, it has plenty smaller buds, called lateral buds, emerging at the nodes, where the leaves are growing.

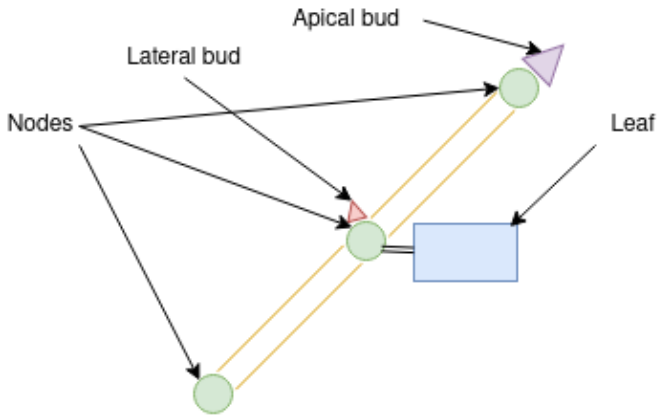

Figure 1: Simplified structure of a monopodial growing shoot. The shoot is separated in inter-nodes, which are connected by nodes. The buds and the leaves are connected to the nodes.

2.2 Monopodial growth

The focus of this work is on European monopodial trees, which have one stem and all branches can be categorized by their so called orders. As seen in figure 2 monopodial trees have one dominant apical shoot. In this way a mostly linear trunk is developing [6].

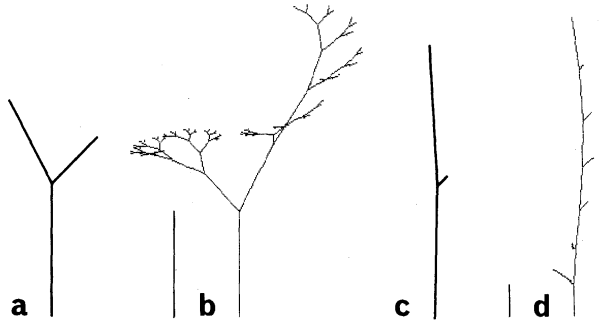

Figure 2: Simple illustration of the differences between sympodial growth (a,b) and monopodial growth (c,d) [6].

2.3 Branching order

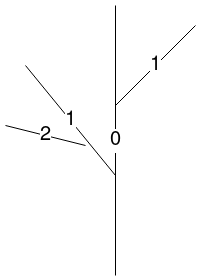

Figure 3: Simplified concept of a monopodial branching order. The trunk has order zero, the branches emerging from the trunk have order one and the branches emerging from these branches have order two.

This project has been designed for monopodial trees, therefore the definition of the branching order is quite simple. The trunk has order 0. A shoot growing out of a lateral bud has the order of the mother shoot plus 1 (Fig 3).

2.4 Leaves

As known from basic biology, leaves only grow on nodes and not on internodes. It can also be assumed that leaves are most likely to direct orthogonal to the sun. The last important factor considered in this project is the approach that buds are most likely to be in the axil of the leaf. Therefore, the position of a leaf is related to the position of the connected bud.
Based on the fact that this project does not contain a water model, the focus is on photosynthesis as the main function of the leaf.

3 GroIMP

GroIMP implements graph rewriting as a generalization of Lindenmayer systems (short L-Systems), therefore it handles every model as a graph. In this way, it is possible to analyze the model by using the graph as some kind of database. To achieve this, GroIMP comes with its own query language, which is a part of GroIMP's own programming language XL (eXtended L-systems). The model specification language XL extends Java as a high-level programming language, which enables full featured computer programs. XL implements, amongst other things, a syntax for L-systems. Due to the software architecture GroIMP can be extended by several plugins [5]. To work with GroIMP and XL it is useful to understand the basic concepts and theories behind the software.

3.1 Graph rewriting

In contrast to the well known string rewriting grammar, a graph rewriting grammar is able to handle complex structures [7]. A graph can be separated in nodes and edges. This means, each graph has an alphabet A consisting of labels of the nodes and edges, a collection of nodes G_n, a function to label the nodes G_λ and a set of edges G_e. Each edge contains a starting node, a destination node and an edge label. A simple example can be found in figure 4.

A graph rewriting grammar is defined as the combination of the initial graph (axiom) and a collection of productions. The productions are working quite similar to other rewriting grammars. The production

$$G \to H \tag{1}$$

is the rule to transfer the graph G to the graph H. [5]

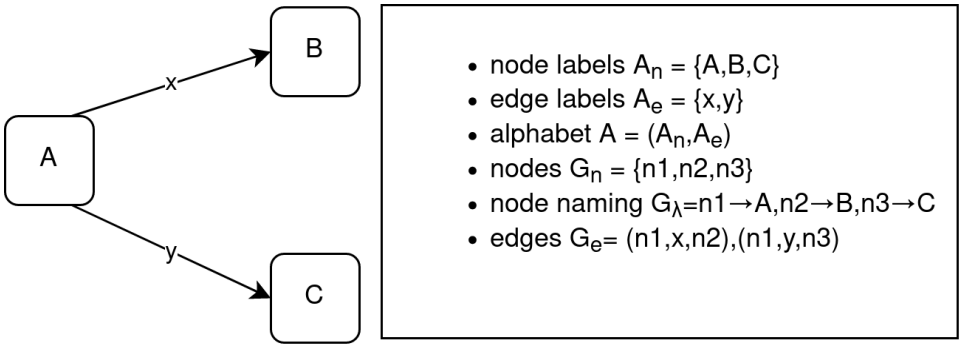

Figure 4: Example graph for graph rewriting with the defining sets and functions.

3.2 Lindenmayer-Systems

Lindenmayer-Systems (short L-systems) are formal grammars, which are defined as parallel string rewriting systems [8]. Similar to Chomsky-grammars,

L-systems contain an alphabet and production rules, as well as an axiom that defines the initial string.

The main difference between Lindenmayers work and the work of Chomsky is the handling of productions. While Chomsky-grammars use productions sequentially, in L-systems the productions are changing all letters of a word parallel [9]. This way they are more useful to simulate biological behavior [10]. An example from thwe book by Prusinkiewicz and Lindenmayer can be seen in figure 5.

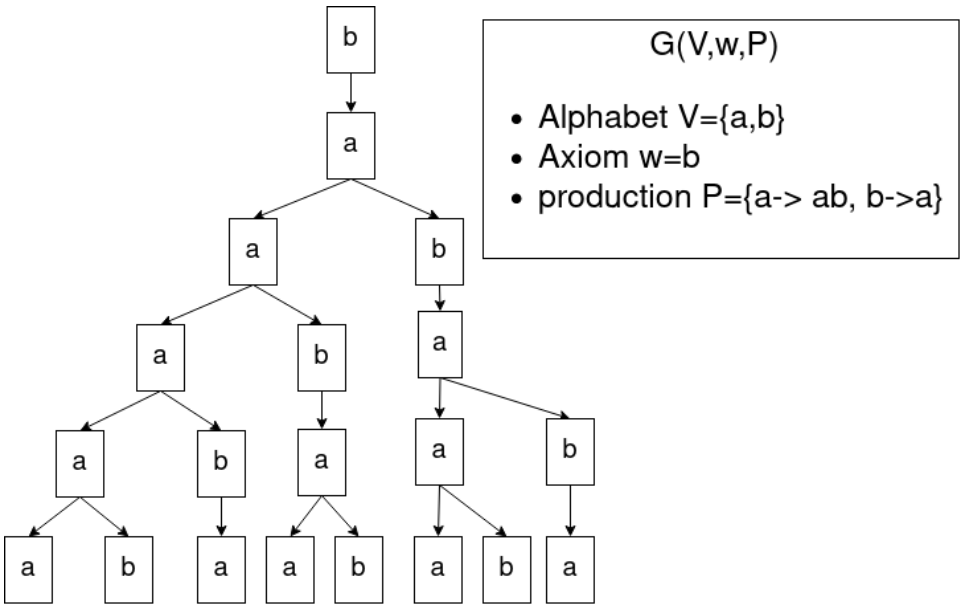

Figure 5: A redesigned example from "The algorithmic beauty of plants" [10] for L-system rewriting. Showing the change of the string by steps based on the grammar G including an alphabet (V), an axiom (w) and the production rules (P).

3.2.1 Types of L-Systems

There are different types of L-Systems, which are similar to the Chomsky hierarchy [8], yet the generated string languages are not equal [10]. It is possible to create a context free L-System that results in a language for which no context free Chomsky grammar exists. There are deterministic context free (DOL-Systems), context free (OL-Systems) and context sensitive L-Systems. Moreover, there are parametric L-Systems which are quite important to create more complex structures [10]. An other group of L-Systems are the **bracketed L-systems** these grammar using square brackets to display of more complex structures [11]. This group can be interpreted as specific types of graph rewriting grammars with only two types of edges, successors (white space) and branches (square brackets).

3.3 XL-Programming language

Each node of a graph represented within GroIMP can be understood as an instance of a module in the language XL. A module can include variables, functions and submodules, and it can be customized by parameters. A module can also extend other modules or even Java-classes. XL-code can be quite similar to the L-System syntax (Lst. 1).

```
1  module A();
2  module B();          // V = {a,b} node alphabet
3
4  protected void init(){
5   Axiom ==> B;        // w = b initial graph
6  }
7  public void grow(){         // P=(b->a),(a->ab)    grammar
8   B ==> A;
9   A ==> A B;
10 }
```

Listing 1: Simple XL code with similarity to L-Systems syntax, including the definition of the modules (alphabet) in line 1 and 2, the init function and a grow function to run a simple grammar.

3.3.1 Syntax

XL-code can be separated in rule based code, which is orientated on the L-System syntax and in imperative code, which is similar to java. Rule based code written in an imperative code block has to be put in square brackets while imperative code written in a rule based code block has to be written within curly brackets. The ground level of the code is imperative, yet it is possible that a function only contains rule based functions (compare Lst. 1).
There are three sorts of production rules in XL:

- "A ==> B" change A to B and connect B with the remaining host graph
- "A ==>> B" change A without new connections to B
- "A ::> $\{code\}$" trigger XL code whenever a node of type A is found.

On the left side a query is selecting the subgraph which will be transformed and on the right side the new subgraph is defined. A new graph can be designed by lining up modules, such as

$$[RL(90)F]F \tag{2}$$

In this case the edges are getting defined by either a white space (successor) or square brackets (branches).
Moreover, it is possible to use control flow constructions from Java such as for-loop or if/else-conditionals in rule based XL-code.

3.3.2 Visualization

Graph grammars only create mathematical graphs as output. Therefore, similar to other L-System software, GroIMP uses turtle geometry to create 3D objects based on these graphs. XL comes with a collection of predefined turtle commands, including visual objects, such as cylinders and spheres, and transforming commands, for instance move or rotate [12]. Furthermore, any defined module can be used as a turtle command.

3.3.3 Lambda expressions

A lambda expression, also known as anonymous function, is a function that is not addressed by name, but by pointer or reference. In consequence one is able to handle a lambda-function like a variable, including the possibility to use it as a parameter [13].

Besides Java's own lambda functions, XL comes with its own set of interfaces designed for this purpose. There are implementations for all common data type combinations in the style of: "IntToFloat()", "DoubleToDouble()" or "ObjectToFloat()". The syntax of these functions is always similar (Lst. 2).

```
1  static IntToFloat f = int x => float Math.pow(x,2.0);
```

Listing 2: Lambda code example IntToFloat. Implements the function $f(x) = x^2$.

The line of code in Listing 2 is similar to the following mathematical expression:

$$f(x) = x^2 \tag{3}$$

The range of Java functions that can be used in lambda expressions is similar to the functions used in a normal variable declaration. This also includes the conditional operator. The conditional operator is a binary operator that can be used quite similar to the if/else concept.

```
1  static IntToFloat f = int x => float x<9 ? Math.pow(x,2.0) : x ;
```

Listing 3: conditional operator example. Implements the equation 4.

The line of code in Listing 3 is equivalent to the following mathematical expression:

$$f(x) = \left\{ \begin{array}{ll} x^2 & : x < 9 \\ x & : else \end{array} \right\} \tag{4}$$

This project only uses the interfaces **ObjectToFloat, ObjectToInt and ObjectToBoolean**. With these interfaces it is possible to use complex data types as parameters. Therefore, one could for example use a whole bud model including all variables and functions to calculate the length of the new shoot.

It is necessary to mention that all lambda interfaces in XL are not extending the interface "**serializable**". Thus, if used outside of a function, they have to be static (compare Lst. 2, Lst. 3). If they are not static, the Java virtual machine inside GroIMP will break down.

3.3.4 Queries

Graphs inside of GroIMP can be used similar to a database. Therefore, XL has its own query-language, which can detect single nodes or groups of nodes [5]. This query-language can select nodes based on their module, the modules of nodes next to the node and the edges connecting the node with other nodes. In table 1 a collection of queries and their results are listed based on the graph in figure 6.

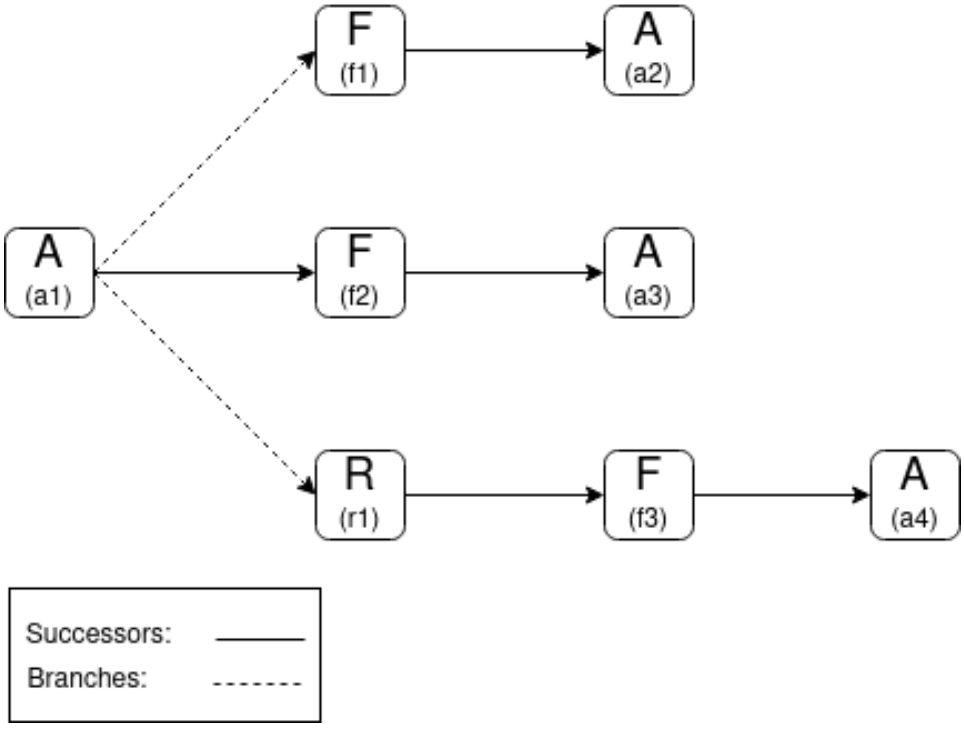

Figure 6: Simplified tree structure similar to the graph generated by the GroIMP code: Axiom ==> A[F A][R F A] F A ;.

Query	Result	Description
Basic		
A or a:A	a1,a2,a3,a4	addresses all instances of module A
a:A,a.len==1 or A(1)	all A with len==1	unary predicates
Node based path pattern		
(* A F *) A	a3	direct combination of nodes
(* F *) A	a2,a3,a4	other example
Edge based path pattern		
ANY_EDGE		
(* A −−> *) F	f1,f2	next node
A (*<−− F*)	a2,a3,a4	prev node
(* F −− *) A	a1,a2,a3,a4	undirected

SUCCESSOR_EDGE		
(* A > *) F	f2	next node
A (* < F *)	a2,a3,a4	prev node
(* F − − − *) A	a1,a2,a3,a4	undirected
BRANCH_EDGE		
(*A +>*) F	f1	next node
A (* <+ F *)		prev node
(*F -+- *) A	a1	undirected
Transitive closures		
(* A >> *) A	a3	Path described by single edges
(* A +> > *) A	a2	other example
(* A (−−>)* *) A	a1,a2,a3,a4	(0-to-n edges possible)
(* A (−−>)+ *) A	a2,a3,a4	(1-to-n edges possible)
(* A (−−>)? *) F	f1,f2	(0-to-1 edges possible)
(* A (−−>){2} *)F	f2	min 2 edges
(* A (−−>){1,2} *) A	a2,a3	min 1 max 2 edges
Combined		
f:F, A +> f, A < f	f1	patterns combined by comma

Table 1: List of basic query commands exemplified by the graph in figure 6 and their results.

Basic The simplest pattern is a module. This pattern selects all nodes which are instances of this module. This also works with generic classes of modules such as ShadedNull. Furthermore, it is possible to select nodes based on parameters, variables and unary predicates. For instance, a module X(int a, int b); could be used with x:X,x.a==1 && x.b==2 or X(1,2) or even X(1,) and X(,2).

Node based path A simple, but also less powerful, way to select nodes based on their position and connection. It can be used to find a node based on its parent nodes.

Edge based path Based on the concept of GroIMP there are different kinds of edges. In this project only successors and branches will be discussed. It is also possible to rename or even add types of edges.
The default notation for edges is:

- "−−>" all types of edges
- ">" successors (A B)
- "+>" branches (A[B])
- "/>" refinement (not used in this project)

Edges can be directed such as A −−> B or B <−− A, undirected A − − − B or even bidirected A <−−> B. The bidirected case is not used in this project.

Transitive closures All queries so far were addressing the immediate neighbourhood of a node, yet it is often necessary to find nodes based on a more flexible pattern.
Therefore, XL comes with the possibility to define repetitions of edges. Basically, a minimum amount of repetitions of edges or even a range between two values can be defined. The basic transitive closure is $(>)\{x, y\}$ where x is the lower limit and y the upper one. It is also possible to only define one. All corresponding expressions can be defined with this basic one, as seen in listing 4.

```
1 (>)*  := (>){0}
2 (>)+  := (>){1}
3 (>)?  := (>){0,1}
```

Listing 4: Definitions of advanced transitive closures based on the basic closure. line 1: Kleene star, line 2: Kleene plus, line 3: this or the next node.

Combined By working with complex graphs it is always possible that one query is not sufficient to find the right nodes. Therefore, it is possible to combine several queries. To do so, the queries are separated by commas, such as : "declaration, query1, query2, query3" (example Lst: 5). In this case the comma is interpreted as a logical "and".

```
1 b:B, X<+b, Y>b
```

Listing 5: Example for combined patterns, selecting a node b of the module B, in relation: Y B [X].

Usage The query language is used in two different parts of the XL language. On one hand it is needed for **production rules** to select the nodes on the left side of the production.
On the other hand it is used for **arithmetical expressions** to analyze the model. They can be used in calculations such as sum() or count(). A simple example is shown in listing 6. A search expression is a query written like this "(* query *)", and delivers a set of graphs.

```
1 float x =sum((* A +> (>)* F *).length);
```

Listing 6: simple arithmetical expression to sum up the combined length of all selected nodes.

3.4 Light Model

Beside the light used in 3D rendered images, GroIMP provides light calculation for analytical usage. Therefore, a radiation model is implemented using ray tracing to calculate the amount of light entering an object.

The idea is that each implemented light source sends out a number of rays, they follow one direction until they collide with a visual object (shown in figure 7).
In this way the amount of rays colliding with an object can be used to calculate the light absorbed by the object.

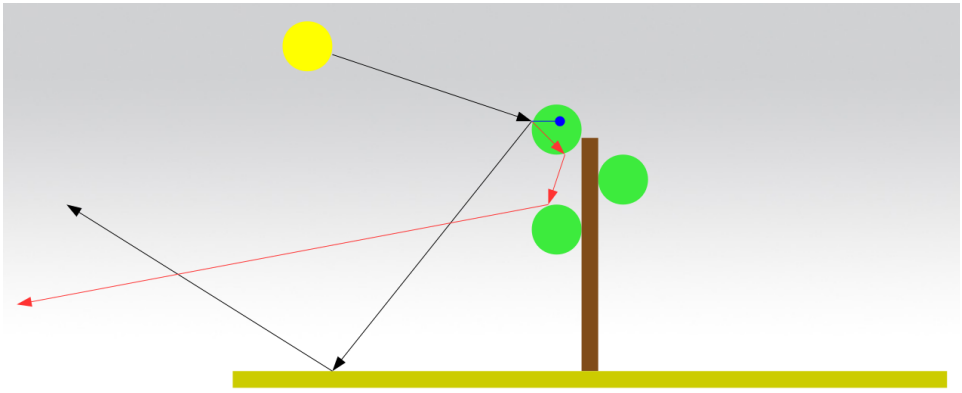

Figure 7: Ray tracing concept used by Ole Kniemeyer [14]. Simulating the light interception and the reflections.

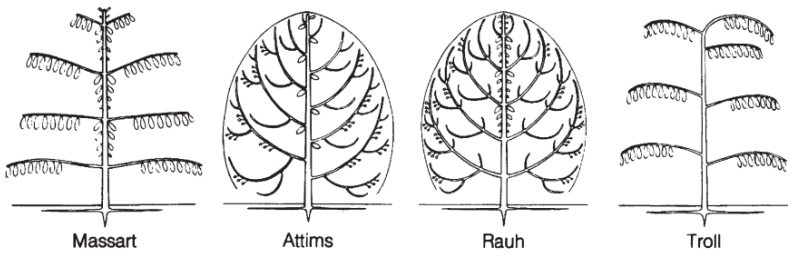

Figure 8: The crown architecture models used in this project, all of them are monopodial tree models with representatives in Europe. [15]

4 Biological models

There are several approaches to simulate and understand trees. In the following, the models used in this project will be explained. This project follows the concept of FSPM and separates the tree into two parts: A functional part and a structural part. The structural part is about the architecture and the shape of trees. The functional part is about the changes on the tree over time, for instance growth and mortality [1]. Both parts are not independent of each other, yet to understand the concepts, a separated view is useful.

4.1 Structural

The stature of a tree can mostly be described by the crown architecture, but additional models are needed to customize branches of higher orders. The orientation and bending of the branches needs to be approximated, and also the distribution of new lateral buds must be considered.

4.1.1 Crown architecture

The theory of crown architecture is mainly based on the aspects of the stature and the growth behavior, of the trunk and the crown of a tree. The original work categorizes tropical trees in 23 models [4]. The main criteria are the amount of trunks, the position and amount of branches of the first order and the growth direction of the first order. As seen in figure 9, trees where the branches of the first order are growing upwards are called orthotropic, while trees with horizontal growth in the first order are called plagiotropic.

Since these models were quite general, D.Bathra et al. [15] were able to categorize all European trees in 8 of these models. Due to the fact that this project is focused on monopodial trees, only 4 of them will be considered: Rauh, Massart, Troll and Attims.

All monopoidal European trees can be classified in these four models (seen in figure 10). Troll and Attims grow lateral buds in spirals around the trunk, while Rauh and Masssart are growing them oppositional. The branches of the first order by Massart and Troll are growing plagiotropic,

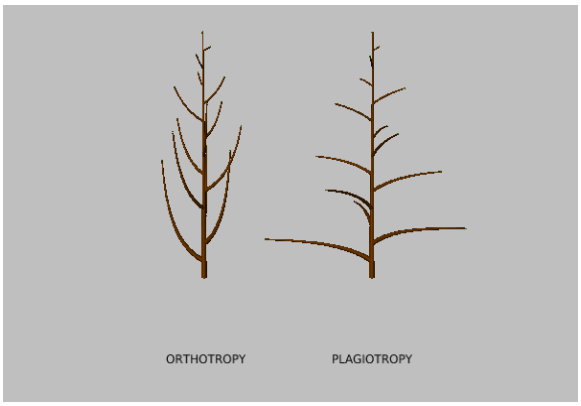

Figure 9: Simulation of growth directions in GroIMP, design by K. Smoleňová .[16]

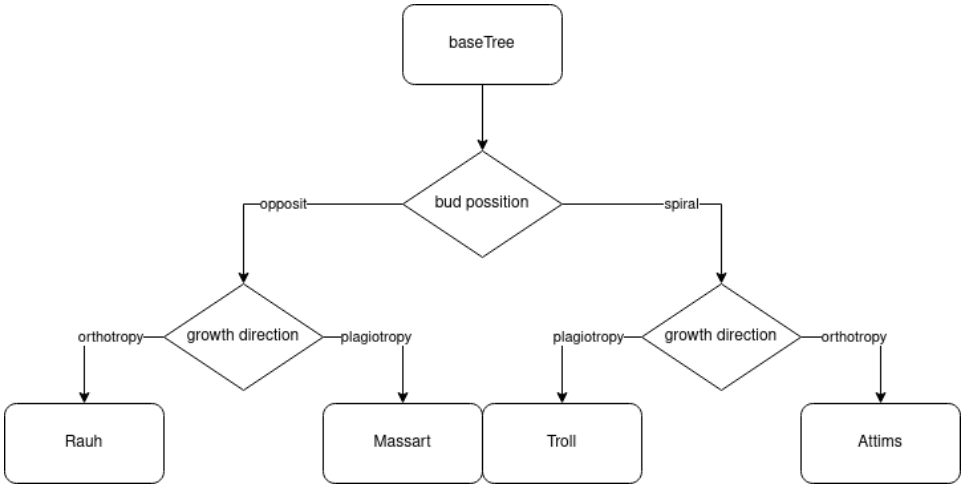

Figure 10: Simplified categorization of monopodial European trees based on the position of lateral buds on the trunk and on the orientation of the branches of the first order.

while by Rauh and Attims they are orientated orthotropically. There are several other specifications such as the position of the blossoms, or the position of the lateral branches on higher orders, which are not needed for this categorization.

Based on this categorization, the information needed to create a simple tree structure are the number of nodes on the stem in a year, the number of lateral branches on each node, the growing direction of the branch and the rotation on the head axis between the nodes of the trunk.

Branch orientation and bending The shape of a branch can be simulated by two factors, the apical orientation and the bending. The orientation is based on stimuli, such as light, and on the basic architectural concepts seen in figure 8 and figure 9.

The gravity based bending is mainly influenced by the length, the diameter and the hardness of the branch. In some cases, for instance *Salix babylonica*, the orientation is highly influenced by the species and the order of the branch.

4.1.2 Higher orders

The amount, position and orientation of lateral buds in the higher orders has a huge influence on the detailed appearance of a tree. There are two basic approaches to simulate the position of lateral buds, the discrete model and the continuous model. The discrete model considers the neighboring nodes and their relation and distance to each other, while the continuous model is placing buds along the shoot based on a probability density function.

This density function can be either randomized, in regular distance or the shoot can be separated into clusters in which the new buds are distributed [17]. In this project the continuous model is used with regular distances, because the amount of buds can be highly variable in the group of trees for which this model is designed. Therefore, the other concepts would be hard to customize and for trees with a large amount of nodes the performance would decrease significantly.

Short shoot The shoots of most trees can be categorized into long and short shoots to describe the different tasks they are fulfilling. In most cases short shoots are mainly used to hold leaves for photosynthesis. Often short shoots have their own physiology and morphology, including the length and the missing ability to have lateral buds [4].

Sleeping bud The greater part of trees grow so called secondary buds, meaning buds that grow next to the "normal" bud and keep sleeping until they are triggered [4]. This trigger can be caused by damage or can occur spontaneously. Yet there is evidence that the spontaneous growing of sleeping buds comes in waves [17]. These so called proventitious buds are used for inner tree recovery to build a more consistent foliage.

4.1.3 Tree forks

A tree fork is a phenomenon of a shoot turning into two shoots of the same order. Tree forks can be classified into 4 groups: temporary forks, recurrent forks, main forks and accidental forks [18].

Temporary forks can appear if trees grow in bad light conditions to get more light. If a shoot got enough light, it will become the dominant shoot, turn into the trunk of the tree and the fork will slowly straighten out.

On trees with **recurrent forks**, every shoot ends in a fork and in most cases one of the new shoots will become dominant and it continues the branch or trunk.

Main forks are appearing on the trunk of older trees. These forks are strong and they are often preceded by forks on the branches.

The last kind of forks are the **accidental forks**, these forks can appear if the apical shoot dies for some reason such as animals foraging.

4.2 Functional

The main changes on trees over time are the increases of length and diameter as well as the mortality. In this project two approaches will be considered, a statistical approach and a carbon balance approach.

4.2.1 Radius increase

Based on the fact that the water transported from the roots to the leaves can not be compressed, the infrastructure must be capable of holding the same amount of water at any place between the root and the leaf. The simplification of this theory is the pipe model, that assumes that every leaf is connected to the roots by a small pipe (Fig 11).

With this theory the radius of the branches and the stem is completely reliant on the amount of leaves [19]. Based on the fact that the cross section area of the new trunk is the sum of the cross section area of the old trunk and that of the new tubes, the following calculation can be used. Let R_n be the new radius, R the old radius, R_p the pipe radius and n the number of pipes:

$$R_n = \sqrt{n \cdot {R_p}^2 + R^2} \tag{5}$$

To initialize a new branch, the model is even simpler since there is no previous radius.

$$R_n = R_p \cdot \sqrt{n} \tag{6}$$

4.2.2 Mortality

An important part of the development of a tree is the mortality of branches, in this model three aspects are used to assume the death of a shoot. First

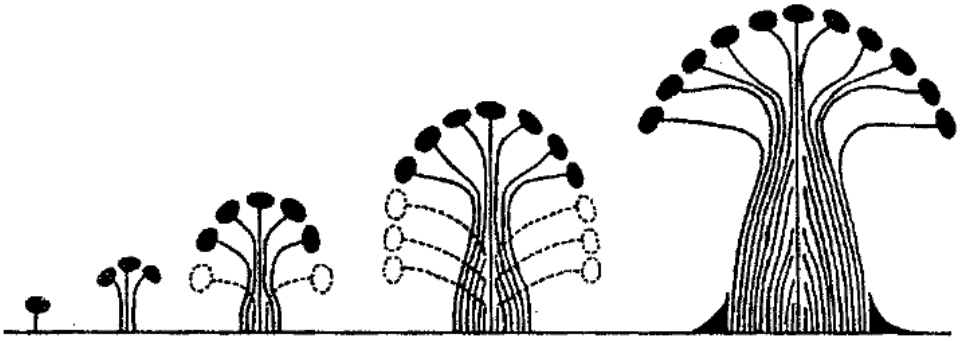

Figure 11: The original drawing of the pipe model from Shinozaki et al. [19]. The model shows the connection between the ground and the leaves and the diameter increase by growing trees. Moreover, it is shown that the tubes are turning into dead wood if the leaf dies.

it can be assumed that a shoot without leaves and child-shoots is dead. It also has to be mentioned that a new shoot with a length of zero is defined as dead and will not grow.

The last aspect is the indirect mortality, this is based on the definition that a shoot is declared dead if is is not sprouting. The therefore needed sprouting possibility can depend either on statistical approaches or the carbon budget.

4.2.3 Morphogenetic gradients

Beside the following approaches there are simple structural aspects influencing the length of a new shoot: the akrotony, the trend and the order. The **acrotony** is describing the phenomenon, that the length of a lateral shoot is depending on the position on the mother shoot. More exactly, acrotony means that a shoot is longer the closer it grows to the apical bud [17]. This phenomenon is visualized on the left side of figure 12.

The **trend** is influencing the length based on the idea that apical shoots are getting shorter depending on the distance between the bud and the mother branch or stem of lower order [17]. The influence of the **order** justifies the fact that the orders are the main aspect used to categorize branches in this structural model. The higher the branching order, the shorter the shoots.

4.2.4 Statistical approach

A common way to predict growth in simulations is the statistical approach. It requires the observation of the physical appearance over time and the concluding of mathematical models out of the results. In case of the height the model can be a growth function $f(t)$ depending on the age t of the tree, such as Gompertz or HossfeldIV [21] (Fig 13) whose parameters have to be fitted according to the measured data. The derivative of this curve $f'(t)$ can be interpreted as the height increase Δ_h. The increase is basically the length of the new apical shoot of the trunk.

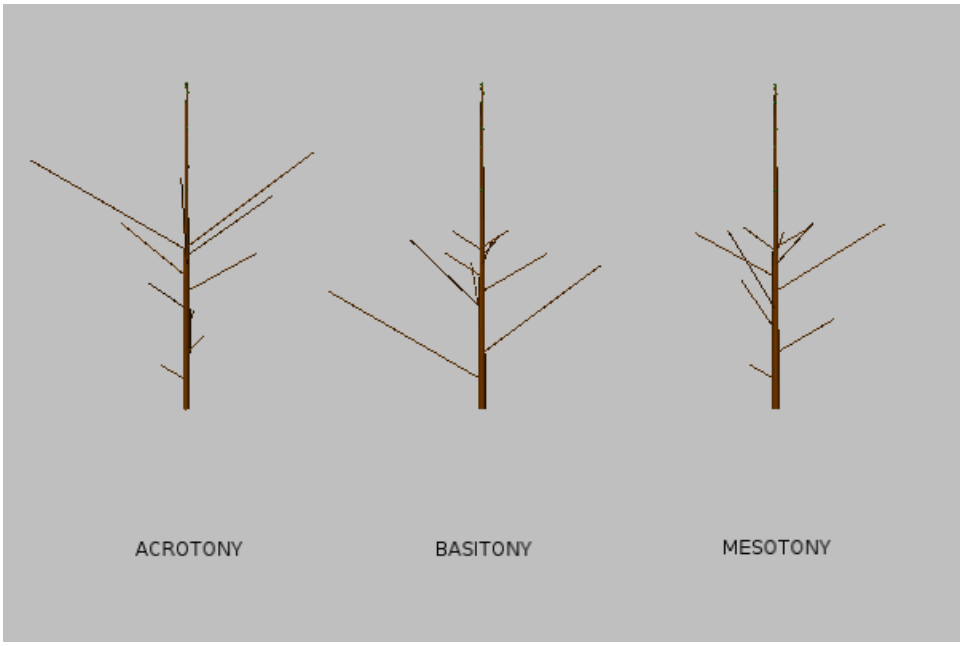

Figure 12: GroIMP simulation of the concepts of longitudinal symmetry. These symmetries are separated into acrotony, basitony and mesotony.[20]

Now we assume a connection $k_n(f'(t))$ between the height increase and the length increase on order n. This new function g_n can be used as a basic calculation of the length increase of the order n.

$$f(t) = h \tag{7}$$

$$f'(t) = \Delta_h \tag{8}$$

$$g_n(t) = k_n(f'(t)) \tag{9}$$

Still it will be necessary to modify the basic calculation based on the morphogenetic gradients and for instance some randomization.

To simulate **indirect mortality** in a statistical model, a sprouting possibility can be concluded, based on observation. This possibility is likely to respect general aspects for instance order or age.

name of the function	growth	increment
Hossfeld IV	$y = t^c/(b + t^c/a)$	$y' = bct^{c-1}/(b + t^c/a)^2$
Gompertz	$y = ae^{-be^{-ct}}$	$y' = abce^{-be^{-ct}}$
Logistisch	$y = a/(1 + ce^{-bt})$	$y' = abce^{-bt}/(1 + ce^{-b})^2$
Monomolekular	$y = a(1 - ce^{-bt})$	$y' = abce^{-bt^{-ct}}$
von Bertalanffy	$y = a(1 - e^{-bt})^3$	$y' = 3abe^{-bt}(1 - e^{-bt})^2$
Chapman-Richards	$y = a(1 - e^{-bt})^c$	$y' = abce^{-bt}(1 - e^{-bt})^{c-1}$
Levakovic III	$y = a(t^2/(b + t^2))^c$	$y' = 2bcy/t(b + t^2)$
Korf	$y = ae^{-bt^{-c}}$	$y' = abct^{-c-1}e^{-bt^{-c}}$

Figure 13: Table of growth functions and growth increase functions (translated) [21].

4.2.5 Carbon balance approach

This approach is, contrary to the statistical approach, less about the appearance itself than about the inner functioning of plants. The main idea is that there is a budget of resources for each tree. Since there is no root or ground model included in this project the important resource is carbon. The sprouting possibility and the length of the shoots is depending on this budget of resources [22]. This influence also triggers a kind of light competition where shoots grow towards the light source.

A simple way to handle this budget is to assume the **autonomy of branches**. This is often done in simulation to simplify calculations. On a biological level this model is justified with the fact that plants are less integrated than animals, this way they are able to handle stress such as a lack of carbon, an injury or an infection, on a branch level, without straining the whole tree [23]. Moreover, there are carbon based observations indicating the autonomy of branches [24].

Nevertheless, a full autonomy can not be assumed, since there is carbon needed to support the parts of a tree that cannot generate carbon, such as the trunk, the roots. Carbon is also needed to create a stack for the winter. This can be visualized by multiple carbon sinks of different strength, for the different carbon consumers [23]. Therefore, it can be assumed that one part of the budget of each shoot is given to the tree [22]. This part of the budget could be imagined as a **carbon tax**, this tax would depend on, for instance, the volume of the wood or the height of the tree. Even though root-systems are not a part of this project, it is possible to estimate the volume of the roots based on the volume of the branches [25].

5 Design

5.1 Data

The different calculations of the simulation need different variables and values. Therefore, the data needed for the calculations is important for the design of the code.

5.1.1 Scale levels

Based on the data they need, there are three groups of calculations in this project, tree calculations, shoot calculations and bud calculations. The **Tree calculations**, such as carbon-tax or base length, only need data from the tree. The next level of calculations are the **shoot calculations**. Based on structural and functional concepts, the annual shoot is an important organisational unit, for instance to simulate trend, order or carbon increase.

Nevertheless, most calculations only use shoot variables indirectly through the bud. Only the bending, the mortality and the diameter increase can be calculated with shoot data. The calculations needed to create new shoots are most likely to need bud specific information. Therefore, these calculations forming the **bud calculation** layer.

5.1.2 Connection

Due to the fact that any bud calculation could also need information from the parent shoot or the tree, a simple hierarchy comes to mind:

$$Tree\_Variables \subset Shoot\_Variables \subset Bud\_Variables \tag{10}$$

This fits perfectly with relations of the sub-graphs and modules in the graph structure that is representing the simulated tree:

$$Bud\_Module \subset Shoot\_Graph \subset Tree\_Graph \tag{11}$$

Based on these two concepts, the growth function has to start on the tree level, calculate the tree variables and then go through all shoots and all buds of each shoot.

5.1.3 Variables

The variables that can be used in calculations are listed in table 2, sorted by layers.

name	**description**
Tree level	
int age	The age of the tree.
float carbonTax	The carbon tax of the tree.
float baseLength	The basic length increase based on the statistical approach.

float carbonMean	The mean of carbon increase of all leaves.
float activeShoots	The number of shoots which have growing leaves.
float sumLeaves	The number of leaves on the whole tree.
float woodVolume	The volume of wood in branches and the stem.
Shoot level	
int age	The age of the shoot.
BasicTree tree	Pointer to the tree instance.
float len	The length of the shoot.
float diameter	The diameter of the shoot.
int nodeNumber	The number of nodes on the shoot.
int order	The order of the shoot.
int trend	The trend of the shoot.
float volume	The wood volume of the shoot
float carbon	The mean of carbon increase of all leaves growing on this shoot.
Bud level	
int order	The order of the bud.
int trend	The trend of the bud.
float position	The relative position of the bud on the shoot.
boolean sleep	The boolean value deciding if the bud is sleeping.
boolean shortShoot	The boolean value deciding if the bud will grow a short shoot.
BasicTree tree	Pointer to the tree instance.
Shoot getParentShoot()	This function is returning the shoot instance on which the bud grows.

Table 2: All variables that can be used in calculations sorted by layers.

5.2 Modules

The visual tree can be separated in wooden structures, leaves and buds. The wooden structures will be separated into annual shoots. This way the order, orientation, bending, diameter increase and autonomy of branches can be simulated in a realistic and simple way.

Nevertheless, the largest organizational unit of the model is the tree. The tree-module is the point where the simulation is interacting with the model, the place where tree wide variables are stored and the module the user can customize by parameters. Therefore, all other modules will be sub-modules of the tree module, all parameters will be given to the tree module and functions such as growth will be implemented here. All modules, their variables, and functions are illustrated in figure 14.

5.2.1 Shoot

In nature the annual shoot would be decomposed into inter-nodes and nodes where leaves and buds would grow. Yet there is no need for separation into several modules in this project. It is sufficient to create one cylinder and place the buds and leaves on the position where the node would be. The apical bud is connected to the shoot by an angle module, which handles

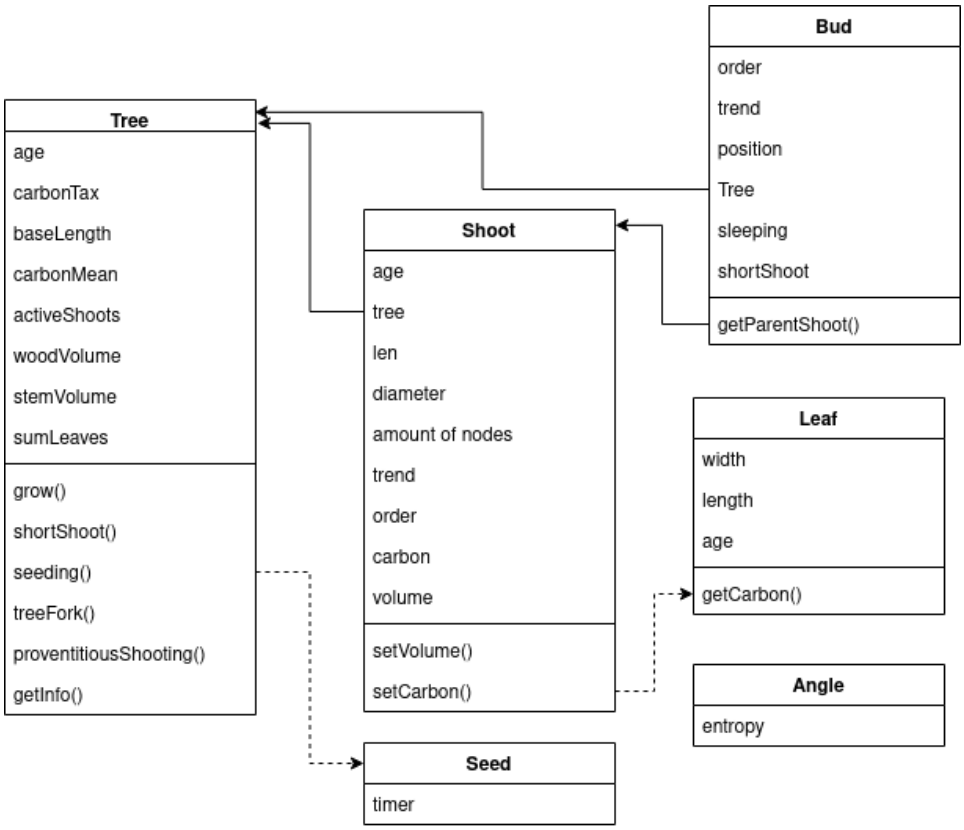

Figure 14: Module structure of the tree module and its sub-modules visualized as UML diagram. Each box is representing a module or submodule including the functions and variables, except the parameters of the tree module. The solid arrows are representing references of variables as well as return values. The dashed arrows are visualizing which modules are used in which functions, to keep clarity the grow function is excepted from the visualization of these usages.

the bending of the branch. This way a branch is represented by a chain of shoots connected by angle modules.
Moreover, the shoot is also used as an organizational unit, to calculate for instance carbon balance or diameter increase and handles all data needed in an autonomous branch, such as carbon, age, trend or order. The carbon increase is calculated by the sum of the carbon increases of the leaves of the shoot. The implementation of the module can be seen in listing 7.

```
1 module Shoot(int age, BasicTree tree, float len, float diameter,
      int nodeNumber,int order, int trend) extends Cylinder(len,
    diameter){...
```

Listing 7: The shoot module and its parameters: age of the shoot, pointer to the tree on which the shoot grows, length of the shoot, diameter of the shoot, number of the nodes on the shoot, order of the branch and trend of the shoot. The module is extending the Cylinder module to visualize the branch.

5.2.2 Bud

The bud is the initial point for a new shoot. Therefore, the module "Bud" has to include all variables possibly needed for the calculation of a new shoot. The bud-module also has to have the ability to sleep to simulate proventitious shoots. The bud module itself is mainly for storing data, this includes the function to get the parent shoot, using the graph based back end of GroIMP. The implementation of the module can be seen in listing 8.

```
1 module Bud(int order,int trend,float position,boolean sleep,
    BasicTree tree, boolean shortShoot){...
```

Listing 8: The bud module and its parameters: order of the bud, trend of the bud, relative position of the bud on the shoot, boolean value deciding if a bud is sleeping or not, pointer to the tree on which the bud grows and boolean value deciding if short shoot.

5.2.3 Leaf

Leaves are highly simplified in this model. They are represented by green squares of equal size which are orientated towards the top. The main function of a leaf is to calculate the amount of radiation that hits it. Therefore the getCarbon() function is using the LightModel provided by GroIMP to calculate the maximum increase of radiation, and the predefined maxLightSquare variable to get the carbon increase. The maxLightSquare variable is depending on the sky module and represents the amount of radiation shining on a 1x1 parallelogram which lays flat on the ground. The maxLightSquare value is needed to calculate a relative carbon increase, independent from the size of the leaf. The implementation of the module can be seen in listing 9.

```
1 module Leaf(float l, float w, int age) extends Parallelogram(l,
    w){...
```

Listing 9: The leaf module and its parameters: length and width of the leaf and the age on which the leaf will die. The module is extending the parallelogram module to visualize the leaf.

5.2.4 Seed

The seed is, similar to nature, used to place the tree and initialize it. The seed includes a timer variable with which it is possible to let different trees grow at different times. The implementation of the module can be seen in listing 10.

```
1 module Seed(int timer) extends Sphere(0.1);
```

Listing 10: The Seed module is extending the Sphere module to visualize the seed as a small dot. The time parameter is used to keep the seed waiting the specific amount of grow steps before it grows.

5.2.5 Angle

The angle module is only an alias for the RV module. The RV module is a rotation module which simulates gravitropism. In contrast to other rotation modules RV is only setting the strength of the influence of gravity, not an angle in degrees. The implementation of the module can be seen in listing 11.

```
1 module Angle(float tropism) extends RV(tropism);
```

Listing 11: The Angle module is extending the RV module with tropism as a parameter to declare the intensity of gravity based orientation. With a negative value it is orientating towards the top.

5.3 Parameters

The main goal of this project is to create a model that can be customized nearly without coding, by using parameters. These parameters will be given to the tree module and can be handled like tree wide variables.

5.3.1 Background

Growth calculations Based on the pipe model, the diameter increase is depending on the pipe size, which has to be modifiable since the number of leaves can be different.

Due to the two different approaches to calculate the length increase, there are calculations needed for the statistical length increase (base length) as well as for the carbon balance approach (mainly the carbon tax). In both cases, a lambda function is used to handle the formulas as parameters. To calculate the specific length of each order, an array of lambda functions is used.

Bud positioning The position of the bud is depending on the position of the node on which the bud grows, as well as the position on the node. To distribute the nodes, the continuous model [17] with regular distances is used. Therefore, only the number of nodes and the length of the shoot are needed.

The positioning of buds on a node is separated in trunk and branches. On the trunk only the number of buds is needed which are getting evenly

distributed around the stem. On the branches each bud has a rotation around the nodes. These rotations are stored in a float array. Also, to enable spiral branching a rotation of the head axes between the nodes is implemented.

Moreover, there are two values needed to set the possibility of buds growing on a node and of a specific bud growth. On this way a shoot can appear more realistic and can have more leaves than buds. If one of these possibilities is not true the bud will be declared sleeping and can be used for a proventitious shoot.

Leaf definition A leaf is simulated by a parallelogram, therefore the length and the width are needed. Also, the age at which the leaf will die has to be declared.

Branch bending The orientation of the branches is simulated by a rotation module between the shoots. This rotation module can be customized by two lambda functions, the orientation to set the direction of growth, and the bending to customize the change over time.

Mortality To simulate the mortality of the branches, a lambda function is used to decide if the shoot is sprouting. Moreover, a branch with no leaves and no child branches will die and disappear.

Specific growth behaviors The more specific growth behaviors can be implemented with the combination of a lambda function to trigger them and a function to run the growth. This function can be overwritten if needed. This concept will be used for: tree fork, short shoot and proventitious shoots.

5.3.2 Specification

The parameters in table 3 are also used in the project in the same order.

name	**type**	**description**
maxOrder	int	maximal order for a branch
baseLengthLambda	ObjectToFloat<Tree>	calculation of the basic length increase in cm
carbonTaxLambda	ObjectToFloat<Tree>	calculation of the carbon tax
specificLengthLambda	ObjectToFloat<Bud>[ ]	array of order specific length increase calculations in cm
numNodeLambda	ObjectToInt<Bud>[ ]	array of order specific calculations of the number of nodes
sproutingPossibilityLambda	ObjectToBoolean<Shoot>	logical expression to check if the shoot is sprouting
proventitiuosShootLambda	ObjectToBoolean<Bud>	logical expression to decide if a bud "wakes up"

apicalSplitLambda	ObjectToBoolean<Bud>	logical expression to decide if a branch splits into a tree fork
shortShootLambda	ObjectToBoolean<Bud>	logical expression to decide if a bud grows a short shoot
possibilityNode_0	float	If a random number is smaller than the value the node on order 0 grows buds. The value has to be between 0 and 1.
possibilityNode_x	float	If a random number is smaller than the value the node on higher orders grows buds. The value has to be between 0 and 1.
possibilityBud_0	float	If a random number is smaller than the value the bud of order 0 grows. The value has to be between 0 and 1.
possibilityBud_y	float	If a random number is smaller than the value the bud of higher orders grows. The value has to be between 0 and 1.
numberNewShoots_0	float[ ]	distribution array to calculate the number of buds per node on the stem
shootPoss	float[ ]	head-rotation angle in degrees of each new bud on a node of a higher order shoot.
shootangle	float[ ]	order based array of the axial angle in degrees
leafL	float	length of a leaf in cm
leafW	float	width of a leaf in cm
leafDieAge	int	the age a leaf dies
pipeSize	float	size of a single pipe in the pipe model (in cm)
orientationLambda	ObjectToFloat<Shoot>	Calculation for the orientation of an apical shoot. The returned value is an entropy.
bendingLambda	ObjectToFloat<shoot>	calculation of the bending of branches by determining angles over time. The returned value is added to the previous entropy.
offset_0	float	head-rotation in degrees between two nodes of the trunk
offset_x	float	head-rotation in degrees between two nodes of branches

Table 3: Collection of all parameters needed to configure a tree.

5.4 Functions

5.4.1 Grow

The grow function is a static function that only has to be started once for all trees in the simulation. The function goes through all trees in an organized way to minimize the number of calculations and simulate the semi-autonomy of branches in the tree.

To implement the connection of the data-layers it iterates shoot by shoot through the tree and bud by bud through the shoots (compare figure 15). This way is supported by the graph based backend of GroIMP. On tree level the carbon tax and the base length are getting calculated, these values are available for all other calculations on shoot and bud level.

For each selected shoot the first step is to calculate the carbon increase, this value can be used for the outlast-lambda and any bud calculations. If the shoot sprouted in the last year and the sprouting possibility is true, it will start to iterate over all buds. On bud level, the first step is to initialize the new shoot based on the lambda calculations and parameters. The main part of the growth function is about the development of new shoots. As described before the growth of the new shoot is different on trunk and branch level. As seen in figure 16 the main concept in both cases is that the new shoot is drawn, then the turtle returns to the initial point, and moves step by step to all the nodes. On each node the algorithm tests if the bud will produce a shoot based on the dependencies and then will distribute the buds over the shoot.

Afterwards other shoot related calculations are getting done, including the diameter increase, the bending and testing for proventitious shoots. The last step is to remove the old leaves.

5.4.2 Specific growth

The proventitiousShooting and shortShoot function, are getting a bud module as parameter. The apicalSplit function needs the length, the radius and the number of nodes which the shoot would have without the split. The seeding function only needs a seed as a parameter. They can all be customized by simply overwriting them.

5.4.3 Information

The getInfo() function can be used to get tree specific information as a string.

5.5 Sky model

Since the original sky model of GroIMP is not ready to use yet, this project has its own basic sky-modules. Nevertheless, it is important that it will be simple to change the module if the original one is ready to use. To change the light model the maxLightsquare variable in base.rgg has to be replaced.

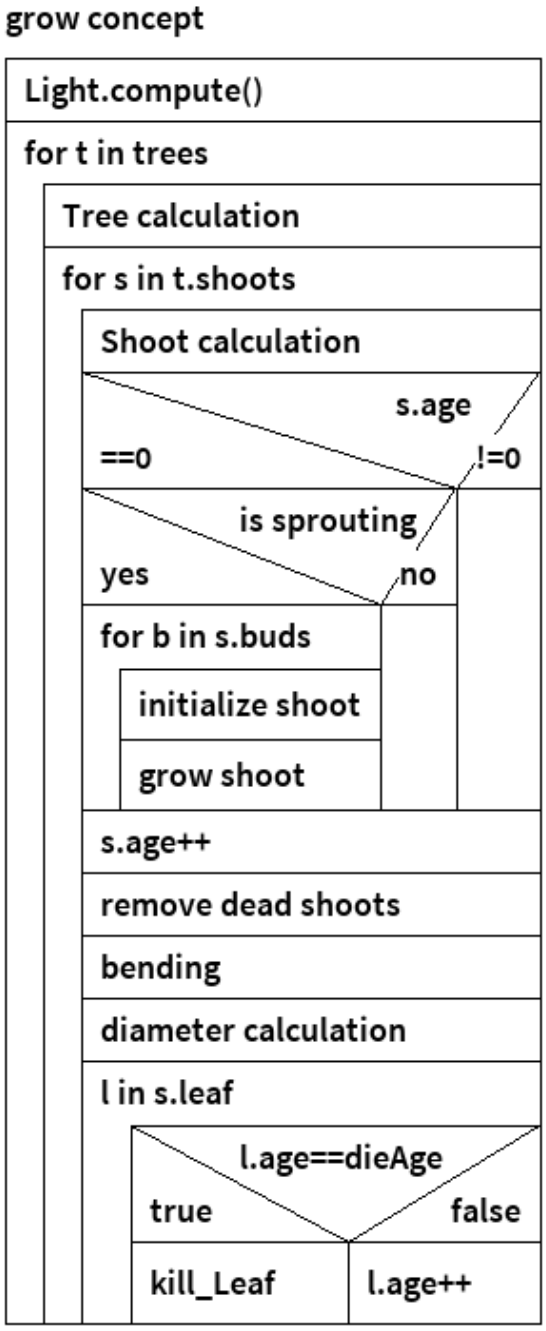

Figure 15: Nassi–Shneiderman diagram of the growth function.

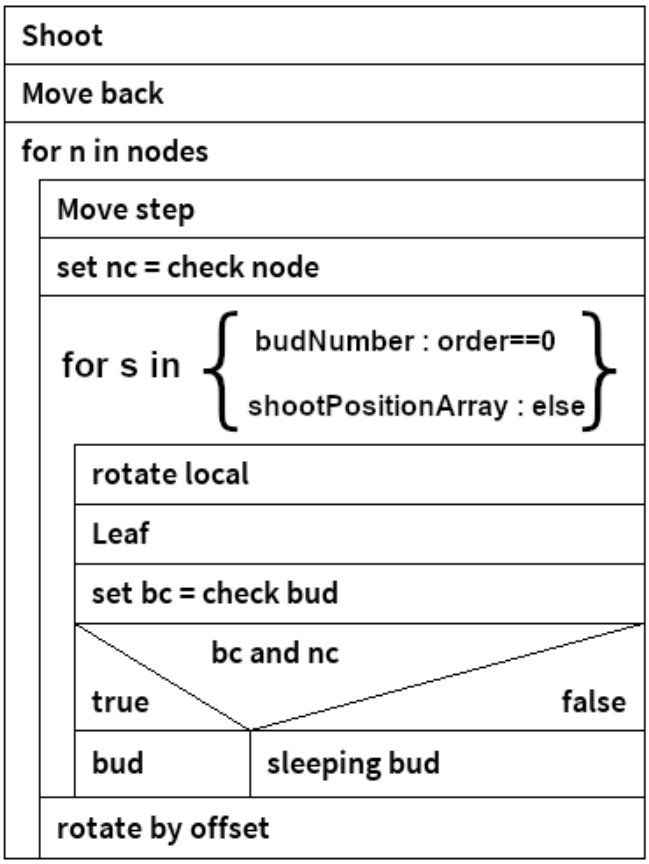

Figure 16: Nassi–Shneiderman diagrams of the development of a new annual shoot. The growth of the trunk and the higher orders differs in the way new buds are placed.

To calibrate the maxLightsquare variable a leaf of size 1m x 1m has to be placed flat on the center of the simulation and the resulting amount of radiation is the new maxLightsquare value.

5.5.1 Light dome

One way to simulate light is to create a dome of directional lights. This way a sky can be simulated quite well. A light dome is focused on one specific point, all directional light sources are placed in the same direction in different angles around this specific point. The light sources in the middle are slightly stronger then the ones on the "horizon".

Yet to use this model on multiple trees, which are growing in competition, can cause the problem that the light is not equally distributed.

5.5.2 Light field

Based on the problems with the light dome, the light field is a more useful approach for multiple trees. The light field is similarly to the light dome a constellation of directional lights. Contrary to the light dome in the light field all lights are organized in a grid and have the same distance to the ground.

6 Usage and examples

In the following, the major results can be seen in several examples. All of these examples can be found in the "Example.gsz" file.

6.1 Basic usage

To model a tree based on this module, there are some basic steps to follow. First it is recommended to create a new rgg file in a copy of "PFSTM.gsz", This way it is less likely to mess up the different lambda-functions. In this new file the "base.rgg" has to be imported. The next step is to create the new tree module (example in listing 12) which extends BasicTree. Now all parameters seen in table 3 must be defined, except for modules which are planned to be sub frameworks. In this case the undefined parameters have to be parameters of the new model.

```
1 module Base_Curve() extends Base.BasicTree(\\parameters);
```

Listing 12: Basic setup example. The phrase parameters is representing the needed parameters.

The different lambda functions have to be set as static variables outside the module similar to the example seen in listing 13.

```
1 static ObjectToBoolean sproutingPoss= Base.BasicTree.Shoot s =>
      boolean s.order==0 ? true : Math.random()>0.1*s.trend-0.05*s
     .tree.age;
```

Listing 13: Example sprouting possible lambda expression. This expression means that the trunk will always sprout and a branch is less likely to sprout the greater the distance to the mother shoot gets. Moreover, shoots of older trees are more likely to sprout.

If it is needed to overwrite one of the specific growth functions (an example can be found in the seeding example in listing 14), it can just be written behind the baseTree module. It is recommended to orientate on the original functions in "base.rgg".

6.2 Architecture

The first and most important step of the structural part of designing a tree using this project is the architecture. The basic architecture models, like the ones seen in figure 8, can be simulated by the parameters seen in table 4. Given these, the trees in figure 17 can be generated.

	Massart	**Attims**	**Rauh**	**Troll**
numNodeLambda[0]	Bud b => int 2	Bud b => int 4	Bud b => int 2	Bud b => int 4
numberNewShoots_0	{0, 0, 0, 0, 0, 0.95 ,0.05}	{0, 0.95, 0.05}	{0, 0, 0, 0, 0, 0.95, 0.05}	{0, 0.95, 0.05}
shootangle	{80,70,50}	{60,40,50}	{60,40,50}	{80,70,50}
shootPos	{0,180}	{-30,-150}	{-30,-150}	{0,180}
offset	75	75	75	75
orientationLambda	Bud b => float -0.01	Bud b => float -0.4	Bud b => float -0.4	Bud b => float -0.01

Table 4: Specification of the four basic tree architecture models.

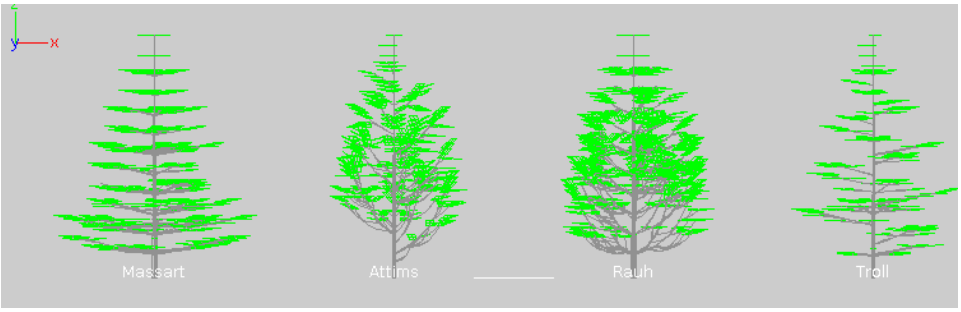

Figure 17: Simulated basic tree architecure models based on the values in table 4. From left to right: Massart, Attims, Rauh and Troll.

The opposite branching, seen in figure 18 and figure 20 can be simulated by a smaller amount of nodes and more new shoots on each node. Contrary to this the spiral branching, seen in figure 19 and figure 21 can be simulated by more nodes, less new shoots on each node and the usage of the offset factor for head rotation.

The orthotropy, seen in Attims (Fig. 19) and Rauh (Fig. 20) is simulated by a negative orientation, while the plagiotopy is simulated by a orientation value close to zero, this behavior can be seen in the simulation of Massart (Fig. 18) and Troll (Fig. 21). Moreover, the position of the lateral buds on the first order can be seen simulated in different ways.

6.3 Growth

6.3.1 Morphological gradients

Beside the statistical and carbon based growth models used in this project, the morphological gradients are important to shape the crown in its horizontal extension. The different parts of the morphological factors of length increase can be included into the calculations in several ways. The provided parameters are the length of the parent shoot l_p, the relative position of the bud p which is 1 for apical buds, the trend δ and the order o. A simple way to use this parameter could be

$$l_m = l_p \cdot p - \delta \cdot x, \tag{12}$$

with x as a factor of the intensity of the trend impact. In this calculation the order is only included in an indirect way. With higher order, the influence of the position is more intense. Simplified if a shoot of order n would grow on a position p and all its ancestor shoots had grown on the exact same position p on their mother shoot, all positions can be defined as p. This way the influence of the order is seen clearly in this formula:

$$\begin{aligned} l_n &= l_{n-1} \cdot p \\ l_n &= l_0 \cdot p^n \end{aligned} \tag{13}$$

Figure 18: Simple simulation of the crown architecture model Massart.

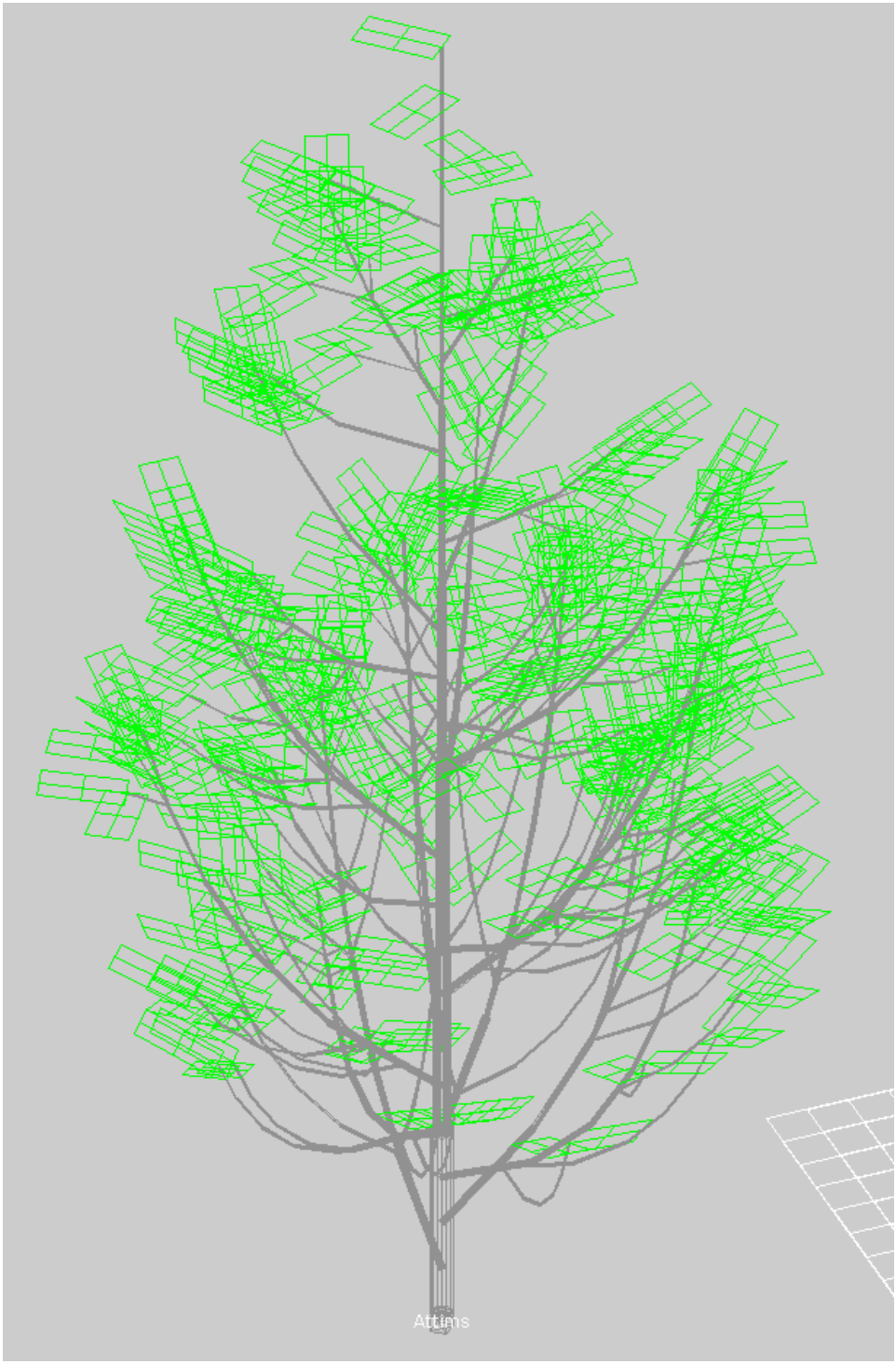

Figure 19: Simple simulation of the crown architecture model Attims.

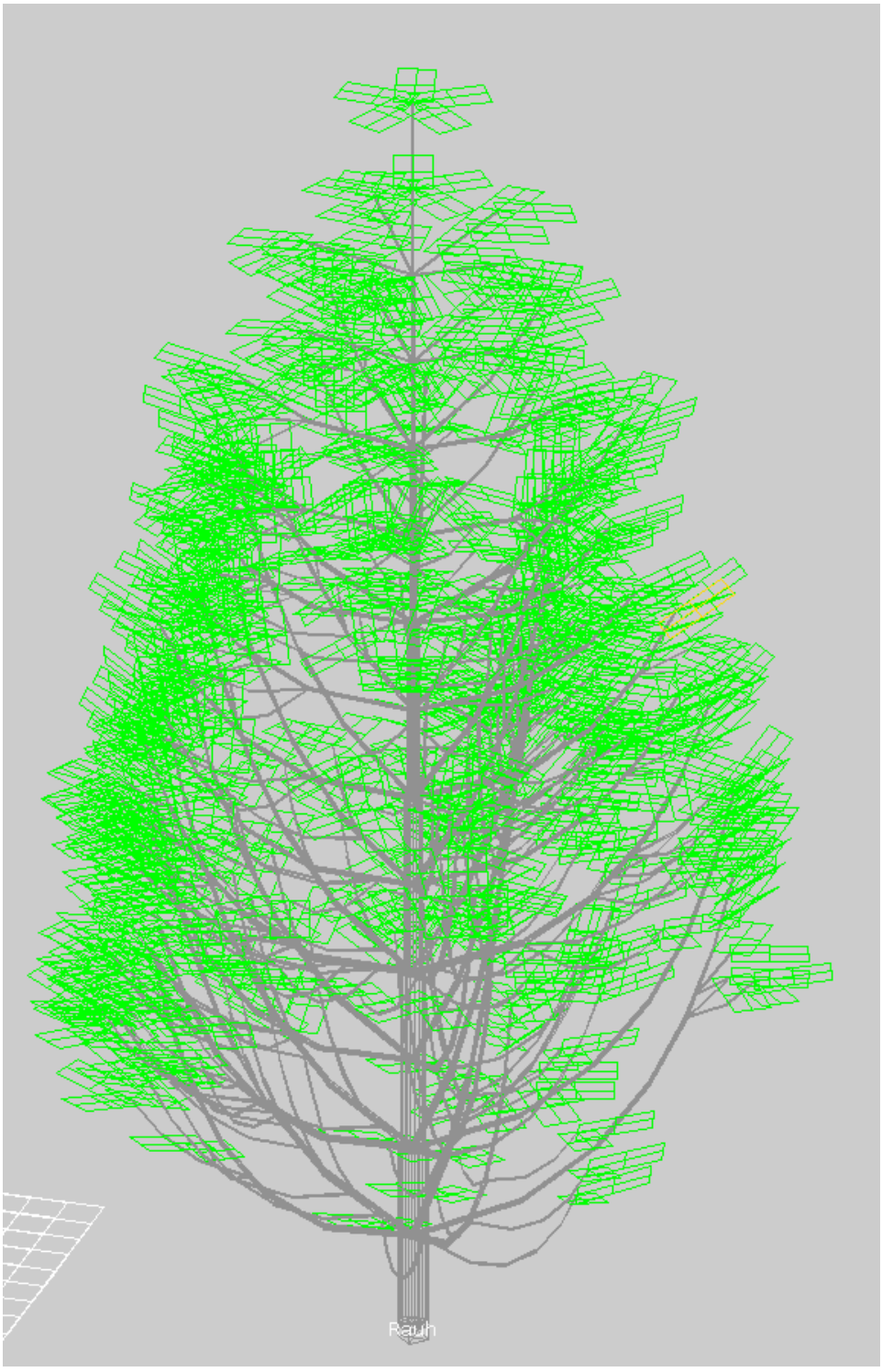

Figure 20: Simple simulation of the crown architecture model Rauh.

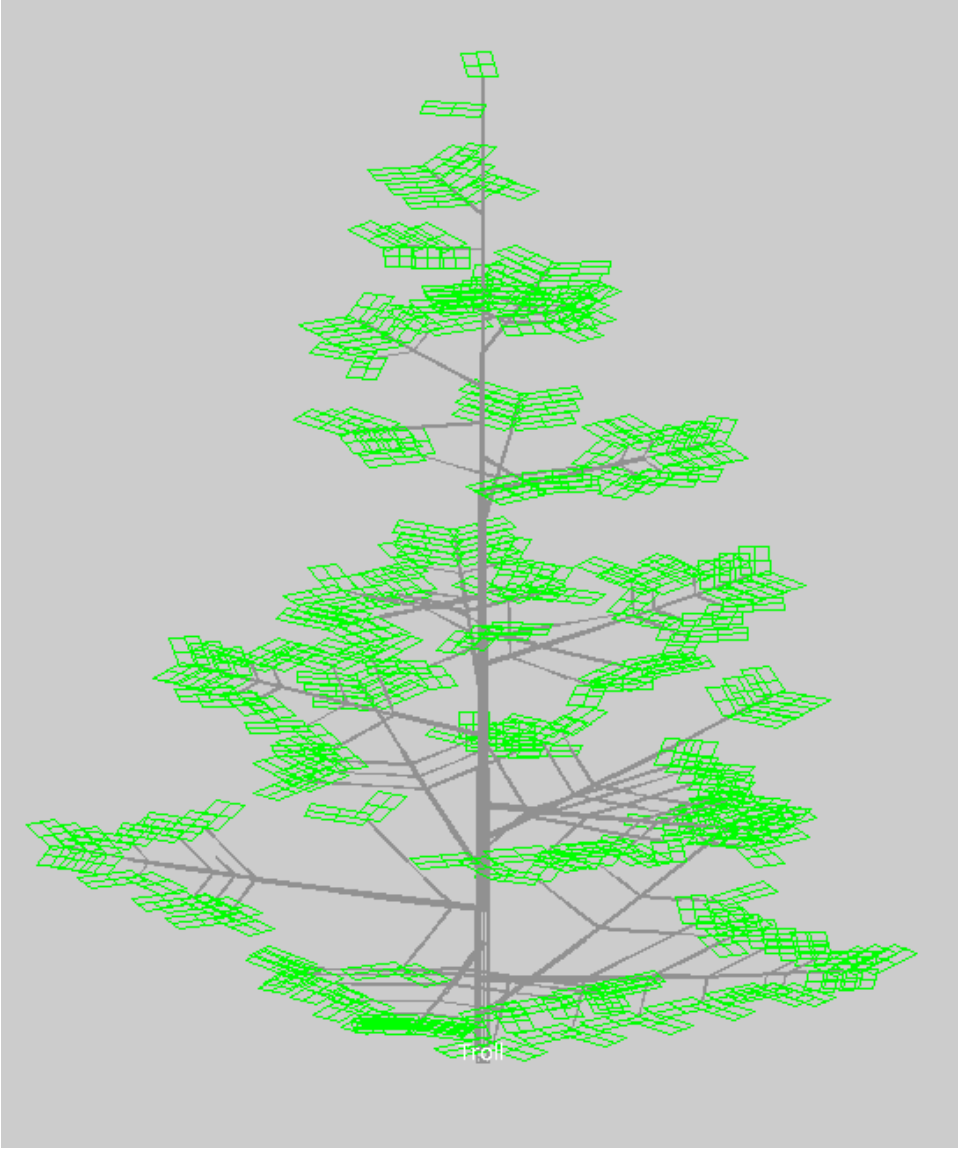

Figure 21: Simple simulation of the crown architecture model Troll.

6.3.2 Statistical growth

To show the concept of statistical growth, the growth function of Yoshimoto et al. [26] will be used to simulate a *Cryptomeria*. The used function is a Richards function:

$$h(t) = 24.9506 \cdot (1 - e^{-0.06438 \cdot t})^{1.9736} \tag{14}$$

Yet in this project the derivative of this function is needed to simulate the basic length increase of the first order:

$$h'(t) = 3.17023 \cdot e^{-0.06438 \cdot} \cdot (1 - e^{-0.06438 \cdot t})^{0.9736} \tag{15}$$

Since this example is mainly about using height curves, the calculation of the length increase of the trunk will be set equal to the base length l_b, which is the result of $h'(t)$:

$$l_0 = l_b \tag{16}$$

The length increase of the first order is calculated by a combination of morphological factors and the base length. The morphological factors are represented by the length of the parent shoot l_p times the relative position p of the bud on the mother shoot. Both increases are getting balanced by the factors ϵ and ω.

$$l_1 = \epsilon \cdot l_p \cdot p + \omega \cdot l_b \tag{17}$$

The second order length is calculated only depending on the morphological factors. The calculation is done the same way as in the first order.

$$l_2 = l_p \cdot p \tag{18}$$

The result of these calculations, used together with some architectural specifications, can be seen in figure 22.

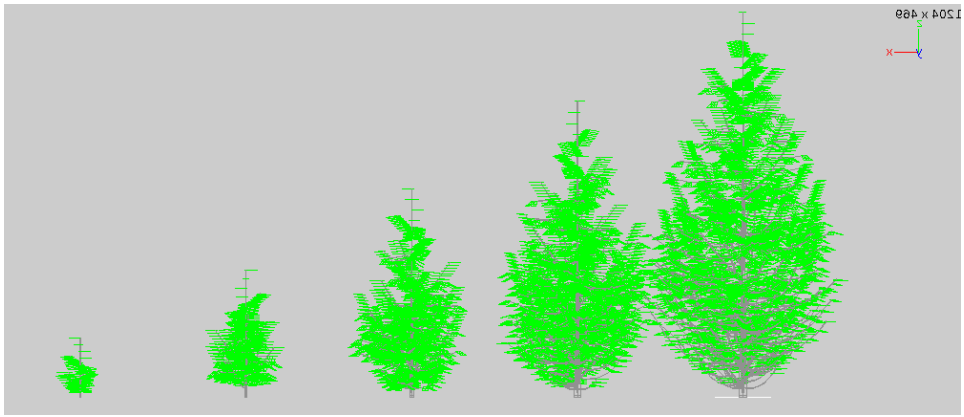

Figure 22: Statistically based simulation of the growth of *Cryptomeria*: age from left to right 4, 6, 8, 10, 12 years.

6.3.3 Carbon balance

The main part of the carbon balance approach is the carbon tax calculation. One of the main uses of the carbon tax is the maintaining of wooden structures. Therefore, the carbon tax c_{tax} can be assessed based on the volume of wood v_w of the tree. Based on the fact that in older trees a part α of the trunk is dead wood, it is also necessary to consider the trunk volume v_t. Now the important question is how much wood has to be supported by one active shoot. Therefore, the number of shoots n is needed. Furthermore, a parameter β is needed to describe how much carbon is needed to maintain one part of wood.

$$c_{tax} = \frac{v_w - (\alpha \cdot v_t)}{1+n} \cdot \beta \tag{19}$$

Based on the fact that the carbon budget c is a relative value, the simplest way to calculate the length increase of the trunk is to define a maximal length increase of order zero $l_{max;0}$:

$$l_0 = c \cdot l_{max;0} \tag{20}$$

To calculate the length increase of the higher orders the morphological aspects are considered. Therefore the carbon based length is calculated the same way as for the trunk and a morphological length is added. These two values have to be combined therefore the carbon based length is multiplied with the factor ϕ and the morphological length with the factor ξ.

$$\begin{aligned} l_1 &= \phi \cdot (c \cdot l_{max;1}) + \xi \cdot (l_p \cdot p - \delta \cdot x) \\ l_2 &= \phi \cdot (c \cdot l_{max;2}) + \xi \cdot (l_p \cdot p - \delta \cdot x) \end{aligned} \tag{21}$$

The result of these calculations, used together with some architectural specifications, can be seen in figure 23.

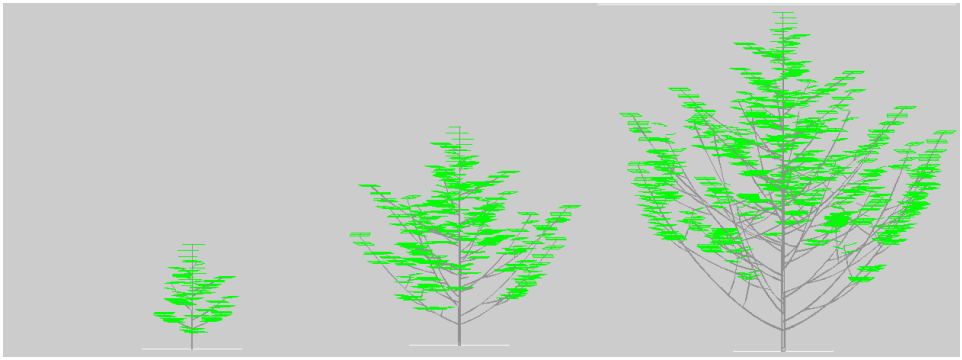

Figure 23: Carbon balance based growth after 5, 10 and 15 years.

To simulate light competition in carbon balance models, the sprouting possibility s can be resolved by a minimal amount of carbon. In many cases it is suitable to use a conditional operator to prefer apical shoots or the trunk.

$$s = \left\{ \begin{array}{ll} c > 0.05 & o == 0 \\ c > 0.2 & else \end{array} \right\} \tag{22}$$

This results in a behavior seen in figure 24.

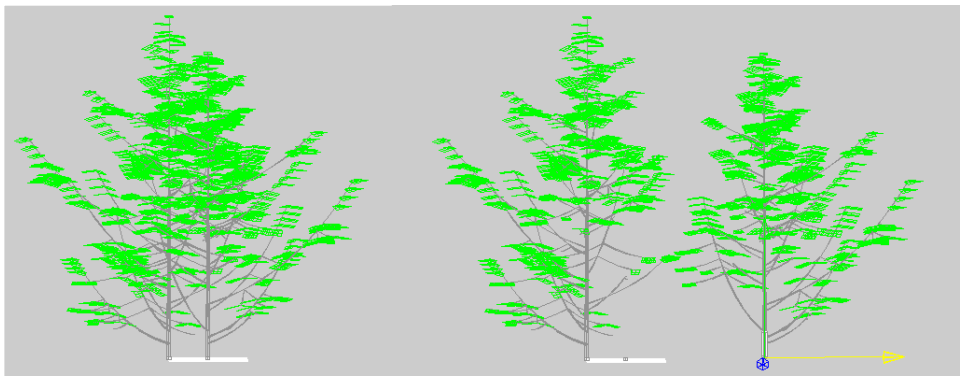

Figure 24: Light competition of two close growing trees. Left: Trees while growing, right: Trees pulled apart to show the influence of the competition.

6.3.4 Combination

It has to be mentioned that there is always the possibility to combine these two approaches. For instance it is possible to simulate the growth of the tree based on statistical functions and to use the carbon budget to handle the mortality. Moreover, it could be imaginable, to only apply the statistical approach to the basic growth, while the carbon based approach is added to simulate carbon competition.

6.4 Specific growth

6.4.1 Proventitious shoot

One way to "wake" a sleeping bud is to use a probability, which can be specified by different factors. For instance there are morphological and statistical factors such as the trend, the order or the age that can affect proventitious shooting.

An advanced way can be to use the carbon tax as an indicator of stress. A limit can be set to trigger once the carbon tax is above this limit. In that case the tree would need more carbon and is more likely to "wake" sleeping buds. Thus it is possible to control the carbon tax as shown in figure 25, moreover the foliage is then more realistic (compare Fig. 26).

6.4.2 Seeding

To simulate different growth behavior out of the seed, the seeding function can be overwritten. With the new seeding function in listing 14 multiple trunks can be simulated (Fig: 27).

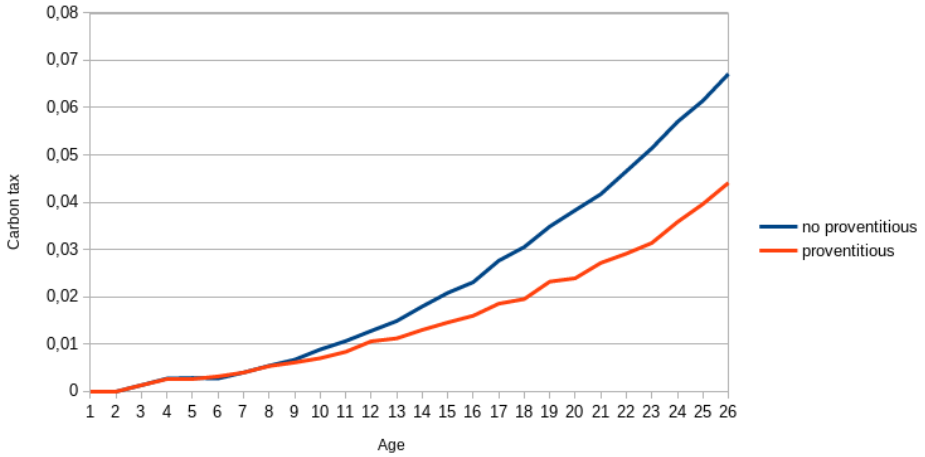

Figure 25: A diagram showing the influence of carbon tax based proventivious shoots on the example trees seen in figure 26. The proventitious shoots start to grow if the carbon tax is rising, increasing carbon availability.

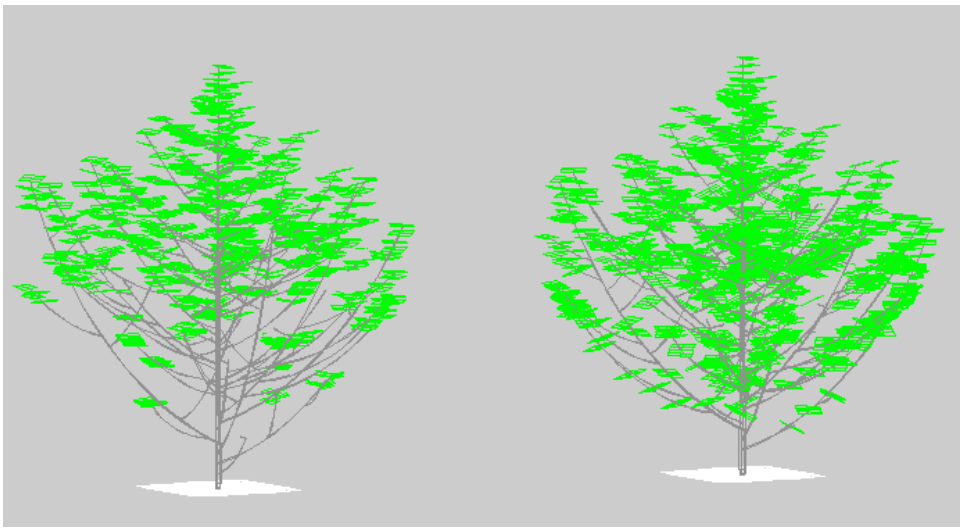

Figure 26: Two trees one with carbon triggered proventitious shoots (right) and one without (left). This shows the influence of proventitious shoots on the foliage.

```
1  void seeding(Seed s)[
2   s,(s.timer!=0)==>Seed(s.timer-1);
3   s==>
4    for(int i=0; i<5; i++)(
5     RH(60)
6     {
7      //new Shoot(int trend, float len, float diameter, int
           numNode, int order, int trend)
8      Shoot sh = new Shoot(0,this,Math.random()*5,Math.sqrt(2*
           Math.pow(this.pipeSize,2))*4,2,0,0);
9     }
10    [
11     RL(30)
12     sh
13     M(-sh.len/2)
14     [
15      //new Leaf(float width, float lenght, int age)
16      RV(-2)RL(90)RU(0)Leaf(this.leafL,this.leafW,0)
17     ]
18     [
19      RV(-2)RL(-90)Leaf(this.leafL,this.leafW,0)
20     ]
21     M(sh.len/2)
22     //new Bud(int order, int trend, float position, boolean
           sleep, BasicTree tree, boolean shortShoot)
23     Bud(0,0,1,false,this,false)
24
25    ]
26   );
27  ]
```

Listing 14: A seeding function to create 5 trunks of random length growing out of one seed.

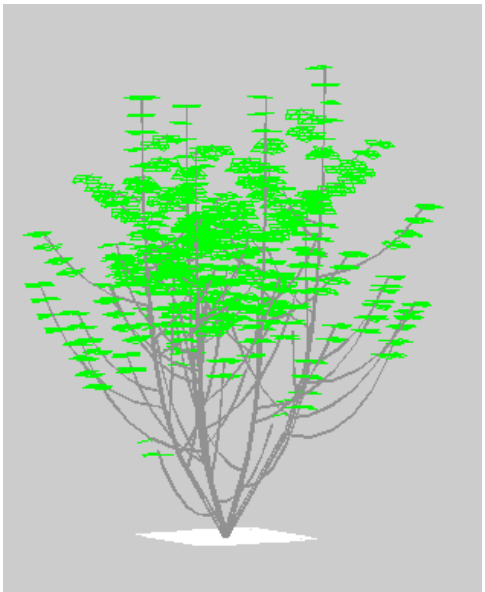

Figure 27: Example of a multi stem tree based on an overwritten seeding function (compare Lst. 14).

6.4.3 Tree fork

To simulate a tree fork the treeForkLambda expression and the treeFork function can be customized. A main fork as seen in figure 28, can be designed by a simple expression based on the age of the tree or the trend of the trunk.

Beside the main fork a simple way to simulate a temporary fork can be to predict the light situation with the average amount of carbon. In this way trees in a bad position are using tree forks to find a better way to get light (Fig. 29).

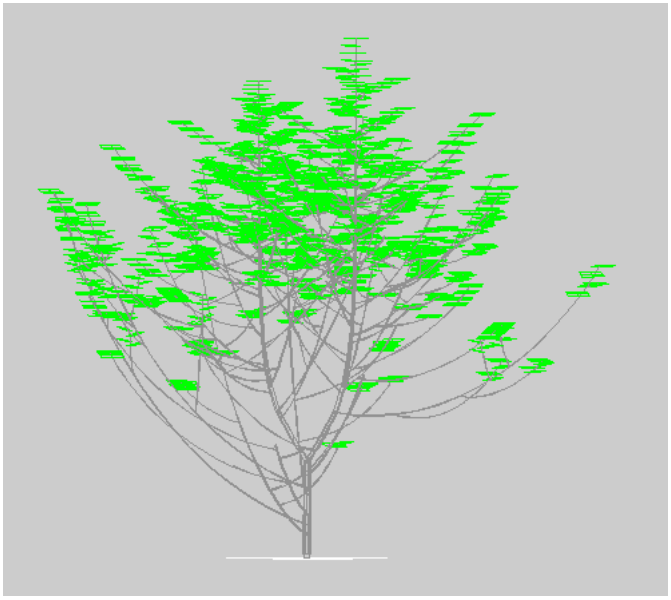

Figure 28: A simple example for a tree fork where the forking is triggered by the trend of the trunk.

Figure 29: A younger tree growing in the shadow of a higher tree. The younger tree "used" tree forks to find a better position for light.

7 Discussion

7.1 Evaluation

Looking at the results of my project, based on the four aspects from the introduction: customizability, performance, realistic growth and usability, I can say that most things worked out quite well.

The performance is comparable to other GroIMP projects which is good, considering the scope of the module. With the exception of the leaves the appearance is also satisfying. The architectural aspects can be found easily in the simulations and the growth behavior is simulating the theoretical concepts quite well. I believe that most users can work with the model without advanced knowledge in Java and XL. I was able to simulate nearly every behavior seen in monopodial European trees, yet for some of the more specific phenomena a bit more programming is needed. In addition it has to be said that there are still several species specific phenomena that can not be simulated such as sylleptic shoots, polycyclism or the separation in preformed and neoformed shoots.

In summary I can say that I believe that I was able to achieve all goals of this thesis project and I hope that others can use this project.

7.2 Outlook

There are several perspectives to further develop this project, the following is only a small selection of ideas I found interesting to consider.

7.2.1 Other architecture models

Next to the four architecture models on which this project is focused, there are 19 other models [4]. Therefore, it would be interesting to extend this project to simulate these models as well.

For the monopodial models, that are not respected in this project, it should be possible to simulate them without changes on the model, simply via changing the parameters.

The other models, for instance the other four European models, can also already be simulated by overwriting the treeFork function to simulate non monopodial growth behavior. Nevertheless, it has to be mentioned that the treeFork function was not designed to be used in that way.

To properly simulate all tree architecture models some changes would be necessary. To modulate a more curvy appearance of models, such as Champagnat, the separation of shoots in inter-nodes could be considered. Moreover, to simulate more complex shoots the grow function could be separated into several small functions. This way the sprouting of the shoot could be customized more specifically by overwriting a specific part of the grow function.

7.2.2 Leaf

As said in the evaluation the main inaccuracy of the simulation are the leaves. To have a more realistic model the size and the shape of the leaf could be improved. The size could further depend on more aspects such as position, order or the length of the mother shoot. Moreover it could be considered to use the leaf3d module. This was not implemented in this project since the focus of this work was the architecture and the growth behavior.

7.2.3 Root system

Based on the pipe model it would be possible to approximate the volume of roots. Moreover, this could be combined with different approaches of root architecture. To simulate the growth behavior of roots, a ground model would be needed.

7.2.4 Seed distribution

A simple extension would be to consider that a tree spreads seeds under certain conditions. These conditions could be for instance the age, the size or the carbon availability. Then an amount of seed could be placed at random in a specific area around the tree. This way the development of a stand could be simulated over several generations of trees.

7.2.5 Graphical user interface

An achievement to approve the usability would be to create a graphical user interface (GUI) to parameterize the model.

This GUI could be implemented directly as a GroIMP plugin and would enable users to generate specific tree models without using any code. This would create an even lower threshold for tree simulation and would make it possible to create trees faster. Moreover, it would be an idea to implement an import and export function so the parameters could be stored in a separate file.

List of Figures

List of Tables

Listings

References

[1] J. Vos, J. Evers, G. Buck-Sorlin, B. Andrieu, M. Chelle, and P. De Visser. Functional–structural plant modelling: a new versatile tool in crop science. *Journal of Experimental Botany*, 61(8):2101–2115, 2010.

[2] P. De Reffye, T. Fourcaud, F. Blaise, D. Barthélémy, F. Houllier, et al. A functional model of tree growth and tree architecture. *Silva Fennica*, 31:297–311, 1997.

[3] M. Henke, W. Kurth, and G. Buck-Sorlin. FSPM-P: Towards a general functional-structural plant model for robust and comprehensive model development. *Frontiers of Computer Science*, 10(6):1103–1117, 2016.

[4] F. Hallé, R. Oldeman, and P. Tomlinson. *Tropical trees and forests: an architectural analysis*. Springer Science & Business Media, 1978.

[5] O. Kniemeyer. *Design and implementation of a graph grammar based language for functional-structural plant modelling*. PhD thesis, Brandenburg University of Technology, Cottbus, Germany, 2008.

[6] S. Sakai. Sympodial and monopodial branching in Acer (Aceraceae): Evolutionary trend and ecological implications. *Plant Systematics and Evolution*, 171(1-4):187–197, 1990.

[7] R. Echahed and A. Maignan. Parallel graph rewriting with overlapping rules. *arXiv preprint arXiv:1701.06790*, 2017.

[8] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems*. Academic Press, 1980.

[9] G. Ochoa. An introduction to Lindenmayer systems. *web site:* `http://csweb.wooster.edu/dbyrnes/cs220/lectures/pdffiles/LSystems.pdf`, Accessed: 28.05.2020.

[10] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. New York: Springer-Verlag, 1990.

[11] J. Togelius, N. Shaker, and J. Dormans. Grammars and L-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*. Springer, 2016.

[12] GroIMP. Turtle Kommandos in der Sprache XL. *web site:* `http://134.76.18.36/wordpress/wp-content/uploads/2018/02/xl_turtle.pdf`, Accessed: 28.05.2020.

[13] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in Java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

[14] O. Kniemeyer. Extensions to the light model of GroIMP. *web site:* *http://ufgb966.forst.uni-goettingen.de/GroIMP/doc/Ole%20Kniemeyer%20-%20Lightmodel.pdf*, Accessed: 28.05.2020.

[15] D. Bartha. Architekturmodelle und Typen von Gehölzen des Pannonischen Raumes. *Acta Botanica Hungarica*, 53(3):215, 2011.

[16] K. Smoleňová. Simulation of growth directions. *web site:* *http://wwwuser.gwdg.de/~groimp/grogra.de/gallery/Plants/growthDirection.html*, Accessed: 28.05.2020.

[17] W. Kurth. Die Simulation der Baumarchitektur mit Wachstumsgrammatiken. *Wissenschaftlicher Verlag Berlin*, 1999.

[18] C. Drénou. Pruning trees: the problem of forks. *Journal of Arboriculture*, 26(5):264–269, 2000.

[19] K. Shinozaki, K. Yoda, K. Hozumi, and T. Kira. A quantitative analysis of plant form the pipe model theory: I. basic analyses. *Japanese Journal of ecology*, 14(3):97–105, 1964.

[20] K. Smoleňová. Longitudinal symmetry. *web site:* *http://wwwuser.gwdg.de/~groimp/grogra.de/gallery/Plants/longSymmetry.html*, Accessed: 28.05.2020.

[21] H. Pretzsch. *Grundlagen der Waldwachstumsforschung.* Springer-Verlag, 2019.

[22] R. List, M. Küppers, and F. Schröder. Madeira ein Programm zur Simulation des Wachstums (Biomasse und Architektur) von Holzgewaechsen. *Verhandlungen der Gesellschaft für Ökologie*, 23:39–48, 1994.

[23] DG. Sprugel, TM. Hinckley, and W. Schaap. The theory and practice of branch autonomy. *Annual Review of Ecology and Systematics*, 22(1):309–334, 1991.

[24] A. Lacointe, E. Deleens, T. Ameglio, B. Saint-Joanis, C. Lelarge, M. Vandame, GC. Song, and F. Daudet. Testing the branch autonomy theory: a 13C/14C double-labelling experiment on differentially shaded branches. *Plant, Cell & Environment*, 27(9):1159–1168, 2004.

[25] A. Mäkelä. A carbon balance model of growth and self-pruning in trees based on structural relationships. *Forest Science*, 43(1):7–24, 1997.

[26] A. Yoshimoto and R. Marušák. Evaluation of carbon sequestration and thinning regimes within the optimization framework for forest stand management. *European Journal of Forest Research*, 126(2):315–329, 2007.