FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Algorithmische Botanik durch Lindenmayer Systeme in Blender

Diskussion von Lindenmayer Systemen und Möglicher Vorteile ihrer Einbindung in die 3D-Grafik-Software Blender

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Nikole Leopold
Matrikelnummer 1327344

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Associate Prof. Michael Wimmer

Wien, 24. März 2017

Nikole Leopold	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Algorithmic Botany via Lindenmayer Systems in Blender

Discussion of Lindenmayer Systems and Potential Advantages of their Integration in the 3D Computer Graphics Software Blender

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Nikole Leopold
Registration Number 1327344

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Michael Wimmer

Vienna, 24th March, 2017

Nikole Leopold	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Nikole Leopold
Linzerstraße 429, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. März 2017

Nikole Leopold

Acknowledgements

I am very grateful to my supervisor Michael Wimmer, who let me choose this topic of personal interest. Had I not discovered the intricate beauty of Computer Graphics, I would likely be studying biology: So fascinating and full of miraculous patterns is the world around us, the world that grows by itself. While I am now very much at home in the field of bits and pixels, the exploration of computer-generated plant-like patterns is a long-held personal interest. Very well am I aware that in my future workings I will most likely pursue matters of a more practical nature. I am therefore especially happy to have had the opportunity in my Bachelor thesis of taking such an interesting excursion into a field beyond the practical, a field of flowers of a beauty that is in no need of legitimization.

Moreover I am very happy for having been able to follow my ethical motivation of working entirely with open-source software projects. Nonsubtractible goods like information and software enrich the world if we end the unnecessary artificial scarcity in their distribution.

My deep gratitude goes to late philosopher and entertainer Alan Watts for opening my eyes to the world around us to which we connect, the world that goes on of itself:

> *"When you look at the clouds they are not symmetrical. They do not form fours and they do not come along in cubes, but you know at once that they are not a mess. They are wiggly but in a way, orderly, although it is difficult for us to describe that kind of order. Now, take a look at yourselves. You are all wiggly. We are just like clouds, rocks and stars. Look at the way the stars are arranged. Do you criticize the way the stars are arranged?"*
>
> —Alan Watts (1915–1973)

Kurzfassung

Lindenmayer-Systeme, kurz L-Systeme, stellen ein lange bestehendes und weitreichend untersuchtes Konzept im Bereich der Computergrafik dar. Ursprünglich durch den theoretischen Botaniker Aristid Lindenmayer eingeführt, um die Entwicklung einfacher multizellulärer Organismen zu modellieren, werden sie heute häufig mit der Modellierung von Pflanzen oder anderen abstrakten Verzweigungsstrukturen in Verbindung gebracht. Verschiedene Varianten wie stochastische, parametrische und kontextsensitive L-Systeme erweiterten den Formalismus über die Jahre, was die Modellierung von stochastischem, kontinuierlichem Wachstum sowie komplexen Wechselwirkungen von Pflanzenorganen untereinander und mit der äußeren Umgebung ermöglicht. Spezialisiertere interaktive Techniken sind mit Sicherheit besser geeignet, um Pflanzenstrukturen auf intuitivere und mehr vorhersehbare Weise zu produzieren, da wo künstlerische Kontrolle unerlässlich ist. L-Systeme stellen aber jedenfalls nichtsdestotrotz eine faszinierende und eindrucksvolle Methodik dar, da sie die Beschreibung von Mustern von erstaunlicher Vielfalt mittels einfacher formaler Produktionsregeln, sowie grafischer Interpretation der Ergebnisse, ermöglichen. Kleine Änderungen an diesen Regeln ergeben oftmals unerwartete, aber ästhetisch faszinierende Ergebnisse und eine Vielfalt an Formen und Mustern deren genauere Untersuchung schon um ihrer selbst willen lohnenswert ist.

Der Schwerpunkt dieser Arbeit liegt nicht darin, neue Techniken für die ästhetische oder biologische Modellierung von Pflanzen zu präsentieren. Diese Arbeit zielt vielmehr darauf ab, den bestehenden Formalismus parametrischer, kontextsensitiver L-Systeme in einer weit verbreiteten open-source Computergrafiksoftware wie Blender in Form eines Add-ons zu integrieren und die möglichen Vorteile einer solchen Integration zu diskutieren. In dieser Hinsicht wird besonderes Augenmerk auf die Modellierung der Interaktion einer wachsenden Struktur mit ihrer virtuellen Umgebung gelegt.

Software, die im Rahmen dieses Projekts entstanden ist, ist unter einer open source Lizenz zur Verfügung gestellt:
https://github.com/mangostaniko/lpy-lsystems-blender-addon. [Leo17]

Abstract

Lindenmayer systems, or L-systems, are a well-established and thoroughly studied concept in the field of computer graphics. Originally introduced by theoretical botanist Aristid Lindenmayer to model the development of simple multicellular organisms, they are now commonly associated with the modeling of whole plants and complex branching structures. Various extensions such as stochastic, parametric and context-sensitive L-systems have been introduced to the formalism, allowing the modeling of stochastic, continuous growth and complex interactions of plant organisms with each other and with the external environment. More specialized interactive techniques are arguably better suited to more intuitively and predictably produce plant structures where artistic control is essential. Nonetheless, L-systems remain a fascinating and powerful methodology as they allow for the description of patterns of astonishing diversity via simple formal rules of production and graphical interpretation of the results. Small changes to these rules often yield unexpected but aesthetically fascinating results and the plethora of forms and patterns thus produced constitute a subject of study that is highly worthwhile in itself.

The focus of this work is not to present novel techniques for the aesthetic or biological modeling of plants. This work aims at integrating the existing formalism of parametric, context-sensitive L-systems in a widely used open-source computer graphics software like Blender in the form of an add-on, as well as to discuss the potential advantages of such an integration. In this regard, special consideration is given to allow the modeling of environmental interaction of a growing structure with a Blender scene.

Software developed in the course of this project is available under an open source license: https://github.com/mangostaniko/lpy-lsystems-blender-addon. [Leo17]

Contents

CHAPTER 1

Introduction

1.1 Motivation and Overview

Lindenmayer systems or L-systems are a well-established and thoroughly studied concept in the field of computer graphics. Introduced in 1968 by theoretical botanist and plant physiologist Aristid Lindenmayer [Lin68] to model the development of simple multicellular organisms, they are now commonly associated with the modeling of whole plants and complex branching structures. Various extensions, such as stochastic, parametric and context-sensitive L-systems, have been introduced to the formalism, allowing the modeling of stochastic, continuous growth and complex interactions of plant organs with each other and with the external environment.

It is important to note that L-systems are not necessarily the tool of choice regarding the modeling of plant structures. A few rather different approaches previously worked out in this field will briefly be discussed in Chapter 2, including some that prove to be much better suited for artistic aesthetic use, yielding more predictable and intuitive results to work with as an artist, compared to the fascinating but often hard to predict structures produced by L-systems.

The beauty of L-systems is arguably found in the astonishing diversity of patterns emerging from a set of simple rules. It is the intriguing appearance and fascinating patterns of growth of plants and branching structures, especially when interacting with their environment, that is the main motivation inspiring this work. Despite the often unpredictable nature of results of a procedural method such as the L-system formalism, its use in this work is motivated by the plethora of fascinating forms and patterns, the study of which is highly interesting and worthwhile in itself. In this light, it is important to note that the focus of this work is not to present novel techniques for the aesthetic or biological modeling of plants. The main motivation of this work is rather to bring the existing formalism of parametric context-sensitive L-systems to the open-source

computer graphics software Blender, with special consideration to allowing the modeling of environmental interaction with a Blender scene, in order to discuss the potential of such an integration.

The objectives of this work can thus be outlined as in the following:

- Make parametric context-sensitive L-systems available in the open-source computer graphics software Blender.
- Demonstrate results and discuss the potential advantages of Blender integration regarding the graphical interpretation, rendering, postproduction modeling, but most importantly also directly during L-system production to simulate environmental interaction with a Blender scene.

Notably, the main goal of this thesis is to discuss possible advantages and disadvantages of an integration of parametric context-sensitive L-systems in Blender, not so much the details of possible implementations.

Moreover, it should be emphasized again that it is not the main objective to have a tool for artists to easily create aesthetic but also intuitively controlled plant models. L-systems are not so well suited for this purpose, better ways to achieve such means are discussed in Section 2.2.

In Chapter 3 the well-established theory of L-systems will be laid out as an overview, followed by a brief review of the various extensions that have been introduced to the L-system formalism over the years.

Chapter 4 gives an overview of the steps taken to implement an integration of parametric context-sensitive L-systems in Blender. For this purpose, an add-on based on an adapted version of the L-Py open-source Lindenmayer system framework was developed using custom graphical interpretation code based on the Blender Python API with extended interpretation commands to facilitate environmental interaction with a Blender scene. This chapter also describes possible advantages of this integration in Blender from a theoretical standpoint, to be verified via application results in Chapter 5.

Finally, Chapter 5 will give examples of what can be achieved using L-Py integration in Blender. The production of models of plant inflorescence and trees will be demonstrated, as well as their graphical interpretation via the Lindenmaker add-on, with results being rendered directly in Blender. More importantly, however, several examples modeling environmental interaction of a growing structure with a Blender scene during L-system production are presented, demonstrating this most promising advantage of Blender integration.

1.2 On Graftals and the Fractal Appearance of Plants

The visual patterns exhibited by many “naturally” grown phenomena, such as plants, are often described as aesthetically pleasing, yet chaotic at the same time. It may be

appropriate to note that the term chaos is often used to describe patterns that exhibit an order not immediately apprehended by the human mind. This kind of aesthetics is also often attributed to a class of phenomena called *fractals*, objects that exhibit recurring self-similar structure at any scale. While such patterns had been subject of examination over a century before his time, the term fractal was first used by Benoit Mandelbrot [Man83], whose devotion to their study and especially his use of computer-generated imagery rapidly led to widespread popularity of the concept in the scientific community and in popular imagination.

Mandelbrot roughly characterized a fractal as "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole". Lacking an agreed precise definition, fractals are commonly attributed the aforementioned quality of (strict or approximate) self-similarity at different scales, and thus detailed structure even at arbitrarily minute scales. While exhibiting seemingly complex detail, fractals typically emerge from the application of comparatively very simple, mostly recursively defined rule sets. Another characteristic is the so-called fractal dimension, regarding the way in which fractals scale: Doubling the length of the sides of a square will yield an area scaled by a factor four. For any polygon, the area is proportional to the square of the length of the one-dimensional unit of measure of the space it resides in. However, for fractals the two-dimensional area measure often scales proportional to a non-integer power of the one-dimensional measure. An intuitive way to think of this is to imagine that the rough surfaces of fractals made up of infinitely many minute segments with differing orientations often do not fit the classical measures of one- or two-dimensional lines or areas, which are defined by smooth continuous means of measurement. The concept of fractal dimension is also often defined as a measure of the space-filling capacity of the structure, which is the degree to which its boundary tends to fill up the given space. More precise delimitations of the term are given by Kenneth Falconer [Fal90].

At first glance, plants and the branching structures obtainable via Lindenmayer systems may exhibit such fractal patterns, yet they are mostly not true fractals per definition: They do not exhibit self-similarity at all scales and typically do not have infinite detail, as they are usually not defined in the limit like fractals. Alvy Ray Smith [Smi84] suggested the term *graftal* referring to the structures produced by parallel rewriting systems such as L-systems, to distinguish them from fractals in the strict sense, since even though some graftals qualify as fractals, mostly they do not. While this terminology was not widely adopted, it is mentioned here to suggest a cautious use of the term fractal when referring to the products of L-systems.

This note aside, L-systems and fractals still have a lot in common. Simple rules applied in a recursive manner yield complex and intriguing structures that are often aesthetically pleasing not only in their graphical interpretation but also in the very nature of their growth. These perceptions of aesthetics often attributed to plants as well as fractals indicate that what is often associated with chaotic growth and randomness is in fact

symptomatic of an underlying implicit order that the human mind may find hard to trace back symbolically, yet intuitively appears to arouse a kind of familiarity.

CHAPTER 2

Related Work in the Field of Plant Modeling

Lindenmayer systems are among of the first techniques to have been used for the procedural modeling of plants in computer graphics. They are however not necessarily the most effective way of generating plant models. For artistic purposes especially, algorithms specialized for the modeling of specific plant species or more interactive and intuitive methods may give the graphical artist more precise control over the desired results, as opposed to the often unpredictable nature of the results of using highly generic and predominantly procedural methods like L-systems. It shall be pointed out again that it is the focus of this thesis to discuss the theory and application of L-systems specifically, due to the wide range of fascinating patterns they allow to describe. Yet the existence of other techniques for the modeling of plants that can be much more viable in production systems shall be emphasized in this chapter, which will present some examples of the more well known related work in algorithmic botany.

2.1 Models of Morphogenesis

Related to procedural methods of modeling plant *morphogenesis*, i.e. the development of forms and patterns in plants, some well known approaches are summarized by Przemysław Prusinkiewicz [Pru93]. An important notion in the context of morphogenesis is the concept of *emergence*, which denotes a quality of aggregations of interacting units or cells to exhibit new properties that can not be attributed to a mere superposition of separate individual contributions. Quite often, the rules describing the behavior of the individual components are relatively simple, yet the processes and structures resulting from their interaction can be astonishingly intricate.

Prusinkiewicz distinguishes structure-oriented and space-oriented models of morphogenesis. Structure-oriented models focus on describing the development of the structure (e.g.

plant) itself, whereas space-oriented models involve a description of the space or medium in which the structure is situated and with which it interacts. The structure-oriented models involve L-systems in their various forms, which will be thoroughly discussed in the next chapter. It shall be noted at this point that L-systems are capable of reproducing structures such as the tree models described by Aono and Kunii [AK84] or de Reffye et al. [dREF+88], as well as grass models simulated via particle systems by Reeves and Blau [RB85], among others. An interesting variant called *Map L-systems* was introduced by Lindenmayer and Rozenberg [LR79] that allows the modeling not only of branching structures as with conventional L-systems, but also of botanical structures that involve cyclic graphs, such as sections of plant tissue, leaves or other cell aggregations.

The space-oriented models include reaction-diffusion models introduced by Alan Turing [Tur52] that are well studied in the field of theoretical biology. Reaction-diffusion models simulate the diffusion and interaction of morphogens, i.e. substances involved in morphogenesis, in a given medium via systems of partial differential equations. In the simulation, the morphogens may fulfill various functions: They may induce cell differentiation and determine whether a cell belongs to the medium or to the growing structure. They may also act as an inhibiting agent to the initiation of other branches in the vicinity of the tip of a growing branch tip, or model nutrient gradients determining growth direction. Another approach is that of cellular automata, which operate on a uniform grid of cells. Each cell has a state that may transition to another state according to a set of rules shared among all cells, such that the next state of a cell depends only on its current state and those of its neighboring cells. Cellular automata can be used to model branching structures as well as collisions between branches in the medium. Extensions to three-dimensional space, known as voxel automata, have been used to simulate structures in strong interaction with their environment, such as climbing plants or roots.

2.2 Effective Modeling of Plants for Artistic Use

The depiction of plant structures is of relevance in various applications of computer graphics, yet in most cases an accurate simulation of plant development is not required. Architectural visualization, graphic design, animated movies etc. focus their attention on a convincing and aesthetically pleasing appearance of plant models, regardless of related botanical phenomena. Graphical artists need tools that enable them to create suitable plant models in a predictable and controlled manner, that lead to good results without much effort while allowing for precise customization when needed.

Weber and Penn [WP95] presented a highly popular model that found broad adoption in various 3D graphics tools, including the open-source software project Blender. The aim of their project was to devise a method of generating a wide variety of 3D tree models of realistic appearance in a computing resource efficient and user-friendly way, to be used in simulation of natural environments for image detection and recognition studies. Specifically, the goal was to detect vehicles among possibly lush vegetation,

requiring accurate visual properties of tree models and efficient drawing of great numbers of trees to prevent excessive bias. In their approach, they deliberately abstained from the inclusion of biophysical simulation and used only simple mathematical models to ensure ease of use for implementation and end users. They emphasized the importance of hiding aspects that may be difficult to control and present to the user a selected set of intuitive parameters, that still leave the user extensive control and freedom of design. Thus, they recommended to avoid models such as pure fractals that allow only limited control via the specification of a small set of rules that lead to incredible yet unpredictable results, but rather expose parameters to allow for precise tweaking of specific features of the model. Many of these parameters include an additional variable specifying the degree of random variation the respective feature may exhibit, thus allowing for the generation of a variety of unique instances of a tree species via the choice of a single pseudorandom number generator seed value.

Their model consists of multiple levels of branches, commonly limited to a trunk which can include a dichotomous split, branches and subbranches as well as leaves. Notably, the display of different branching levels can be disabled during modeling, making it easier to adjust parameters for each level individually. Attributes of child structures, such as the branch length or radius, can differ from those of their parents, but are typically defined relative to them. Trunk and branches may be refined by a specific number of segments with adjustable curvature and their radius may be tampered or lobed. Child structures may be placed around the parent branch in various arrangements relative to other children and with different branching angles. A useful feature is the definition of common tree shapes (such as conical, spherical, flame-like among others) via predefined functions determining the length of a branch in relation to its position on the trunk. Moreover, the length of branches can be pruned to fit a custom envelope shape. The number of branches is determined via a specified density or probability of their occurrence. The same is the case for leaves, which are typically assigned to the third or fourth level of branching. The shape of leaves may also be adjusted or chosen from a few characteristic presets. Some implementations, such as the Sapling add-on for Blender, rely on the use of leaf textures with an alpha channel to avoid the need for detailed leaf geometry.

The model also includes a simple but convincing simulation of wind sway of branches for animation, as well as parameters for vertical attraction to simulate phototropism, i.e. the tendency of shoots to orient and grow toward regions of increased light exposure. Moreover, Weber and Penn describe a method to smoothly degrade the quality of tree structures with increasing distance to the viewer. Instead of choosing from a small set of predefined models at different levels of detail, which would lead to sudden changes in tree appearance during animation, they continuously degrade the geometry of individual components of branches and leaves, e.g. reducing curved polygonal stems to simple triangular tubes and eventually lines. Altogether the model described by Weber and Penn offers highly effective and easy to use means of quickly generating a variety of detailed trees for artistic use or realistic visualization of individual trees and canopy.

A more interactive approach to plant modeling was introduced by Lintermann and Deussen [LD98], [LD99]. Like Weber and Penn they focused on the visual aspects and on allowing the direct modification of global characteristics of a plant, unlike L-systems which specify implicit and unpredictable growth rules. Their modeling approach differs from the Weber and Penn model in that it is highly interactive and allows for the creation of not just trees and shrubs but a wide range of branching structures, also including flowers and even nonbotanical objects. Their system is based on a graph-based description, where plants are generated according to a hierarchical arrangement of components that the user can manipulate via a graphical interface by dragging iconic representations of the components from one place to another in the hierarchy (Fig. 2.1). These components are powerful tools that encapsulate data structures and algorithms related to the generation of plant structures. While rearranging the components in the graph allows for a effective and time-efficient way of specifying the general high level structure, each component also offers a set of parameters to adjust more specific details.

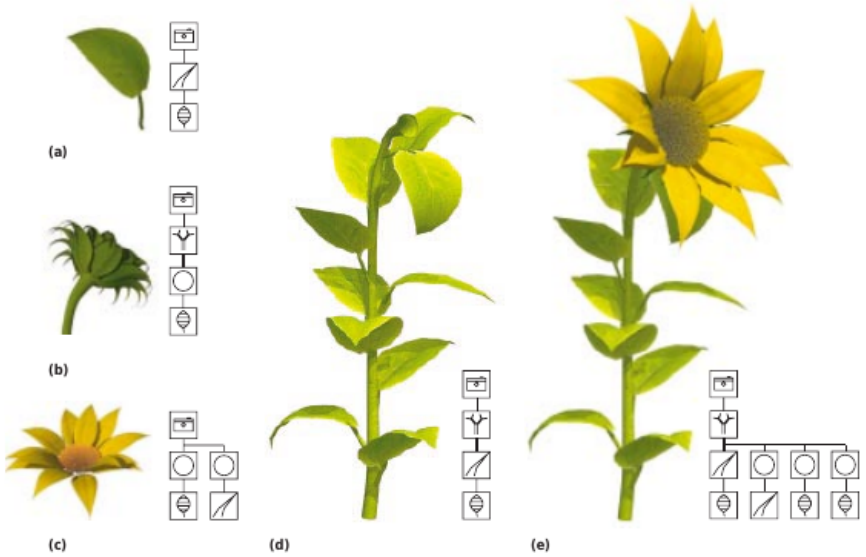


Figure 2.1: The plant modeling approach by Lintermann and Deussen builds on a hierarchical iconic representation of components defining the structure and geometry of the plant, which the user can interactively rearrange. In this example of a sunflower, a leaf is scanned and used as texture for a Leaf component that is placed underneath a Horn component specifying the leaf stem (a). These leaves are later arranged around a stem via a Tree component (d), note that the bold connecting line indicates a branch relationship. The flower itself is modeled via Phiball components for the petals and sepals as well as the seeds (b and c), which are and then placed underneath the Tree component via a simple child relationship so that the flower is simply appended to the stem (e). Image taken from Lintermann and Deussen [LD99].

The types of components include some that define simple geometric primitives or more complex structures such as surfaces of revolution or the Horn component which defines a sequence of point sets used as cross sections that can be oriented in various ways and are later connected and triangulated to form a stem or twig. A Leaf component allows for interactive modeling of leaves by adjusting polygonal curves and also allows to project textures.

The Tree component extends the Horn component by arranging its child components as branches on itself, where the user can easily specify branch density, angular deviation etc. Other components that also create instances of subsequent components include Hydra and Wreath, creating a circular arrangement where instances are oriented perpendicular or parallel respectively. The Phiball component places components on a section of a sphere according to the golden section, a common pattern in phyllotaxis.

Additionally, there are components to further adjust the final plant structure. The FFD and HyperPatch components for free form deformation by specifying functions or interactively moving controls points of a 3D patch, respectively. A Pruning component can be used to limit the growth to a specified CSG volume, either cutting or scaling geometry for that purpose. Finally the World component defines global constraints to plant growth by simulation of gravitropism and phototropism, i.e. by specifying the tendency of growth to orient according to gravity or light fields. Further variation is introduced by allowing each component parameter to be individually transformed by a user-specified function or a random factor. To model exceptions such as dead branches, components that iteratively create instances also hold a list of boolean flags, that for each iteration number indicates whether an instance will be created or not. Thus an exception could be introduced by switching of the flags e.g. for iteration numbers 2, 3, and 5, and using another component that has these flags enabled and produced a different geometry.

All in all, the technique introduced by Lintermann and Deussen offers a highly effective means of quickly modeling diverse plant structures of realistic appearance. According to an evaluation by Lintermann and Deussen [LD98] in two groups involving 18 inexperienced users, most of them found the component and hierarchy based modeling approach intuitive, yet found the great numbers of parameters a bit confusing. The groups were however able to model structures like a tulip in less than an hour, being allowed to ask questions. Experienced users were reported to be able to create convincing models of complex trees within 2 hours. Notably, existing component hierarchies can easily be reused to create variations of existing trees in a few minutes.

2.3 Particle Systems and Space Colonialization

Two other techniques that shall be described here are based on particle systems and point clouds. Rodkaew and Chongstitvatana [RCSL03] presented a surprisingly simple method for modeling three dimensional branching structures yielding very plausible results. They first describe the synthesis of leaf venation patterns via particle systems in a two-dimensional space, which they then extend to the modeling of trees and shrubs,

as well as root systems. For the purpose of modeling leaf venation, a boundary shape is chosen in which particles are randomly distributed. Particles then iteratively move toward the petiole (the leaf stalk) while being simultaneously attracted by the nearest neighboring particle: Simply adding the respective unit vectors and multiplying with a given step size is deemed sufficient for determining the particle displacement. Veins then correspond to the trails of the particles. When particles come close enough to each other, they are merged to form a single particle with combined attributes. An abstract attribute called "energy" is initially assigned in uniform distribution to each particle, to model the thickness of veins determined as a function of this energy. As particles move toward each other and merge, their energies are added up, leading to thicker veins as particles move closer to the petiole, until eventually only one particle remains. The model is easily extended to the three dimensional case. For the modeling of trees, Rodkaew and Chongstitvatana recommend distributing particles in a volume between an inner and an outer boundary shape. Leaves can then be placed at the initial particle positions. While not being suitable for some kind of plants, this method yields realistic visual models of leaf venation and tree branching structures from a very simple set of rules and a very limited set of parameters including the boundary shape determining the initial distribution, the number of particles and their initial energy, the merging radius and the movement step size. To model the effect of light they employ a ray-tracing technique to simulate light distribution, which can then be used to determine the density of initial particles.

A similar approach was introduced by Runions and Lane [RLP07], who interestingly also based their model for trees on a model of leaf venation. They explicitly mention that the development of tree branches is often much more influenced by the competition for space and other environmental factors, as opposed to intrinsic branching patterns of the plant. Their method differs from that of Rodkaew and Chongstitvatana in that branches are formed in a base-to-leaves order. Very similarly, they also assume a three dimensional envelope shape as a boundary for the tree crown, yet instead of distributing initial particles growing down from the tips to the base, the envelope is seeded with attraction points. These attraction points can be seen as indicators of empty space to be filled by branch growth. The space colonialization algorithm described by Runions and Lane treats the growing structure as a collection of tree nodes that grow toward the attraction points, as illustrated in fig. 2.2. Starting from a single tree node at the base, existing tree nodes are influenced by any number of attraction points, where an attraction point only influences its closest tree node (as in Voronoi partitioning), if it is within a radius of influence. For each node influenced by at least one attraction point, a new node is generated one step of constant distance away toward the arithmetic mean direction to all influencing attraction points. This direction of growth may be deliberately biased by another vector to account for tropisms and branch weight. Attraction points are removed as soon as a tree node is generated within a given threshold distance, to indicate that the space is filled. This process is repeated until all attraction points have been removed, no more nodes are within the radius of influence of any attraction point or a user specified number of iterations is reached. Like the model by Rodkaew

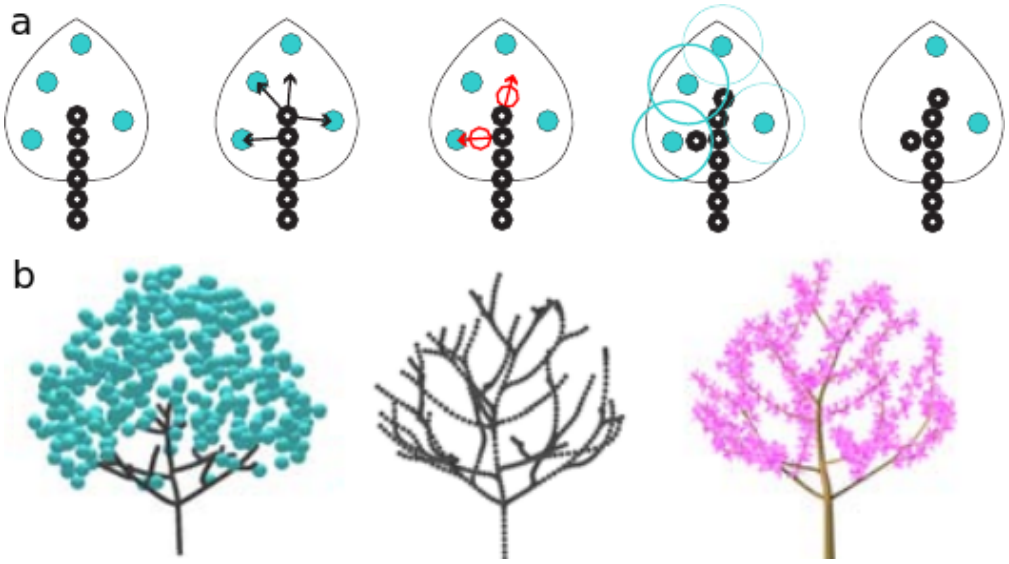


Figure 2.2: (a) The space colonialization algorithm: New tree nodes are added to the existing nodes in average direction of influencing attraction points, while any attraction point may only influence a single tree node. Attraction points within a threshold radius of a tree node are removed. (b) Three dimensional illustration of the process. In the last step, geometrical detail and flowers are addded. Images taken from Runions and Lane [RLP07].

and Chongstitvatana this approach allows the generation of a variety of realistic tree and shrub structures from a simple set of parameters such as the number of attraction points and their spatial and temporal distribution, the attraction point influence and removal distance etc. Notably, a small number of attraction points may lead to rough and irregular branches, as the removal of a single attraction point may significantly affect the subsequent growth direction. To more realistically model the influence of light, Runions and Lane recommend to more densely distribute attraction points near the rim of the envelope shape. The influence of obstacles can be accounted for by simply removing attraction points that lie within the volume of collision. Due to the nature of the algorithm, branch intersections among the tree itself do not occur. Flowers and other organs can be placed at the tips of growing branches after a certain number of iterations.

CHAPTER 3

Lindenmayer Systems

The formalism of Lindenmayer systems known today has been the result of many years of interdisciplinary research with contributions from several fields of study, most notably computer graphics and botany. As such, it has attracted interest for its possible applications both for image synthesis purposes and for scientific visualization.

As mentioned in the last chapter, there are other techniques of algorithmic botany that may offer more effective means of generating plant models for artistic uses. While L-systems are very well capable of producing images evoking aesthetic pleasure in a human observer, they do not allow for intuitive artistic control in the modeling of plants without more interactive extensions. The beauty of L-systems lies not in artistic freedom of a human user, but much more in the astonishing diversity of patterns emerging from the procedural productions themselves. A small set of simple definitions and rules spiced with a bit of pseudorandom variation can yield the most fascinating forms. As noted by Prusinkiewicz [Pru00], L-systems however are not so much a method to model the mere appearance of a plant, i.e. a "descriptive" model focused on the geometric definition of plant shape. They much more fall into the category of "functional-structural" models, which try to reflect the physiological growth processes intrinsic to the development of plant form. Plants can be described as modular structures consisting of individual units, such as branches, leaves and flowers. Instead of explicitly specifying the structure and arrangement of each individual modules in a grown plant, functional-structural models try to implicitly capture the growth of arbitrarily complex structures via a small set of rules, describing the development of each class of module.

The original formulation of Lindenmayer systems is attributed to Aristid Lindenmayer, who in his work as a biologist was first and foremost inspired by the study of plant physiology. In the year 1968 Lindenmayer published his studies on a mathematical framework to simulate the development of simple multicellular organisms [Lin68]. His approach is based on a parallel string rewriting method, in which an organism is defined by a string of symbols, sometimes called an *L-string*, from a limited alphabet that

represent the individual modules or cells, and a set of rules governing the patterns by which each type of cell will be replaced by a string of other cells. The strings resulting from the iterative application of such replacement rules can be interpreted geometrically in various ways to create visual models of the organisms described. As emphasized by Prusinkiewicz [Pru98] these strings themselves form the key data structure most implementations are based on, thus offering a very compact representation that leaves a lot of freedom to the graphical interpretation. Lindenmayer originally employed the formalism to model simple branching filamentous organisms such as *Callithamnion roseum*, a species of red algae that according to more recent taxonomy is now officially recognized under the name of *Aglaothamnion hookeri*.

3.1 Basic Definition of L-systems

In the most basic form of L-Systems, the development of plant modules is exclusively modeled in terms of lineage, i.e. without any considerations regarding potential interaction between individual modules and with the environment. Such L-Systems are referred to as *context-free*, and commonly denoted *0L-System* to indicate that modules interact with zero other modules. Context-sensitive or interactive L-Systems will be described in a later section.

An 0L-System according to Prusinkiewicz and Lindenmayer [PL96] as well as Hanan [Han92] is defined as an ordered triplet $G = \langle V, \omega, P \rangle$, consisting of

1. an *alphabet* V of symbols corresponding to the different types of plant modules,
2. an initial nonempty string of symbols $\omega \in V^+$ called the *axiom*,
3. as well as a finite set of *productions* $P \subset V \times V^*$.

A production $(a, \chi) \in P$ is a parallel rewriting rule defining that each occurrence of a module $a \in V$ in the current string is to be replaced by a string of modules $\chi \in V^*$, where a is called the *predecessor* and χ the *successor*. Such a production would be commonly written as $a \to \chi$. The identity production $a \to a$ is assumed for modules that are not mentioned as predecessor in any explicitly specified production. If for each module as a predecessor there exists exactly one possible successor (i.e. exactly one applicable production), such an 0L-system is called *deterministic* and written as D0L-system.

As opposed to Chomsky grammars where productions are applied sequentially, all rewriting in L-Systems formally occurs simultaneously (or rather as a single atomic operation). This is motivated by the biological background of L-systems to describe multicellular organisms, where division of individual cells may occur at more or less the same time independent of each other.

The classical example of a D0L-system given in Lindenmayer's original paper [Lin68] models the development of algae. Its definition (here using different letters for better

visualization) involves four types of modules $\overleftarrow{O}, \overrightarrow{O}, \overleftarrow{o}, \overrightarrow{o}$, where O and o refer to the state a cell in the algae is in regarding its readiness for cell division, and the arrows model the orientation or direction of future growth. It shall be made clear that this is an arbitrary choice of nomenclature of no general semantics in the formalism of L-systems. The axiom and productions are given in the following:

$$\begin{aligned} \omega &: \quad \overrightarrow{O} \\ p_1 &: \quad \overrightarrow{O} \rightarrow \overleftarrow{O}\overrightarrow{o} \\ p_2 &: \quad \overleftarrow{O} \rightarrow \overleftarrow{o}\overrightarrow{O} \\ p_3 &: \quad \overrightarrow{o} \rightarrow \overrightarrow{O} \\ p_4 &: \quad \overleftarrow{o} \rightarrow \overleftarrow{O} \end{aligned}$$

Applying the productions starting from the axiom yields the following sequence of strings:

$$\begin{gathered} \overrightarrow{O} \\ \overleftarrow{O}\overrightarrow{o} \\ \overleftarrow{o}\overrightarrow{O}\overrightarrow{O} \\ \overleftarrow{O}\overleftarrow{O}\overrightarrow{o}\overleftarrow{O}\overrightarrow{o} \\ \overleftarrow{o}\overrightarrow{O}\overleftarrow{o}\overrightarrow{O}\overrightarrow{O}\overleftarrow{o}\overrightarrow{O}\overrightarrow{O} \end{gathered}$$

Some of the key properties and limitations of this basic definition of L-systems are identified by Prusinkiewicz [Pru99]:

- L-systems describe the development of structure over time, as opposed to a static description of shape.
- The states of individual subunits are discrete and different states may only be modeled by the addition of symbols and productions for each state of a cell and their transitions. This limitation is overcome by parametric L-systems which are the subject of a later section.
- Production application happens in discrete time steps, with no continuous development between subsequent applications. A time-continuous alternative offered by Differential L-systems will be covered later.
- An L-system string by itself defines only topology, not geometry. The most common techniques of geometric interpretation of strings is subject of the following section.
- For context-free L-systems, changes in topology occur by cell division (lineage) only. Thus the modeling of freely moving cells like in animal tissue as well as the interaction with other cells or with the environment is not possible without extensions to the formalism. Such extensions are the context-sensitive and open L-systems which are also covered later.

3.1.1 Branching Structures and Bracketed L-systems

So far no specifications have been made regarding any methodology to define branching structures, which needless to say is of central importance to the description of plant structure. Lindenmayer in his original work [Lin68] proposed a notation of bracketed strings, known as bracketed L-systems. It involves the addition of left and right brackets, "`[`" and "`]`" as special symbols to the productions. Each opening left bracket indicates the start of new branch attached to the module to its left, containing all subsequent modules up to the next unmatched closing right bracket. More specifically, whenever a right bracket is found while traversing the string from left to right, it is matched with the last unmatched occurrence of a left bracket. Obviously for each left bracket there should be a matching right bracket. The nesting of matching pairs is possible, allowing for the definition of branches of higher order. The interpretation of bracketed strings is facilitated by the use of stack data structures, as described in the next section, which also demonstrates possible geometric interpretations of bracketed L-systems.

3.2 Interpretation of Strings via Turtle Graphics

To be used for the visualization of plant models, the strings produced by an L-system need to be interpreted graphically. By themselves they only describe the topology of modules, i.e. how they are arranged relative to each other within the structure, but not where in space they shall be positioned, how they shall be oriented etc.

As noted by Prusinkiewicz [Pru99], two different approaches for the geometric interpretation have been suggested:

- One might keep the specification of any geometric information separate from the production of modules, thus leaving the details about how each module or successions of modules shall be positioned and oriented geometrically in the interpretation step entirely to the system interpreting the strings. This would however make it very difficult to achieve consistent results on different systems without sharing additional interpretation instructions. More importantly however, the geometric aspects of plant growth would be separate from the development process itself.
- Another approach would be to introduce special symbols to be used as standardized instructions for the interpreting system. These symbols would be added or removed as part of the production process itself and thus would allow the specification of geometric detail right within the L-system formalism, allowing a predictable interpretation.

Most L-system implementations today include symbols related entirely to the geometrical and graphical interpretation as a central part of the production process and use a technique commonly known as *turtle interpretation*, that is based on the turtle graphics methodology. Turtle graphics was first used as part of the educational programming

language LOGO, where the turtle was used to conceptualize a relative cursor, that would draw lines as it moved around the screen like a pen-carrying turtle. The turtle is controlled by a sequence of commands that would change its state and create a figure to be drawn on the screen.

The state of the turtle consists of its *position* $\vec{P}$ in 3D space, as well three perpendicular unit vectors *heading* $\vec{H}$, *left* $\vec{L}$ and *up* $\vec{U}$ defining its *orientation*, i.e. its local coordinate system, where $\vec{H} \times \vec{L} = \vec{U}$, as well as some drawing attributes like color and line width. Using homogeneous coordinates, the 4 vectors can be conveniently represented by a 4x4 matrix allowing for efficient state transformations.

Turtle interpretation in the context of L-systems involves reading the string produced from left to right, until a symbol relevant to the graphical interpretation is encountered. These special symbols correspond to commands that alter the state of the metaphorical turtle and potentially the resulting graphic. The most commonly implemented commands and their corresponding symbols, according to Lindenmayer and Prusinkiewicz [PL96], are listed in the following:

`F` Move forward one step in direction of the heading vector $\vec{H}$ and draw a line in between.

`f` Move forward one step without drawing a line.

`+` Turn left around the up vector $\vec{U}$.

`-` Turn right around the up vector $\vec{U}$.

`&` Pitch down around the left vector $\vec{L}$.

`^` Pitch up around the left vector $\vec{L}$.

`\` Roll left around the heading vector $\vec{H}$.

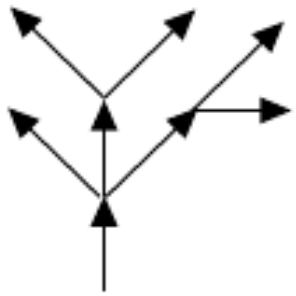
`/` Roll right around the heading vector $\vec{H}$.

`|` Turn halfway around the up vector $\vec{U}$.

The modeling of branching structures is facilitated by special commands that make use of a pushdown stack data structure to be able to easily return to the last state on the mother branch after complete traversal of a branch, as illustrated in Figure 3.1:

`[` Push the current state onto the stack.

`]` Pop the last state from the stack and make it the current state.



F[+F][-F[-F]F]F[+F][-F]

Figure 3.1: A simple branching structure modeled via bracketed string notation (note that arrows are used instead of lines for clarification. Initially the turtle is positioned at the base of the lowermost arrow, and oriented in its indicated direction. As the string is read from left to right, it first encounters a `F` command that moves the virtual turtle forward and draws a line. Next a left bracket `[` is read, indicating the beginning of a new branch, thus pushing the current state of the turtle onto the stack. The two commands in the first pair of brackets are a + that rotates the turtle and another `F` that moves it forward again, to the end of the lower left arrow. The closing right bracket `]` indicates the end of the branch and the last state on the mother branch is popped from the stack and applied to the turtle. The turtle is now back at the end of the first arrow, continuing accordingly. Image taken from Lindenmayer and Prusinkiewicz [PL96].

These simple commands already allow interpretations of L-systems that resemble the branching structure of plants, as shown in Figure 3.2, which for ease of illustration focuses on structures growing in a plane.

Additionally, some often used commands related to drawing attributes are listed below, here following the notation used in the open-source L-Py framework as specified by Boudon et al. [BPC+12], that is introduced in Chapter 4:

- `_` Increase line width (for subsequent drawing).
- `!` Decrease line width.
- `;` Increase the color map index.
- `,` Decrease the color map index.

As mentioned by Hanan [Han92] and others, apart from the drawing of simple lines (or cylinders) via the `F` command, individual modules may be associated with predefined 3D surfaces (vertex meshes, splines, ...) that are placed according to the current position and orientation of the turtle. This allows for the efficient graphical interpretation of certain modules that may represent more complex structures like flowers or leaves, the specific geometry of which may more conveniently be defined in an external editor. Some

implementations also allow the specification of mesh surfaces on a vertex level within L-systems themselves via specialized commands that are however beyond the scope of this chapter.

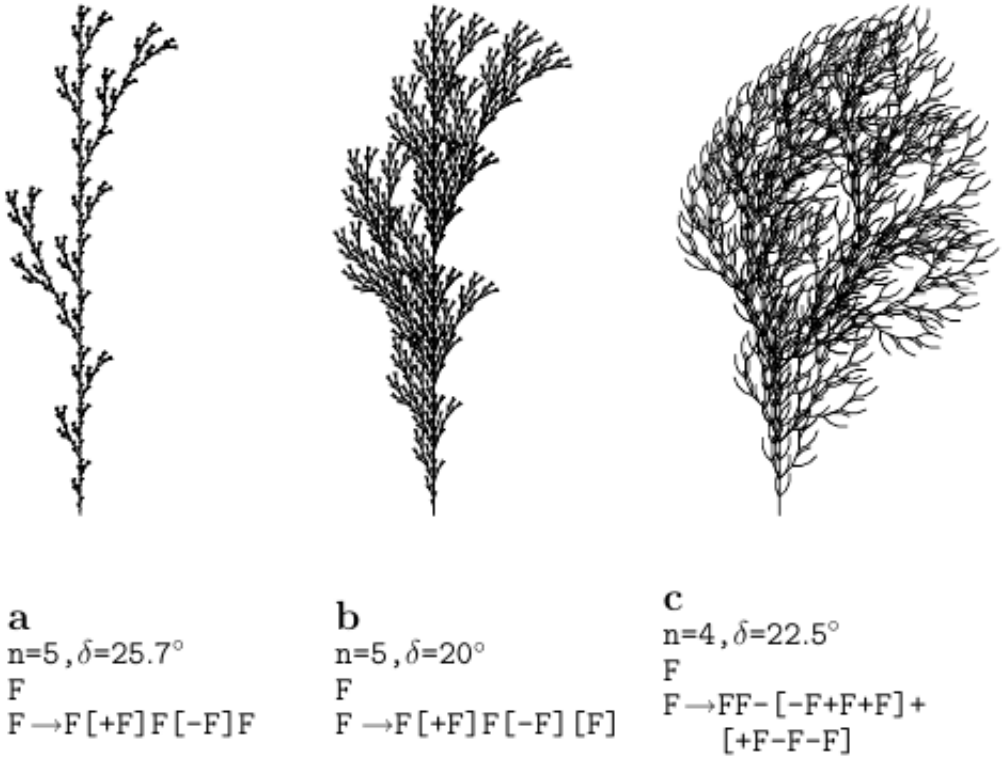


Figure 3.2: Simple interpretation commands allow the visualization of a variety of plant-like branching structures. Here, the variable n defines the number of derivation steps, i.e. the number of times the productions shall be applied, and δ defines the turn angle. Note that with the introduction of parametric L-systems (as described in a later section), parameters like the branching angle can be specified on an individual basis for each command. Image taken from Lindenmayer and Prusinkiewicz [PL96].

3.3 Stochastic L-systems

No two plants grow exactly in the same way, even if they are of the same species and share the same genetic instructions. For L-systems, even if the same productions are used it may be desirable to introduce some random variation. Variation in parameters like branching angles, length of line segments etc. will be covered in the section on parametric L-systems. Variation in branching angles etc. does not really alter the

topology of the structures produced. Much more striking random deviations can be achieved by randomly choosing between different productions to be applied for a given predecessor. For this purpose, the notion of *stochastic L-systems* was introduced by Eichhorst and Savitch [ES80], where productions are assigned a probability in $(0, 1]$ such that the probabilities of all productions sharing a common predecessor sum up to 1. If multiple productions are specified for a given predecessor, the production to be applied is selected randomly according to the given probability distribution. If only one production is specified for a module, the probability of its selection is 1.

The probabilities may be commonly written on top of the derivation symbol (production arrow) or at the end of the production following a colon:

$$
\begin{array}{lcl}
\omega & : & \mathtt{F} \\
p_1 & : & \mathtt{F} \xrightarrow{.33} \mathtt{F[+F]F[-F]F} \\
p_2 & : & \mathtt{F} \xrightarrow{.33} \mathtt{F[+F]F} \\
p_3 & : & \mathtt{F} \xrightarrow{.34} \mathtt{F[-F]F}
\end{array}
$$

A simple stochastic L-system like the one above may already yield a great diversity of branching patterns, as illustrated in Figure 3.3, even though it does not yet involve any random variation of geometrical parameters like branching angles.

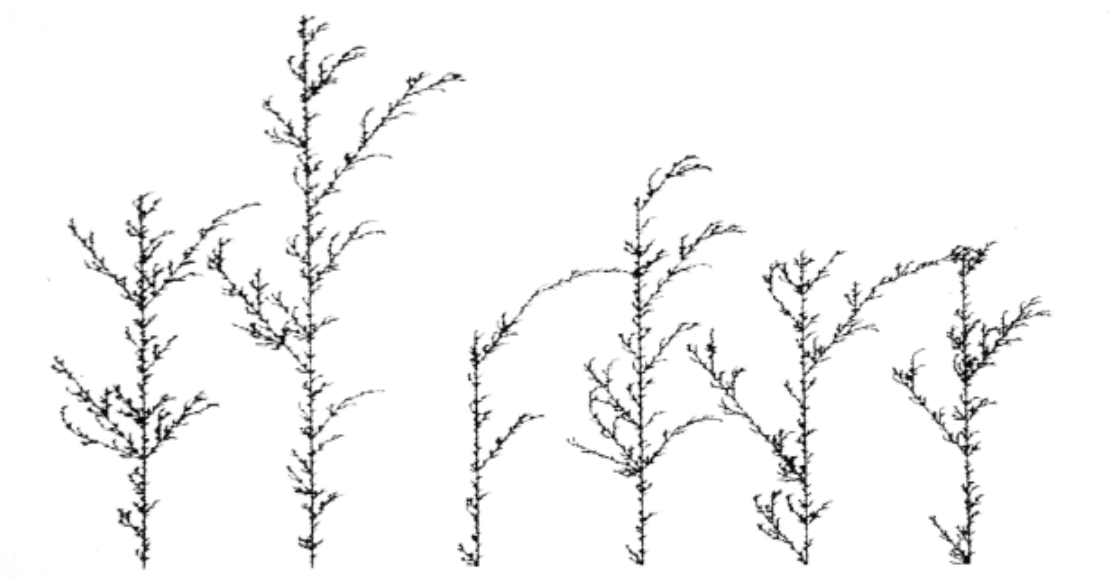
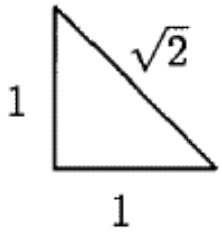


Figure 3.3: Variety produced from a single stochastic L-system, where the productions applied are selected according to predefined probabilites. Image taken from Lindenmayer and Prusinkiewicz [PL96].

3.4 Parametric L-systems

While L-systems with turtle interpretation as described so far can already produce a variety of geometric structures and visual models of simple plants, the formalism is still limited in that all interpretation commands may be specified in discrete steps only. It would for example not be possible to describe a simple isosceles right triangle with just three `F` commands, since line length is a predefined constant uniformly applied at all occurrences of the command. The line length may be set to the length of the catheti, or to the length of the hypotenuse, but not both. The only way to describe both lines would be to approximate them as integer multiples of a common sufficiently small unit length. However since the hypotenuse of an isosceles triangle depends on the length of its sides by a factor of $\sqrt{2}$, an irrational number, a visually satisfying approximation would require a great number of small steps, or `F` commands.



This also poses a great limitation to the modeling of plants, which typically involve structures that vary greatly in respect of geometric parameters like size and angles, especially during development when modeling their growth over time. Lindenmayer thus proposed an extension to the discrete formalism, in which the modules and interpretation commands would be associated with numerical parameters. The first formal definition of *parametric L-systems* was given by Hanan [Han92].

In parametric L-systems a module A may be associated with any number of real-valued formal parameters $a_1, a_2, ..., a_n \in \mathbb{R}$, written in parenthesis as $A(a_1, a_2, ..., a_n)$. Additionally, parameters may be combined with other parameters or numeric constants in arithmetic expressions involving common operators like $+, -, *, /$ and the exponentiation operator `^`, with operator precedence as in common programming languages, and parenthesis for custom order of evaluation. Most implementations also provide basic mathematical functions like sine, cosine, tangent, arccos, arcsin, arctan, floor, ceiling, truncate, absolute value, exponential, logarithms as well as a random function which may provide values from algorithmic pseudorandom number generators or the system's hardware random number generators.

Additionally, productions in a parametric L-system may specify a *condition* that must be met in order for the production to be applied. A condition C consists of logical expressions that may combine parameters or arithmetic expressions via relational operators such as $>$, $<$, $\geq$, $\leq$, `==`, `!=` that evaluate to boolean values, and may themselves be combined via logical operators `&&` and `||`, as known from common programming languages.

Productions (a, C, χ) with predecessor a, successor χ and condition C are commonly written as $a : C \rightarrow \chi$, like in the following example:

$$\texttt{A(s,t)} \;:\; \texttt{s/2} < 7 \;\&\&\; \texttt{t} < 4 \;\rightarrow\; \texttt{B(s} * 2\texttt{)CD(s\^{}2,t} - 2\texttt{)}$$

A production will match and be applied to a module if the following three conditions can be satisfied: The symbol of the production predecessor must be the same as the symbol of the module, the number of actual parameters (arguments) given by the module must be equal to the number of formal parameters specified by the production predecessor, and the condition of the production must be met. The production given above would match a module $\texttt{A(4,3)}$, since the symbol and the number of formal parameters of the production predecessor match the symbol and the number of arguments of the module, and the condition evaluates to *true* for the given arguments. After evaluation of the parameters in the successor, the module is replaced by the resulting string $\texttt{B(8)CD(16,1)}$. In this example, the module `C` does not have any parameters. It shall be noted that for drawing commands, parameters may have different meaning depending on the implementation. The commands `_` and `!` to increase and decrease line width for example are often assumed to both set the line width to a given value if a parameter argument is given.

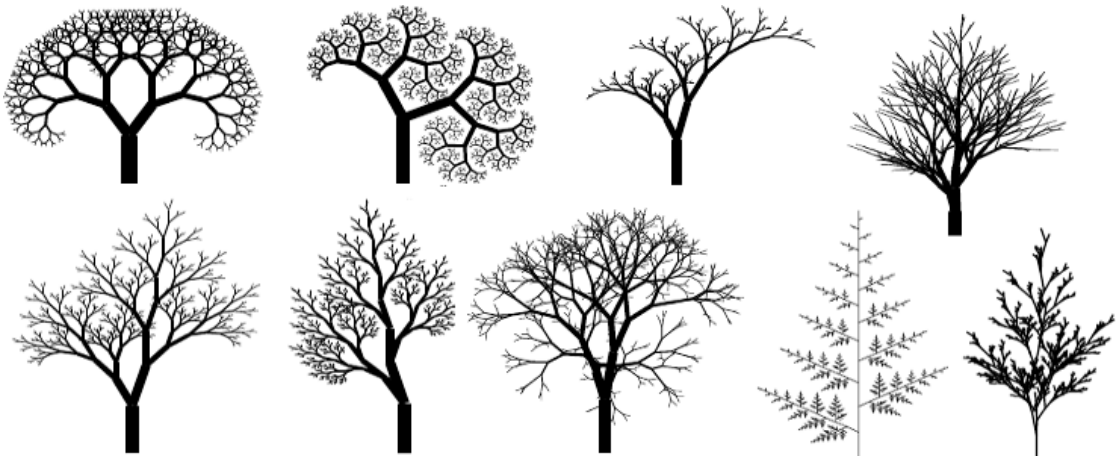


Figure 3.4: Variety produced from the following simple parametric L-system. Growing branches, or apices, are modeled by $\texttt{A(s,w)}$ with branch length `s` and width `w` gradually decreasing according to the factors $\texttt{r}_1, \texttt{r}_2$ and $\texttt{q}, \texttt{e}$ respectively. Branching angles are determined by $\alpha_1, \alpha_2, \varphi_1, \varphi_2$. Image taken from Prusinkiewicz et al. [PHHM97b].

$$\begin{array}{lll} \omega : & \texttt{A(100,w}_0\texttt{)} & \\ p_1 : & \texttt{A(s,w)} \quad : \; \texttt{s} >= \texttt{min} & \rightarrow \; \texttt{(w)F(s)} \\ & & \quad [+(\alpha_1)/(\varphi_1)\texttt{A(s} * \texttt{r}_1\texttt{,w} * \texttt{q\^{}e)}] \\ & & \quad [+(\alpha_2)/(\varphi_2)\texttt{A(s} * \texttt{r}_2\texttt{,w} * \texttt{(1} - \texttt{q)\^{}e)}] \end{array}$$

3.5 Context-sensitive L-systems

The formalism thus far only accounted for the propagation of information via lineage, i.e. from a module to its successor. The exchange of information between neighboring modules in the structure (*endogenous interaction*) and with external structures simulating the environment (*exogenous interaction*) was not considered in the model. L-systems are however very well capable of modeling both types of interaction. This section shall explore the simulation of endogenous interaction via an extension to L-systems known as *context-sensitive L-systems* or interactive L-systems (IL-systems).

The capability of modeling the interaction between modules of a growing structure is an essential aspect of the biologically motivated formulation of L-systems. As discussed by Prusinkiewicz et al. [PHHM97a], it allows the description of complex dynamic phenomena such as the propagation of signals or growth hormones that control the morphogenetic development of a plant, the distribution and flux of resources (minerals, water, photosynthates) among the growing structure or the response of a plant to pathogens like certain insects. While such models are still far from biologically accurate descriptions of the processes mentioned, they allow visualizations of interaction for educative purposes and to test hypotheses by visual comparison of simulated models based on empirical data of the actual structures. The expressive power of context-sensitive L-systems clearly distinguishes them from purely geometric models.

A module in a string may have a left and a right context, given by the modules to its left and to its right. Depending on whether just one side or both are considered, context-sensitive L-systems are also referred to as 1L-systems or 2L-systems, respectively. The predecessor of a production in a 2L-system may specify a left and / or right context as strings of modules, and will be applied to a module if it matches the actual context of a module. The common syntax is $\texttt{lc} < \texttt{a} > \texttt{rc} : \texttt{C} \rightarrow \chi$, where the triplet $(\texttt{lc}, \texttt{a}, \texttt{rc})$ is the predecessor that includes the left and right context `lc` and `rc`, `C` is the condition, and χ the successor. `a` is called the *strict predecessor*, that is going to be replaced on production application. While the strict predecessor must be a single module, the left and right context may involve a string of modules.

A simple example is given below on the left, modeling the propagation of a signal `B` through the structure, eventually inducing the growth of, for example, a flower `C`. The result of the productions can be seen on the right:

$$
\begin{array}{lll}
\omega & : & \texttt{BAAA} \\
p_1 & : & \texttt{B} < \texttt{A} \rightarrow \texttt{B} \\
p_2 & : & \texttt{B} > \texttt{A} \rightarrow \texttt{A} \\
p_3 & : & \texttt{B} \rightarrow \texttt{C}
\end{array}
$$

```
BAAA
ABAA
AABA
AAAB
AAAC
```

As emphasized by Hanan [Han92], when considering context within branching structures, it is important to note that the left context corresponds to the modules "closer to the root" from the current module in the branching hierarchy and the right context to those "farther out along the branches" it. When matching the left context, the string is not just traversed from the module toward the left, but rather (considering the topology implied by the branches) toward the root or mother branches, skipping all those branches that grow further out from them. For example, the left context `EF` would match the string `EF[A][B[C]]` for the strict predecessor `A`, but also for `B`, where in the latter case `[A]` would be skipped. It would not however match the module `C` as it would require a left context of `EFB`. When matching the right context, the string is traversed from the module toward the right, including all subbranches, until the end of the branch the module itself is on, i.e. until the next unmatched right bracket or the end of the string. For example, the right context `T[K]L` matches the string `YT[KR[ZO]W]L` for the module `Y` where those parts `R[ZO]W` of the subbranch that are not required by the context are skipped. It would, however, not match the string `T[RK]L`, as `K` would be required to directly follow `T`. An example of context-sensitive matching in a branching structure is illustrated in Figure 3.5.

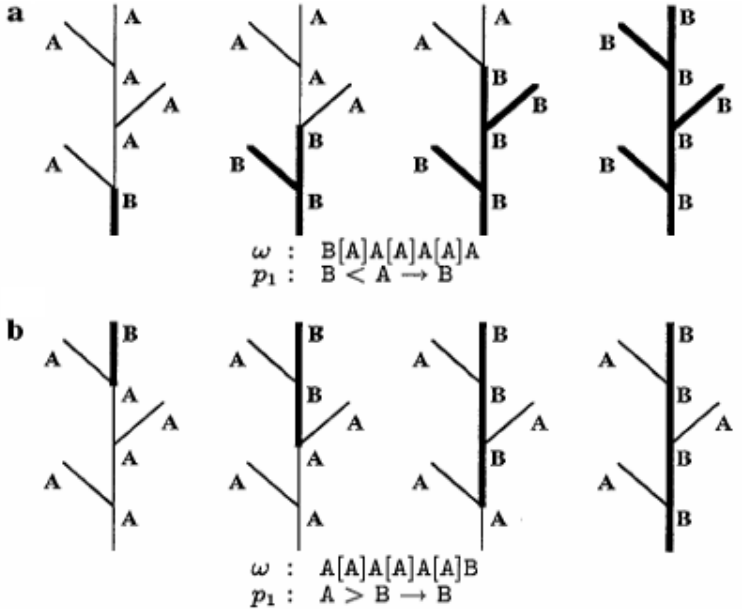


Figure 3.5: Propagation of a signal in a branching structure. Module `B` describes the parts of the structure that have been reached by the signal. In (a) the signal travels from the base to the branches (acropetally), where all modules `A` including the branches gradually match `B` as their left context or so to say their mother branch. In (b) the signal travels from the end of a branch to the base (basipetally), notably being ignored by the other branches as they never see `B` as their right context. Image taken from Hanan [Han92].

A more detailed example of modeling plant structures illustrating the capabilities of context-sensitive L-systems is given in Chapter 5.

3.6 Other Variants and Extensions

Various extensions to the L-system formalism exist beyond those covered so far. While not all of them can be demonstrated in this thesis, this section shall give a short overview over some of the other commonly known methodologies in the field. A good overview that includes other extensions is given by Prusinkiewicz [Pru99].

3.6.1 Shedding of Branches

Sometimes it can be useful to be able to remove a large number of different modules at once, without requiring individual productions for the removal of each one of them. Plants may shed branches or other organs at certain stages of their development, or loose them due to pruning. To facilitate the simulation of such processes, Hanan [Han92] introduced the *cut symbol* %. When encountered in a string, all symbols of the current branch (i.e. up to the next unmatched right bracket or the end of the string) would be skipped, and thus rendered invisible to production application. For example, the string `AB[C%DE[FG]H]I[%JK]L%MNO` would be read as `AB[C]I[]L` unless the cut symbols were removed.

An example of the application of the cut symbol is the modeling of palm trees. Growth in palms is limited to the extremities, the diameter of the trunk of palms does not increase by continuous outward growth. Thus an almost constant amount of nutrients and water transported in the stem only permits a more or less constant number of leaves in the crown. Palms thus shed old branches to be able to continue upward growth. As formulated by Prusinkiewicz et al. [PHMH95], the simple L-system below demonstrates this kind of shedding of lateral branches, visualized in Figure 3.6:

$$
\begin{array}{lllll}
\omega & : & \texttt{A} & & \\
p_1 & : & \texttt{A} & \rightarrow & \texttt{F(1)[-X(3)B][+X(3)B]A} \\
p_2 & : & \texttt{B} & \rightarrow & \texttt{F(1)B} \\
p_3 & : & \texttt{X(d)} \quad : \quad \texttt{d} > 0 & \rightarrow & \texttt{X(d} - \texttt{1)} \\
p_4 & : & \texttt{X(d)} \quad : \quad \texttt{d} == 0 & \rightarrow & \texttt{U\%} \\
p_5 & : & \texttt{U} & \rightarrow & \texttt{F(0.3)}
\end{array}
$$

3.6.2 Differential L-systems

So far the development of growing structures was described by L-systems in discrete steps of time: No consideration was given to the development between two successive steps of production applications. Motivated especially by the prospect of more realistic animations of the growth processes described, extensions to the formalism have been developed to enable the specification of development in continuous time. The most commonly used method in this respect was introduced by Hammel [Ham96], known as

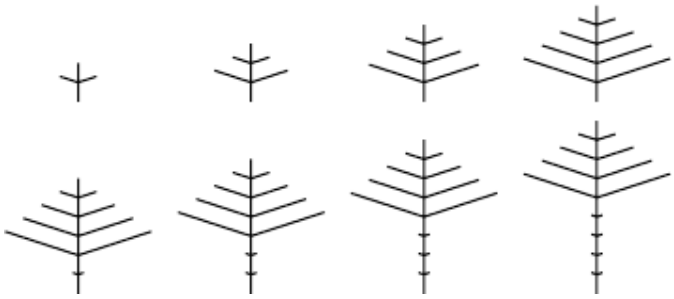


Figure 3.6: The apex of the main branch `A` grows upward and produces two lateral apices `B` that are preceded by breaking points `X`. After a number of steps `d` the breaking points are replaced by a cut symbol to shed the old lateral branches, leaving a small remaining scar. Image taken from Prusinkiewicz et al. [PHMH95].

differential L-systems. It is based on an approach that combines discrete and continuous aspects of simulation. Modules themselves are replaced in discrete events instead of discrete uniform time steps. The changes to the parameters resulting from production application may be applied in an interpolated way such that they vary continuously between the discrete events or may vary according to specialized growth functions.

3.6.3 Open L-systems and Environmental Interaction

While endogenous interaction can be simulated via context-sensitive L-systems, no mechanism has been introduced to cover *exogenous interaction*, i.e. interaction of modules with the environment. An early approach known as table L-systems employed the usage of different production sets or tables that would be exchanged in accordance to global environment factors, such as temperature or daylight. To cover the effect of the local environment on individual modules in the structure, such as due to different concentrations of nutrients in the soil or different density of radiation received, *environmentally-sensitive L-systems* have been developed. According to their original definition they are based on one-sided influence of the environment on the plant and do not yet simulate the reciprocal influence of a plant with its environment.

Finally, *open L-systems* were introduced by Mech and Prusinkiewicz [MP96] that allow the expression of plant-environment interaction in the framework of parametric context-sensitive L-systems. Open L-systems are based on a simulation of the growing structure and its environment as individual processes that develop according to their own kind of rules. The environment is typically simulated by algorithms distinct from L-systems. The two processes communicate via a standard interface that within the L-system formalism is accessed via special communication symbols that request the exchange of parameter values with the environment.

Chapter 5 will discuss some of the possibilities to simulate environmental interaction of parametric context-sensitive L-systems via integration of their production and graphical interpretation into the open-source computer graphics software Blender.

3.6.4 Genetic / Evolutionary Approaches

One last topic that shall be briefly mentioned is the use of genetic algorithms as known from search, optimization and machine learning problems. L-systems modeled by human designers typically undergo a process of artificial evolution, where changes are made to productions and the results evaluated to select the model that best fits the requirements. A first approach to automatically simulate this process of evolution, by inducing mutations to reach a greater variety of productions to choose from, was presented by MacKenzie [Mac93]. He demonstrated the use of common genetic algorithms to optimize L-system productions based on criteria such as the amount of light received by a growing structure and similar. Another method was introduced by Burt [Bur13], as a kind of Interactive Evolutionary Computation technique, where the user would guide the evolutionary process by interactively selecting from a visualized set of variations produced from the current productions, to be used as the starting point for the next round of mutations. Lastly it must be noted that, while these approaches are based on processes known from the development of genes via evolutionary processes, they operate on a highly abstract level far from the complexities of actual plant genetics.

CHAPTER 4

Integrating the L-Py Framework as an Add-on for Blender

One of the steps relating to the practical part of this bachelor thesis was to bring the expressive power and aesthetic beauty enabled by L-systems to Blender, an open-source 3D computer graphics software sustained by a broad community of users in many fields, with applications as diverse as animated films, architectural and scientific visualization, 3D printing and game design. [FtBC17c].

Designing a full implemention of L-systems from the ground up would be beyond the scope of this thesis and would not be viable, especially since the main goal of this thesis is to discuss possible advantages or disadvantages of an integration in Blender, not so much the details of possible implementations. Therefore, an existing implementation is used and adapted for use as an add-on for Blender. Regarding implementations for Python, the L-Py L-system simulation framework described in the next section can be considered the most feature-rich option available.

This chapter shall give a short introduction to the L-Py framework and the Blender Python API, and will then focus on describing the steps taken to implement *Lindenmaker*, a small wrapper add-on of L-Py for Blender, as well as giving an overview on its main features and user interface. This chapter also discusses potential advantages of L-Py integration in Blender from a more theoretical standpoint. To test this potential on practical examples, applications of the add-on for the modeling of the visual structure of plants as well as their interaction with a Blender scene will be the subject of Chapter 5.

Lindenmaker employs an adapted version of L-Py as the basis for L-string production, while aspects related to the graphical turtle interpretation are rewritten from the ground up via the Blender Python API. The core functionality of Blender, written in C and C++, can be extended via the Blender Python API, allowing access to most operators and data that is accessible to the user via the GUI. Extensions can be made in the form

of small Python scripts up to full-fledged add-ons that may be shipped with the offical Blender release.

The software source code for both the adapted version of L-Py as well as the Lindenmaker add-on is publicly available under an open source license on GitHub: https://github.com/mangostaniko/lpy-lsystems-blender-addon. [Leo17]

To summarize, the steps taken to integrate parametric context-sensitive L-systems in Blender can be outlined as in the following:

- Adapt the L-Py L-system framework so that it can be used as a Python module in the Blender Python interpreter
- Write a Blender add-on that forms a wrapper around the L-string production capabilities of L-Py and offers graphical turtle interpretation using the Blender Python API, including special functionality related specifically to advantages of integration in Blender, most notably use of modeled objects and environmental interaction with the Blender scene.

Note that to avoid ambiguity in the context of programming languages, the term L-string is used throughout this section to refer to the string of modules an L-system operates on.

4.1 L-Py: An open-source L-system Framework for Python

L-Py is an open-source framework and library for the simulation of L-systems based on Python, developed by Boudon et al. [BPC+12].

Prior to L-Py, numerous L-system implementations have been developed, most of them based on statically typed languages. One of the most significant contributions was the *pfg* (plant and fractal generator) by Prusinkiewicz and Hanan, which was the central part of the Virtual Laboratory plant simulation framework described by Mercer et al. [MPH90], and its successor *cpfg* (pfg with continuous parameters) by Hanan [Han92]. Pfg and cpfg first allowed the use of general programming language constructs in the specification of parametric context-sensitive L-systems, including local and global variables, functions and flow-control statements [PHM99]. Notably, pfg and cpfg did not build upon an existing framework for an imperative language, but rather used their own implementation of an interpreter for C-like language constructs to extend the formalism of L-systems. For simplicity, only a subset of features of the C language are included in the cpfg language.

A different approach was taken by Karwowski and Prusinkiewicz [KP03] in developing the L+C language and the corresponding lpfg simulation framework. The L+C language is fully based on C++ and uses features of the language itself, like classes, libraries etc. to extend its syntax to the formalism of L-systems. L+C thus has the same expressive power as the C++ language and allows for more complex L-system constructs such as

module parameters with compound data types (structures), as described by Karwowski et al. [PKL07]. Both cpfg and lpfg are part of the current implementation of the Virtual Laboratory suite, which is also distributed with a different user interface under the name of L-studio [Pru04]. Another notable contribution was made by Kniemeyer and Kurth [KK08] with *XL* (eXtended L-system language), which is based on the interpreted Java language, focusing on cross-platform portability. All frameworks mentioned so far are however still based on statically typed languages, contrary to the dynamically typed Python language employed by L-Py.

The core of L-Py, as described by Boudon et al. [BPC+12], is itself written in C++ for improved performance of compiled binaries. To benefit from the flexibility, simple syntax and concise high-level expressions of the dynamically typed and interpreted Python language, the core features are exposed as a Python module via the Boost.Python library, which can provide an almost seamless mapping of the most relevant C++ constructs to Python constructs. Apart from the considerably more shallow learning curve and great expressive power of Python, another significant advantage of its dynamic nature is that once the L-Py C++ framework itself has been compiled to a Python module, users of the framework are not required to compile their L-system definitions prior to use.

The L-Py interpreter compiles the code of an L-system model (typically stored in a file with extension `.lpy`) to pure Python code via the L-Py parser. Predecessors are stored in dedicated data structures, and rules are reformulated as Python functions. The interpreter then iteratively applies production rules to the previously produced L-string, starting with the axiom. To this end, the L-string is parsed and scanned for modules that match a predecessor in any of the defined productions by module name and number of parameters. Once a match is found, the respective production is applied by calling its corresponding Python function and appending the string of modules it returns (the successor of the production) to the L-string resulting from the current production application step. Thanks to the dynamic language evaluation of Python, introspection and runtime modification of L-system models is possible.

The L-Py source code and precompiled binaries are made available as part of the OpenAlea distribution developed at INRIA. OpenAlea comprises as set of tools related to plant architecture analysis and modeling [dReIeeA14]. For all aspects of graphical turtle interpretation, L-Py relies on the PlantGL library, which is also part of the OpenAlea suite. While L-Py is very well suitable and convenient to use as a component library in other Python projects, L-Py and PlantGL exhibit a deliberately strong coupling as noted by Boudon et al. [BPC+12]. This dependency is hard-coded to an extent that makes it difficult to build the L-Py core as a standalone module without aspects related to graphical turtle interpretation of PlantGL, unless making changes to parts of the core itself and thereby breaking compatibility.

Similar to cpfg and lpfg, the L-Py interpreter has also been embedded into its own integrated development environment. The L-Py IDE features a code editor with syntax highlighting, an editor for parameters related to graphical turtle interpretation, visual editors to create and manipulate objects like curves and patches to be used for the

interpretation of modules, an interactive 3D view of the model resulting from graphical turtle interpretation, a Python shell as well as a specialized debugger and profiler.

The syntax of L-system related elements of L-Py is, for compatibility, very similar to that of cpfg and L+C, with some aspects being simplified due to dynamic typing. An example L-system definition in L-Py is given in the following, with rendered graphical interpretation results shown in Figure 4.1:

Listing 4.1: L-Py L-system Code Example

```
1  MAX_AGE, dr = 10, 0.02 # constants
2  Axiom: A(0)
3  derivation length: 10
4
5  production:
6  I(l,r) --> I(l,r + dr) # Internode
7  A(age):                # Apex
8    if age < MAX_AGE:
9      produce I(1,0.1)/(137.5)[+(40)A(0)]A(age+1)
10
11 homomorphism:
12 I(l,r) --> F(l,r)
```

The L-Py code in this example can be seen to contain standard Python elements, like variables (here used for constants), comments and control flow statements. It is important to point out that Python code may be used in various ways to introduce additional structural diversity to the model, e.g., via the Python random module. Syntactical and semantic elements related to L-Py that can be observed from the example include the `Axiom` keyword followed by the axiom definition and the `derivation length` keyword to specify how often productions are to be applied. Lines following the `production` keyword relate to production rule definitions. A production can be written in two ways: Following the cpfg syntax `Predecessor --> Successor` or by separating the predecessor via a colon from more involved instructions of regular Python combined with the keyword `produce` to indicate potential successors. Context-sensitive predecessors can be defined via `LeftContext < StrictPredecessor > RightContext`. Notably, not all modules need to correspond to valid commands for graphical turtle interpretation. The `homomorphism` keyword is used to specify rules to replace abstract module names with defined turtle commands in a postproduction step.

4.2 The Blender Python API

The core elements of the open-source 3D graphics software Blender are written in C and C++. The codebase, however, also includes elements of Python, which are related mostly to UI definitions but also to a handful of essential internal tools. An integral aspect of Blender is that it comes with a built-in Python interpreter and its own bundled Python

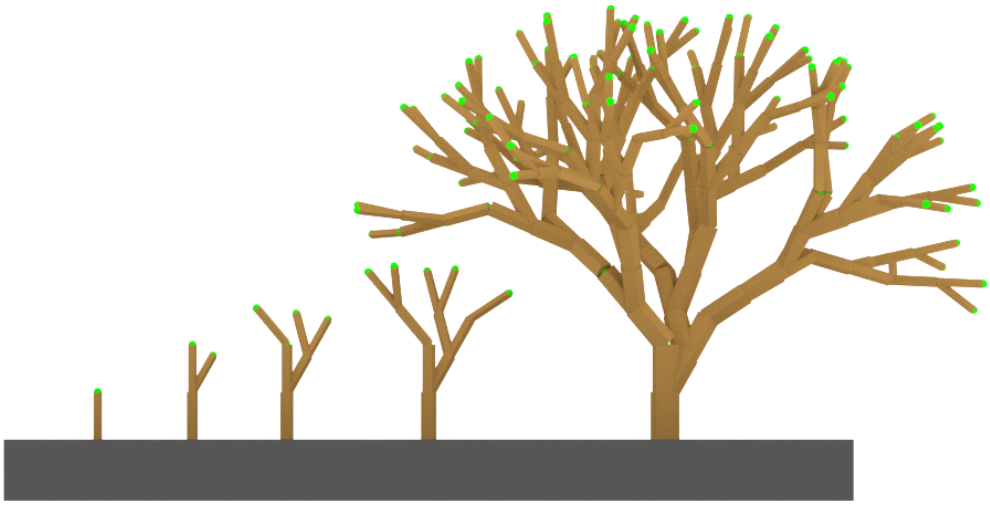


Figure 4.1: The results obtained from the L-system specified in Listing 4.1 using L-Py after 1, 2, 3, 4 and 10 production steps, respectively, after applying graphical interpretation via the Lindenmaker add-on in Blender. The geometry associated with the apices is highlighted in green. In each step, each Apex module is replaced by an Internode module with two new Apices at its end, one growing straight on to continue a branch until a certain age is reached and one growing in a new branch at a slight angle.

environment. The built-in interpreter is exposed to the user via an interactive Python shell embedded into the UI.

The Blender Python API [FtBC16] is implemented as a Python module called `bpy`, which is provided by Blender to expose its internal data and operators to the Python environment. This allows the flexible and modular extension of Blender's core functionality via simple Python scripts without having to recompile the core C/C++ code.

One of the most important motivations for the Blender Python API is its application for the development of add-ons. An add-on is simply a Python script or module that defines specific metadata via the `bl_info` dictionary and defines callbacks for its UI elements to be registered and unregistered via the `bpy`. When an add-on is placed in the add-ons search path it is listed in and can be enabled or disabled via the Blender User Preferences.

A central aspect the `bpy` module offers is the *context* structure `bpy.context` that exposes the current state of the open Blender scene.

For more complex operations instead of directly altering context, so called *operators* should be used. Operators are a core design convention of Blender, to bundle functions that modify a scene's context into coherent units that can be accessed and run with given parameter arguments via the graphical user interface, that are integrated into the state history to allow undoing as well as to Blender's event handling system.

Among many other things, the API also exposes the creation of custom *panels* in the Blender user interface, to graphically represent a diversity of input elements that may directly expose properties of the scene context among others to be modified by the user via these UI elements.

For more detailed documentation refer to the official Blender API documentation pages [FtBC16].

4.3 Lindenmaker: A Small Add-on for L-systems in Blender via L-Py

4.3.1 Implementation Considerations

The first step in the development of the Lindenmaker add-on for Blender was to consider how existing software could be reused to avoid reinventing the wheel. L-Py was developed with this in mind and is available under an open-source license.

Not all parts of L-Py would be fit for reuse however. Lindenmaker thus aims to form a light wrapping layer around L-Py and uses it as a Python module for the production of parametric, context-sensitive L-strings. Notably the framework allows to express far more than these two aspects: stochastic L-systems, for example, can be implemented simply by using the Python random module within the L-system definition code.

Aspects related to graphical turtle interpretation had to be rewritten from the ground up for Blender via the Blender Python API, thus not relying on the PlantGL framework used by L-Py for turtle interpretation.

This section gives an overview over the implementation steps and design decisions involved in the Lindenmaker add-on. These steps can roughly be outlined as follows:

- Adapt L-Py to be usable in the Blender Python interpreter, and thus by the Lindenmaker add-on
- Implement graphical turtle interpretation tailored specifically for environmental interaction with a Blender scene, using the Blender Python API.

Adapting L-Py for use as a Python Module in Lindenmaker

An important design consideration was the question of whether it would be possible to use an L-system production framework built entirely in Python, to avoid any need for compilation, relying only on code directly interpreted by Python. This would be advantageous especially for ease of use and installation, requiring only the placement of the Python module into the respective Blender add-ons directory, as well as for ease of maintenance. However, since L-Py is written in C++ and only uses a Python wrapper via Boost.Python, it still requires compilation. It was considered to completely rewrite the L-Py L-system production framework in Python based on the existing open-source

C++ code, but this was deemed unrealistic given that L-Py consists of over 18000 lines of code. Notably, run-time performance of an entirely interpreted production system would likely be significantly worse compared to the compiled C++ code via the Boost.Python wrapper.

Using the existing C++ and Boost.Python wrapper code of L-py as an L-system production framework, it soon became evident that a number of adaptations would be required. Most notably, the Boost.Python aspects of L-Py were written for Python 2.x as a target, but Blender has migrated to Python 3 with version 2.5 and has since dropped support for Python 2 [FtBC09]. Moreover, being part of the OpenAlea suite, L-Py relied on a build process involving custom build tools that would first have to be build themselves in order to build L-Py and PlantGL. Noting that the last substantial changes to the L-Py code base date several years back [dReIeeA16b], the decision was taken to avoid trying to maintain compatibility with the original L-Py source, and build a custom version for Python 3.5 with dependencies to PlantGL removed (the two frameworks as mentioned being rather strongly coupled) and using the well documented and widely supported CMake for the build process.

To get an idea on which parts of L-Py would need adaptation, especially regarding the removal of dependencies to unused libraries such as to PlantGL for graphical interpretation code, the first step was to get an overview over the L-Py code base and study its structure. While the code is very clearly structured, there is an evident lack of documentation in the code itself but more importantly also of the L-Py API on the web [dReIeeA16a]. However, it is still possible to identify parts dependent on the PlantGL library via extensive use of text search tools like `grep` and inspecting relevant header files. Most dependencies on PlantGL can be removed via simple `#ifdef` C++ preprocessor directives. Some of the more general PlantGL code that L-Py heavily relies on (like utility math structures and functions) had to be integrated into the L-Py core, given that compatibility to the original L-Py is not a requirement as mentioned before.

Regarding adaptions of the Boost.Python wrapper for Python 3, `#ifdef` preprocessor directives were used to conditionally select statements depending on whether Python 2 or Python 3 libraries were being used. The differences in Python versions are mostly subtle changes in function names which can be hard to find without using tools to inspect symbol references in the resulting object files [Fou09]. L-Py adaptation also raised a number of other minor issues, such as the use of double underscore names in the L-Py code, which for the GNU C++ compiler are typically only allowed for reserved names.

Using L-Py as a Python Module in the Blender Python Interpreter

Once the L-Py Python module is built as a shared library (`.so` on UNIX or `.dll` on Windows) it can be readily imported to be used by the Python interpreter. Usage of L-Py is very straightforward. After importing the module via `import lpy` an Lsystem object can be created from a source `.lpy` L-system definition file via a simple command `lsystem = lpy.Lsystem("/path/to/lpyfile.lpy")`.

A call to `lsystem.derive()` will trigger the application of production rules to the axiom for the specified number of derivation steps and return the result in the internal `AxialTree` representation, which can be converted directly to a string to yield the resulting L-string. The member function `lsystem.interpret()` can be used to apply homomorphism substitution rules.

To use the L-Py module in Blender's Python interpreter, it should be placed within its search path [FtBC17a]. This is required for the Lindenmaker add-on to make use of the L-Py framework for L-system productions directly within Blender.

Implementing Turtle Interpretation via the Blender Python API

To provide graphical turtle interpretation in Blender, all functionality related to this aspect of the Lindenmaker add-on relies on the Blender Python API, instead of the PlantGL library originally used by L-Py.

The central components for drawing, i.e. placement of 3D geometry in this context, are formed by a Python class called `Turtle` and a subtype of it called `DrawingTurtle`.

`class Turtle:`

- stores turtle transformation matrix
- implements movement and branching turtle commands
- allows branching hierarchy to be inspected via Blender scene manager

The `Turtle` class implements basic turtle interpretation commands related to movement, i.e., translation, rotation, look at, as well as branching. The state of the turtle relating to its orientation in space is stored via a single 4D matrix, which is manipulated for movement via multiplications with basic transformation matrices. Branching is implemented via a stack data structure to which turtle states can be pushed to on branch initialization as well as retrieved from on branch termination. This turtle is used where draw commands are not necessary. It can create a branching hierarchy tree of objects, which only store transformation information, but have no mesh information associated with them that could be visualized via rendering and rasterization. The root of this hierarchy tree is formed by an object labeled "Root" in the Blender scene. The hierarchy of objects produced is managed via the Blender object parenting system, and can thus be inspected and modified in the Blender scene after graphical interpretation.

`class DrawingTurtle:`

- stores turtle state and configuration related to drawing
- implements turtle drawing commands using 3D mesh instances

The `DrawingTurtle` subtype extends the simple turtle by providing various methods implementing the draw commands related to graphical interpretation. Its state is extended by a number of fields related to drawing, such as internode width and length as well as material index. Drawing in terms of the Blender Python API module `bpy` corresponds to the creation of 3D objects with the current turtle matrix state that also include mesh information. For the standard draw command `F`, which is used

to draw an internode of a plant structure, instances of a predefined simple cylinder mesh named "LindenmakerDefaultInternodeMesh" are created, which all share the same mesh data to avoid redundancy. If desired, another predefined mesh named "LindenmakerDefaultNodeMesh" can also be used to draw simple spheres to indicate branching points. Both meshes can be customized or replaced entirely – the same being the case for colors and materials used for rendering – as described in the following User Interface Overview subsection. Additionally, objects associated with instances of a custom mesh defined in the Blender scene can be generated in the current turtle matrix state via the ~ command, which will be discussed in more detail in Section 4.4.

`function turtle_interpretation.interpret():`

- parses L-string produced by L-Py
- splits L-string into individual commands with associated parameters
- forwards commands to `Turtle` member functions
- applies branch cut commands

The turtle classes just described interface closely with the `bpy` module, but are independent of the L-Py library. Calls to the turtle are done mostly by the function `turtle_interpretation.interpret()`, which parses a simple input string representing the L-string produced by L-Py or any other L-system production framework, and interprets it graphically according to drawing parameters specified by the user via the Lindenmaker UI. This function uses Python's regular expression functionality to split the L-string into the individual interpretation commands together with their associated parameters contained in parentheses, and calls the respective Turtle methods for valid commands and parameter values. Additionally, it applies the cut branch command % on the L-string level before turtle interpretation. This is done by removing all command symbols following the cut command symbol until the end of the branch, i.e., until the next unmatched closing bracket or the end of the string.

`class LindenmakerPanel:`

- provides main Lindenmaker user interface

The interface of Lindenmaker to the user is implemented via the `LindenmakerPanel` class. This is a subtype of `bpy.types.Panel`, which is one of the main ways of specifying UI elements through the Blender Python API. The user interface will be described in more detail in the following subsection.

`class LindenmakerOperator:`

- used to embed Lindenmaker functionality into the Blender event handling system
- required to provide undo/redo capabilities

Buttons specified by the `LindenmakerPanel` are used to evoke methods of another class named `LindenmakerOperator`. This is a subtype of `bpy.types.Operator`, which, as previously described, is a central aspect of the Blender Python API to achieve modularity. Essentially, it is used to integrate custom functionality that change a Blender scene's state into the state history to allow undoing as well as to Blender's event handling system in a unified way. The `LindenmakerOperator` not only makes calls to the

`turtle_interpretation.interpret()` function, but also interfaces with the L-Py library module. It takes the input `.lpy` L-system specification file and calls the L-Py `lsystem.derive()` method to apply a given number of production steps according to production rules to the current L-string, and then calls `lsystem.interpret()` to apply L-string substitutions according to homomorphism rules specified in the `.lpy` file. The latter function, however, does not do any graphical turtle interpretation. It is used only to translate abstract L-string symbols into actual commands relating to graphical turtle interpretation. Graphical interpretation is done as described via the custom `turtle_interpretation.interpret()` function provided by Lindenmaker, which should thus not be confused with `lsystem.interpret()`.

4.3.2 User Interface Overview

To use the Lindenmaker add-on, it is installed by being placed in the add-ons search path, and enabled from the user preferences panel. When it is enabled, a new tab labeled `Lindenmaker` is apparent in the `Tool Shelf` panel of the `3D View` area, as seen on the left in Figure 4.2. This is where all UI elements of Lindenmaker reside, which shall be explained in the following.

[filepath] L-Py File: Path of .lpy file containing L-system definition.

[float] Default Turtle Step Size: Default length value for `F` (move turtle and draw) and `f` (move turtle) commands if no arguments given.

[float] Default Rotation Angle: Default angle in degrees for turtle rotation commands if no argument given.

[mesh] Internode: Name of mesh to be used for drawing internodes via the `F` command. Default is “LindenmakerDefaultInternodeMesh", a cylinder mesh generated at first use.

[float] Default Line Width: Default width of internode and node objects drawn via `F` command.

[float] Width Growth Factor: Factor by which line width is multiplied via **;** (width increment) command. Also for **,** (width decrement) command as $1-(factor-1)$.

[float] Internode Length Scale: Factor by which move step size is multiplied to yield internode length. Used to allow internode length to deviate from step size.

[checkbox/mesh] Nodes: If enabled, the selected node mesh is drawn at branching points. If not enabled uses Empty objects if hierarchy is used. Default is “LindenmakerDefaultNodeMesh", a simple icosphere mesh generated at first use.

[checkbox] Recreate Default Internode / Node Meshes: If enabled, recreates “LindenmakerDefaultInternodeMesh” and “LindenmakerDefaultNodeMesh", according to attribute values specified via the following fields.

[integer] Default Internode Cylinder Vertices: Number of base circle vertices for default cylinder “LindenmakerDefaultInternodeMesh".

[integer] Default Node Icosphere Subdivision: Number of subdivision steps for default icosphere “LindenmakerDefaultNodeMesh".

[checkbox] Force Flat Shading: Force flat shading for all parts of the generated structure.

[checkbox] Single Object (No Hierarchy, Faster) If enabled, generate a single object with a single joined mesh. If disabled, generate a branching hierarchy of objects. Having this option enabled is significantly faster.

[checkbox] Remove Last Interpretation Result: If enabled, the last object labeled “Root” generated from the previous interpretation is replaced. Useful for stepwise production and interpretation, to avoid cluttering the scene.

[button] Add Mesh via Lindenmayer System: Do the whole process from L-system definition to graphical interpretation. Pass .lpy file to L-Py to produce L-string. Apply production rules as many times as specified in file, then apply homomorphism substitution rules. Finally create a graphical interpretation of the L-string based on the UI options.

[button] Clear Current L-strings: Clear the currently stored L-strings. Note that while the L-system yields just one L-string, two copies of the L-string are stored: One used for production and one used for graphical turtle interpretation. See below for details.

[button] Apply One Production Step: Apply a single production step to current L-string, or to the L-system axiom in the first production step. No graphical turtle interpretation is done.

[textfield] L-string for Production: The produced L-string used for further stepwise production. Manual modifications are possible by editing the textfield contents directly, usage of copy/paste to edit externally is however more convenient, as L-strings tend to become very long with increasing number of production steps.

[textfield] Homomorphism (For Interpretation): The same produced L-string but with homomorphism substitution rules applied (if given), used for graphical turtle interpretation. “Homomorphism” is a L-Py feature intended as a final postproduction step to replace abstract module names with actual interpretation commands. In L-Py these rules are preceded by the keyword `homomorphism:` or `interpretation:` (including the colon). Once homomorphisms are applied the L-string can no longer be used for stepwise production, thus two L-strings have to be stored.

[button] Interpret L-string via Turtle Graphics: Interpret current L-string via graphical turtle interpretation. No production steps are applied.

[button] Produce Step and Interpret: Apply a single production step and interpret.

The result of the graphical interpretation is generated as an object labeled “Root", which is displayed in the central `3D View` of the Blender user interface, as seen in

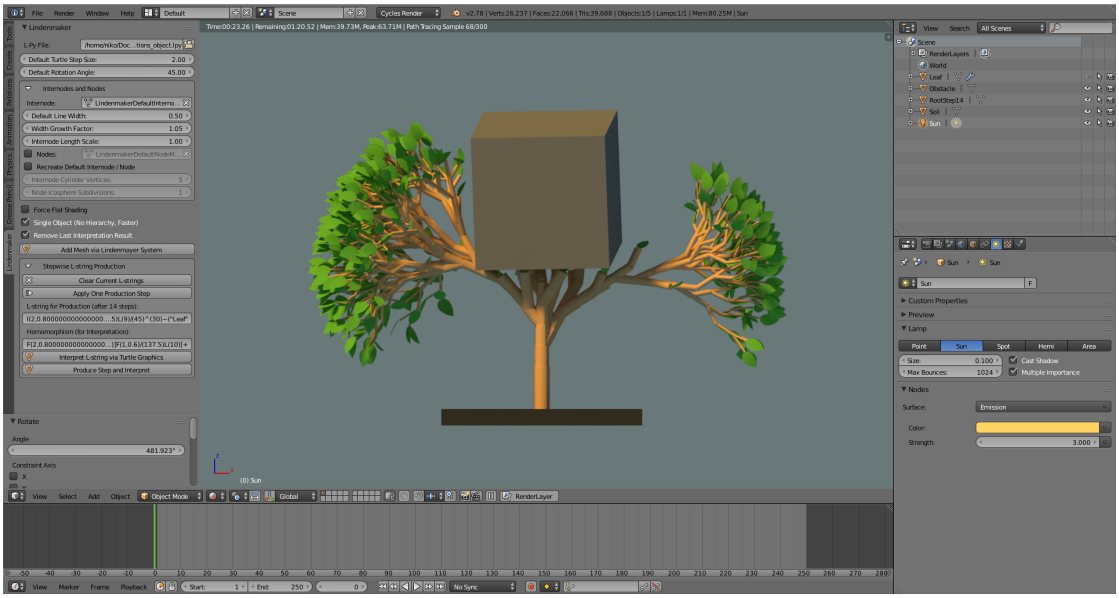


Figure 4.2: Blender User Interface with the Lindenmaker panel on the left. Results of graphical interpretation are displayed in the central `3D View`, where the generated objects can also be transformed and the corresponding meshes can be edited. Object hierarchies produced (unless configured to produce a single object) can be inspected via the `Outliner` area in the top right, materials can be edited via the `Material` panel of the `Properties` area in the bottom right.

Figure 4.2. The `3D View` also allows for the generated objects to be transformed and the corresponding meshes to be edited (for detailed documentation of basic Blender usage refer to the Blender Manual [FtBC17b]). If the checkbox `Single Object` is enabled, this is a single object with all interpreted plant modules merged into a single mesh. When disabled, the result will be a hierarchy of objects with individual meshes, which can be individually selected via the `Outliner` area in the top right of the Blender user interface, but will take significantly longer during graphical interpretation.

Materials used by the objects produced can be edited in the `Material` panel of the `Properties` area. Lindenmaker adds material slots to the default internode object (or for corresponding mesh polygons if a single object is produced) and alphabetically assigns existing materials to them from the material list of the Blender scene depending on the currently set turtle material index. The idea is that these generated material slots can then be edited after interpretation, when the result is inspected.

4.4 Advantages of L-Py Integration in Blender

The integration of L-Py in Blender via the Lindenmaker add-on is motivated by several advantages both during production and in further modeling and visualization after

production. While examples of the applications of these features are given in the following chapter, this section shall give a more abstract overview and especially introduce some of the specific new interpretation modules in Lindenmaker that can be used to model environmental interaction.

Some of the advantages in the post-production stage include:

- direct integration with the Blender path-tracing renderer Cycles, allowing for the realistic simulation of light propagation in the modeled structure
- access to modifiers and sculpting tools to further improve a model after production
- possibility to visualize the hierarchy of branches in a growing structure via the Blender parent-child system
- integration with simulation tools like particle system, wind simulation etc., allowing for example the modeling of a field of flowers

Of greater interest, however, are the possibilities during L-Py production, most notably the following:

- access to mesh and interpolated curve modeling tools to model custom objects, such as specific parts of a plant, that can be referenced in L-Py production
- interaction with a Blender scene via direct access to Blender structures during L-Py production, together with Lindenmaker *turtle state query* and *look at* modules, to model environmental interaction

The inclusion of custom-modeled objects is an essential feature of most modern L-system frameworks. In Lindenmaker, it follows the conventional syntax and is done via the interpretation module ~, which takes as arguments the referenced object's name in double quotes as well as optional scaling factors for uniform scaling or scaling along individual local turtle coordinate directions. Examples would be `~("Flower")`, `~("Flower", 0.5)` and `~("Flower", 0.2, 0.3, 0.4)`.

The most notable advantage of the integration with Blender is that it allows an interaction of the growing structure with its environment similar to the approach of open L-systems discussed in Section 3.6.3. This interaction is facilitated by the fact that the Blender Python API, being based on Python just like L-Py, may be used to access information about the currently open Blender scene directly during production. This is achieved by importing the `bpy` module directly in the L-Py code via `import bpy`, allowing access to `bpy.context.scene` and similar structures. Without Blender integration, it is not straightforward to integrate production rules into an environment as complex as a Blender scene.

To use the current state of the turtle to influence production, Lindenmaker in analogy to the original L-Py offers a query command `?`. This command can be used to query any of the turtle matrix column vectors (heading, left, up, position) via `?("H"|"L"|"U"|"P",0,0,0)`, e.g. for the turtle position vector `?("P",0,0,0)`. The three zeros `0,0,0` are placeholders that will be replaced by the queried vector x,y,z values. To use the queried values, a production rule for the `?` module should be added, as demonstrated by the following example:

Listing 4.2: Lindenmaker Query Module Example

```
1 production:
2 A: # Apex
3     produce ?("P",0,0,0)I/(137.5)
              ?("P",0,0,0)[+(40)?("P",0,0,0)A]A
4
5 ?(vector,x,y,z):
6     if vector == "P":
7         print("turtle position: {} {} {}".format(x,y,z))
8         if y > 5:
9             produce ~("Flower")
```

In this rather arbitrary example, multiple query modules are inserted by the Apex module. Lindenmaker will replace the `0,0,0` placeholders with the actual x,y,z values of the queried vectors. The production rule for the query module can then be used to replace the query module with something else, depending on the vector values which are given as parameter arguments. Here, positional queries are replaced by a flower object if the y value of the position vector is bigger than 5. By comparing the turtle state to other variables from `bpy.context.scene`, this allows the modeling of environmental interaction of plants with a Blender scene.

Another new turtle interpretation command added in Lindenmaker is the *look at* command `@(x, y, z)`. This command will reorient the turtle so that the heading vector is pointing toward the given target position x, y, z, while the heading, left and up vectors are guaranteed to have the same relative orientation (handedness) as before. This can be used together with Blender scene information via the `bpy` module, to orient a growing structure toward an object, e.g. toward a light source to simulate phototropism. The Python `random` module can be used to add random deviation from the look at direction.

It is important to note that some of the customizations necessary to adapt L-Py to Blender mean that some of the original L-Py interpretation modules can not be used via the Lindenmaker add-on, and vice-versa. This partial break of compatibility is however deemed acceptable since L-Py appears to be no longer actively maintained [dReIeeA16c].

CHAPTER 5

Modeling Plant Structures via L-systems in Blender

This chapter will show some examples of what can be achieved using the Lindenmaker add-on to integrate L-Py L-system production in Blender.

First, some results of the graphical interpretation via the Lindenmaker add-on will be given, focusing on well-known L-system models of plant inflorescence. Moreover, some of the advantages of having the full feature set of a 3D computer graphics suite like Blender at hand are shown, such as the use of particle systems and simulation tools, e.g., to model a field of flowers. All results will be rendered directly in Blender using the Cycles path tracer, which is also a notable advantage of this integration.

More detailed consideration, however, is given to another main advantage of Blender integration for L-system production. It is important to note that this integration not only enables the direct graphical interpretation, i.e., the creation of 3D meshes, in a widely used modeling tool: It also means that L-System production code, being based on Python, can have access to the whole context of a Blender scene via the Blender Python API. Having access to the structures of a Blender scene during production is very useful to model environmental interaction. Special attention will thus be given to this aspect, which is one of the main motivations to integrate L-system production into a general purpose modeling tool like Blender.

5.1 Modeling Herbaceous Plants and Inflorescence

5.1.1 *Mycelis muralis*

One of the more well-known examples of an L-system used to model plant inflorescence is that of *Mycelis muralis* (wall letuce), introduced by Janssen and Lindenmayer [JL87] and

popularized by Lindenmayer and Prusinkiewicz [PL96]. Notably, it is a good example for the modeling of signal propagation in a plant, in this case of a signal triggering inflorescence.

The flowering sequence of *Mycelis* is basipetal, i.e., flowers start to develop starting from the top, toward the bottom. Additionally, the plant exhibits acrotonic shape in some of its intermediate developmental stages, meaning that branches further up are more developed than lower ones, at some points in time. These two aspects make the modeling of *Mycelis* an interesting task, and one well suited for context-sensitive L-systems, to model signal propagation.

The model presented here in Listing 5.1 is based on the original model by Janssen and Lindenmayer [JL87], with adaptations regarding the graphical interpretation via Lindenmaker in Blender. The result of the graphical interpretation can be seen in Figure 5.1.

Listing 5.1: L-Py / Lindenmaker L-system Code for *Mycelis muralis*

```
1  Axiom: I(25)FA(0)
2  derivation length: 100
3
4  # ignore certain symbols in context check
5  ignore: +/
6
7  production:
8  S < A(t) --> T(0)O(t)
9  A(t) :
10     if t > 0:
11         produce A(t-1)
12     elif t == 0:
13         produce [+(30)[+(30)L]G]F/(90)A(2)
14 S < F --> FS
15 F > T(c) --> T(c+1)FU(c-1)
16 U(c) < G --> I(c)FA(2)
17 I(c) :
18     if c > 0:
19         produce I(c-1)
20     elif c == 0:
21         produce S
22 S --> *
23 T(c) --> *
24
25 homomorphism:
26 G --> F
27 L --> ~("Leaf")
28 O(t) --> ~("Flower", 0.3*t+0.7)
```

In this L-system, the axiom is given by an initial internode `F`, the main apex `A` as well as the module `I`, which is used to model the time-delayed release of a plant hormone known as florigen, here represented by the module `S`, which as the name implies, is responsible for inducing inflorescence. Initially, the main Apex produces subapical segments growing upwards, producing lateral leaves as well as lateral apices for branches, however the growth of these is initially inhibited. After a certain delay, the florigen signal travels acropetally (toward the top) until it reaches the main apex, which is then replaced by a flower, terminating growth on the main axis. It is important to note that in order for the signal module to propagate, certain modules used for interpretation must be ignored in the context check, which is done using the L-Py keyword `ignore`.

Termination of the main apex also releases another signal `T`, traveling basipetally (toward the base), which triggers the development of the lateral apices, which happens homologous to that of the main apex. In this way, the inflorescence continues downwards. Importantly, the parameter `c` is used to model the number of segments produced by a lateral branch apex before the florigen signal terminates its growth and produces a terminal flower. This number of segments increases toward the base, thus eventually producing a final basitonic shape.

Figure 5.1: The results obtained for the L-system of *Mycelis muralis* specified in Listing 5.1 after 1, 10, 41, 60 and 100 production steps, respectively, rendered in Blender. The flowers and leaves are custom meshes modeled in Blender, referenced directly by Lindenmaker for graphical interpretation using the Blender Python API.

While this is a classical example to demonstrate the capabilities of context-sensitive L-systems, the integration of L-Py production directly in Blender via Lindenmaker has some advantages: For one, the resulting structure can be rendered directly using the Blender Cycles path tracer using diffuse and slightly translucent surface scattering distribution functions. Moreover, the flowers and leaves are custom meshes modeled in Blender that are directly referenced by Lindenmaker during graphical interpretation, thus making it easy to introduce adjustments to these flower organs directly using a highly capable mesh modeling tool like Blender.

5.1.2 *Lychnis coronaria*

A fairly different model of plant inflorescence is given by the following L-system for *Lychnis coronaria* (rose campion). This model does not rely on signal propagation to control development of plant organs and growth of branches, and is arguably less intricate than the previous one. It was formulated by Robinson [Rob86] and was also included in well known publications by Lindenmayer and Prusinkiewicz [PL96]. The L-system shown in Listing 5.2 is an adapted version of the original. The result of the graphical interpretation can be seen in Figure 5.2.

Listing 5.2: L-Py / Lindenmaker L-system Code for *Lychnis coronaria*

```
1  Axiom: A(7)
2  derivation length: 30
3
4  production:
5  A(t) :
6      if t == 7:
7          produce FI(20)[+(20)L(0)]/(90)[&(30)A(0)]
                   /(90)[+(20)L(0)]/(90)[&(30)A(4)]FI(10)K(0)
8      elif t < 7:
9          produce A(t+1)
10 I(t) :
11     if t > 7:
12         produce FFI(t-1)
13 L(t) --> L(t+1)
14 K(t) --> K(t+1)
15
16 homomorphism:
17 L(t) --> ~("Leaf", 0.7+t*0.05)
18 K(t):
19     if t >= 5:
20         produce ~("FlowerBud", 1.0)
21     elif t < 5:
22         produce ~("Flower", 2.5-t*0.02)
```

The axiom here consists of a single module `A` describing the apex. After a certain amount of time, an apex will develop two internodes, two lateral leaves `L` and lateral apices, as well as a terminal flower bud `K`. The main apex is thus not continued, however the module `I` is introduced to model elongation of the internodes.

While each apical meristem is quickly terminated, apical growth is continued according to the same pattern on the lateral apices. This is called sympodial (cymose) growth, as opposed to monopodial (racemose) growth where the terminal apex keeps growing and is not consumed by a terminal flower. In sympodial growth, apices are terminated and the (in this case two) lateral apices take over.

Over time, the size of the leaves increases, and the flower buds change into fully developed flowers. Notably, the two lateral apices develop at different rates (modeled by differing time delays related to their differing parameter arguments), thus yielding an asymmetric inflorescence overall. As in the previous example, the interpreted meshes are rendered directly using the Cycles path tracer, and leaves and flower meshes are modeled in Blender.

Figure 5.2: The results obtained for the L-system of *Lychnis coronaria* specified in Listing 5.2 after 1, 5, 15 and 30 production steps, respectively, rendered in Blender. The flowers and leaves are custom meshes modeled in Blender.

5.1.3 Modeling a Field of Flowers in Blender

One advantage of integration of L-Py with Blender is that the generated plant models can easily be assembled into bigger scenes, for example a field of flowers. This can be done in Blender using the particle systems / hair modifier. Blender also offers simulation tools to for example simulate wind. This can be combined with particle systems / hair to produce a field of flowers moving in the wind.

Figure 5.3 shows a simple example where flower object instances are emitted at random positions from a subdivided and slightly curved ground surface. Apart from the position, the size and orientations of the flower instances also exhibit some slight random variation. The grass does not build on a model generated using L-Py / Lindenmaker, it consists of simple curves and is generated directly using Blender's hair modifier. It is important to note that this example is far from biologically accurate, yet it shows what can be achieved using L-Py models directly in Blender.

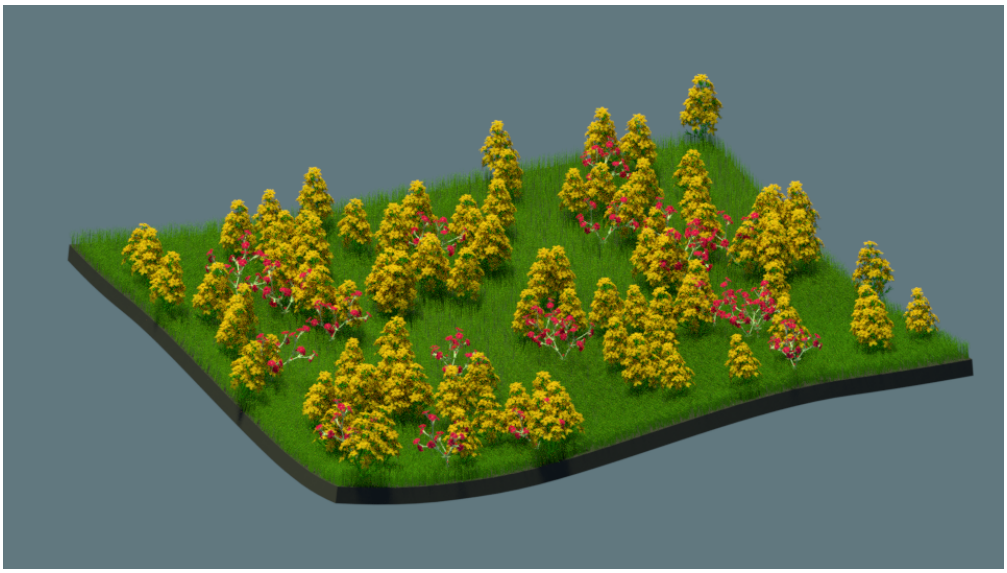


Figure 5.3: Integration with Blender makes it easy to create a field of plant models produced using Lindenmaker, by generating randomly placed, sized and rotated instances of the plants using Blender's particle system / hair modifiers. The grass in this example are simple curves generated using a hair modifier.

5.2 Modeling Trees

5.2.1 Deterministic Monopodial Tree

While the modeling of branching patterns via L-systems had been studied before, the models for trees proposed by Honda [Hon71] were among the first to be interpreted via turtle graphics and served as a basis for other well-known tree models.

The following Listing 5.3 presents a slightly adapted version of Honda's model for a simple monopodial tree-like structure, i.e., where the main growth axis and lateral axes are clearly distinguished, based on deterministic growth patterns. The model by Honda is based on the following assumptions:

- Segments of the tree grow in straight lines
- One mother segment produces two child branches (one continuing the main axis, one being a lateral branch)
- Lengths and widths of the segments are in each step reduced by constant ratios
- Child segments lie in the same plane and exhibit constant branching angles to the mother segment

Listing 5.3: L-system Code for a Simple Monopodial Deterministic Tree

```
aBr      = 60  # branching angle
aBrV     = 30  # branching variation angle
aDiv     = 120 # divergence angle
dL1      = 0.8 # length decrease rate trunk
dL2      = 0.6 # length decrease rate lateral
dW       = 0.7 # width decrease rate

Axiom: A(10,1)
derivation length: 10

production:
A(l,w) --> F(l,w)[&(aBr)B(l*dL2,w*dW)]/(aDiv)A(l*dL1,w*dW)
B(l,w) --> F(l,w)[-(aBr)L(1)C(l*dL2,w*dW)]&(aBrV)C(l*dL1,w*dW)L(1)
C(l,w) --> F(l,w)[+(aBr)L(1)B(l*dL2,w*dW)]^(aBrV)B(l*dL1,w*dW)L(1)
L(age) --> L(age+1)

homomorphism:
L(age):
    if age > 1 and age < 6:
        produce &(20)\(40)~("Leaf", 2+age/5)
```

Like in the original L-system, a mother segment `A` on the main axis continues its development by producing another segment `A` (with a certain rotation around its own axis), as well as a lateral branch segment `B`. The lateral axes follow the same monopodial pattern as the main axis in each segment, by also producing another segment to continue their current branch, as well as producing a lateral segment. However, instead of producing a module `A`, two modules `B` and `C` are produced in an alternating manner, each exhibiting

slightly different branching angles. The length and width for newly produced segments decreases by a constant multiplicative factor in each step.

In the adapted version here, branching angles are customized such that main and lateral segments do not lie in the same plane, but rather exhibit some variation. Also, simple leaves have been added growing at the outermost branches, which change size slightly depending on their age. Results of the graphical interpretation can be seen in Figure 5.4.

Figure 5.4: The results obtained for the adapted L-system specified in Listing 5.3 of a monopodial tree-like structure based on Honda [Hon71] after 1, 2, 3, 6 and 10 steps.

5.2.2 Randomized Tree with Stochastic Branch Termination

The previous tree model was characterized by deterministic production rules. More realistic and aesthetically pleasing results can be achieved by introducing randomness in the patterns of growth, to better simulate the many factors influencing growth that can not be included directly in the model. Since L-Py and Lindenmaker are based on Python, this can be achieved by using the Python `random` module directly in the L-system production code.

The following model in Listing 5.4 is based on a model presented by Lindenmayer and Prusinkiewicz [PL96], which however exhibits deterministic production rules. In order to add randomness in the growth patterns, the model presented here is thus a somewhat modified version that does not resemble the original too much.

Listing 5.4: L-system Code for a Tree with Randomized Tertiary Branching Characteristics and Stochastic Branch Termination

```
1  import random
2
3  aDiv1          = 95    # divergence angle 1
4  aDiv2          = 133   # divergence angle 2
```

```
 5 aBr           = 25   # branching angle
 6 dL            = 1.05 # elongation rate
 7 dW            = 1.2  # width increase rate
 8 maxApexAge    = 3    # age as branch stops terminal growth
 9 leafStartTime = 2    # time as leaves start to grow
10
11 Axiom: I(3,0.3)/(45)A(0,0)
12 derivation length: 5
13
14 production:
15 I(l,w): # Internode
16     rnd = random.random()/4+1
17     produce I(l*dL*rnd, w*dW)
18 L(age, t) --> L(age+1, t) # Leaf
19 A(age, t): # Apex
20     rnd = random.random()/2+0.75
21     if age <= maxApexAge and rnd > 0.9:
22         produce I(1,0.3)
                   [&(aBr*rnd)I(1,0.3)L(0,t+1)A(0,t+1)]I(0.5,0.3)
                   /(aDiv1*rnd)[&(aBr*rnd)I(1,0.3)L(0,t+1)A(0,t+1)]
                   /(aDiv2*rnd)[I(1,0.3)A(age+1,t+1)]
23
24 homomorphism:
25 I(l,w) --> F(l,w)
26 L(age, t):
27     if t > leafStartTime and age < 3:
28         produce ~("Leaf", 1.5+age/5)
```

An apex module `A` forms the basis of production, each producing three child branches at differing angles, thus exhibiting ternary branching characteristics. Each child branch also has its own apex that continues growth in the same manner as the parent branch. In this modified model, slight random variation has been added to branching angles.

Like in the original model, this model differs from the previous tree-like model in Listing 5.3 in that branch segments are not treated as static dead matter, but rather as evolving parts of a growing structure. Therefore, segments are represented by an internode module `I` that also changes its length and width dynamically over time, instead of just having internodes produced at different scale in their initial production. Here, the change in length is also varied via a random factor.

The most drastic difference to the original model in terms of random variation here is that a random factor is checked against a fixed value to determine with a given probability whether a branch should be terminated. This is essential to add to the plausibility of the resulting structure, especially for large scale structures like trees that are heavily influenced by external factors and where fully deterministic unfolding of fixed patterns

is fairly unlikely. In this model, leaves have also been included in the production for added realism. Results of the graphical interpretation rendered in Blender can be seen in Figure 5.5. It must be noted, though, that due to the random influences, results are different each time.

Figure 5.5: The results obtained for the L-system of a Tree with Randomized Tertiary Branching and Stochastic Branch Termination, as specified in Listing 5.4, after 1, 3, 5, 6 and 8 production steps.

5.3 Simulating Environmental Interaction via Integration with a Blender Scene

As has been discussed before, modeling environmental interaction with a Blender scene is one of the main advantages of integrating L-Py with Blender via Lindenmaker. The following models of environmental interaction will be based on the same "ground truth" L-system, which is a simple regular tree model presented in Listing 5.5. The result of the graphical interpretation of this model can be seen in 5.6.

Listing 5.5: L-system Code for a Tree as Ground Truth for Environmental Interaction

```
1  maxApexAge = 3 # age as branch stops terminal growth
2  dW = 0.05      # width increase factor
3
4  Axiom: I(2,0.1,0)A(0,0)
5  derivation length: 10
```

```
 6
 7  production:
 8  I(l,w,age) --> I(l,w+dW,age+1)  # Internode
 9  L(age) --> L(age+1)             # Leaf
10  A(age,time):                    # Apex
11      if age <= maxApexAge:
12          produce I(1,0.1,0)/(137.5)L(0)
                    [+(40)A(0,time+1)L(0)]A(age+1,time+1)L(0)
13
14  homomorphism:
15  I(l,r,age):
16      produce F(l,r)
17  L(age):
18      if age > 0 and age < 3:
19          produce ^(30)~("Leaf", 1.0+0.1*age)
```

This model is based on monopodial branching, however due to the termination of apices of a certain age, this characteristic is no longer given and growth occurs in all directions without a clear main apex. Leaves and internodes increase in width and size over time.

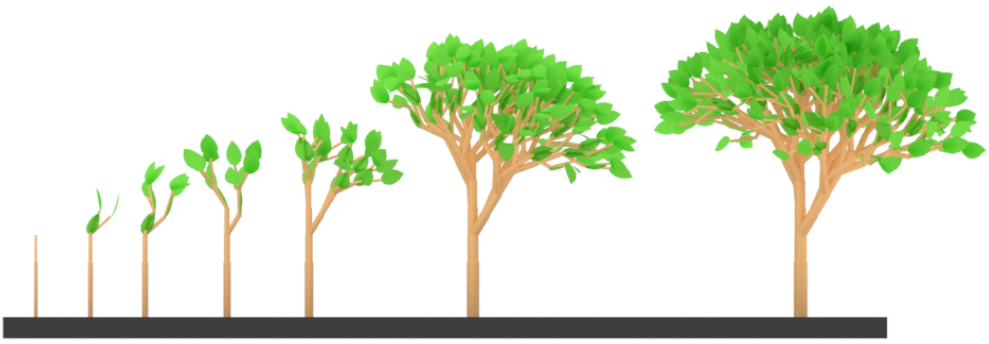


Figure 5.6: The results obtained for the L-system of a regular tree used as ground truth for environmental interaction, as specified in Listing 5.5, after 1, 2, 3, 4, 5, 8 and 9 production steps.

5.3.1 Modeling Phototropism

Phototropism, as discussed in previous sections, is the tendency of plants and other growing organisms to reorient in response to electromagnetic stimuli. The growth inducing hormone auxin accumulates in cells farthest from the light, thus causing a greater elongation at the side in shadow, which in turn bends the growing structure toward the light source.

To simulate this effect in Lindenmaker, a custom command for graphical interpretation has been added that implements a *look at* function, as discussed in Section 4.4, to reorient the turtle heading toward a target point.

Listing 5.6: L-system Code Modeling Phototropism in a Blender Scene

```
1  import bpy
2  import random
3
4  sunPos = bpy.data.objects['Sun'].location
5
6  maxApexAge = 3          # age as branch stops terminal growth
7  phototropismStartT = 3  # time as phototropism takes effect
8  dW = 0.05               # width increase factor
9
10 Axiom: I(2,0.1,0)A(0,0)
11 derivation length: 10
12
13 production:
14 I(l,w,age) --> I(l,w+dW,age+1) # Internode
15 L(age) --> L(age+1)            # Leaf
16 A(age,time):                   # Apex
17     if age <= maxApexAge:
18         produce I(1,0.1,0)/(137.5)L(0)[+(40)A(0,time+1)L(0)]
                   X(time)A(age+1,time+1)L(0)
19 X(time):
20     if time >= phototropismStartT:
21         rndX = (random.random()-0.5)*5
22         rndY = (random.random()-0.5)*5
23         rndZ = (random.random()-0.5)*5
24         produce @(sunPos.x+rndX,sunPos.y+rndY,sunPos.z+rndZ)
25
26 homomorphism:
27 I(l,r,age):
28     produce F(l,r)
29 L(age):
30     if age > 0 and age < 3:
31         produce ^(30)~("Leaf", 1.0+0.1*age)
```

The model in Listing 5.6 builds on the previously described regular tree model. Using the *look at* command `@` very few changes have to be made to simulate the effect of phototropism. In the first two lines, the Blender Python API module `bpy` and the `random` module are imported. The Blender scene in which this model is integrated contains a light emitting object labeled "Sun". Using the `bpy` module, the location of the sun object can be accessed directly from the L-Py / Lindenmaker production code.

A new module `X` is added to apply the `@` *look at* command in the direction of the sun position, with a slight random deviation. Notably, the *look at* command is applied only to one of the branches, to avoid an overly strong phototropic effect.

The result of the graphical interpretation can be seen in 5.7. It can be seen that the tree has significantly altered its growth according to the position of the sun object. It is important to note that the ground-truth tree model has a tendency to grow toward the right, as seen in Figure 5.6, due to its fixed angle branching characteristic. The influence of the phototropic effect, however, adjusts the growing structure to bend toward the light source even in the tree to the right of the sun object.

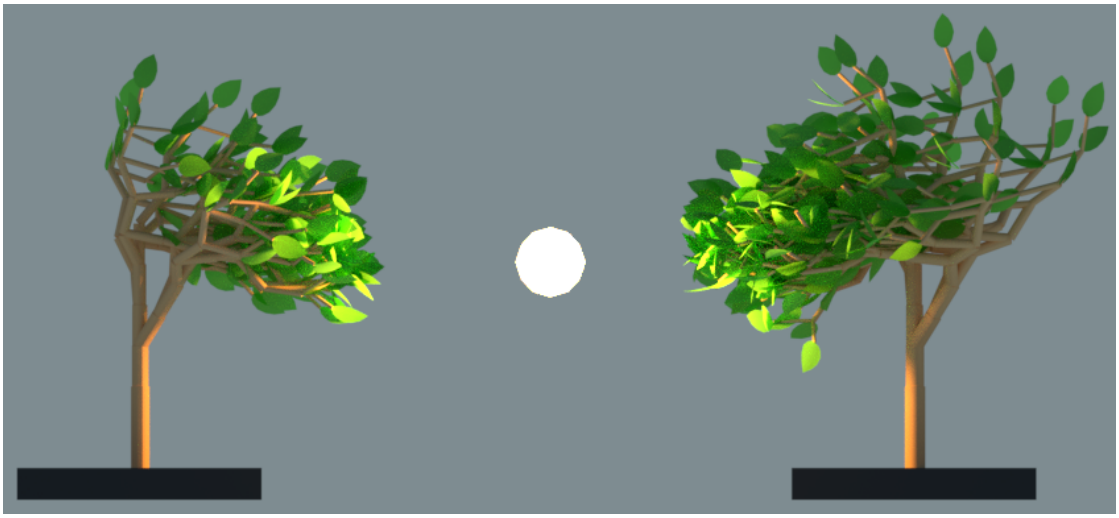


Figure 5.7: Simulating the effect of phototropism via the Lindenmaker *look at* command, using the model specified in Listing 5.6 after 8 (left) and 9 (right) production steps.

5.3.2 Pruning Branches near Object Intersections

Another interesting example of how a growing structure may interact with the environment, i.e. the Blender scene, is the pruning of branches near intersections with other objects. The L-system in Listing 5.7 gives an example of how such an effect can be achieved using L-Py and some of the specific extensions offered by the Lindenmaker Blender add-on: The turtle state query command, and the branch cutting command based on the integration with a Blender scene.

Listing 5.7: L-system Code Modeling the Pruning of Branches near Object Intersections

```
1  import bpy
2  import mathutils
3
4  obstacle = bpy.data.objects['Obstacle']
5
```

```
 6 pruningDist = 1.0 # radius around obstacle to cut branches
 7 maxApexAge = 3    # age as branch stops terminal growth
 8 dW = 0.05         # width increase factor
 9
10 Axiom: I(2,0.1,0)A(0,0)
11 derivation length: 10
12
13 production:
14 I(l,w,age) --> I(l,w+dW,age+1) # Internode
15 L(age) --> L(age+1)            # Leaf
16 A(age,time):                   # Apex
17     if age <= maxApexAge:
18         produce I(1,0.1,0)/(137.5)L(0)
                   ?("P",0,0,0)[+(40)A(0,time+1)L(0)]
                   A(age+1,time+1)L(0)
19 ?(vector,x,y,z):
20     if vector == "P" and is_nearby(x, y, z, obstacle):
21         produce /(45)^(30)~("Leaf", 1.0)%
22
23 homomorphism:
24 I(l,r,age):
25     produce F(l,r)
26 L(age):
27     if age > 0 and age < 3:
28         produce ^(30)~("Leaf", 1.0+0.1*age)
29
30 def is_nearby(x, y, z, obj):
31     p = mathutils.Vector((x,y,z))
32     pLocal = obj.matrix_world.inverted() * p
33     surfacePoint = obj.closest_point_on_mesh(pLocal)
34     surfacePoint = obj.matrix_world * surfacePoint
35     pToSurfacePoint = surfacePoint - p
36     return pToSurfacePoint.length < pruningDist
```

The model is based on the same code of a regular tree previously used. Again, the Blender Python API module `bpy` is imported, as well as the `mathutils` module, which is required to describe common math structures used in Blender, such as vectors and matrices. Instead of a light source, this time a mesh surface labeled "Obstacle" is referenced via the `bpy`. As described in Section 4.4, the turtle state query command `?` is used to retrieve the position of the turtle after each production step. The production rule for the query module itself is then used to replace it by a leaf and a cut command %, if the turtle position is under a certain distance to the obstacle mesh. Results of the pruning of branches can be seen in Figure 5.8 and Figure 5.9.

Figure 5.8: Pruning branches near object intersections, according to the model specified in Listing 5.7: In the model on the left where 12 steps are applied, the obstacle object is a simple cuboid, on the right a second tree is used where 9 and 12 steps are applied respectively. Branches are cut when the distance from turtle position to the closest point on the obstacle mesh falls below a certain threshold.

To determine whether a branch is close enough to the obstacle to be cut, the custom Python function `is_nearby` is used directly in the L-Py code, which in turn uses the function `closest_point_on_mesh` offered by `bpy.types.Object`.

5.3.3 Modeling Growth Effects of Soil Nutrient Distribution

One of the essential factors influencing growth is the soil a plant may be embedded in. The following final model of environmental interaction presented here attempts to simulate the effect of differing nutrient density in the soil on plant growth. Integration with Blender proves advantageous for this purpose, as it allows assigning a weighting to the vertices of a mesh simply by drawing on the mesh using virtual brushes similar to those known from common image manipulation software. Such a vertex weighting may be used in various ways, here it is used to model the distribution of nutrients with varying density over the soil. Figure 5.10 shows the weights painted onto a simple mesh representing the soil, where weights are visualized based on hue, with red indicating a weight of 1, blue a weight of 0 and interpolated values in between.

The model shown in Listing 5.8 interprets this distribution of vertex weights as nutrient density in the soil, and uses it to adjust growth factors based on the value closest to the root of the growing plant. Plants growing in regions with greater nutrient density will exhibit more lush and dense growth of branches and leaves, whereas plants growing in rather barren regions where nutrients are scarce will yield rather meagre growth. The result of the graphical interpretation of this model is shown in Figure 5.11, rendered in Blender. Notably, the grass shown in this illustration is produced not by L-systems but directly by Blender's hair modifier and is used as a more intuitive indicator for nutrient density.

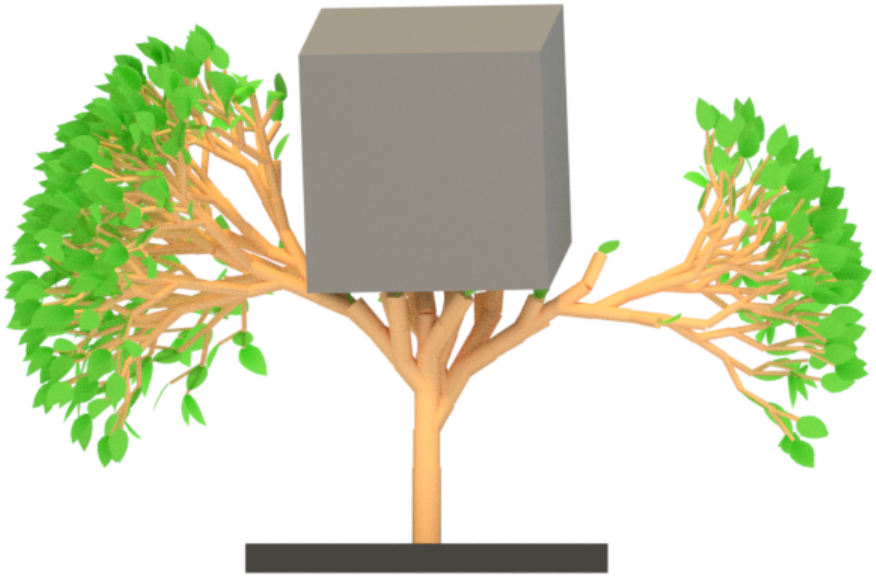


Figure 5.9: Interesting shapes can be achieved by pruning the growing structure based on objects placed in different relative positions in the Blender scene. In this example 14 production steps are applied, according to the model in Listing 5.7.

Listing 5.8: L-system Modeling Effects of Soil Nutrient Density on Plant Growth

```
1  import bpy
2  import mathutils
3  import random
4
5  soil = bpy.data.objects['Soil']
6  rootpos = bpy.data.objects['Root'].location
7
8  # determine soil nutrient density (vertex weight)
9  # around turtle root position
10 def get_soil_nutrient_density():
11
12     # build kd tree to find mesh vertices by position
13     mesh = soil.data
14     kd = mathutils.kdtree.KDTree(len(mesh.vertices))
15     for index, vert in enumerate(mesh.vertices):
16         kd.insert(vert.co, index)
17     kd.balance()
18
19     # build list of mesh vertex weights
20     vgroup = soil.vertex_groups[0]
```

```
21      weights = []
22      for index, vert in enumerate(mesh.vertices):
23          for group in vert.groups:
24              if group.group == vgroup.index:
25                  weights.append(group.weight)
26
27      p = rootpos
28      pLocal = p * soil.matrix_world.inverted()
29      vPos, vIndex, vDist = kd.find(pLocal) # closest vertex
30      return weights[vIndex]
31
32  nutr = get_soil_nutrient_density()+0.25
33  maxApexAge = (3*nutr)+1 # age as branch stops terminal growth
34  dW = 0.05 # width increase factor
35
36  Axiom: I(2,0.1,0)A(0,0)
37  derivation length: 10
38
39  production:
40  I(l,w,age) --> I(l,w+dW,age+1) # Internode
41  L(age) --> L(age+1)            # Leaf
42  A(age,time):                   # Apex
43      rnd = random.random()
44      if age <= maxApexAge and nutr > rnd/2:
45          produce I(1,0.1,0)/(137.5)L(0)
                    [+(40)A(0,time+1)L(0)]A(age+1,time+1)L(0)
46
47  homomorphism:
48  I(l,r,age):
49      produce F(l,r)
50  L(age):
51      rnd = random.random()
52      if age > 0 and age < 3 and nutr > rnd:
53          produce ^(30)~("Leaf", 1.0+0.1*age)
```

To determine the nutrient density in the soil closest to the root of the growing structure via the Blender Python module `bpy`, a custom function `get_soil_nutrient_density` has been added. This function first initializes a k-d tree structure of soil mesh vertices by their positions, using the Blender `mathutils.kdtree` module. It also generates a list of all vertex weights sorted by vertex index. Using this information, the root position in the local coordinate system of the soil mesh model is then used to find the index of the closest vertex in the k-d tree, to return the associated vertex weight. This vertex weight is used via the constant `nutr` to adjust certain growth characteristics: For one, the

maximum age of apices after which a branch stops its terminal growth may be reduced if the nutrient density is low. Moreover, random probabilities of branch termination and leaf growth have been added that are dependent on nutrient density. As can be seen in Figure 5.11, these simple changes already yield highly distinct results for positions with different nutrient density.

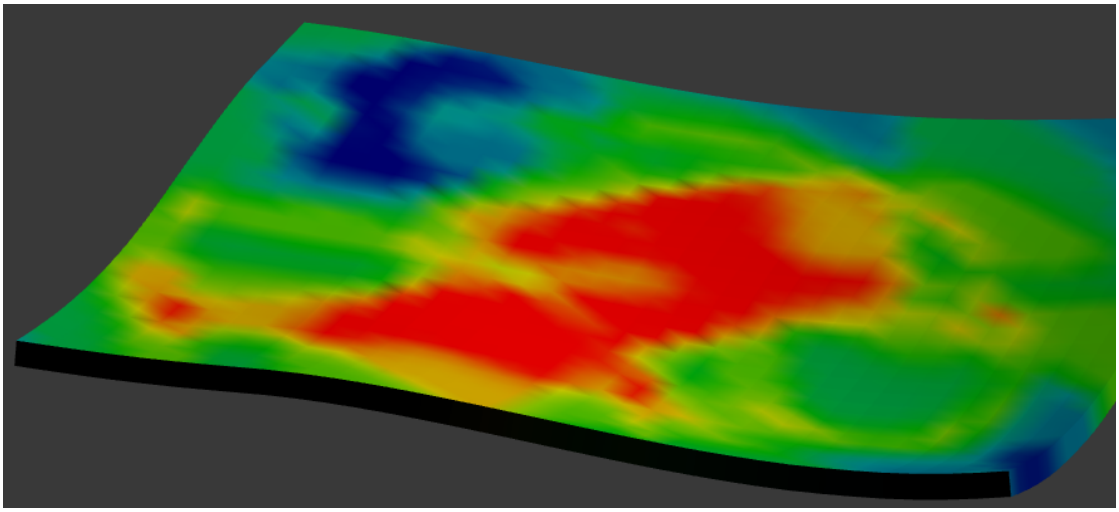


Figure 5.10: Vertex weights painted onto a simple mesh in a Blender scene, to model the distribution of nutrients in soil. These weights are used to affect growth characteristics in the L-system specified in Listing 5.8. Weights are visualized based on hue, with red indicating a weight of 1, blue a weight of 0 and interpolated values in between.

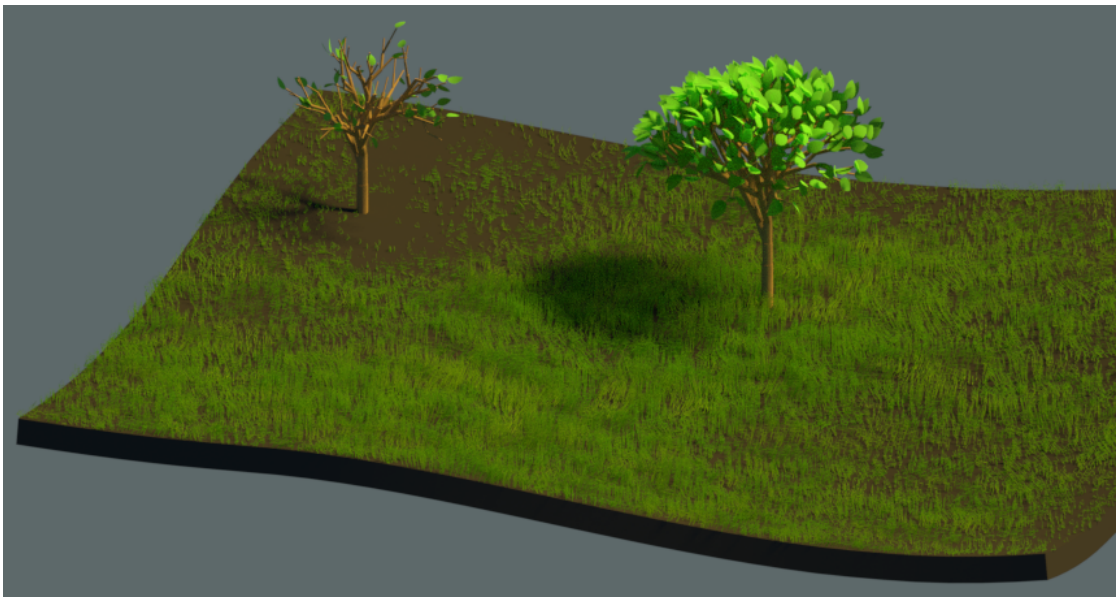


Figure 5.11: The interpreted results of the model in Listing 5.8, demonstrating the simulated effect of nutrient density in the soil on tree growth characteristics. The tree on the left is embedded in more barren soil and exhibits sparse growth even after 14 production steps. The tree on the right growing in richer soil can be seen to yield much denser growth.

CHAPTER 6

Conclusion and Possible Improvements

Today, Lindenmayer systems are used for a wide range of applications in the field of procedural modeling. It may be argued that the focus has shifted more toward architectural man-made structures, as far as the modeling of whole urban city structures based on L-systems as presented by Parish and Müller [PM01]. However, as emphasized by Wonka and Wimmer [WWSR03] in their approach to modeling buildings, the generation of architectural models is generally characterized by a partitioning process rather than a growth process, thus proving other shape grammars more suitable.

The use of L-systems for the modeling of plants likewise is more of a niche interest today. This is mainly due to them being rendered obsolete by the existence of other modeling approaches: Methods better suited for artistic aesthetic use that yield more predictable and intuitive results to work with as an artist, as discussed in Chapter 2, as well as more detailed biological models hard to capture via L-systems. Nonetheless, L-systems in the field of plant modeling still produce a plethora of fascinating forms and patterns, the study of which is highly interesting and worthwhile in itself. Likely the most potential application today would thus be in the educational field, as discussed among others by Hanan [Han92] and Boudon et al. [BPC+12].

The focus of this work was, however, not to present novel techniques for the aesthetic or biological modeling of plants. The main goal of this work was to integrate the existing formalism of parametric context-sensitive L-systems in the open-source computer graphics software Blender, with special consideration on allowing the modeling of environmental interaction with a Blender scene, and more importantly to discuss the possible results and potential of such an integration.

In Chapter 3 the well-established theory of L-systems was laid out as an overview, followed by a brief review of the various extensions that have been introduced to the L-system formalism over the years.

Building on this foundation, Chapter 4 discussed the implementation steps required to bring L-systems to Blender. This was done in the form of a small add-on that uses an adapted version of the open-source L-Py L-system framework for L-string production, as well as independent graphical interpretation code based specifically on the Blender Python API with extended interpretation commands to facilitate environmental interaction with a Blender scene. Chapter 4 also discussed potential advantages of this integration in Blender from a theoretical standpoint, which were then verified via application results in Chapter 5.

Finally, Chapter 5 demonstrated results of L-Py integration in Blender for environmental interaction with Blender scene context during L-system production, post-production modeling and simulation tools as well as rendering. This was done via examples of modeled results for plant inflorescence and tree structures, the main focus however being the discussion of environmental interaction which is arguably the most promising application of Blender integration.

To conclude, it can be said that integration of L-system production and interpretation with a highly versatile computer graphics software like Blender may greatly enhance the overall visibility and interest of L-systems with regards to a large audience of users. Direct integration with a commonly used modeling and rendering tool makes it easier to visualize produced models and customize them after L-system production, as well as presenting them in the context of a complex environment such as a Blender scene. Most importantly, however, there is great potential to influence the actual production of the growing structure based on such context to model environmental interaction.

Several improvements to the add-on implementation may be desirable, some of them mentioned in the following:

- Rebuild L-Py in pure Python to achieve platform independence and greater ease of use. The performance penalty compared to the current C++ wrapper might be considerable though.
- Allow for the generation of smoothly curved internodes.
- Generate Blender mesh structures from individual vertices directly in L-Py, as opposed to using existing mesh objects. This may be possible via the features offered by the Blender Python API. While it may not be very intuitive to model mesh structures from code, it would be useful for the modeling of procedural variation in the structures of leaves and other plant organs.
- Implement modular L-systems where a module in one L-system may trigger production rules of another L-system, i.e., to allow for two mutually influencing growing structures without having to manually produce and then reference resulting objects.

- Add more sophisticated tools to model environmental interaction and influence of the environmental context on the growing structure with more precise control.

Bibliography

[AK84] Masaki Aono and Tosiyasu Kunii. Botanical tree image generation. *IEEE Comput. Graph. Appl.*, 4(5):10–34, May 1984.

[BPC+12] Frédéric Boudon, Christophe Pradal, Thomas Cokelaer, Przemyslaw Prusinkiewicz, and Christophe Godin. L-Py: An L-system Simulation Framework for Modeling Plant Architecture Development Based on a Dynamic Language. *Frontiers in Plant Science*, 3(76):doi: 10.3389/fpls.2012.00076, May 2012.

[Bur13] Thomas Burt. *Interactive Evolutionary Modelling by Duplication and Diversification of L-Systems.* PhD thesis, University of Calgary, 2013.

[dREF+88] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant Models Faithful to Botanical Structure and Development. *SIGGRAPH Comput. Graph.*, 22(4):151–158, June 1988.

[dReIeeA14] INRIA Institut National de Recherche en Informatique et en Automatique. OpenAlea Software Environment for Plant Modeling. `http://openalea.gforge.inria.fr`, 2014. Acessed: 2017-01-31.

[dReIeeA16a] INRIA Institut National de Recherche en Informatique et en Automatique. L-Py Open Source Lindenmayer Systems for Python. Developer Documentation. `http://openalea.gforge.inria.fr/dokuwiki/doku.php?id=packages:vplants:lpy:doc#developper_documentation`, 2016. Accessed: 2017-03-03.

[dReIeeA16b] INRIA Institut National de Recherche en Informatique et en Automatique. L-Py Open Source Lindenmayer Systems for Python. GitHub Repository. `https://github.com/openalea/lpy`, 2016. Accessed: 2017-03-03.

[dReIeeA16c] INRIA Institut National de Recherche en Informatique et en Automatique. L-Py Open Source Lindenmayer Systems for Python. Main Page. `http://openalea.gforge.inria.fr/dokuwiki/doku.php?id=packages:vplants:lpy:main`, 2016. Accessed: 2017-02-04.

[ES80] Peter Eichhorst and Walter J Savitch. Growth functions of stochastic lindenmayer systems. *Information and Control*, 45(3):217–228, 1980.

[Fal90] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, 1990.

[Fou09] The Python Software Foundation. What's New In Python 3.0. `https://docs.python.org/3.0/whatsnew/3.0.html`, 2009. Accessed: 2017-03-03.

[FtBC09] Blender Foundation and the Blender Community. Blender Python 3.x Migration Proposal. `https://wiki.blender.org/index.php/Dev:2.5/Source/Python/API/Py3.1_Migration`, 2009. Accessed: 2017-03-03.

[FtBC16] Blender Foundation and the Blender Community. Blender 2.78 Python API Documentation. `https://docs.blender.org/api/blender_python_api_2_78_release`, 2016. Accessed: 2017-01-31.

[FtBC17a] Blender Foundation and the Blender Community. Blender 2.78 Configuring Platform Dependent Paths. `https://blender.org/manual/getting_started/installing/configuration/directories.html`, 2017. Accessed: 2017-03-03.

[FtBC17b] Blender Foundation and the Blender Community. Blender Reference Manual. `https://docs.blender.org/manual/en/dev`, 2017. Accessed: 2017-03-04.

[FtBC17c] Blender Foundation and the Blender Community. blender.org: Home of the Blender project. Free and Open 3D Creation Software. `https://blender.org`, 2017. Accessed: 2017-01-31.

[Ham96] Mark Stephen Hammel. *Differential L-systems and their application to the simulation and visualisation of plant development*. Computer Science, University of Calgary, 1996.

[Han92] James Scott Hanan. *Parametric L-systems and Their Application to the Modelling and Visualization of Plants*. PhD thesis, 1992.

[Hon71] Hisao Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of theoretical biology*, 31(2):331–338, 1971.

[JL87] J. M. Janssen and A. Lindenmayer. Models for the control of branch positions and flowering sequences of capitula in mycelis muralis (l.) dumont (compositae). *New Phytologist*, 105(2):191–220, 1987.

[KK08] Ole Kniemeyer and Winfried Kurth. *The Modelling Platform GroIMP and the Programming Language XL*, pages 570–572. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[KP03] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. Design and Implementation of the L+C Modeling Panguage. *Electronic Notes in Theoretical Computer Science*, 86(2):134–152, 2003.

[LD98] Bernd Lintermann and Oliver Deussen. A Modelling Method and User Interface for Creating Plants. *Computer Graphics Forum*, 1998.

[LD99] Bernd Lintermann and Oliver Deussen. Interactive Modeling of Plants. *IEEE Comput. Graph. Appl.*, 19(1):56–65, January 1999.

[Leo17] Nikole Leopold. Lindenmaker: A Blender add-on for Lindenmayer systems (L-systems) using the L-Py Python library. `https://github.com/mangostaniko/lpy-lsystems-blender-addon`, 2017. Accessed: 2017-08-31.

[Lin68] Aristid Lindenmayer. Mathematical Models for Cellular Interactions in Development II. Simple and Branching Filaments with Two-sided Inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[LR79] Aristid Lindenmayer and Grzegorz Rozenberg. Parallel generation of maps: Developmental systems for cell layers. In *Graph-grammars and their application to computer science and biology*, pages 301–316. Springer, 1979.

[Mac93] Cameron MacKenzie. *Artificial Evolution of Generative Models in Computer Graphics*. Computer Science, University of Calgary, 1993.

[Man83] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Comp., New York, 3 edition, 1983.

[MP96] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410. ACM, 1996.

[MPH90] Lynn Mercer, Przemyslaw Prusinkiewicz, and James Hanan. The concept and design of a virtual laboratory. *Proceedings of Graphics Interface. Halifax, Nova Scotia*, 90:149–155, 1990.

[PHHM97a] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Mech. L-systems: From the Theory to Visual Models of Plants. In M. T. Michalewicz, editor, *Plants to Ecosystems. Advances in Computational Life Sciences, I.* CSIRO publishing, Melbourne, 1997.

[PHHM97b] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomír Měch. Visual Models of Plant Development. In *Handbook of formal languages*, pages 535–597. Springer, 1997.

[PHM99] Przemyslaw Prusinkiewicz, Jim Hanan, and Radomír Měch. An L-system-based Plant Modeling Language. In *Applications of graph transformations with industrial relevance*, pages 395–410. Springer, 1999.

[PHMH95] Przemyslaw Prusinkiewicz, Mark Hammel, Radomir Mech, and Jim Hanan. The Artificial Life of Plants. *Artificial life for graphics, animation, and virtual reality*, 7, 1995.

[PKL07] Przemyslaw Prusinkiewicz, Radoslaw Karwowski, and Brendan Lane. The L+C Plant-Modelling Language. *Frontis*, 22:27–42, 2007.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[PM01] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[Pru93] Przemyslaw Prusinkiewicz. Modelling and Visualization of Biological Structures. In *Proceedings of Graphics Interface '93*, GI '93, pages 128–137, Toronto, Ontario, Canada, 1993. Canadian Human-Computer Communications Socitey.

[Pru98] Przemyslaw Prusinkiewicz. Modeling of Spatial Structure and Development of Plants: A Review. *Scientia Horticulturae*, 74(1):113–149, 1998.

[Pru99] Przemyslaw Prusinkiewicz. A Look at the Visual Modeling of Plants Using L-systems. *Agronomie*, 19(3/4):211–224, 1999.

[Pru00] Przemyslaw Prusinkiewicz. Simulation Modeling of Plants and Plant Ecosystems. *Communications of the ACM*, 43(7):84–93, July 2000.

[Pru04] Przemyslaw Prusinkiewicz. Art and Science of Life: Designing and Growing Virtual Plants with L-systems. In *XXVI International Horticultural Congress: Nursery Crops; Development, Evaluation, Production and Use 630*, pages 15–28, 2004.

[RB85] William T Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *ACM Siggraph Computer Graphics*, volume 19, pages 313–322. ACM, 1985.

[RCSL03] Yodthong Rodkaew, Prabhas Chongstitvatana, Suchada Siripant, and Chidchanok Lursinsap. Particle Systems for Plant Modeling. *Plant growth modeling and applications*, pages 210–217, 2003.

[RLP07] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling Trees with a Space Colonization Algorithm. In D. Ebert and S. Merillou, editors, *Eurographics Workshop on Natural Phenomena*. The Eurographics Association, 2007.

[Rob86] D. F. Robinson. A notation for the growth of inflorescences. *New phytologist*, 103(3):587–596, 1986.

[Smi84] Alvy Ray Smith. Plants, Fractals, and Formal Languages. *SIGGRAPH Comput. Graph.*, 18(3), January 1984.

[Tur52] Alan Mathison Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 237(641):37–72, 1952.

[WP95] Jason Weber and Joseph Penn. Creation and Rendering of Realistic Trees. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 119–128, New York, NY, USA, 1995. ACM.

[WWSR03] Peter Wonka, Michael Wimmer, Franc cois Sillion, and William Ribarsky. Instant architecture. *ACM Transaction on Graphics*, 22(3):669–677, July 2003. Proceedings ACM SIGGRAPH 2003.