CHALMERS

Generating Climbing Plants Using L-Systems

Master of Science Thesis in the Programme Software Engineering and Technology

JOHAN KNUTZEN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, March 2009

Generating Climbing Plants Using L-Systems

Johan Knutzen

Examiner: Ulf Assarsson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2009

Contents

Abstract

We propose a novel method of procedurally generating climbing plants using L-systems. The goal of this research is to generate geometry and textures for 3D-modelers, where procedurally generated content is used as a base for the final design.

The algorithm is fast and efficiently simulates external tropisms such as gravitropism, heliotropism and other pseudo-tropisms. The structure of the generated climbing plants is discretized into strings of particles expressed using L-systems. The tips of the plant extend the branches by adding particles in its path, forming internodes. The tip holds the growth direction of an internode, and is affected by external factors such as scene objects and tropisms.

A climbing heuristic has been developed that uses the environment as leverage when the plant is climbing, and effectively covers objects on which it grows. A fast method that sprouts leaves on the surface on which the plant is growing has also been developed, along with a heuristic that simulates the decrease in length, radius and leaf size.

Sammanfattning

Vi har tagit fram en modell som procedurellt genererar klängväxter med hjälp av L-system. Målet med vår forskning är att generera geometri och texturer till 3D-modelerare, där procedurellt genererat material används som bas för den slutgiltiga produkten.

Vår algoritm är snabb och kan simulera externa tropismer såsom fototropism, gravitropism och andra pseudo-tropismer. De genererade klängväxternas struktur beskrivs av sekvenser av partiklar uttryckta i L-system. En växts gren förlängs genom att lägga till partiklar i dess väg, där ändnoden bestämmer växtens riktning. Ändnoden påverkas av externa faktorer såsom objekt i scenen och tropismer.

En klätterheuristik har utvecklats som använder omgivningen som hjälp för att klättra samt effektivt täcka de objekt som växten växer på. En snabb metod som får löv att gro på ytan som växten växer på har även utvecklats, samt en heuristik som simulerar hur längd, radie och lövstorlek minskar.

Acknowledgements

First of all I would like to thank Ulf Assarsson and Suguru Saito for their guidance throughout this project. I would also like to thank Ole Kniemeyer and Reinhard Hemmerling for their support on using GroIMP and the XL language.

Further appreciation goes to Nelson Max, Masayuki Nakajima, Alexis Andre, Klaus Petersen,Ramin Honary, Jacob Lopez Montiel and David Gavilan for reviewing the paper and participating in fruitful discussions about the research.

Much kudos goes to Shinoyama Noriaki for providing me with 3d-models and invaluable help in the preperations of my presentation. I would also like to thank all of the members of Nakajima/Saito lab at Tokyo Institute of Technology, for their support and comments during the seminar meetings.

Much appreciation also goes out to Deniz Dogan, Shahrouz Zolfaghari, Ivan Majdandzic and Alexander Göransson for reviewing my paper for the IWAIT2009 workshop, and for their friendly support when writing this thesis.

Last but not least, I would like to thank Eric Knutzen for reviewing the final drafts of my paper.

1 Introduction

This thesis proposes a method of generating climbing plants procedurally using L-systems [21]. An abstract of this thesis was published at the International Workshop on Advanced Image Technology, 2009[12]. The research field of this paper is Functional Structural Plant Modeling and Computer Graphics. The L-system used is a graph grammar based language designed by Ole Kniemeyer [18]. This is an abstract language which extends L-systems with a graph rewriting formalism called Relational Growth Grammars(RGG) [43]. The implementation of this language is called the XL programming language [17] and is a derivative of Java. The software used for developing is called GroIMP [27] and is an Open-Source implementation of the XL language. GroIMP's features include the ability to export geometry, raytracing with PovRay[26] and ray-mesh collision detection.

1.1 Motivation

One motivation behind this research is that in recent years content creation for computer games has become increasingly demanding. More content requires handcrafting by the designers, and much of the work is often repetitious. In the extreme case of modeling a forest, the designer would have to handcraft every visible tree in order to create a unique scene. An example of this kind of scene is illustrated in fig. 1. This enables the designer to have

(a) *Elder Scrolls IV: Oblivion* developed by Bethesda Game Studios.

(b) *Too Human* developed by Silicon Knights.

Figure 1: *Screenshots from games using the middleware SpeedTree [8].*

complete control in creating high quality custom scenes for games, drawing parallels to the content creation for movies. Roden and Parberry [33] argued that although content creation for computer games increasingly mirror that of film development, this trend is at its end. They argued that procedural content will soon become the dominant form of content creation for computer games. According to Roden and Parberry handcrafted game content suffer from four drawbacks [33]:

- Advances in technology mean artists need increasingly more time to create content.
- Handcrafted content is often not easy to modify once created.
- Many widely used content creation authoring tools output content in a different format than that used by proprietary game engines
- Interactive games have the potential to become much more expensive than even the most epic film.

Recent research in the field of procedural content creation for the movie and game industry presents similar evidence of this [16, 39, 5, 34]. We argue that these drawbacks are related to the fact that regardless of how much processing power the user has, it will never suffice to describe real life. Although we are coming closer with the speed of Moore's law, we are bound to what hardware can process in real-time for computer games. With more processing power, we need to render more things and one solution to this could be to simply hire more designers, though often cost prohibitive.

One approach to minimize costs for the creation of game worlds regards reusing textures and geometries for objects that occur frequently throughout the game, such as trees and crates. Another approach is to generate recurring objects or even entire game worlds in a so-called procedural approach, yielding unexpected outcomes in form of appearance [4].

Not only can procedural techniques reduce costs and speed up the development process, they can also yield better looking results. In the spirit of Mandelbrot: "The geometry of Euclid describes ideal shapes - the sphere, the circle, the cube, the square. Now these shapes do occur in our lives, but they are mostly man-made and not nature-made" [20].

In the case of decorating a vase with the Koch snowflake[19] by hand, this fractal can easily be generated procedurally at virtually any depth, if the heuristic is known. Plants also exhibit patterns which can be derived from evolution, and are ideal candidates for procedural generation.

When describing the contents of a given scene in nature, the most obvious objects that could come to mind would be the trees, stones, mountains etc. We perceive these as being the main contents of the scene, but just modeling these objects in a 3D-modeler would create a dull scene for rendering.

What makes nature so beautiful in a sense is its power to make every one of these objects look dynamic in synthesis. Grown trees in nature did not just suddenly appear. The growth of a tree reflects interaction with the environment for years before claiming their place in the scene. Stones are not just geometry with a texture painted on, as they are often covered by moss and other plants which grow on everything in the scene.

These plants rely on the environment to strive for some optimum condition, such as optimum coverage or sunlight exposure etc. They interact with the environment and create unique structures which blend and bind the scene together. Climbing plants are excellent examples of these types of plants. They try to dominate an area, striving for sunlight and coverage, with the aid of the environment. We argue that by adding these kinds of climbing plants, realism of a scene is increased.

1.2 Problem Statement

The goal of this research is to create a system which generates climbing plants which interact with the environment. The problems of such a method can be categorized as following.

Branches and Leaves How branches and leaves are represented with geometry, textures and materials. They way in which radial growth is simulated; older branches should be thicker. The leaves also play a major role in plants, how they sprout, their size etc.

Tropisms The effects of gravity, wind, and light exposure and their influence on the growth of the plant.

Collision Detection In what way the plant probes the environment for intersections with its growth direction.

Collision Reaction How the plant reacts to the environment in the event of a collision.

Climbing Heuristic This is arguably the most important part of our method. It is not only important to consider how the plant grows on planes, it is also important to consider how it interacts with the environment. Some plants (e.g. Japanese Ivy) try to attach to a close object in hopes of climbing it. Different climbing plants accomplish this in different ways, by using thorns or adhesive materials for instance.

1.3 Application

The application of the system is generating content for the gaming or movie industry. The output of the system is geometry and textures, which is essentially the same in both industries. The gaming industry, however, requires a more strict level of detail scalability, and this must be taken into consideration when representing the geometry of the plant. Other unique applications also arise in games, such as in the design of levels. One most interesting way of using the system could be to integrate it into a game so that whenever a level loads, procedural content such as climbing plants are generated anew. This would result in a slightly different world where not only permanent objects such as trees, walls and stones are present, but new objects have been added as well. Repetitive game artwork can be recognized by players and has a detrimental effect on the illusion of a 'realistic' virtual space [34]. Through the introduction of dynamic new worlds when entering new levels, game artwork can instead stimulate player satisfaction, and potentially reduce player fatigue.

We must stress the point that the motivation behind this research has not been to simulate ivy in a biophysiologically accurate way. We are only interested in the final shape of the branch; e.g. we do not take into account the biomechanical properties of the material under stress by forces over time etc. For our application in the gaming and movie industries, the designer is not interested in whether or not the simulation is as close as possible to the real thing. The focus has instead been put on creating a tool where the designer can bend the rules in order to create what he wants.

1.4 Climbing Plants

Japanese Ivy was a major influence and the rules of the system have been defined to model this type of plant. Pictures of this plant is illustrated in fig. 2. That is to say, the system has not been designed to specifically model Hedera Helix, but can be parameterized to generate it. Any type of climbing plant which *climbs* in the sense that it grows around pre-existing geometry in the scene can be parameterized. Climbing roses for instance do not have the same adhesive mechanism as ivy when it grows up a wall, but generate similar structural patterns. This is illustrated in fig. 3. The system has been generalized to generate climbing plants so that the designer can

(a) Growth around a tree.

(b) Closeup of the leaves.

Figure 2: *Pictures taken of Japanese Ivy at the Ookayama campus of Tokyo Institute of Technology.*

specify plant-specific rules like branching angles, leaf size and stem-length for what he is modeling. Apart from specifying plant-specific characteristics the system is automated in simulating different forces such as gravity, wind, light conditions etc.

There is always a trade-off between user interaction and a fully automated system in terms of results. If too much stress is put on the designers, they might as well have modeled the plant using a 3D-modeler instead. On the other hand, if too much is automated, the results may be satisfactory for a specific scene but not for another. One set of parameters might generate good results for a scene consisting of a flat wall, but not one where the scene consists of a sphere.

The proposed system can use the same predefined parameters for generating plants for any type of scene. The designer can also add forces, make them weaker, add more climbing plants, make the plant thicker etc.

2 Previous Work

2.1 Functional Structural Plant Models

FSPM (Functional Structural Plant Models) combines disciplines from various fields of research, so it is hard to file it under a specific branch of science. In 1968 a biologist named Lindenmayer developed a formal language that is now called L-systems or Lindenmayer language [21]. Initially

(a) Climbing roses growing up a wall..

(b) Ivy growing up a wall.

Figure 3: *Climbing plants in the wild.*

L-systems were used to model cell growth but Lindenmayer later applied them to plants. This application of L-systems is extensively covered in his book with Prusinkiewicz [30].

As computer hardware became increasingly faster in the 80s, computer scientists used L-systems for simulations of plant growth with the help of computer graphics. This allowed for a particular plant described by the Lindenmayer language to be represented graphically [29]. This spurred the interest of plant biologists, who were interested in a graphical model for their simulations.

However, purely biological models are usually not sufficient, so physicists got involved in the research of biomechanics in plants. Mathematicians and computer scientists also showed an interest in FSPM, in the development of algorithms and graph theory for instance.

There are many different levels of research in FSPM. Some focus on the cellular level [21], some on the individual plant level [30] and others on a forest level. Sometimes a combinations of these are required, for instance in the research of tissue development (cellular, individual plant). Therefore, FSPM is an area of research where scientists from many backgrounds can collaborate.

2.2 L-Systems

When modeling the growth of trees and bushes, L-systems have traditionally been chosen because of their simplicity. Much research has been put into expanding existing L-systems to reflect the characteristics of different vegetation. L-systems are not universal and one technique for modeling a particular tree can not easily be transferred to model a tree of different characteristics. Highly parameterized L-system languages have been developed to address this issue, and one good example is the L+C modeling language [15].

It is important to be aware, however, that one set of rules that can generate the structure of a bush, cannot be bent to generate a tree without a change of parameters or even an expansion of them. Some argue that this is an acceptable trade-off since L-systems have proven adequate enough to generate realistic vegetation. Commercial middle-ware such as X-frog [22] and Speedtree [8] use L-systems to generate plants and are highly used in the industry.

Some authors argue that L-systems suffer from the drawback that they recursively generate branches regardless of previous data and parallel generated branches [14]. Newly expanded branches usually only consider the branch from which it adheres. This brings up the problem of dealing with highly intersecting branches, which L-systems in particular suffer from [14]. However, L-systems are not restricted to the simple structure of that originally outlined by Lindenmayer. Some L-systems have been developed that consider the biomechanics involved in growth such as tropism and gravity [37, 25, 10, 11, 38].

The reaction of the stem of the plant to gravity is called negative gravitropism, and can be simulated in different ways. Jirasek [11] simulates this with torsional springs interconnecting internode segments which resulted in S-shaped branches. This corresponds to the biomechanical model presented by Fournier et. al [24], which takes radial growth in account with negative gravitropism. Jirasek [11] has good results when simulating hanging plants and branches that grow freely without interaction with the environment, but assumes that all rotations are occurring in the model are infinitesimal - as such the model cannot properly handle branching points [38]. There is also a problem of keeping the steps small enough for the numerical solver to the differential equation at each torsional spring, in order to keep the system stable [38].

Our method does not ignore the effects of gravity, and limits it to affecting the transformation of the apical buds. The internode segments remain static once placed and do not weigh down with time. The visual difference in not having internode segments weigh down is an upside-down U-shape in the branch with initial vertical elevation, instead of the more physiologically correct S-shape.

Another drawback of traditional L-systems is that they lack a sense of object-orientation and essentially just query a one-dimensional string for patterns to rewrite. Kniemeyer et. al proposed an approach transitioning from string rewriting grammars to graph grammars called Relational Growth Grammars [43]. A concrete implementation of Relational Growth Grammars is illustrated by the high-level language XL which combines the rules-based programming of graph grammars and L-systems with the object-oriented programming paradigm of Java [17].

2.2.1 Definition

The recursive nature of L-systems leads to self-similarity and thereby fractal-like forms that are easy to describe with an L-system. Plant models and natural-looking organic forms are similarly easy to define, as by increasing the recursion level the form slowly *grows* and becomes more complex. L-systems are now commonly denoted as **parametric L-systems**, defined as a tuple:

$G = \{V, S, \omega, P\}$

V is a set of symbols containing elements that can be replaced (variables)

S is a set of symbols containing elements that remain fixed (constants)

ω (start, axiom or initiator) is a string of symbols from V defining the initial state of the system

P is a set of production rules or productions defining the way variables can be replaced with combinations of constants and other variables. A production consists of two strings - the predecessor and the successor.

Fibonacci Example
variables: A B
constants: none
start: A
rules: (A → B), (B → AB)
This L-system produces the following sequence of strings:
n = 0 : A
n = 1 : B
n = 2 : AB
n = 3 : BAB
n = 4 : ABBAB
n = 5 : BABABBAB
n = 6 : ABBABBABABBAB
n = 7 : BABABBABABBABBABABBAB
If we count the length of each string, we obtain the famous Fibonacci sequence of numbers:
1 1 2 3 5 8 13 21 34 55 89 ...

Figure 4: *A fractal plant for $n = 6$*

Fractal Plant Example
variables: X F
constants: + -
start: X
rules: (X $\rightarrow$ F-[[X]+X]+F[+FX]-X),(F $\rightarrow$ FF)
angle: 25°

Here, F means "draw forward", - means "turn left 25°", and + means "turn right 25°". X does not correspond to any drawing action and is used to control the evolution of the curve. [corresponds to saving the current values for position and angle, which are restored when the corresponding] is executed. See fig. 4 for an example.

2.3 Particle Systems for Modeling Plant Growth

Another way of modeling plant growth which is different from using L-systems is using a particle system instead. This idea was first introduced by Reeves and Blau [32] but did not allow particle-particle and environment interaction. Arvo and Kirk [1] later extended the model to deal with environment interaction. Greene [7] presented a similar model for simulating climbing plants, with the biggest difference from the work of Arvo and Kirk being collision detection by means of a voxel space instead of ray-casting [2]. These models are extended by the work of Benes and Millan [2] to model climbing plants, adding e.g. traumatic reiteration to cope with collision problems.

When using a particle system, plants are discretized into strings of particles where each particle grows depending on local information. The behavior of a particle is defined by its internal state and external conditions. External conditions can be e.g. light exposure, gravity and wind forces. The internal state of a particle defines the role of the particle in the plant, whether it be e.g. a tip, leaf or internode. A particle system using these kinds of *intelligent* particles is AMAP[3], which is now commercially available.

The differences between using an L-systems or a particle system can be vague. If a plant is discretized into strings of symbols using only local information in an L-system, then it can be translated into a particle system by representing each symbol with a particle. Conversely, the behavior of the particles can be modeled with rules in an L-system and the particles can be replaced by symbols. In this sense, they are interchangeable. However, there are benefits from describing a particle system *using* L-systems. The proposed system models particles with L-system rules and is therefore better mathematically tractable as the rules are described using a formal grammar such as L-systems.

2.4 Relational Growth Grammars

Relational Growth Grammars is an extension of L-systems where a graph is being rewritten instead of a string of symbols.

2.5 XL Language and GroIMP

In the XL programming language, nodes and edges can represent segments of a branch, transformations, shader setters and user defined attributes. Rules define how graph nodes and edges should be rewritten and the state of the graph can be saved in XML format. GroIMP parses the state of the graph and replaces it with turtle graphics commands. This includes e.g. NURB surfaces, rotations, translations and user-defined commands. Because the XL programming language is a Java derivative, it is object-oriented and any code in XL can be translated to Java. Moreover, since XL is graph-based, a given graph state can easily be converted to a scene-graph for efficient rendering.

(a) An ivy designed by Jerome Prevost. (b) An ivy designed by Leonardo Merlos.

Figure 5: *Images generated using Thomas Luft's Ivy Generator [23].*

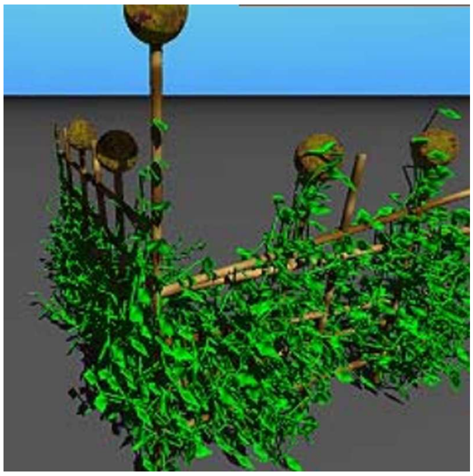
2.6 Climbing Plants

There are two approaches typically used in computer graphics to model climbing plants: particle systems and L-systems. Benes and Millan [2] and Luft [23] use a particle system approach to grow climbing plants, where each branch is a string of particles. Another way of modeling climbing plants is to discretized the space into voxels and have the models grow from predefined geometric elements, according to rules based on intersection, proximity, and occlusion [7].

The branch shape of climbing plants is important since their branches are highly elastic. Biomechanical models of the shape of branches have been evaluated, such as the ones proposed by [37, 25, 10, 11, 38]. These demonstrate realistically looking branch shapes but add extra computational overhead in order to reorient all segments in the plant when new segments are added.

Another way of modeling biomechanics is to disregard the impact of added weight caused by newly added segments, and have the newly added segments orient only dependent on the general growth direction of the branch. Since climbing plants grow upwards around geometry in the scene, newly added segments are less burdened by the gravity of previous segments than in trees. This is due to each segment being self-upholding by attaching to the geometry in the scene with the help of an adhesive material. This is a property of voluble plant species, which ensures that the entire branch is close to its supporting object [2]. We believe that from a Computer Graphics point of view the biomechanical shape of the branching model correctness is not that

(a) Growth in a 300x300x300 voxel space, using Greene's method [7].



(b) A fence with a climbing plant, generated using the method of Benes and Millan [2].

Figure 6: *Images of generated climbing plants of previous work.*

important in providing visual plausibility. Consider fig. 5(a), this illustrates an ivy growing around a torus using the Ivy Generator by Thomas Luft [23]. The software uses a stochastic approach where branches do not have any biomechanical properties but have plausible visual results.

The method proposed in this paper is similar to that of Thomas Luft [23] in that branches grow towards a general growth direction, affected by gravity. Other factors are also considered, such as phototropism, collision detection and random influence.

3 Generating Climbing Plants

3.1 Introduction

In this section, our method is explained in further detail. We start by first establishing a common terminology for the structure of the branches and leaves and then outline how these are formalized by symbols in an L-system. Further evidence is presented to support using L-systems as a formalism to simulate climbing plants and forces dictating growth. The justification of how the parameters are chosen and how branching and sprouting heuristics have been developed is explained in subsection 3.1.1 so that the reader is familiarized with our way of simulating climbing plants from observations in nature.

In order to generalize our system, emphasis is not put on strict definitions in the XL programming language. The system is instead abstracted to high-level L-system pseudo code, as the XL programming language is simply an implementation of an L-system.

3.1.1 Justification of Parameters

Patterns in nature arise as a result of evolution where plants strive for some optimum. These patterns form fractal patterns in a recursive fashion, creating structures in plants such as branches and leaves. Climbing plants have adopted a strategy of covering as large an area as possible to compete with other plants for sunlight. The plant is affected by heliotropism; it grows towards a light source, the sun. They are also affected by gravitropism, which they counter by levering themselves with surrounding structures e.g. trees and stones.

If we study these patterns and map these as rules, we can effectively simulate plants. Lindenmayer saw this in his original work of simulating the growth of algae [21]. In the case of climbing plants, we studied climbing plants in the wild and derived rules from our observations. Branching is symmetrical and a classic example is the disc phyllotaxis of daisies and sunflowers, as depicted in fig. 7. The shape of the florets follow fermats spiral, as proposed by Vogel in 1979 [41]:

$r = c\sqrt{n}$

$\theta = n \times 137.5\,^{\circ}$

$137.5\,^{\circ}$ is the golden angle which is approximated by the ratios of Fibonacci numbers [30]

In our work, we studied Japanese Ivy in the wild and through our observations we sought the parameters and heuristic that dictate its growth.

Figure 7: *A picture of the florets of a daisy, creating the fractal pattern of Fermats spiral.*

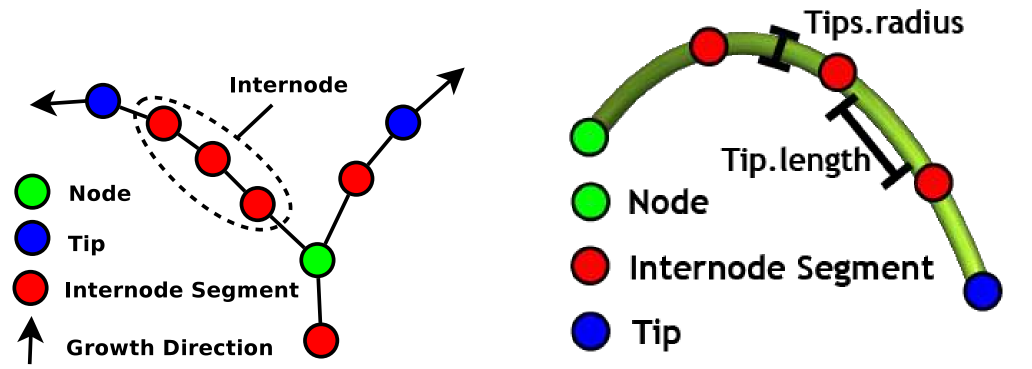
3.2 Variables and Constants

The most important component of our system is the Tip class. The tips extend the branches, split branches and add leaves according to our heuristics. A branch, which is also denoted as an internode, is comprised of internode segments. An internode segment is an affine transformation, which attributes its contribution to the curvature, radius and length of the branch. A tip also contains an affine transformation that defines the growth direction of the internode.

Refer to appendix A for a summary of the variables and constants of a tip, these will be referenced throughout this section.

3.3 Growth of a Tip

Consider fig. 8. In fig. 8(a) a branch is discretized into a string of particles that are called internode segments. Each internode segment contains an affine transformation that represents its contribution to the curvature, radius and length of the branch. The radius and length contribution of each internode segment is depicted in fig. 8(b). The transformation of an internode segment



(a) A tree with one internode split into two internodes; separated by a node. Two tips extend the tree by adding new internode segments.

(b) A tip with constant length and radius.

Figure 8: *The growth of a tip.*

is defined in eq. 1.

$$
\begin{array}{rl}
D: & \text{direction vector of the internode segment} \\
N: & \text{normal vector of the internode segment} \\
X: & D \times N \\
P: & \text{The length of the internode segment}
\end{array}
$$
$$
M_i = \begin{pmatrix} X_x & N_x & D_x & P_x \\ X_y & N_y & D_y & P_y \\ X_z & N_z & D_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1}
$$

Internode segments are followed by either a node which represents a branching point, or by a tip that represents that the branch is still growing. A tip extends an internode segment in the sense that it has the same transformation but adds extra functionality to change its transformation as it grows.

A tip uses the growth direction D in eq. 1 to specify in what direction it is growing. The tip extends the branch by placing Internode segments in its path, i.e. placing an internode segment with the same transformation as the Tip. The tip also passes its radius to the internode segment A, its contribution to the width of the branch. While the radius of the branch is kept as a separate variable in an internode segment, the length and curvature contribution is kept in its transformation matrix.

An L-System with such properties can be written as following:
variables: Tip
constants: A (an internode segment)
start: Tip
rules: Tip → A(Tip.transformation, Tip.radius) Tip
n = 0 : Tip
n = 1 : A Tip
n = 2 : A A Tip

In every time-step (L-system rewrite operation) the tip extends the internode and changes its transformation due to the effect of tropisms e.g. gravitropism and heliotropism that change the growth direction. The P vector in eq. 1 is the length of the internode segment and in the ideal case of infinitesimal steps should be near zero in length.

The rules in above L-System produce a string of internode segments forming the curvature of the branch, and make up an internode. Consider eq. 2. Each time an A is added in the path of the Tip, it inherits the transformation of the Tip as well as the radius. The radius is used to create a Vertex node which is used when generating the geometry for the NURBS surface of the internode. An internode segment can therefore be seen as a tuple of $\{Transformation, Vertex\}$.

The transformation from worldspace to the localspace of the Tip can be written as a product of the internode segments from the root of the tree to the Tip.

$$T = \prod_{i=0} M_i, \text{ where } M_i \text{ is the transformation of an internode segment.} \quad (2)$$

Moreover, since our L-System production rules are kept in a graph, all transformations can be found in $O(n)$ by traversing with e.g. depth-first.

At every time-step, the variable holding the amount of successive nodes placed by the Tip is incremented. Referring to appendix A, this corresponds to incrementing the segmentLength of the tip. The tip *dies*, i.e. is removed from the productions, when segmentLength exceeds the maximum allowed successive internode segments with some probability. This can be written as:

rules:
1. if (segmentLength modulo maxBranchLength = 0
    and probability(deathProbability))
   then
    Tip →
   else
    Tip → A Tip
2. Tip.segmentLength += 1

3.4 Tropisms

Apart from adding internode segments in its path, the tip undergoes rotation due to tropisms. By default the plant is affected by gravitropism and heliotropism. Gravitropism means that the tip is rotated towards the earth, and heliotropism means that the tip is rotated towards the general direction of the sun. The strengths of the tropisms are chosen empirically, to meet the effect sought by the designer and to fit well with a given scene.

Artificial tropisms can also be added; the effect of wind or just a vector that ensures that the plant grows towards a desired growth direction. For example, the designer might want the plant to skew across a wall with less impact of gravity when growing up a wall, as if the plant was growing up a flat wall in a windy environment.

To rotate the tip towards a tropism direction, we first need to calculate the axis of rotation. With the worldspace to localspace transformation from eq. 2, the axis of rotation in worldspace can be evaluated as in eq. 3.

$$\begin{aligned} T^{-1}: &\quad \text{localspace to worldspace transformation} \\ \tfrac{D}{|D|}: &\quad \text{direction of the tip in localspace} \\ a': &\quad \text{axis of rotation in localspace} \\ a: &\quad \text{axis of rotation in worldspace} \\ v: &\quad \text{direction of the tip in worldspace} \\ d: &\quad \text{direction of the tropism in worldspace} \\ & v = T^{-1}\frac{D}{|D|} \\ & a = v \times d \\ & a' = Ta \end{aligned} \tag{3}$$

Now, using the axis of rotation in localspace $a\prime$ from eq. 3, the equivalent rotation matrix of the tropisms impact on the tip can be calculated. This is done by using the strength of the tropism ϕ, as shown eq. 4. A derivation of eq. 4 can be found in [6, Chapter 5].

```
class Tip {
    // The transformation of the tip.
    transformation t;
    ...
    // dir : Direction of the tropism
    // e   : Strength of the tropism
    tropism(Vector3d dir, float e) {
        Vector axis; // Axis of
                     // rotation.
        // The rotational matrix from localspace
        // to worldspace of the tip.
        Matrix toLocal = transformation(this);
        Matrix toWorld = toLocal.inverse();
        // Transform dir into localspace of
        // of the Tip.
        Vector worldDir = toWorld.transform(t.z);
        worldDir.normalize();

        // Get the rotational matrix of
        // the rotation from the
        // direction of the tip towards
        // the direction of the tropism,
        // with a strength of e.
        axis = crossproduct(worldDir, dir);
        toLocal.transform(axis);
        Matrix rot = axis_rotation(axis, e);

        // Apply the rotation to the local
        // transformation of the tip.
        t = t * rot;
        t.normalizeRotation();
    }
}
```

Figure 9: *Pseudo code for a function that rotates the Tip towards the a tropism. Note that the direction of the tip is the z composant of its affine transformation.*

$$
\begin{aligned}
a' : &\quad \text{axis of rotation} \\
\phi : &\quad \text{angle} \\
\alpha : &\quad cos(\phi) \\
\beta : &\quad sin(\phi) \\
\gamma : &\quad 1 - cos(\phi) \\
R(\phi, a') = &\begin{pmatrix} (a'_x)^2\gamma + \alpha & a'_y a'_x \gamma + a'_z \beta & a'_z a'_x \gamma - a'_y \beta \\ a'_x a'_y \gamma - a'_z \beta & (a'_y)^2 + \alpha & a'_z a'_y \gamma + a'_x \beta \\ a'_x a'_z \gamma + a'_y \beta & a'_y a'_z \gamma - a'_x \beta & (a'_z)^2 + \alpha \end{pmatrix}
\end{aligned}
\tag{4}
$$

By multiplying $R(\phi, a')$ with the local transformation of the tip, the tip can be rotated towards the direction of the tropism. Pseudo-code for this is provided in fig. 9.

3.5 Collision Avoidance

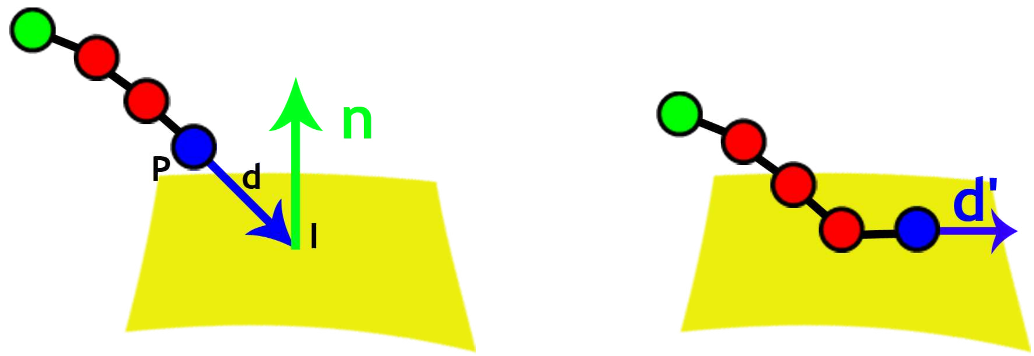
Benes and Millan [2] and Greene [7] use voxels for collision detection to check occupancy of a voxel by either the plant or some scene object. Our method uses ray-mesh intersection tests to query the environment for collisions and is different from previous work in that we use a BVH (Bounding Volume Hierarchy) instead of a uniform space subdivision scheme.

Greene argues that a voxel representation makes it easier to obtain geometric information by scanning or sampling a voxel environment than by ray casting a conventional model[7]. It is true that inspecting the nearest neighbor has constant cost, but (in a naive implementation) requires memory usage scaling at a rate of $O(n^3)$ where n is the size of a voxel. This can be unacceptable for large scenes consisting of millions or more triangles, but then again this depends on the implementation of the voxel data structure.

An alternative data structure to voxels is the use of a BVH (Bounding Volume Hierarchy), for instance a kD-tree. Constructing a kD-tree can be done in $O(n \; log \; n)$ [42]. Moreover, intersection tests can be done in $O(log \; n)$ [40, Chapter 9], which is not significantly higher than the constant time required to find the nearest-neighbors in a voxel data structure. Wald et. al also suggests that finding the k-nearest-neighbors can efficiently be done using kD-trees, by reorganizing the tree for k-nearest-neighbors queries [9].

In our implementation we used another type of BVH, called an Octree, to check the environment for intersections. KD-trees outperform octrees at ray-mesh intersection queries, but the difference would not had sped up the system drastically.

Consider fig. 10. When the tip encounters an intersection it is rotated so that it its direction D is parallel with the plane of the intersected triangle,



(a) Before the collision avoidance: the tip intersects the plane with a direction d.

(b) After the collision avoidance: the tip grows parallel with the plane with a new direction d'.

Figure 10: *Collision of a tip with a plane with normal n.*

and so that the normal N is aligned with the normal of the triangle. This simple heuristic ensures that the orthonormal vector X to N and D is parallel to the plane, and is used for sprouting leaves on the plane on which the tip is growing.

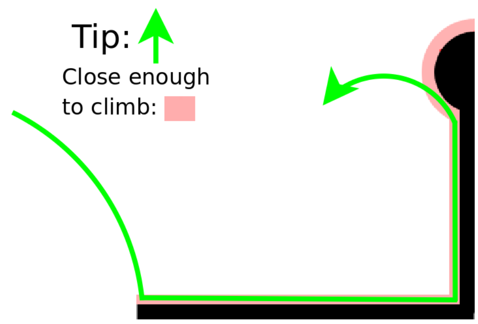
The rotation required by the transformation of the tip in response to an intersection with a plane with normal n is written in eq. 5, and is illustrated in fig. 10.

$$\begin{aligned}
T &: \quad \text{worldspace to localspace of the tip transformation} \\
D &: \quad \text{direction of the tip in localspace} \\
N &: \quad \text{normal of the tip in localspace} \\
X &: \quad \text{x composant of the transformation of the tip} \\
I &: \quad \text{intersection point of the tip in the plane} \\
P &: \quad \text{translation of the transformation of the tip} \\
d' &: \quad \text{new direction of the tip in worldspace, parallel to the plane} \\
d &: \quad \text{direction of the tip in worldspace} \\
n &: \quad \text{normal of the plane in worldspace} \\
& d' = d - n(n \cdot d) \\
& D = Td' \\
& N = Tn \\
& X = D \times N \\
& P = I - T^{-1}P
\end{aligned} \tag{5}$$

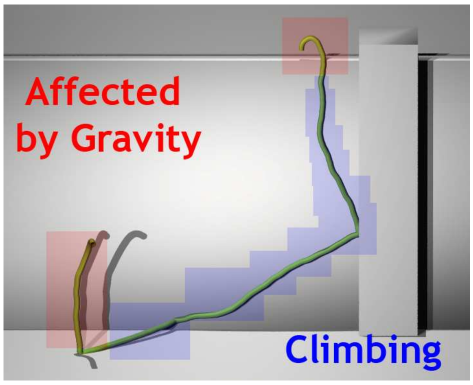
Every time a tip intersects a triangle in the environment, its normal N is aligned with the normal of the triangle. This ensures that the orthonormal vector X to D and N can be used as a direction for leaves to sprout out, and is detailed in section 3.8.

3.6 Climbing Heuristic

A tip can have two states depending on its distance from the objects in the scene, either it is *climbing* or *not climbing*. If near enough, the tip ceases to be affected by gravity and instead starts to grow upwards, clinging to the nearest object in the scene. This heuristic means to simulate how climbing plants use for example adhesive materials or thorns to clasp branches onto objects in the scene to climb upwards.



(a) When the tip is close to the pink area, it grows close on the black walls.



(b) The branch has been selected in blue when it is in a climbing state, otherwise pink.

Figure 11: *Climbing heuristic of the system, depicted in 2D and 3D.*

At every time-step, the tip queries the environment for the nearest triangle. If the triangle is near enough, gravity is replaced by a directional tropism towards the triangle. An up vector is also added, so that the tip grows upward. This ensures that the tip climbs upward and closely to the geometry it is using as leverage. The strengths of these tropisms are decided empirically, and might be stronger or weaker depending on what kind of effect is desirable.

t_{nc} :	tropism direction when not climbing
t_c :	tropism direction when climbing
s_{nc} :	tropism strength when not climbing
s_c :	tropism strength when climbing
$up' = up \times s_{up}$:	up tropism vector with length s_{up}
$g' = g \times s_{gravity}$:	gravitropism vector with length $s_{gravity}$
$l' = l \times s_{light}$:	heliotropism vector with length s_{light}
$r' = r \times s_{random}$:	random tropism vector with length s_{random}
$a' = a \times s_{adhesive}$:	adhesive tropism vector with length $s_{adhesive}$

$$\widehat{v}_c = \frac{v_{gravity} + v_{light} + v_{random}}{||v_{gravity} + v_{light} + v_{random}||} \tag{6}$$

$$\widehat{v}_{nc} = \frac{v_{up} + v_{light} + v_{adhesive} + v_{random}}{||v_{gravity} + v_{light} + v_{adhesive} + v_{random}||} \tag{7}$$

Consider eq. 6 and 7. In eq. 7 the summed tropism direction and respective strength is evaluated, for the case when the tip is not close enough to any scene object. Whether or not it is close enough, is evaluated by querying the environment for the distance between the nearest point on the nearest triangle to the tip.

This corresponds to the distance between the barycentric coordinate of the tip projected onto the plane of the triangle, and the position of the tip. If short enough (empirically chosen), tropism vectors $v_{adhesive}$, and v_{up} are added, so that the tip is rotated upwards towards the nearest triangle.

In either case, the final tropism of the tip is affected by a random tropism vector v_{random} and a heliotropism vector v_{light}. The randomness has been incorporated as a tropism to prevent artificial regularity, which deterministic L-systems suffer from [30].

In our implementation this is done by traversing all of the triangles in the scene, which might seem costly but did not turn out to be a bottleneck for the models used. An obvious improvement to the system would of course be to query the kD-tree for the nearest triangles instead, perhaps using a technique similar to the one described by Wald [9].

In any case, the corresponding rotation matrix to the tropism vector $\widehat{v}_{nc}$ or $\widehat{v}_c$ is calculated, as described in section 3.4.

Refer to fig. 11 for two examples of our climbing heuristic; one drawn, and one a screenshot from the system.

3.7 Branching Heuristic

Whereas section 3.3 is concerned with how a tip expands a branch, this section deals with how branches are split to form trees. Internodes are split at **nodes**, and form the tree structure of the plant. This splitting operation is denoted in L-systems with $[x]$, where x is affected by all transformations up until the split. These rules can be written in an L-system as:

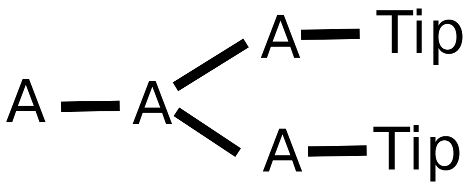
rules:

1. if (segmentLength modulo branchLength = 0
 and probability(branchingProbability))
 then
 Tip → [Mark Tip] Tip
 else
 Tip → A(Tip.transformation, Tip.radius) Tip
2. Tip.segmentLength += 1

The splitting rule of the above L-System results in two tips, sharing the

n = 0 : Tip
n = 1 : A Tip
n = 2 : A A Tip
n = 3 : A A [Tip] Tip
n = 3 : A A [A Tip] A Tip

(a) Visualized as a string of particles.



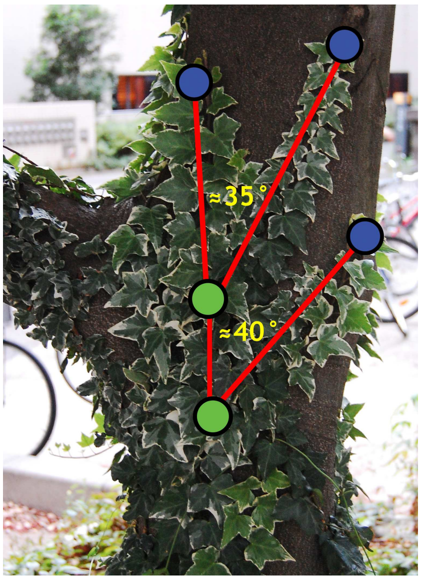
(b) Visualized as a tree.

Figure 12: *With x = 2 and y = 1.0.*

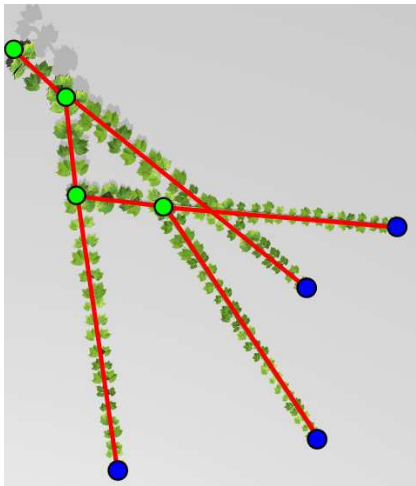
transformation as before the split, as illustrated in fig. 12. A mark node is also placed in the beginning of the new internode along with a new Tip. This Mark node signifies a beginning of a new node and is used when generating geometry, as outlined in sec. 3.10.1.

The branching heuristic of the plant needs to conform to our observations in nature to produce convincing results. The branching angle of Japanese Ivy generally differ with circa $35\,^{\circ} - 40\,^{\circ}$ from the growth direction, as depicted in fig. 13(a). This is equivalent to placing a node at the splitting point, followed by a new tip. The node rotates the new tip away from the growth direction of the splitting tip, and results in two tips growing independently of each other. The result of this is depicted in fig. 13(b).

As with tropisms, we add randomness to the branching frequency in order to reduce regularity. The amount of randomness is proportional to how many nodes make up a branch and how much variance in the branching pattern the designer aims for.



(a) Branching of a Japanese Ivy at the Ookayama Campus of Tokyo Institute of Technology.



(b) Branching of a climbing plant, based on our observations.

Figure 13: *The branching of Japanese Ivy.*

3.8 Sprouting of Leaves

Leaves sprout in a similar manner to the heuristic controlling the splitting of branches, as described in sec. 3.7. If the segmentLength of a tip exceeds segPerLeaf, a ToLeaf node is placed:

rules:

1. if (segmentLength modulo segPerLeaf = 0)
   and probability(sproutingProbability))
   then
   Tip → ToLeaf(Tip.leafsize) Tip
   else
   Tip → A(Tip.transformation, Tip.radius) Tip
2. Tip.segmentLength += 1

A ToLeaf node signifies that leaves should be sprouted, and is comprised by one variable: leafScale, which holds a scalar that is multiplied with the default size of the leaf. This scalar is set by the constructor of the ToLeaf and is retrieved from the Tip. The leafsize is decreased at each time-step as described in sec. 3.9. In this way, newer branches have bigger leaves and vice versa. A ToLeaf node is a dummy node that is used when generating geometry for the leaves, as explained in sec. 3.10.2.

To view an example of this heuristic that sprouts leaves, please refer to fig. 14.

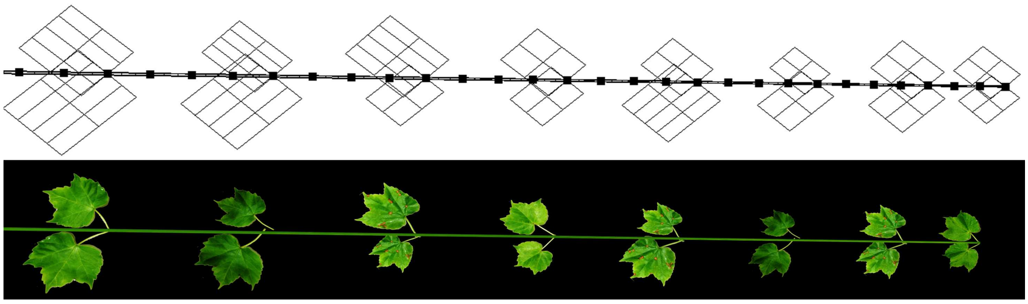


Figure 14: *The designer wants leaves to sprout every fourth segmentLength, with a probability of 1.0. The above rendering was rendered in wireframe-mode using OpenGL, and the one below was rendered using PovRay [26].*

3.9 Internode Segment Length, Radius and Leaf Size

Consider our observation of Japanese Ivy in fig. 15. From this observation, we extracted the decrease in length, radius and leaf size between each internode segment. A general formula for how the radius, length and leaf size of branches is decreased can be written mathematically as in equation 8.

Figure 15: *An observation of a Japanese Ivy growing outside of Ookayama Campus at Tokyo Institute of Technology.*

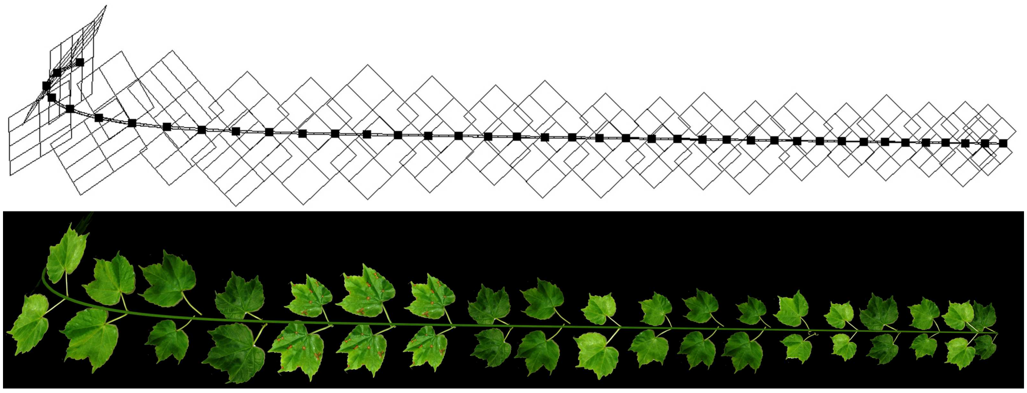


Figure 16: *An internode with a segPerLeaf (segments per leaf) of 2, leafSizeDec (decrease in leaf size per segment) of 0.94, and a radius and length decrease of 0.97 per segment.*

$$\begin{aligned} i: &\quad \text{ith segment in an internode.} \\ radius_{i+1} &= radius_i \times a \\ length_{i+1} &= length_i \times b \\ leafsize_{i+1} &= leafsize_i \times c \end{aligned} \tag{8}$$

In our system we apply a rule which decreases these three variables at each time-step in the Tip. A rendering with these relations is depicted in fig. 16.

As stated throughout the text, the transformation of each internode segment holds the length and curvature contribution to the internode, but an internode segment is also comprised of a radius node containing the radius of the internode segment. The radius node is used when generating the geometry of the branch, which is generated as a NURBS surface. More of how this is done, is explained in sec. 3.10.

3.10 Geometry Representation

The system consists of two procedures. The first one builds up a graph of productions with the XL-language until the production rules come to an end. By default the plant stops growing when a certain depth in the tree has been met, and the designer can also stop the production manually. The first procedure places dummy nodes which will be used in the second pass to generate geometry. These dummy nodes are stated in 9.

$$\begin{aligned} ToSurface: &\quad \text{replaced by the type of NURBS surface.} \\ ToLeaf: &\quad \text{replaced by geometry of a leaf.} \end{aligned} \tag{9}$$

The ToSurface node replaces a tip whenever it dies, and signifies the end of an internode. This can happen if the maximum internode length has surpassed, as described in sec. 3.3. The ToLeaf node is placed by the tip as described in sec. 3.8, and signifies that leaves should be sprouted at this point in the internode. The resulting graph is therefore a minimal abstract representation of the plant. This has a great advantage in terms of memory usage when traversing the graph at each time-step, and also greatly reduces the amount of information needed to save a generated plant.

In the second procedure, the system traverses the nodes in the graph and replaces the dummy nodes with geometry with materials and textures. This results in something that can be rendered and exported.

3.10.1 Branches

Branches are visualized as cylindrical NURBS surfaces, and can be generated by replacing the ToSurface node with a NURBSSurface node. A NURBSSurface is a predefined node in GroIMP, and has a shader with a texture and a flatness that defines the level of tessellation. When GroIMP renders a scene, it first generates the geometry in form of triangles. It does this by traversing the graph too look for geometric nodes such as NURBSSurfaces. To enable the system to generate geometry the dummy ToSurface nodes are replaced by NURBSSurface nodes with the following rule:

rules:
1. ToSurface → NURBSSurface

In order for GroIMP to create a NURBSSurface, it needs to find a starting node called a Mark. A Mark signifies the beginning of a new internode, and is used by GroIMP when creating a NURBS Surface. As explained in sec. 3.7, marks are placed whenever a tip branches. GroIMP records the transformation of the mark and continues to traverse until it encounters a NURBSSurface node which marks the end of the NURBS surface. Along the way it also records the vertices and their transformation, and interpolates between these to form the curvature of the surface. The obtains the radius between each internode segment is found interpolating the radius of the vertices.

Fig. 17 illustrates a NURBS surface with varied radius, generated in the same manner as the internodes in the system.

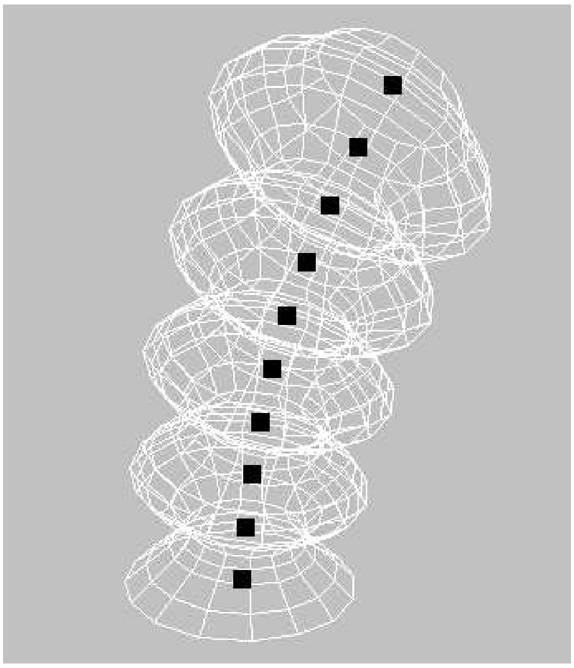


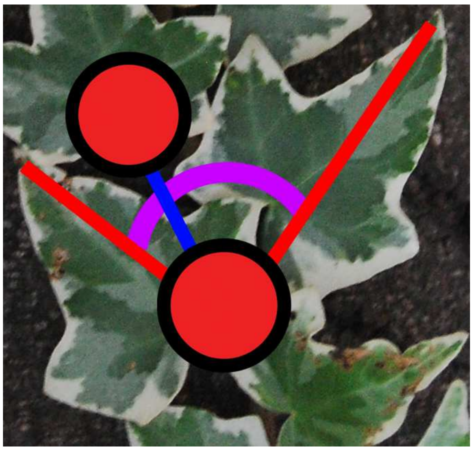
Figure 17: *A cylindrical NURBS surface with varied radius. The black squares are vertices, and the shape of the surface is obtained by interpolating between these.*

3.10.2 Leaves

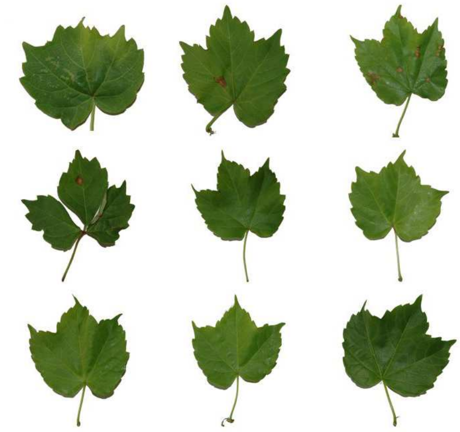
Leaves are generated in a similar manner as branches, but they are not NURBS surfaces; they are parallelograms with textures. The textures for the leaves in the system were created from real photos of Japanese Ivy in the wild, which are portrayed fig. 18(b). A leaf is created by placing a LeafNode, which contains geometry, textures and shading information. When created, a LeafNode randomly chooses a texture from the available ones in memory, and scales the geometry according to the leafScale variable of the ToLeaf. To add some variance in the size of leaves, the sizes are also varied with leafSizeVariance.

To find the transformation for each leaf, the productions graph is traversed until it encounters a ToLeaf node. The transformation is oriented so that when the plant is near enough to any surface in the screen, the normal of the matrix is parallel with the normal of the plane. This ensures that the leaves sprout parallel on the plane by sprouting along the X vector of the internode segment in worldspace; the orthogonal vector to the growth direction and the plane of the matrix.

In the case of Japanese Ivy, two leaves are sprouted with an averaged sprouting angle of $40° - 45°$. One observation of this is illustrated in fig.



(a) An observation of leaves with an averaged sprouting angle of $40° - 45°$. Picture taken at Ookayama Campus at Tokyo Institute of Technology.



(b) Textures used for leaves. Original Photos taken at Ookayama Campus at Tokyo Institute of Technology.

Figure 18: *Climbing heuristic of the system.*

18(a). To conform with our observations, one leaf is extended from the growth axis with a random angle between $40° - 45°$, and another with an angle between $-40° - (-)45°$. The parallelograms are rotated around the normal of the transformation in worldspace of the internode segment, and are placed parallel with its plane.

4 Results

In this section, our results are compared to previous work, and it is prudent to keep in mind that the comparison of results in the form of images can be highly subjective. As most research in the field of procedural content creation is presented in the form of images, we will evaluate the realism of our method in terms of how well it compares to our observations.

Moreover, to evaluate the realism of our method in contrast to another method, both methods must be applied to the same data. This imposes some restrictions as to what methods we can compare our method to. Unfortunately, we could not obtain an implementation of the work of Benes and Millan [2] and Greene [7], so we will focus on comparing our method to the one by Luft [23].

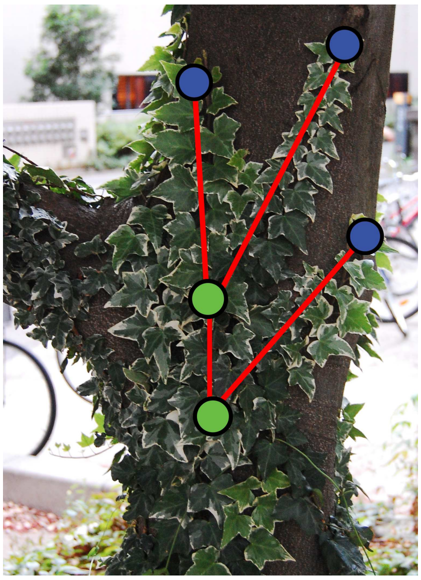
Although unpublished, Luft's implementation is freely available on his website, along with a model of a wall that comes with the software. We will focus most of our comparisons of climbing plants generated using this model, and directly compare our method with the one by Luft.

The biggest difference between our method and previous methods is that we have contributed with improved:

- Branches
- Leaves
- Climbing Heuristic
- Geometry Representation

4.1 Branches

The branching of the internodes in plants is symmetrical, and is a result of the evolution of the plant. Through observations we have captured this symmetry, and created a model to reflect it. An observation with branching nodes and internodes is illustrated in fig. 19(a) and will be used as a reference.



(a) The branching of Japanese Ivy at the Ookayama Campus at Tokyo Institute of Technology.

(b) Our branching heuristic seen from above, with no random influence.

Figure 19: *The branching of climbing plants.*

We will compare our method with the one by Luft's by using the same 3d-model as input, to make this comparison as unbiased as possible. What we have introduced, which Luft's model does not deal with, is a branching heuristic. This model is detailed in sec. 3.6, and introduces a heuristic of how the plant branches. The result of using our branching heuristic, in contrast to the one by Luft is illustrated in fig. 20.

In our model we have implemented a branching heuristic, and the structural difference is most noticeable in fig. 20(a) in comparison with fig. 20(b). It is hard to be objective when comparing the two, but we believe that our method better conforms with the branching rules observed in fig 19(a).

The growth direction of both methods is influenced by randomness and tropisms, but in our case we have accomplished a better self-similarity structure by introducing a branching heuristic. The evidence of this is clear when studying fig. 20(d) and fig. 20(c). In our result in fig. 20(c) we have a clearer branching structure, with self-similar branches at all depths. Luft's result in fig. 20(d) however, does not have any branching heuristic; resulting in a more random structure. Another figure depicting the branching heuristic can be seen from above in fig. 19(b).

When it comes to the shape of the branches and leaves, much has been improved. We have modeled a system where the spacing between internode segments, placement of leaves and the decrease in size of branches and leaves is simulated. To seek a comparison with our observations, please refer to fig. 21(a). In this figure, the length and radius of each internode segment as well as the leaf size decreases over time. We have effectively simulated this effect in branches with a simple heuristic, as described in sec. 3.9.

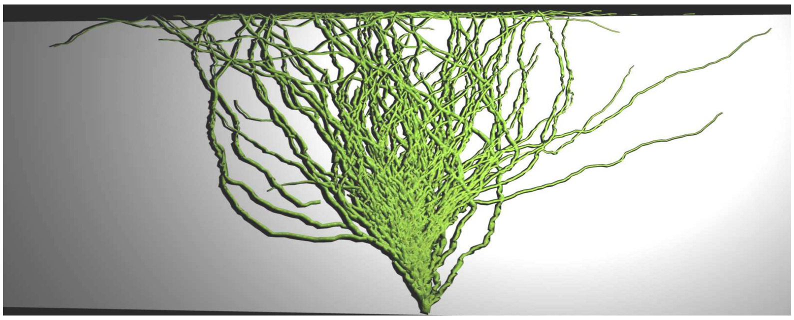
A comparison of the result of using Luft's method to this ours is illustrated in fig. 21. This figure also shows the effectiveness of our method in providing realistic branches using NURBS surfaces, instead of connected cylinders as in Luft's implementation.

4.2 Leaves

Leaves differ in our work from previous methods in that the leaves are kept close and flat to the surface on which they are growing. We accomplished this without the use of any complex collision avoidance heuristic. We have also developed a sprouting heuristic that conforms with that of our observations, as described in sec. 3.8.

This sprouting heuristic corresponds to the observations in fig. 22(a), and the result is illustrated in fig. 22(b).

(a) Branching of a climbing plant with no leaves using our method.



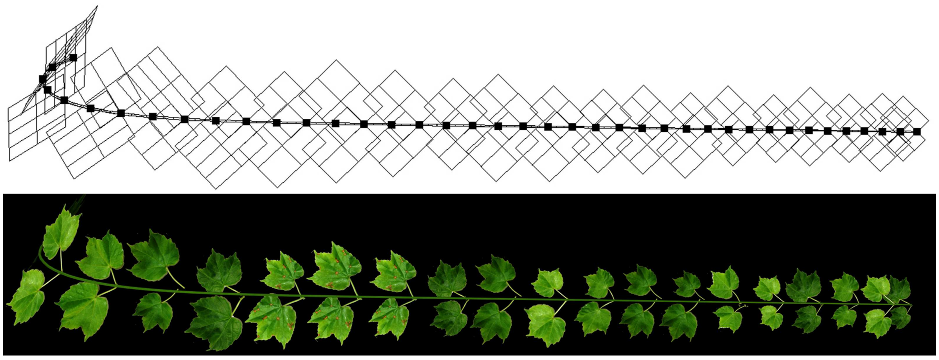
(b) Branching of a climbing plant with no leaves using Luft's method.

(c) Branching of a climbing plant with leaves using our method.

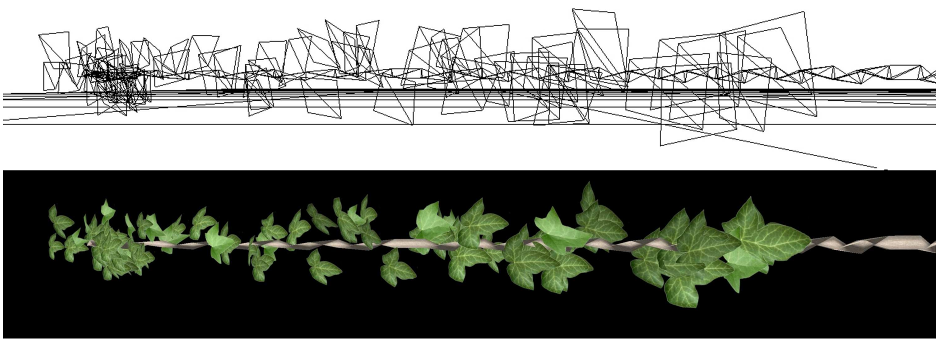
(d) Branching of a climbing plant with leaves using Luft's method.

Figure 20: *The branching of Japanese Ivy.*

(a) Japanese Ivy at the Ookayama Campus at Tokyo Institute of Technology.



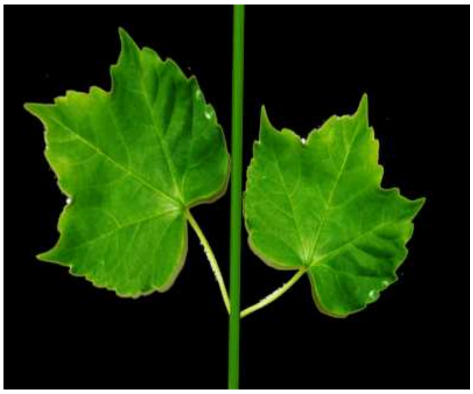
(b) Rendering of our result.



(c) Rendering of Luft's result.

Figure 21: *The decrease in radius, length of the internode segments as well as in the leaf size.*

(a) Japanese Ivy at the Ookayama Campus at Tokyo Institute of Technology.



(b) A rendering using our method.

Figure 22: *The sprouting of leaves of climbing plants.*

4.3 Geometry Representation

Using the XL-language, the climbing plants can be saved as a graph consisting of nodes which can be exported into geometry and textures. This allows for minimal space usage, as well as portability of the geometry representation. In the current prototype, branches are represented as NURBS surfaces and leaves as parallelograms. This implementation can easily be changed depending on how the graph is interpreted.

Moreover, in sec. 3.10 we have provided evidence that NURBS surfaces prove superior over regular cylinder chains which are used in Luft's method.

4.4 Climbing Heuristic

Our climbing heuristic is similar to that of Luft's method in that the plant grows closer to the nearest triangle in the scene. Furthermore, we have improved upon the method by incorporating a leaf sprouting heuristic, which sprouts leaves according to the objects in the scene which the plant is climbing on. More of how this is done, is detailed in sec. 3.6

4.5 Performance

Extending tips of the internodes in the plant takes $O(n)$ if the world to localspace transformation is recalculated at each time-step, and $O(1)$ if the transformation is cached. The n variable is the number of nodes in the productions graph.

With a model of 35,000 triangles and 4 climbing plants, the system can generate internode segments in real-time. The bottleneck of the method lies in the collision detection code, where each tip needs to perform one ray-intersection test and one closest triangle query per time-step.

If optimized, it is not inconceivable that the system can run in real-time, even with highly complex models. Recent advances in real-time raytracing are a proof of this. Examples includes the work of Shevtsov et. al, achieving 15.4 fps at a resolution of 1024x1024 with 1087K triangles, on mediocre hardware [35]. The nearest triangle queries can also be sped up by using distance fields, as surveyed by Jones et. al [13].

Generating geometry from the productions graph can be done in O(n), where n is the number of nodes in the graph. The time required to generate triangles from a NURBS surface depends on the amount of tessellation, but could be calculated in near real-time.

4.6 Renderings

Figures 25, 23 and 24 are results using our method.

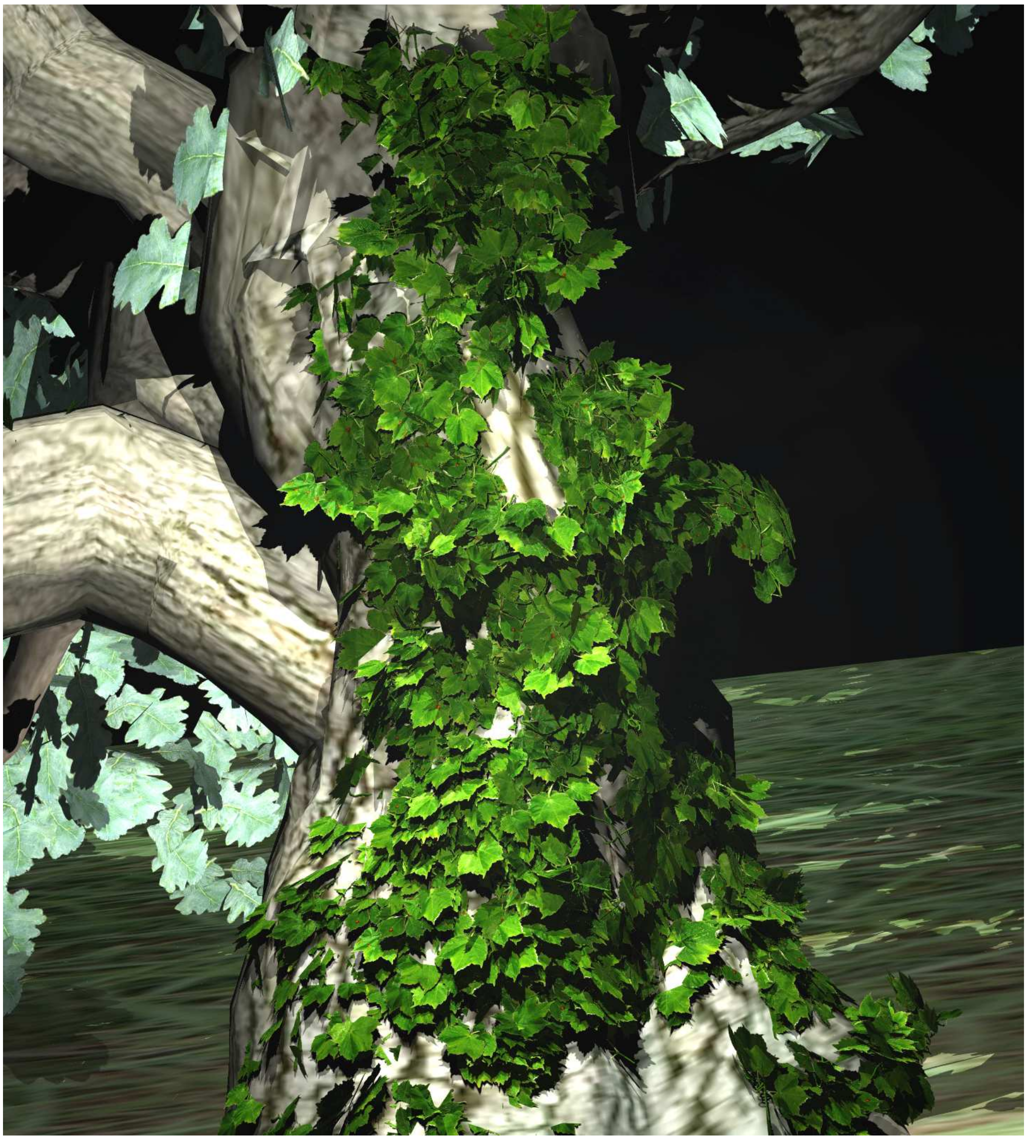


Figure 23: *The plant growing on the trunk of a tree, raytraced using PovRay.*

Figure 24: *The plant growing on the porch of a house, raytraced using PovRay.*

Figure 25: *The plant growing on a wall, raytraced using PovRay.*

4.7 Conclusions

The modeling of plants comes down to choosing the application area. We have shown that simple heuristics can model collision avoidance and the effects of tropisms efficiently to generate geometry of climbing plants. Although biophysiologically inaccurate, our method is fast and easy to implement, with no reliance on infinitesimal steps to satisfy any numerical solver.

The fact that designers prefer tools, which are easy to use and provide powerful ways of procedurally generating content is clearly evident in the popularity of Luft's software. The realism of the climbing plants generated by the tool is apparently not weighed by designers as heavily as the flexibility and ultility of the tool which the designers work with. This is apparent when generating plants using the tool and comparing these to the ones created by artists, as published on his webpage [23].

We have embraced the concept of simplicity in providing a powerful tool that improves and extends the model by Luft. Sticking to simplicity should not be underestimated, as the most complicated fractal shapes can be generated with the simplest rules. Quoting Oppenheimer: "If one can model a complex object through simple rules, one has mastered the complexity. The proof (although subjective) is in the picture [28]".

5 Discussion

As for all things in nature, climbing plants are complex beings. No matter how much more complex the model gets, it will never suffice to describe the real thing. Some research on the modeling of plants focus on the molecular level and some on the structural or forest level. One single model does not adequate to describe a plant to any level of detail. In some cases, one model may fail when a completely different one proves better and vice versa.

It is important for a model to aim at one particular application when being developed. At the beginning of our research, much effort was put into developing biophysiologically accurate models, only to result in slow implementations that hardly provided any increase in realism. It is true that a more biophysiologically accurate model would be nice, but to what extent and at what price is disputable. In this research we have had a clear goal of developing a useful prototype for the application of 3D-modelers, and we believe that an accurate enough model has been developed.

We would also comment on the use of GroIMP as a platform for developing methods to generate plants. The software proved very useful when creating a prototype, despite some parts requiring work-arounds. At the time of this paper's publication, GroIMP is still relatively new, and provides a great implementation of the XL-language. The software is open-source, and has good documentation. When developing a prototype using many low-level features of a framework, the benefits of using an open-source implementation are very clear. Furthermore, whenever something was lacking or buggy, it was easy to get in contact with the developers or simply fix the bug ourselves.

At the inception of the project other frameworks were considered, and in retrospect we are very happy to have chosen GroIMP. An obvious closed-source competitor developed at the University of Calgary is L-Studio [31], and is very powerful in expressing L-systems. Although a valid alternative, we believe that it would have proved limiting, simply in recognizing that L-Studio is closed-source. Using closed-source gratis software can come with serious ramifications. Examples include potentially not being able to fix something broken, not being able to export productions into any format and the most important for us: being limited to the abilities of the language of the framework.

The XL-language implementation in GroIMP is a java derivative. As such, anything coded in Java can also be coded in the XL-language, which that proves very useful in the developing process.

6 Future Work

Although L-System frameworks prove very useful in creating prototypes, the next step would be to write a plug-in for a major 3D-modeler. Maya would be an obvious candidate, as it provides all the requisite tools required to implement our method. Several improvements can be added to the system itself. A gui would provide an easier way of configuring the climbing plants, which can also be written in a plug-in.

When it comes to branches, a more clever way of generating the shape of the internodes comes close to mind. A backward propagating algorithm defining the radius of the internode segments could be developed, perhaps implementing the pipe-model as proposed by Shinozaki et. al [36]. The climbing heuristic of the plant can also be improved by swapping the closest triangle queries with a distance field algorithm, many of which are surveyed by Jones et. al [13].

The effect of gravity on branches could be modeled more realistically. However, one should consider the impact on speed and stability when modeling using differential equations. Especially if incorporating collision avoidance in the same algorithm.

The shape of leaves could also be improved, though at the expense of more complicated geometry. They could be modeled to bend, using e.g. NURBS surfaces. Other improvements include leaf fading, animation of leaves and branches as well as better sewing of the geometry where branches merge.

A Variables and Constants

A.1 Internode Segments

The local variables of an Internode Segment	
transformation	The transformation of the internode segment.
Vertex	Hold the radius of the internode segment.

A.2 Tips

The global variables of a Tip	
aliveTips	Holds the amount of alive tips at any given time.

The local variables of a Tip	
segmentLength	The length of the internode of the tip.
transformation	The transformation matrix of the tip, the same as an internode segment.
depth	The depth of the tip in the tree
radius	The radius of the tip
length	The length of the tip
leafsize	The size of a leaf, if sprouted from the tip.

The global constants of a Tip	
maxRandomAngle	The maximum random angle that the random tropism can differ from the growth direction.
branchLength	The amount of segmentLength at which the tip may branch.
branchingProbability	The branching probability of the tip when segmentLength mod branchLength = 0.
sproutingProbability	The leaf sprouting probability of the tip when segmentLength mod segPerLeaf = 0.
deathProbability	The death probability of the tip when segmentLength mod maxBranchLength = 0.
maxBranchLength	The maximum allowed length of a branch.
maxAliveTips	Maximum allowed alive tips at any point in time.
maxDepth	The maximum depth of all tips in the tree.
segPerLeaf	The amount of leaves per internode segments, i.e. when segmentLength mod segPerLeaf → place a ToLeaf.
minBranchingAngle	The minimum branching angle of a tip when it is split by a node.
maxBranchingAngle	The max branching angle of a tip when it is split by a node.
radiusDec	The decrease in radius at every time-step.
lengthDec	The decrease in length at every time-step.
leafSizeDec	The decrease in leaf size at every time-step.

A.3 Leaves

The local variables of a ToLeaf	
leafScale	The scalar of the size of the leaf.

The global variables of a ToLeaf	
nrOfLeaves	The number of different leaf textures in the system.
leafSizeVariance	Random influence on the size of the leaf.
leafHeight	Height of the starting size of a leaf in the plant
leafWidth	Width of the starting size of a leaf in the plant

A.4 Nodes

The local variables of a Node	
transformation	The Transformation of the node.
Mark	A node symbolizing a branch.

References

[1] J. Arvo and D. Kirk. Modeling plant with environment-sensitive automata. In *Proceedings of Ausgraph '88*, pages 27 – 33, 1988.

[2] Bedrich Benes and Erik Uriel Millan. Virtual climbing plants competing for space. *ca*, 00:33, 2002.

[3] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and developmentr. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1988. ACM.

[4] Greuter et al. Undiscovered worlds – towards a framework for real-time procedural world generation. In *Proceedings of the Fifth Intern. Digital Arts and Culture Conference*, 2003.

[5] Michael Nitsche et al. Designing procedural game spaces: A case study. In *Proceedings of FuturePlay 2006*, 2006.

[6] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics.* Addison Wesley, 1995.

[7] N. Greene. Voxel space automata: modeling with stochastic growth processes in voxel space. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 175–184, New York, NY, USA, 1989. ACM.

[8] Interactive Data Visualization Inc. Speedtree. `http://www.idvinc.com/speedtree/, Accessed 18 February, 2008.`

[9] Johannes Günther Ingo Wald and Philipp Slusallek. Balancing considered harmful – faster photon mapping using the voxel volume heuristic. In *Proceedings of Eurographics). 5*, page 2004, 2004.

[10] C. Jirasek, P. Prusinkiewicz, and B. Moulia. Integrating biomechanics into developmental plant models expressed using l-systems. In Ch and T. Speck, editors, *Plant biomechanics 2000, Proceedings of the 3rd Plant Biomechanics Conference*, pages 615–624, Freiburg-Badenweiler, August 2000. Georg Thieme Verlag.

[11] Catherine Alena Jirasek. A biomechanical model of branch shape in plants expressed using l-systems. Master's thesis, University of Calgary, 2000.

[12] Masayuki Nakajima Johan Knutzen, Suguru Saito. Generating climbing plants using l-systems. In *IWAIT2009: Proceedings of International Workshop on Advanced Image Technology*, 2009.

[13] Mark W. Jones, J. Andreas Bærentzen, and Milos Sramek. 3d distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[14] Jaap A. Kaandorp and Janet E. Kübler. *The Algorithmic Beauty of Seaweeds, Sponges and Corals*. Springer, 2001.

[15] Radoslaw Karwowski. *Improving the Process of Plant Modeling: The L+C Modeling Language*. PhD thesis, University of Calgary, 2002.

[16] George Kelly and Hugh Mccabe. Itb journal a survey of procedural techniques for city generation. *ITB Journal*, 14:87–131, 2006.

[17] Ole Kniemeyer. Rule-based modelling with the xl/groimp software. In Ulrike Brüggemann Harald Schaub, Frank Detje, editor, *The Logic of Artificial Life. Proceedings of 6th GWAL*, pages 56–65. Akademische Verlagsgesellschaft Berlin, 2004.

[18] Ole Kniemeyer. *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling*. PhD thesis, BTU Cottbus, 2008.

[19] H. von. Koch. Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta Math.*, 30:145–174, 1906.

[20] Nigel Lesmoir-Gordon. *Introducing Fractal Geometry*. Totem Books, 2006.

[21] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, Parts I and II(18):280–315, 1968.

[22] B. Lintermann and O. Deussen. Interactive modeling of plants. *Computer Graphics and Applications, IEEE*, 19(1):56–65, Jan/Feb 1999.

[23] Thomas Luft. Ivy generator.
`http://www.ivy-generator.com, Accessed 18 February, 2008.`

[24] B. Chanson M. Fournier, H. Bailleres. Tree biomechanics: growth, cumulative prestresses, and reorientations. *Biomimetics*, 2(3):229–251, 1994.

[25] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, New York, NY, USA, 1996. ACM.

[26] Persistence of Vision Pty. Ltd. (2004). Persistence of vision raytracer (version 3.6).
`http://www.povray.org/download/ , Accessed 22 August, 2008.`

[27] Winfried Kurth Ole Kniemeyer, Gerhard Buck-Sorlin. Groimp as a platform for functional-structural modelling of plants. In *Functional-Structural Plant Modelling in Crop Production*, pages 43–52. Springer, 2006.

[28] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. *SIGGRAPH Comput. Graph.*, 20(4):55–64, 1986.

[29] P Prusinkiewicz. Graphical applications of l-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society.

[30] P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[31] Przemyslaw Prusinkiewicz, Radoslaw Karwowski, Radomir Mech, and Jim Hanan. L-studio/cpfg: A software system for modeling plants. In *Applications of Graph Transformations with Industrial Relevance*, 2000.

[32] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, 19(3):313–322, 1985.

[33] Timothy Roden and Ian Parberry. From artistry to automation: A structured methodology for procedural content creation. In Matthias Rauterberg, editor, *ICEC*, volume 3166 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2004.

[34] G. Leach S. Greuter, N. Stewart. Beyond the horizon. In *Image, Text and Sound 2004: The Yet Unseen: Rendering Stories*. Image Text and Sound Conference 2004, 2004.

[35] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, 2007.

[36] Kichiro SHINOZAKI, Kyoji YODA, Kazuo HOZUMI, and Tatuo KIRA. A quantitative analysis of plant form - the pipe model theory i. basic analysis. *Japanese Journal of Ecology*, 14(3):97–105, 19640601.

[37] J. Taylor-Hell. Biomechanics in botanical trees. Master's thesis, University of Calgary, 2005.

[38] Julia Taylor-Hell. Incorporating biomechanics into architectural tree models. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, page 299, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation for racing games, 2007.

[40] Eric Haines Tomas Akenine-Möller. *Real-Time Rendering*. AK Peters, 2 edition, 2002.

[41] H. Vogel. A better way to construct the sunflower head. *Mathematical Biosciences*, 44:179–189, 1979.

[42] Ingo Wald. On building fast kd-trees for ray tracing, and on doing that in o(n log n). Technical report, 2006.

[43] Gerhard Buck-Sorlin Winfried Kurth, Ole Kniemeyer. Relational growth grammars - a graph rewriting approach to dynamical systems with a dynamical structure. In *Unconventional Programming Paradigms*, volume 3566/2005 of *Lecture Notes in Computer Science*, pages 56–72. Springer Berlin / Heidelberg, 2005.