# Rule-Based Mesh Growing and Generalized Subdivision Meshes

Stefan Maierhofer

January 10, 2002

Dissertation

# Rule-Based Mesh Growing and Generalized Subdivision Meshes

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Prof. Dr. Werner Purgathofer
E186
Institut für Computergraphik

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Stefan Maierhofer
Matr.-Nr. 9425210
Wehlistrasse 51/2/33, A-1200 Wien

Wien, im Dezember 2001

# Kurzfassung

In dieser Arbeit präsentieren wir eine verallgemeinerte Methode zur prozeduralen Erzeugung und Manipulation von Meshes, die im wesentlichen auf zwei verschiedenen Mechanismen beruht: *generalized subdivision meshes* und *rule-based mesh growing*.

Herkömmliche Subdivision-Algorithmen beruhen darauf, dass eine genau definierte, spezifische Subdivision-Vorschrift in wiederholter Folge auf ein Mesh angewendet wird um so eine Reihe von immer weiter verfeinerten Meshes zu generieren. Die Vorschrift ist dabei so gewählt, dass die Ecken und Kanten des Basis-Meshs geglättet werden und die Reihe zu einer Grenzfläche konvergiert welche festgelegten Stetigkeitsansprüchen genügt. Im Gegensatz dazu erlaubt ein verallgemeinerter Ansatz die Anwendung verschiedener Vorschriften bei jedem Subdivision-Schritt. Konvergenz wird im wesentlichen dadurch erreicht, dass die absolute Größe der durchgeführten geometrischen Veränderungen von Schritt zu Schritt geringer wird. Bei genauerer Betrachtung stellt man jedoch fest, dass es in vielen Fällen von Vorteil wäre die stärkere Ausdruckskraft von Subdivision-Vorschriften ohne die oben genannte Einschränkung zu nutzen. Wir schlagen deshalb vor, die Erzeugung eines Submeshs $M^{(n+1)}$ aus einem spezifischen Mesh $M^{(n)}$ in zwei eigenständige Operationen zu zerlegen. Die erste Operation, genannt *mesh refinement*, bezeichnet dabei die Verfeinerung des Meshs durch das Einfügen neuer Eckpunkte und die Festlegung der dadurch neu entstehenden Nachbarschaftsbeziehungen zwischen Eckpunkten, Kanten und Flächen, ohne dabei jedoch bereits die konkreten Positionen der Eckpunkte festzulegen. Erst die zweite Operation, genannt *vertex placement*, berechnet konkrete Positionen für die Eckpunkte. Um dem Anwender eine größtmögliche Flexibilität bei der Spezifikation von Subdivision Surfaces zu bieten, schaffen wir die Möglichkeit verschiedene *refinement* und *vertex placement* Operatoren in sogenannten *mesh operator sequences*, das sind beliebige Sequenzen von Operatoren, zu kombinieren, und diese dann auf konkrete Meshes anzuwenden.

*Rule-based mesh growing* ist eine Erweiterung von parametrisierten Lindenmayer Systemen (pL-Systemen), die jedoch nicht auf der Basis einzelner Symbole, sondern auf der Basis von Symbolen die in einer Nachbarschaftsbeziehung stehen, operieren. Einzelne Symbole repräsentieren dabei Flächen eines Meshs. Dieser Mechanismus erlaubt es, in kontrollierter Art und Weise, komplexe Details in Meshes einzufügen und zwar genau dort wo dies gewünscht wird. Um die Systematik von pL-Systemen auch im Kontext eines Mesh-basierten Rendering-Systems nutzen zu können, führen wir *mesh-based pL-systems* (Mesh-basierte pL-Systeme) ein. Hierbei wird jedes parametrisierte Symbol (linke Seite einer Ersetzungsregel) mit einer oder mehreren Flächen in einem oder mehreren Meshes in Beziehung gesetzt beziehungsweise verknüpft. Die rechte Seite einer Ersetzungsregel ist nun nicht mehr eine lineare Sequenz von Symbolen, sondern ein Mesh dessen Flächen wiederum Symbole zugeordnet sind. Die Topologie eines Objekts, welches mit Hilfe eines solchen Mesh-basierten pL-Systems erzeugt wird, ist automatisch durch die Nachbarschaftsbeziehungen des Meshs festgelegt, und es ist deshalb nicht mehr, so wie dies bei herkömmlichen pL-Systemen der Fall ist, nötig, spezielle Gruppierungssymbole zu verwenden.

Werden beide Mechanismen kombiniert, so erhält man ein Werkzeug mit dem man eine große Anzahl von komplexen Formen und Objekten modellieren kann und mit dessen Hilfe diese auch äußerst kompakt repräsentiert werden können. Wir zeigen dies anhand einer Integration der beschriebenen Mechanismen in ein bestehendes Rendering-System. Die Mesh-basierten pL-Systeme werden dabei mit Hilfe von *directed cyclic graphs* (gerichteten zyklischen Graphen) abgebildet, welche die oben genannte kompakte Repräsentation der Modelle ermöglichen und durch den, von Fraktalen und L-Systemen her bekannten, Effekt der *database amplification* in der Lage sind aus einer solch kompakten Datenbasis komplexe Strukturen zu erzeugen. Auf der Basis dieser Implementierung der grundlegenden Konzepte unseres Ansatzes erstellen wir schliesslich einen Prototypen eines interaktiven Pflanzeneditors mit der Möglichkeit diverse Parameter semiautomatisch aus Photographien von Pflanzen zu extrahieren um so auch die praktische Anwendbarkeit unseres Ansatzes zu demonstrieren.

# Abstract

As a general approach to procedural mesh definition we propose two mechanisms for mesh modification: *generalized subdivision meshes* and *rule-based mesh growing*.

In standard subdivision, a specific subdivision rule is applied to a mesh to get a succession of meshes converging to a limit surface. A generalized approach allows different subdivision rules at each level of the subdivision process. By limiting the variations introduced at each level, convergence can be ensured; however in a number of cases it may be of advantage to exploit the expressivity of different subdivision steps at each level, without imposing any limits. We propose to split the process of generating a submesh $M^{(n+1)}$ from a specific mesh $M^{(n)}$ into two distinct operations. The first operation, which we call *mesh refinement*, is the logical introduction of all the new vertices in the submesh. This operation yields all the connectivity information for the vertices of the submesh without specifiying the positions of these newly introduced vertices. The second operation, which we call *vertex placement*, is the calculation of the actual vertex positions. In order to obtain maximum flexibility in generating subdivision surfaces, we make it possible for the user to independently specify both of these operations, by offering a number of refinement and vertex placement operators, which may be arbitrarely combined in user-specified *mesh operator sequences*, which in turn are applied to particular meshes.

Rule-based mesh growing is an extension of parametric Lindenmayer systems (pL-systems) to not only work on symbols, but connected symbols, representing faces in a mesh. This mechanism allows the controlled introduction of more complex geometry in places where it is needed to model fine detail. In order to use pL-systems in the context of a mesh-based modeling system, we introduce *mesh-based pL-systems*, by associating each parameterized symbol of the system with one or more faces in one or more meshes. Thus the right-hand side of each production rule is not a linear sequence of symbols, but a template mesh with each face again representing a symbol. Thereby the topological structure of an object generated with such a mesh-based pL-system is automatically encoded in the connectivity information of the mesh, and we do not need to introduce grouping symbols in order to encode the hierarchical structure, like it is necessary in standard pL-systems.

Using both these mechanisms in combination, a great variety of complex objects can be easily modeled and compactly represented. We demonstrate this by including the proposed framework in a general-purpose rendering system. *Directed cyclic graphs* are used to represent mesh-based pL-systems, and from this compact representation complex geometry is generated due

to the effect of *database amplification*, known from fractals and L-systems. Finally, this implementation of the main concepts of our approach is used as a basis for an interactive plant editor, and an appositional user interface for semi-automatic parameter extraction from photographs of plants, in order to demonstrate the applicability of our approach to real-world applications.

*To Alexandra*

## Acknowledgements

*I would like to express my gratitude to my parents and everyone else who supported me in one way or the other during my years of studies in Vienna.*

*I would like to thank Werner Purgathofer, Christian Breiteneder, and Dieter Schmalstieg.*

*Finally I would like to thank my advisor Robert F. Tobler who provided much appreciated support and ideas at key times. He was always there with new suggestions and his ongoing confidence in me was essential to the completion of this work.*

# Contents

# Chapter 1

# Introduction

Many of the most beautiful shapes are created by nature. From microscopic pollen to giant Redwood trees, one of the secrets of beauty is complexity. In most naturally grown shapes, complexity is an integral part on every scale. This is still unmatched in state-of-the-art computer graphics. Imagine a tree, exposed to storms, draught and other natural disasters over the course of centuries. In state-of-the-art computer graphics, a model of such a tree would of course exhibit a reasonable appearance on a large scale, but as soon as one would start zooming in, complexity would start to vanish. First, one would recognize that the bark is in fact made up of high resolution texture maps and bump maps which look fine from a distance, but are perfectly flat and smooth otherwise. On a second look, one would recognize the absence of myriads of cracks and knots, and branching structures would probably exhibit sharp edges and unnatural transitions. Altogether, computer generated natural shapes still look too perfect and sterile.

But what is really the difference between a state-of-the-art computer generated tree and a real tree? Real trees grow according to rules which are encoded in their genes. The well-known concept of L-systems is used to simulate and model exactly this type of processes. But this is not enough. No two trees or plants look exactly the same. Even if two identical seeds of the same species are planted next to the other, the resulting shapes would differ, because it is still a long way from the genotype, which is encoded in the genes, to a distinct phenotype, which is the physical appearance of a distinct individual. Each individual plant has to obey the laws of nature and its development is influenced by a score of environmental parameters, like gravity, incident direction of light, amount of water, moisture, distribution of nutriments in the soil and many more. The plants appearance and development is influenced by these constraints on every scale. A score of other incidents, like natural desasters, storms, heavy snowfall, or ice, may damage or break

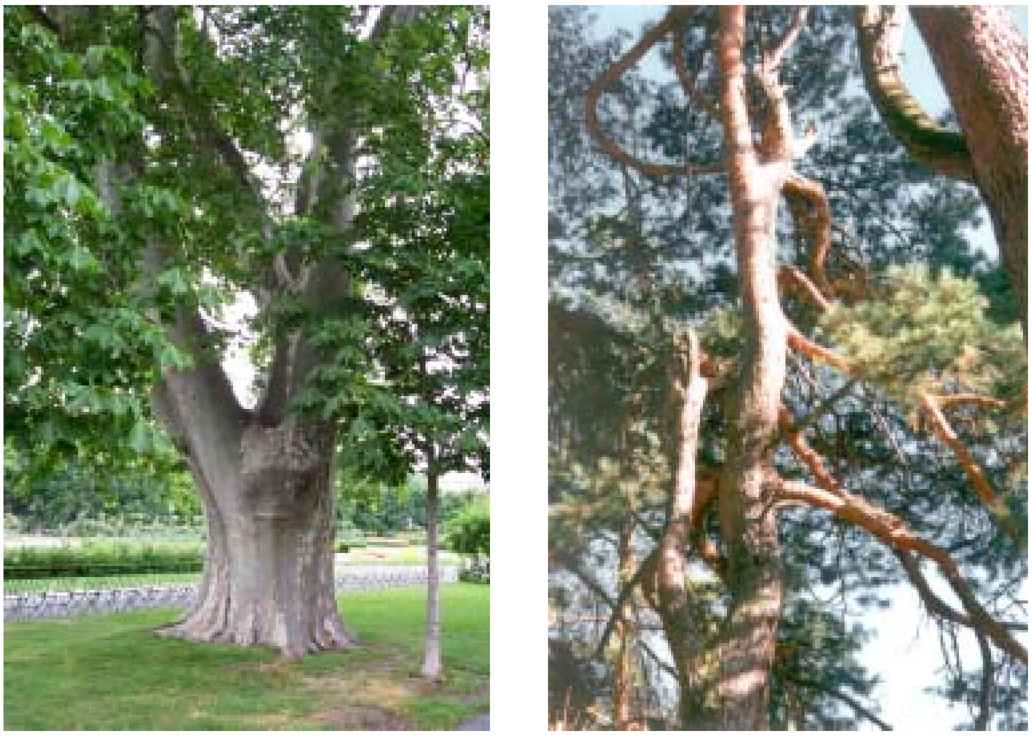


Figure 1.1: A mighty tree (left), which has been spared from natural desaster, could develop a rich and impressive appearance. Another, less fortunate, individual (right) suffered from severe injuries inflicted by storm and/or heavy snowfall and ice, and consists primarely of traumatic reiterations. Nevertheless, it offers another kind of fascinating visual appearance.

stems, branches, twigs, or ruin everything from individual plants to whole populations. Animals, from insects to deer, plagues, chemical reactions, and of course human interference, whether it is direct, like when pruning trees in a garden, or indirect like disturbing the balance in an ecosystem - every single incident will leave a mark on the appearance of an individual plant.

## 1.1 Motivation

As stated above, the main problem of creating realistic and detailed natural shapes is the lack of geometric detail in state-of-the-art applications. Either the overall shape is modeled in sufficient detail, but small scale details are neglected at all or approximated by texture maps or bump maps, or small scale detail is modeled on relatively simple global shapes. The first applies to L-system-based modeling applications, while the latter applies to procedural

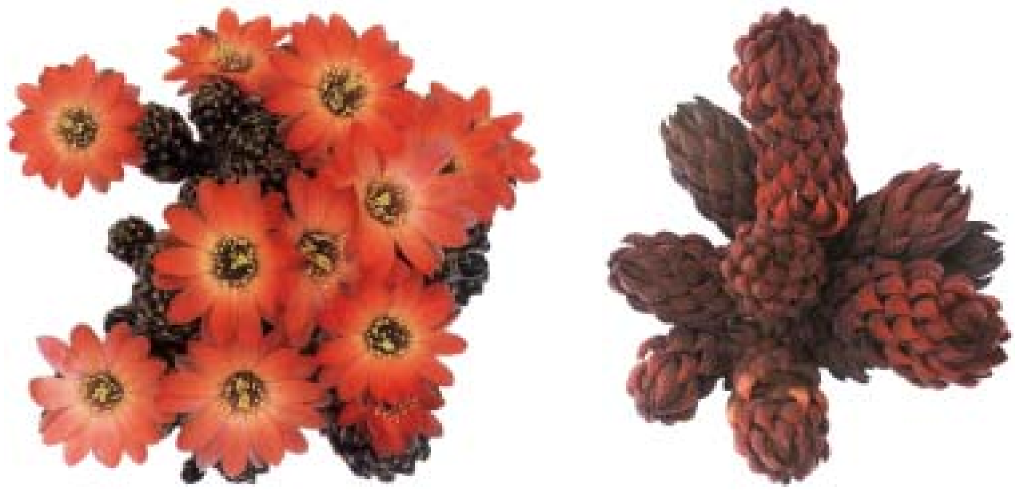


Figure 1.2: Even relatively simple plants like cacti and succulents create amazingly rich shapes. Intensely colored, orange flowers (left) are produced in abundance from the lower stem of a *rebutia pygmaea* [Hew97]. *Haworthia coarctata v. adelaidensis* (right) is one of the few species in its genus to form such striking clumps of tough, leafy columns. Normally dark green, it turns a rich purple-red in full sun [Hew97].

modeling techniques. While both tools can represent very complex geometry in a very compact way, the same holds not true (in general) for the combination of both techniques. Certainly, it is not impossible to combine both strategies to create realistic and detailed shapes as can be seen especially in *computer generated imagery* and *special effects* for a number of high-budget film productions (Final Fantasy [Tri01], Shrek [Dre01], Toy Story 2 [Pix00], to name just a few) but this is associated with an immense modeling effort and even with dozens of specialist designers at hand, most effort is "wasted" for modeling leading characters. One of the main reasons for this is, that due to the lack of a common framework, the integration of different modeling techniques is mostly performed by hand which in turn requires a huge manual effort, as stated above. Of course it is not possible to solve these problems all at once, and certainly much more research has to be performed by the computer graphics community for many years to come.

## 1.2 Purpose and Outline of This Thesis

The goal of this thesis is to improve the way highly complex shapes are modeled, by *combining* and *advancing* a number of existing techniques from subdivision surfaces to parametric L-systems and thus to bridge the gap between different powerful modeling techniques. The goal is a novel framework allowing for the generation of very complex geometric shapes of arbitrary topology. The main focus thereby lies in the *generation* of complex geometry - means of efficient rendering of such complex geometry would go beyond the scope of this thesis and will be subject to future work. A short outlook and basic ideas are given in Chapter 8.

Before presenting our proposed framework, we first give a detailed overview of *subdivision surfaces* and *L-systems* as well as some chosen *plant generation models*. We then introduce *generalized subdivision meshes* which are capable of generating not only smooth surfaces by the application of standard subdivision surface algorithms, but also small scale surface detail by means of procedural modeling techniques. The essential improvement and focal point of this thesis is *rule-based mesh growing* which is an extension of L-systems to not only work on symbols, but on connected symbols, representing faces in a mesh. Finally we conclude this thesis by outlining the application of *generalized subdivision meshes* and *rule-based mesh growing* to a simple plant generation model, which serves as a testbed and reference application for our proposed framework.

# Chapter 2

# Background and Related Work

In this chapter we will provide an overview of related work which will be used throughout the rest of this thesis. After some basic definitions concerning polygonal surface representations, we will provide an overview of subdivision surfaces. Then we will describe Lindenmayer systems, and finally we will review a selected number of plant models.

## 2.1 Polygonal Surface Representation

In the most general sense, a polygonal surface model is simply a set of planar polygons in the three-dimensional Euclidean space $\mathbb{R}^3$. For the rest of this thesis, we assume, that the connectivity of a model is *consistent* with its geometry – if the corners of two polygons coincide in space then those polygons share a common vertex. Further on we will neglect isolated vertices and edges, since their only effect is to complicate the implementation; the underlying algorithms remain the same.

Given these assumptions, we make the following definition: a polygonal surface model $M = (V, F)$ consists of a list of vertices $V$ and a list of polygonal faces $F$. The vertex list $V = (v_1, v_2, ..., v_{nv-1})$ is an ordered sequence; each vertex may be identified by a unique integer $i$. The face list $F = (f_1, f_2, ..., f_{nf-1}n)$ is also ordered, assigning a unique integer number to each face, where each face $f_i = (v_{i_0}, v_{i_1}, ..., v_{i_{nvf-1}})$ is an ordered sequence of vertex indices referring to the vertex list $V$, and enumerating the vertices of the face in counterclockwise order.

In computer graphics polygonal surface models are mostly used to model *manifold* surfaces. The intuitive concept of a manifold surface [Gar99] is that people living on it, their perception limited to a small surrounding area, are unable to distinguish their situation from that of people actually living on a plane. More formally, a *manifold* is a surface, all of whose points have a neighbourhood which is topologically equivalent to a disk. A *manifold with boundary* is a surface all of whose points have a neighbourhood which is topologically equivalent to either a disk or a half-disk.

A polygonal surface is a manifold (with boundary) if every edge has exactly two incident faces (except edges on the boundary which must have exactly one), and the neighbourhood of every vertex consists of a closed loop of faces (or a single fan of faces on the boundary).

### 2.1.1 Winged-Edge Representation of Polygonal Surfaces

The winged-edge data structure ist the most prevalent representation of polygonal surfaces. Originally proposed in the nineteen-seventies by Baumgart [Bau72], [Bau75], it has stood the test of time.

Although there exist countless different implementations, the basic idea is always the same: lists for vertices, edges, and faces are maintained, and each vertex, edge, and face stores indices for adjacent elements which point back into the appropriate lists. Which elements will store which indices mostly depends on which queries a specific application needs to perform. The more adjacency information is stored, the *richer* the data structure becomes. Of course richer structures result in simpler and more performant queries, but also in a higher memory footprint. So in praxis there always exists a trade-off between memory requirements and simplicity of queries. A detailed description of the winged-edge data structure used in our implementation is given in Section 5.4.

### 2.1.2 Non-Polygonal Representations

Polygonal surface representations are not the only available surface representations. A number of alternatives exist [Wat93], [HB97], which all provide certain benefits as well as drawbacks.

#### Implicit Surface Representations

Implicit surface representations exist for a number of *primitive objects* commonly used in rendering packages. *Quadric objects* like sphere, ellipsoid, cylinder, cone, torus, paraboloid, hyperboloid are described with second-degree equations. As an example, the representation of an ellipsoid centered on the origin is defined by

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{2.1}$$

where $r_x$, $r_y$, and $r_z$ denote the radii in direction of the main axes in a Cartesian coordinate system.

A class of objects called *superquadrics* is a generalization of quadric objects which is formed by incorporating additional parameters into the quadric equations to provide increased flexibility for adjusting object shapes. A generalization of Equation 2.1 could look like

$$\left[\left(\frac{x}{r_x}\right)^{\frac{2}{s_2}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s_2}}\right]^{\frac{s_2}{s_1}} + \left(\frac{z}{r_z}\right)^{\frac{2}{s_1}} = 1 \tag{2.2}$$

where $s_1$ and $s_2$ are two additional exponent parameters. For $s_1 = s_2 = 1$, we have an ordinary ellipsoid.

Another class of objects can be used to represent objects whose surface changes according to motion or proximity to other objects, e.g. molecular structures, water droplets, or melting objects - since their shapes show a certain degree of fluidity, they are often referred to as *blobby objects*.

### Spline Representations

Spline representations are one of the most widely used alternatives to polygonal representations. Hearn and Baker [HB97] give a detailed overview and classification of different properties of spline representations.

Splines are mathematically described by *piecewise polynomial functions* whose first and second derivatives are continuous across the various curve sections. We will give a brief overview of *spline curves* (*spline surfaces* can be simply described with two sets of orthogonal spline curves; the mathematical properties stay essentially the same).

Spline curves are specified by a set of *control points*, which define the basic shape of the curve. If the resulting curve passes through the control points, it is said to be *interpolating*, if the curve not necessarely passes through the control points, then it is said to be *approximating*.

In order to create smooth transitions between the different sections of the curve, which are defined by piecewise cubic polynomal functions, a number of *continuity conditions* can be imposed at the connection points.

*Parametric continuity* is achieved, if parametric derivatives of adjoining sections are matched at their common boundary point. $C^0$ continuity (*zero-order parametric continuity*) simply means that the curves meet. $C^1$ (*first-order parametric continuity*) means that the tangent lines at the common boundary point are equal, and $C^2$ (*second-order parametric continuity*) states that the second-order derivatives of two curve sections are the same at their intersection.

Another way of defining conditions for joining sections is to use *geometric continuity conditions*. Here, it is only required that the parametric derivates of two joining curves are proportional at their intersection, not necessarely equal. $G^0$ (*zero-order geometric continuity*) is the same as $C^0$. $G^1$ (*first-order geometric continuity*) states, that the direction of the tangent vector is the same, but not necessarely its magnitude. $G^2$ (*second-order geometric continuity*) is defined accordingly.

Spline curves with cubic basis functions offer a good compromise between flexibility and speed of calculation. Representatives of this class of functions are: *natural cubic splines*, *Hermite splines*, *cardinal splines*, and *Kochanek-Bartels splines*.

Other common spline curves (and their respective surface interpretations) are *Bézier curves and surfaces*, *B-splines*, *uniform B-splines*, *non-uniform B-splines*, *rational splines*, and *non-uniform rational B-splines* (NURBS).

### 2.1.3 Multiresolution Modeling and Levels of Detail

Conventional polygonal models consist of a fixed set of vertices and faces, and as such have only *one* fixed representation. But, this single representation may not be appropriate for each possible configuration in a scene. Take, for example, the model of a cactus (Color Plate 7.2) which is made up of a rather simple global shape, with thousands of small prickles, and with the camera positioned directly in front of one single prickle, which of course is depicted in great detail.

In such a scenario, the sheer attempt to create a single model to be used for all instances of prickles, would be futile. A model which is detailed enough to adequately represent the prickle in front, would result in an incredible waste of resources for all the distant prickles, where in practice thousands of polygons would be projected to the very same pixel in the final image. The opposite holds true for the contrary case, where the model is designed to meet the much weaker requirements for distant prickles.

Garland [Gar99] gives an overview of different approaches to multiresolution representations. The simplest method for creating multiresolution surface models is called *discrete multiresolution*. In this case, a set of increasingly simpler models is generated, and according to parameters like the distance from the camera to the object, the renderer selects the appropriate level of detail to use in rendering.

Unfortunately, this can lead to so-called *popping artifacts* in animations, if in-between two frames, the renderer switches between two representations. To avoid this, *level of detail blending* has been introduced, which extends the level of detail transition over several frames.

In *geomorphing*, two consecutive levels of detail are smoothly interpolated. This technique has been used for terrain-specific applications [FST92] for some time. But imagine a terrain model, where the camera is positioned above the surface and looks out to the horizon. In such a case no discrete level of detail can be found which represents the terrain in an optimal way. Either, the surface is overly detailed in the distance, or it is too crude near the camera. What we need is an approximation which allows the level of detail to vary continuously over the surface. These techniques are called *continuous level of detail* models and are extremely useful in terrain applications [Hop98].

To end this section, we have to annotate, that our proposed *generalized subdivision meshes* and *rule-based mesh growing* schemes are based on polygonal surface representations, and are well suited for multiresolution modeling and level of detail approaches. In the next section we will discuss the creation of smooth surfaces by means of subdivision techniques.

## 2.2 Subdivision Surfaces

Subdivision surfaces have been introduced by Catmull and Clark [CC78] and Doo and Sabin [DS78] in the late nineteen-seventies, and are used to efficiently generate smooth surfaces from arbitrary initial meshes. For a long time the theoretical foundation of the subdivision process was not as thorough as for other modeling techniques such as B-splines and NURBS [PT97], and thus it took a while for subdivision methods to become widely known and used. Recently this has been rectified by the introduction of a method to evaluate subdivision surfaces at any point [Sta98], a method for extending subdivision surfaces for emulating NURBS [SZSS98], the addition of normal control to subdivision surfaces [BLZ00], and a method to closely approximate Catmull-Clark subdivision surfaces using B-spline patches [Pet00]. A number of other extensions to subdivision surfaces, like semi-sharp creases [DKT98], or displaced subdivision surfaces [LMH00], have established them as the modeling tool of choice for generating topologically complex, smooth surfaces. In 1997, Pixar's animated short film *Geri's Game* which was modeled using Catmull-Clark subdivision surfaces, even won an *Academy Award* in the category *Best Animated Short Film*.

There exists a huge number of different subdivision schemes which, at first glance, may appear chaotic. However, most of the different subdivision schemes can be easily classified by three different criteria [ZSD+99]:

- *type of refinement*: *primal* (vertex insertion) or *dual* (corner-cutting)
- *type of mesh*: *triangular* or *quadrilateral*
- *type of scheme*: *approximating* or *interpolating*

In a *primal* refinement scheme each vertex $v_i^{(m)}$ in mesh $m$ has exactly one associated vertex point $v_i^{(m+1)}$ in the submesh $m+1$. *Dual* refinement (corner cutting) means, that the corner defined by a vertex $v_i^{(m)}$ is cut away in the next level of subdivision. Therefore no single vertex point, but a unique face, can be associated with the vertex in the resulting submesh. This also implies, that *dual* schemes can never be *interpolating*, but only *approximating* schemes.

While subdivision surface techniques use recursive refinement to obtain smooth surfaces, the field of procedural modeling uses various similar principles to add detail to surfaces at different levels of resolution. One example for such a procedural modeling strategy is the generation of fractal surfaces by adding random variations at each level of recursive refinement [FFC82]. These surfaces have been demonstrated to be very useful for modeling natural phenomena like terrains and other complex geometry and similar principles are also used in *generalized subdivision meshes* (see Chapter 3).

In the following sections, a number of widely known and practically important subdivision schemes will be explained and classified using the three criteria introduced above.

### 2.2.1 Catmull-Clark Subdivision

The Catmull-Clark subdivision scheme is an approximating, primal scheme and can be applied to an arbitrary polyhedron called the control mesh. The control mesh $M^{(0)}$ is subsequently subdivided to produce a sequence of meshes $M^{(1)}$, $M^{(2)}$, ..., $M^{(\infty)}$ . The averaging rules, or subdivision rules have been chosen such, that the limit surface can be shown to be tangent plane smooth no matter where the control vertices are placed.

In one subdivision step, each face is split into a collection of quadrilateral subfaces. A face with $n$ vertices is split into $n$ subfaces. The vertices of the submesh are computed using certain weighted averages as detailed below. More precisely, for each vertex, edge and face of a mesh $M^{(m)}$, a new vertex point $v_j^{(m+1)}$, a new edge point $e_j^{(m+1)}$, and a new face point $f_j^{(m+1)}$ is created.

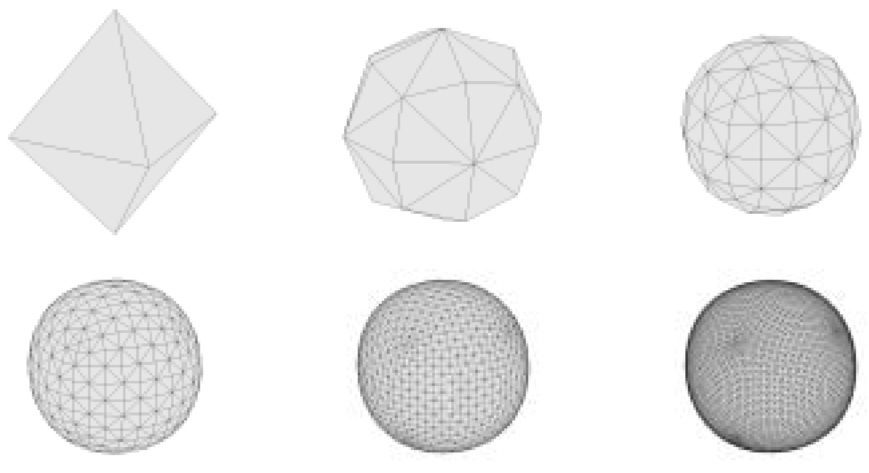


Figure 2.1: Sequence of Catmull-Clark subdivision steps.

The union of all face-, edge- and vertex points forms the mesh at the next level of subdivision.

Face points (see Figure 2.2) are placed exactly at the centroid of the vertices of the corresponding face (where $v^{(m)}_{fi,i}$ is the $i^{th}$ vertex of face $fi$ (face index) of subdivision level $m$):

$$f^{(m+1)}_{fi} = \frac{1}{n}\sum_{i=0}^{n} v^{(m)}_{fi,i} \tag{2.3}$$

Edge points (see Figure 2.2) are placed at the centroid of the edges vertices and the two new face points of the edges neighbouring faces (where $v^{(m)}_{ei,0}$ and $v^{(m)}_{ei,0}$ are the edges $ei$ (edge index) end points and $f^{(m+1)}_{left}$ and $f^{(m+1)}_{right}$ are the face points of the adjacent face, respectively):

$$e^{(m+1)}_{ei} = \frac{v^{(m)}_{ei,0} + v^{(m)}_{ei,1} + f^{(m+1)}_{left} + f^{(m+1)}_{right}}{4} \tag{2.4}$$

For the calculation of a vertex point (see Figure 2.3) all the vertex' neighbouring vertices and the face points of all adjacent faces are used to calculate the position of the new vertex point (where $v^{(m)}_{vi,i}$ is the $i^{th}$ neighbouring vertex of vertex $vi$ (vertex index) of subdivision level $m$). Vertices of valence 4 are called ordinary; others are called extraordinary:

$$v^{(m+1)}_{vi} = \frac{n-2}{n}v^{(m)}_{vi} + \frac{1}{n^2}\sum_{i=0}^{n} v^{(m)}_{0,i} + \frac{1}{n^2}\sum_{i=0}^{n} f^{(m+1)}_{0,i} \tag{2.5}$$

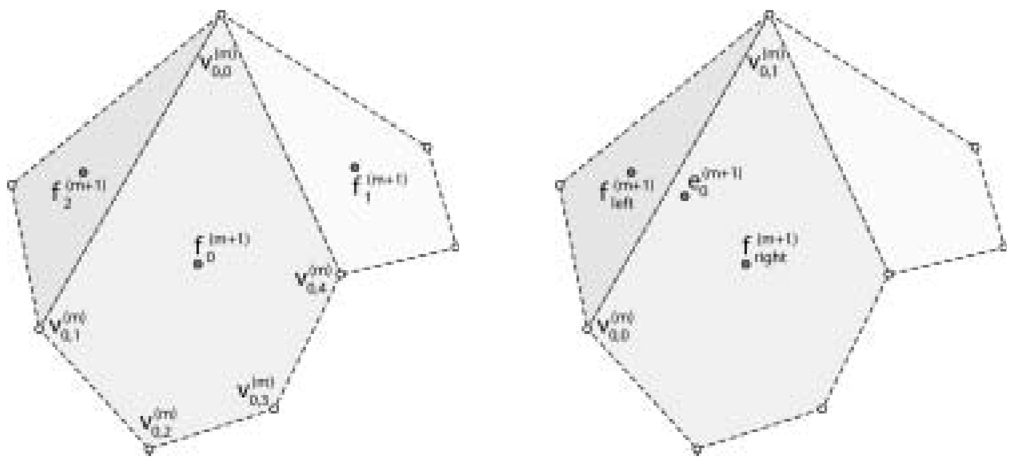


Figure 2.2: Face points (left) are placed exactly at the centroid of the vertices ($v_{0,0}^{(m)}$ – $v_{0,4}^{(m)}$) of the corresponding face. An edge point (right) is placed at the centroid of the end points of the corresponding edge and the face points of the adjacent faces.

Finally, the newly created vertices are connected to form the submesh. See

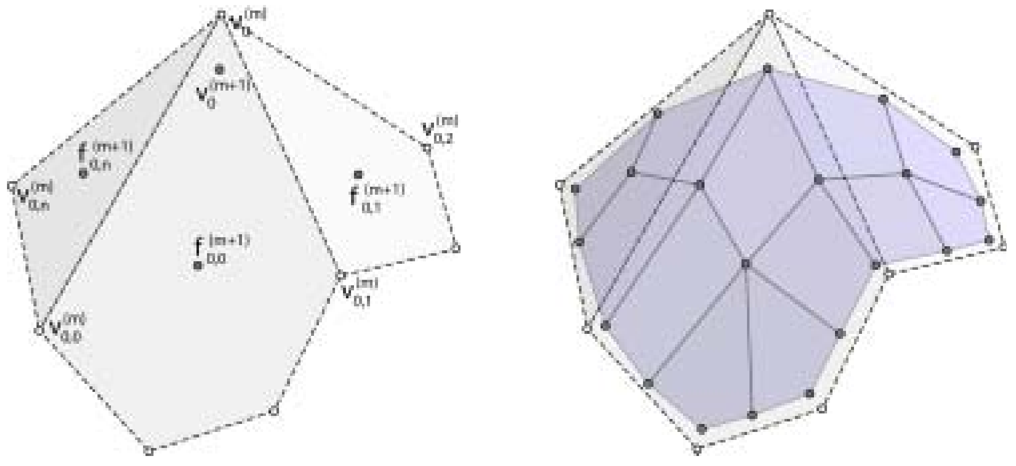


Figure 2.3: A vertex point (left) is the weighted average of the corresponding vertex and the adjacent vertex- and face points. A mesh and its corresponding submesh (right).

Figure 2.3 for the submesh topology.

### Sharp Creases

In order to make subdivision surfaces more useful, DeRose et al. [DKT98] presented an expansion to Catmull-Clark subdivision surfaces which allows for the modeling of fillets and blends. Real world objects are almost never perfectly smooth nor do they have infinitely sharp creases. A corner of a table looks sharp at a distance but is smooth if viewed from a close distance. In order to allow for the modeling of such geometry, an edge sharpness value $s \geq 0.0$ may be assigned to each edge of the control mesh.

If an edge is tagged as sharp, then a different set of subdivision rules is applied. Edge points are placed at the edge midpoint:

$$e_{ei}^{(m+1)} = \frac{v_{ei,0}^{(m)} + v_{ei,1}^{(m)}}{2} \tag{2.6}$$

Vertices having 3 or more incident sharp edges are called corners and are placed using the corner rule:

$$v_{vi}^{(m+1)} = v_{vi}^{(m)} \tag{2.7}$$

Vertices having less than 3 incident sharp edges are called crease vertices and are placed using the crease vertex rule (where the sharp edges are $v_{vi}^{(m)} v_j^{(m)}$ and $v_{vi}^{(m)} v_k^{(m)}$):

$$v_{vi}^{(m+1)} = \frac{v_j^{(m)} + 6v_{vi}^{(m)} + v_k^{(m)}}{8} \tag{2.8}$$

If an edge has an assigned sharpness value of $s$, then the sharpness values for its two corresponding subedges are set to $s - 1$.

If an edge has an associated sharpness value of $s < 1.0$, then the resulting subvertex is calculated as a linear interpolation of the two vertices that result from the application of both the original rules ($v_{smooth}$) and the rules for sharp creases ($v_{sharp}$).

$$v^{(m+1)} = v_{smooth}^{(m)} s + v_{sharp}^{(m)} (1.0 - s) \tag{2.9}$$

Edges with a sharpness value of $s = 0.0$ are not sharp and therefore the original, smooth Catmull-Clark subdivision rules are applied.

### Exact Evaluation

One problem with subdivision surfaces in general and Catmull-Clark subdivision surfaces in particular is, that an explicit subdivision process generates such high numbers of polygons, that it is very difficult to deal with them efficiently. However, Stam [Sta98] presents a method which allows for a direct evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values - without explicitely subdividing. The cost of this technique is comparable to the evaluation of a bi-cubic surface spline. Stam shows that the surface can be evaluated in terms of a set of Eigenbasis functions which depend only on the subdivision scheme and for which analytical expressions can be derived. These expressions do not only work on the regular part of a mesh but also in the neighbourhood of extraordinary vertices.

#### Normal control

Biermann [BLZ00] et al. introduce improved rules for Catmull-Clark and Loop subdivision that overcome several problems with the original schemes, namely, lack of smoothness at extraordinary boundary vertices and folds near concave corners. In addition, it allows the generation of surfaces with prescribed normals, both on the boundary and in the interiour. This considerably improves control of the shape of the surfaces.

### 2.2.2 Loop Subdivision

The Loop subdivision scheme [Loo87] is an approximating, primal scheme, and operates on triangular meshes. The refinement rules are as follows (see Figure 2.4 for subdivision coefficients):

- For each vertex $P_i$ a new vertex $P_i'$ is generated.
- For each edge $e_i$ a new vertex $E_i$ is generated.
- For each triangular face, create four new triangular faces (see Figure 2.4).

### 2.2.3 Doo-Sabin Subdivision

The Doo-Sabin subdivision scheme [DS78], [Joy96] is a dual (corner cutting), approximating scheme which works on meshes of arbitrary topology. After the first subdivision step, all vertices have valence four, which is a characteristic of the Doo-Sabin scheme. Subdivided faces are quadrilaterals, except around vertices which had a valence other than four in the original mesh. In this case the resulting face is a n-sided polygon, where $n$ is the valence of the original vertex. These facets continue throughout the subdivision process and converge to extraordinary points. The subdivision scheme for meshes of arbitrary topology is as follows:

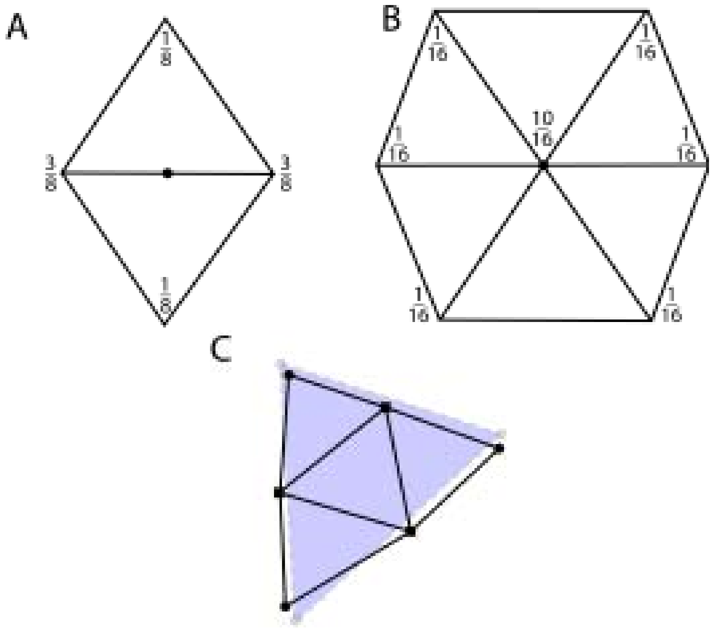


Figure 2.4: Loop subdivision scheme. (A) subdivision coefficients for new edge point (B) subdivision coefficients for new vertex point (C) connectivity of sub-faces

- For each vertex $P_i$ of each face, a new vertex $P_i'$ is generated as the average of the vertex $P_i$, the two adjacent edge points $E_{i,0}$ and $E_{i,1}$, and the face point $F$. An edge point is simply taken as the mid-point of an edge, and a face point is taken as the center-point of the face. See Figure 2.5 A and B.
- For each face, connect the newly generated points for each vertex of this face. See Figure 2.5 C.
- For each vertex, connect the points that have been generated for the faces that are adjacent to this vertex. See Figure 2.5 D.
- For each edge, connect the points that have been generated for the faces which are adjacent to this edge. See Figure 2.5 E.

See Figure 2.5 F for the resulting subdivision mesh after one subdivision step.

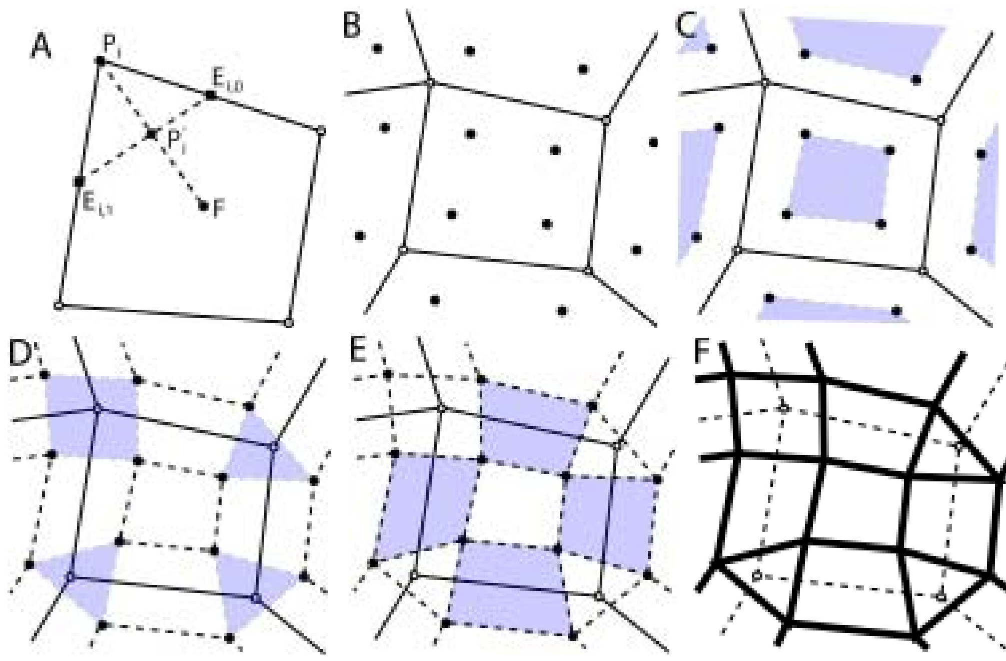


Figure 2.5: Doo-Sabin subdivision scheme. (A) calculation of a vertex point (B) vertex points of a mesh (C) new face-faces (D) new vertex-faces (E) new edge-faces (F) subdivision mesh after one subdivision step

### 2.2.4 Modified Butterfly Scheme

The Butterfly scheme is a primal, interpolating subdivision scheme, which operates on triangular meshes. The original scheme [DLG90] yields $C^1$ surfaces in the topologically regular setting. Unfortunately it exhibits undesirable artifacts in the case of an irregular topology.

Zorin et al. [ZSS96] derive an improved scheme, which fixes the problems of the original approach, but retains the simplicity of the Butterfly scheme, is interpolating, and results in smoother surfaces.

### 2.2.5 Kobbelt Subdivision

The Kobbelt subdivision scheme [Kob96], [Kob98] is an interpolating, primal scheme, which works on quadrilateral meshes. Vertex points are fixed due to the interpolatory property of the scheme. The rules for edge and face points use simple affine combinations of vertices from the unrefined mesh (see Figure 2.6 for details).

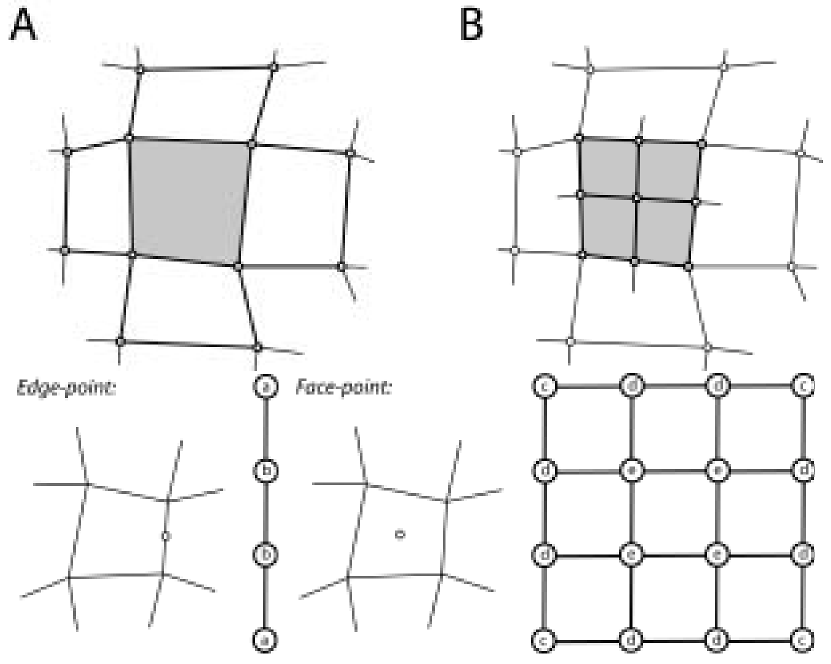


Figure 2.6: Kobbelt subdivision. The refinement operator splits one quadrilateral face into four. The new vertices can be associated with the edges and faces of the unrefined mesh (A)(B). Subdivision masks for regular regions (edge points and face points). $a = -\frac{1}{16}$, $b = \frac{9}{16}$, $c = a^2$, $d = ab$, $e = b^2$.

### 2.2.6 $\sqrt{3}$-Subdivision

$\sqrt{3}$-subdivision [Kob00] is a dual, approximating scheme, which works on triangular meshes and has a number of advantages over the usual *dyadic* subdivision schemes (see Figure 2.7). It performs a *slower* topological refinement than usual split operations. In each step, the number of triangles increases by a factor of 3, and if the subdivision operator is applied twice, it creates a uniform refinement with tri-section of every original edge. Standard *dyadic* split operations increase the number of triangles by a factor of 4 in each step, and two *dyadic* split operations would cause a quad-section of every original edge.

The splitting operation allows for locally adaptive refinement under built-in preservation of mesh consistency without temporary crack-fixing between neighbouring faces from different refinement levels. The size of the surrounding mesh area which is affected by selective refinement is also smaller than for the *dyadic* split operation.

The refinement operator works as follows: for each face, a new face vertex

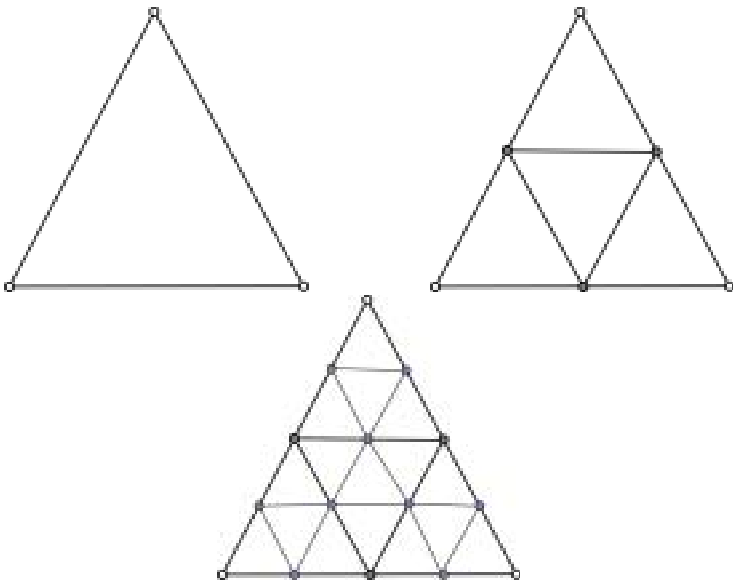


Figure 2.7: Dyadic split. Two dyadic splits will quad-section every original edge.

which is located at the face center is inserted. Next, the new face vertex is connected to the adjacent vertices, creating three new triangles. Finally, the original edges are flipped yielding the final result which is a 30 degree rotated regular mesh.

### 2.2.7 Non-Uniform Rational Subdivision Surfaces

Sederberg et al. [SZSS98] develop rules for Doo-Sabin and Catmull-Clark subdivision surfaces that generalize non-uniform tensor product B-spline surfaces to arbitrary topologies. This added flexibility allows the natural introduction of features like cusps, creases, and darts, while everywhere else the same order of continuity as in their uniform counterparts is maintained. The subdivision scheme works as follows:

Each face is replaced with a new face point.

$$\begin{aligned} F_1 &= [(t_3+2t_4)(s_2+2s_1)P_0+(t_3+2t_4)(s_2+2s_3)P_1+ \\ &\quad (t_3+2t_2)(s_2+2s_3)P_5+(t_3+2t_2)(s_2+2s_1)P_2] \\ &\quad /[4(t_2+t_3+t_4)(s_1+s_2+s_3)] \end{aligned}$$

For each edge a new edge point is created.

$$E_1 = \frac{t_2F_1+t_3F_4+(t_2+t_3)M_1}{2(t_2+t_3)}$$

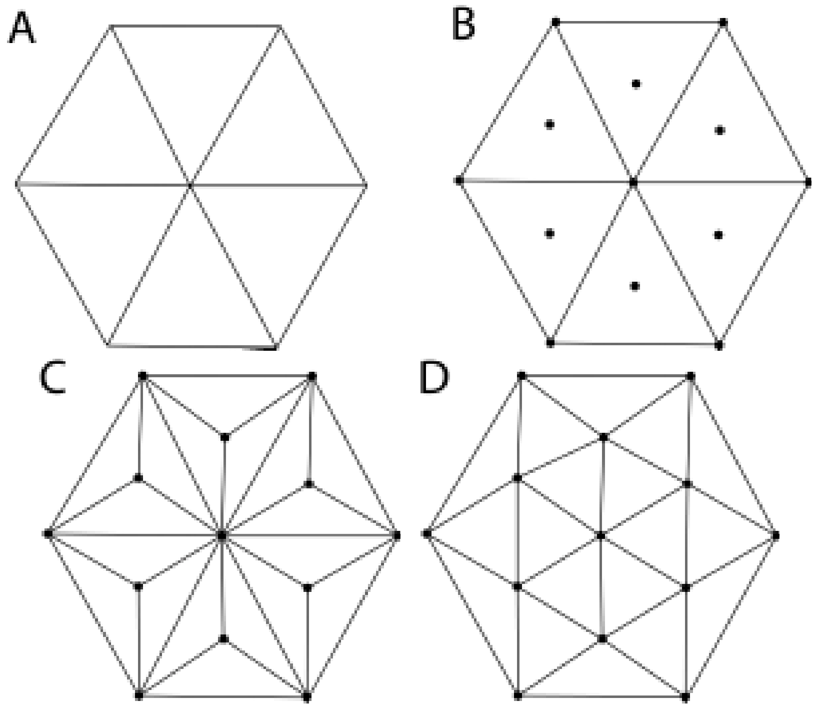


Figure 2.8: $\sqrt{3}$ subdivision. (A) base mesh (B) insert a new vertex for every face (C) connect new face points with adjacent vertices (D) flip original edges

where

$$M_1 = \frac{(2s_1 + s_2)P_0 + (s_2 + 2s_3)P_1}{2(s_1 + s_2 + s_3}$$

Finally, each original vertex is replaced with a vertex point.

$$\begin{aligned} V \quad = \quad & \frac{P_0}{4} + \frac{s_3t_2F_1 + s_2t_2F_2 + s_2t_3F_3 + s_3t_3F_4}{4(s_2 + t_3)(t_2 + t_3)} + \\ & \frac{[s_3(t_2 + t_3)M_1 + t_2(s_2 + s_3)M_2 + s_2(t_2 + t_3)M_3 + t_3(s_2 + s_3)M_4}{4(s_2 + s_3)(t_2 + t_3)} \end{aligned}$$

### 2.2.8 Ray Tracing of Subdivision Surfaces

Kobbelt et al. [KDS98] present the necessary theory for the integration of subdivision surfaces into general purpose rendering systems. The most important functionality is the computation of surface points and normals as well as the ray intersection test. Kobbelt et al. derive the necessary

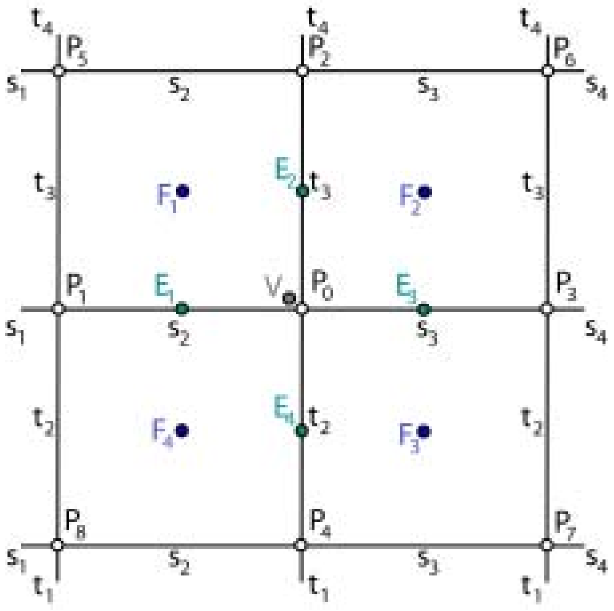


Figure 2.9: Non-uniform rational subdivision surfaces: face, edge and vertex points.

formulaes and present envelope meshes which have the same topology as the control meshes but tightly circumscribe the limit surface. Ray tracing is accomplished by recursively tracing a ray through the forest of triangles emerging from adaptive refinement of a mesh.

## 2.3 L-Systems

Lindenmayer introduced a formalism for simulating the development of multicellular organsisms, subsequently named L-systems [Lin68], [LP89], [PH91], [FPdB90]. Smith [Smi84] introduced state-of-the-art computer graphics techniques to visualize the structures and processes being modeled. Smith also attracted attention to the phenomenon of *database amplification*, which denotes the creation of complex structures from compact datasets, which is inherent in L-systems and forms the cornerstone of L-system applications to image synthesis. The introduction of turtle graphics interpretation of L-systems [Pru86] and a programming language based on L-systems [PL96] made it possible to generate even more realistic images. The recognition of the fractal character of structures generated by L-systems [PH89], [PL96], [HPS91], [PH94], [PH92] also creates a connection to the field of fractals.

Plants are described as a configuration of *modules* in space [PHHM96], [PHMH91], [HHMP96]. The term *module* denotes any discrete constructional unit that is repeated as the plant develops. The goal of modeling at the modular level is to describe the development of a plant as a whole, and in particular the emergence of plant shape, as the integration of the development of individual units.

The development at a modular level can be captured by a *parallel rewriting system* that replaces individual *parent* modules by configurations of *child* modules. All modules belong to a finite *alphabet of module types*, thus the behaviour of an arbitrarily large configuration of modules can be specified using a finite set of *rewriting rules* or *productions*.

In addition, a large body of subsequent work based on the theory of L-systems has been created. Prusinkiewicz presents a model for the animation of plant development [PHM93] which is suitable for animating simulated development processes in a manner similar to time-lapse photography.

Hammel et al. [HP96] creates visualizations of developmental processes by extrusion in space-time by interpreting the entire derivation graph produced by an L-system as opposed to the interpretation of individual words.

Power et al. [PBPS99] explore the problem of interactively manipulating plant models without sacrificing their botanical accuracy. They present a method for interactively manipulating plant structures using a inverse-kinematics optimization technique.

Fowler et al. [FPB92] present a method for modeling spiral phyllotaxis based on detecting and eliminating collisions between the organs while optimizing their packing.

Prusinkiewicz describes a number of selected models of morphogenesis [Pru94], [Pru93b], [Pru93a] and other specialized applications of L-systems [PK96].

## 2.3.1 Context-Free L-Systems

In *context-free* rewriting, a production consists of a single module called the *predecessor* or the *left-hand side*, and a configuration of zero, one, or more modules called the *successor* or the *right-hand side*. A production $p$ with the predecessor matching a given parent module is applied by deleting this module from the string and inserting the child modules as specified by the productions successor. A production has the form

$$pred : succ \tag{2.10}$$

Productions are applied in parallel, with all the modules being rewritten simultaneously in every *derivation step*. Although modules could also be rewritten sequentially, parallel rewriting is more appropriate for the modeling of biological development, since development takes place simultaneously in all parts of an organism. A sequence of structures obtained in consecutive derivation steps from a predefined initial structure (the axiom) is called a *developmental sequence*. It can be viewed as a *discrete-time simulation* of development.

### 2.3.2 Indeterministic L-Systems

If more than one production fits a particular configuration of modules, than the L-system is called *indeterministic* and one of the possible productions has to be chosen randomly. Additionally, a propability can be assigned to each production to have a certain amount of control over the selection of productions.

### 2.3.3 Context-Sensitive L-Systems

Context-sensitive L-systems allow more complex productions - rules are not only matched by comparing the *predecessor* but also the *context* in which the *predecessor* appears. Productions for context-sensitive L-systems are of the form

$$lc < pred > rc : succ \tag{2.11}$$

where *lc* stands for *left context* and *rc* means *right context*.

### 2.3.4 Parametric L-Systems

Parametric L-systems are more expressive and are commonly used in real-world applications of L-systems. They operate on *parametric words*, which are strings of *modules* consisting of *letters* with associated *parameters*. The letters belong to an *alphabet* $V$, and the parameters belong to the set of *real numbers* $\mathbb{R}$. A module with letter $A \in V$ and parameters $a_1, a_2, ..., a_n \in \mathbb{R}$ is denoted by $A(a_1, a_2, ..., a_n)$. Every module belongs to the set $M = V \times \mathbb{R}^*$, where $\mathbb{R}^*$ is the set of all finite sequences of parameters. A production for *parametric context-sensitive L-systems* has the format

$$lc < pred > rc : cond \rightarrow succ \tag{2.12}$$

where *cond* stands for *condition*, which is an arithmetic expression, and the symbol $\rightarrow$ is used to separate the condition from the successor. A parametric production is only matched if the condition evaluates to *true*. For example, a production with predecessor $A(t)$, condition $t > 5$ and successor $B(t+1)CD(t^{0.5}, t-2)$ is written as

$$A(t) : t > 5 \rightarrow B(t+1)CD(t^{0.5}, t-2). \tag{2.13}$$

### 2.3.5 Turtle Interpretation of L-Systems

In order to generate three-dimensional structures out of the one-dimensional strings generated by L-systems, the one-dimensional strings are interpreted sequentially, with reserved modules acting as commands to a LOGO-style turtle [PLH88], [PL96]. At any time, the turtle is characterized by a position vector and three mutually perpendicular orientation vectors indicating the turtles heading, the up-direction, and the direction to the left. A number of reserved modules (mentioned above) cause the turtle to rotate around one of the vectors or to draw a straight line in the current direction. Symbols [ and ] are used to group symbols (for instance symbols representing a single branch). If an opening bracket [ is encountered, the current state of the turtle is pushed onto a stack, if a closing bracket ] is encountered the last state is popped from the stack.

### 2.3.6 Environmentally-Sensitive L-Systems

In [PJM94] Prusinkiewicz extends L-systems in order to simulate the interaction between a developing plant and its environment. As an examplary application the response of trees to extensive pruning is modeled, which yields images similar to sculptured plants found in so-called topiary gardens.

The standard turtle interpretation of L-systems is only designed to visualize models in a postprocessing step, with no effect on the L-system operation. However, position and orientation of the turtle are of importance, when considering environmental phenomena, such as collisions with obstacles and exposure to light. In order to enable the L-system to react to such phenomena, the environmentally-sensitive extension of L-systems makes these attributes accessible during the rewriting process through a number of additional reserved modules. The generated string is interpreted after each derivation step, and turtle attributes found during the interpretation are returned as parameters to reserved *query modules*.

### 2.3.7 Open L-Systems

*Open L-systems* [MP96] augment the functionality of environmentally-sensitive L-systems using a reserved symbol for bilateral communication with the environment. Parameters associated with the occurence of the communication symbol can be set by the environment and transferred to the plant model, or set by the plant model and transferred to the environment. The environment is no longer represented by a simple function, but becomes an active process that may react to the information from the plant. Thus, plants are modelled as open cybernetic systems, sending information to and receiving information from the environment. By using this functionality, phenomena like the development of branches limited by collisions, the colonizing growth of plants competing for space in favorable areas, the interaction between roots competing for water in the soil, and the competition within and between trees for access to light, can be modelled and simulated.

### 2.3.8 PL-CSG-Systems

Based on parametric L-Systems, Gervautz and Traxler [GT95] propose so-called pL-CSG-Systems to build complex CSG-objects which can be rendered very effectively, without having to create an explicit geometric representation of the whole scene, therefore allowing scenes of arbitrary complexity. Instead of using *turtle interpretation* of strings generated by parametric L-systems, cyclic CSG graphs are constructed and used directly inside a rendering system [GT95], [Tra97] allowing for a very compact representation of arbitrarely complex geometry. In [TG95a], [TG95b] Traxler et al. further optimize the rendering process by using tight bounding volumes which are constructed from cyclic CSG-graphs and in [TG96] an approach to improve the visual quality of fractal plants by using genetic algorithms is presented.

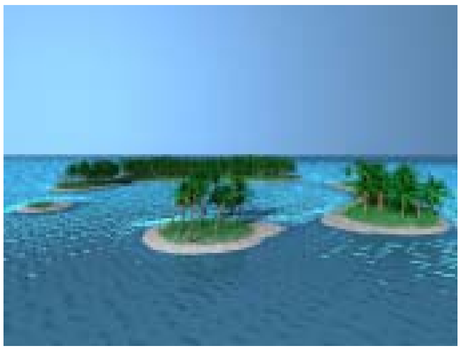
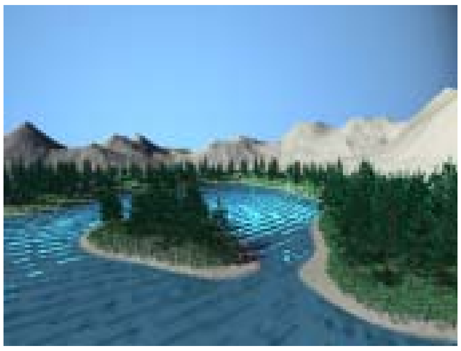



Figure 2.10: PL-CSG-systems where used to model and represent these scenes [Tra98]. They were rendered on a machine with 64 MB of RAM, which was not enough to keep the entire scene in memory [Wil01].

CSG expressions can be interpreted as strings, so it is possible to derive them from a pL-system. As pointed out in detail by Gervautz et al. [GT95], one has to be careful when designing a pL-system that is supposed to generate a valid CSG object. The main problem is that the derivation sequence cannot be stopped arbitrarily as when using a pL-system, where the turtle ignores modules that do not belong to its command set; the result of the derivation process has to be a set of well-formed CSG expressions.

This has two consequences: first, rules can only be applied to modules (generating rules) and second, at least one rule which finally substitutes all variables with a string of terminals (terminating rules) must exist for every module.

## 2.4 Plant Models

Bloomenthal [Blo85] was propably one of the first who tried to not only generate shapes which resembled the overall structure of trees, but he also developed methods to create small-scale detail like branching structures and bark. He connected generalized cylinders with free-form surfaces to model natural looking branching structures and used *blobby* techniques to model the tree trunk as a series of non-circular cross sections. Bump mapping is used to create a natural looking bark-like surface.

### 2.4.1 Botany-Based Models

To the best of our knowledge, de Reffye et al. [dREF+88] present the only plant model in the computer graphics literature which strictly adheres to botanical laws when explaining plant growth and architecture. The model integrates botanical knowledge of the architecture of trees: how they grow, how they occupy space, where and how leaves, flowers or fruits are located, etc. Another very important benefit one can derive from the model is the integration of time which enables viewing the aging of a tree, which includes the possibility of creating different images of the same tree at different ages, and the accurate simulation of the death of leaves and branches. It is also possible to integrate physical parameters such as wind, and the incidence of factors such as insects attacks, use of fertilizers, plantation density and so on, which makes it also a useful tool for agricultural or botanical applications. De Reffye's work is also of importance, because many notions from botany which have not yet been widely used inside the computer graphics community are explained in great detail.

The growth of a plant is the result of the evolution of some specific cellular tissues (internal part of the bud), the so called *meristems*. A bud can, at a given time, die (abort), and it will not produce anything any longer, or it can give birth to a *flower*, or an *inflorescence* (and then the bud dies) or to an *internode*.

The main element of the model is the *leaves' axis* (see Figure 2.11). It is created by the bud situated at its tip, which is called the *apical bud*, and it is comprised of a series of internodes. An internode is a part of a stem made of a ligneous material at the tip of which one can find one or several leaves. In between two internodes there is a *node* which bears leaves and buds. Each node bears at least one leaf. At each leafs axil, one finds a so called *axillary bud*. Axillary stems can either grow immediately (*sylleptic ramification*) or with some delay (*proleptic ramification*).

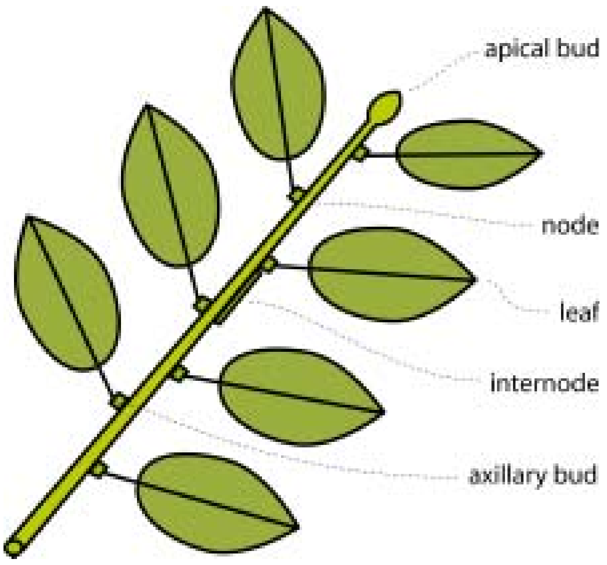


Figure 2.11: The leaves' axis.

Other central notions are the notion of *growth unit* which is a sequence of internodes and nodes produced by the apical bud of the previous node, and the notion of the *order* of an axis. An *order 1 axis* is the sequence of growth units, which originally grew out of the seed of the plant. An *order i axis* (for $i > 1$), is a sequence of growth units such that the first internode of the sequence is born of an axillary bud on an order $i - 1$ axis, called the *bearing axis*.

The relative positions of lateral buds of a node with respect to the lateral buds of the previous node follow regular laws known for each variety of each species and each order; this phenomenon is called *phyllotaxy*. The two most common cases are *spiraled* and *distic* phyllotaxy (see Figure 2.12).

With respect to the ramification process, one distinguishes between *continous ramification*, *rhythmic ramification*, and *diffuse ramification*. All these

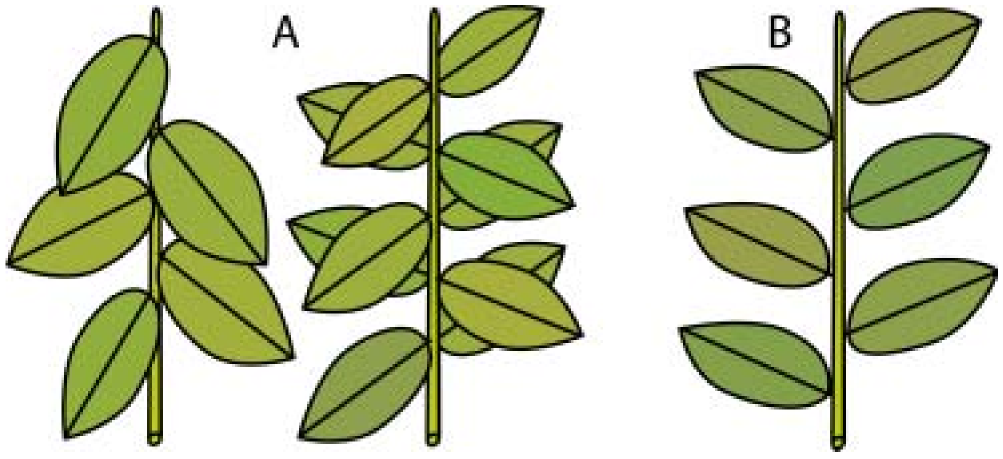


Figure 2.12: Phyllotaxy: (A) spiraled, (B) distic.

kinds of ramifications are functions of the order of the axis for a given variety and species (see Figure 2.13).

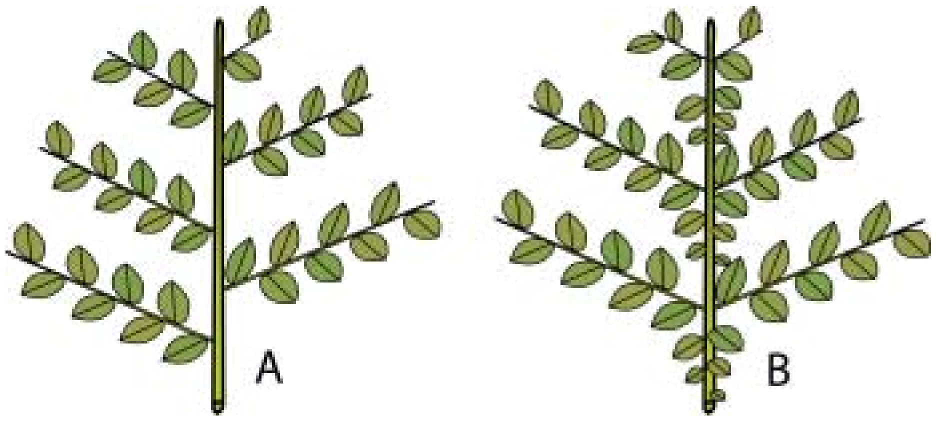


Figure 2.13: Ramification: (A) continuous, (B) rhythmic.

Now, with all these notions in mind, it is also possible to define some notions also well known in the computer graphics community on a more formal basis.

Monopodial trees (*monopods*) are ramified systems which include a unique *order 1* axis and a finite number of axes of higher order. If the orders go up to $k$, the monopod is called an *order k monopod.* The general geometric trend of an axis with respect to its bearing axis leads to further classification. If the general trend is *horizontal* the development is called *plagiotropic*, if it is *vertical* it is called *orthotropic*. There also exists a correlation to phyllotaxy: orthotropy is usually associated with a spiraled phyllotaxy whereas plagiotropy is associated with distic phyllotropy.

*Sympodial* growth (dichotomous branching) occurs when the apical bud of an order i axis dies, and some axillary buds of the previous node produce an axis whose behaviour is of an order $i$ instead of $i + 1$ axis. A similar phenomenon can be observed after the pruning of a tree which is called *traumatic reiteration*. Old trees sometimes exhibit a behaviour called *reiteration* which means, that an axillary bud produces an axis which behaves as an order 1 axis. This behaviour could be interpreted as a "natural" occurence of recursion.

### 2.4.2 Weber and Penn Model

Weber and Penn [WP95] explain a generic tree model which generates a variety of natural looking trees based on a user-editable, large but yet intuitive parameter set. A tree is therein composed of a primary, variably curved trunk similar to a cone. A trunk structure may split multiple times along its length creating a dichotomous branching pattern. Dichotomous branches are called *clones* because they use exactly the same attributes as the original stem. A stem structure and all its clones are considered to be on the same level of recursion. Monopodial branches or *child* branches usually have different attributes than their parent structures and are considered to be one level of recursion below their parents. Usually three or four levels of recursion are sufficient to generate realistic and detailed trees. Different shapes are obtained by scaling child branches according to their position along the stem. Scaling factors are obtained by evaluating a simple 1-dimensional shape function which can be one of: conical, spherical, hemispherical, cylindrical, tapered cylindrical, flame, inverse conical or tend flame. Similarly, leaf shape is selected out of a number of predefined simple leaf shapes like: oval, triangle, 3-lobe oak, 3-lobe maple or 5-lobe maple. In order to completely describe a tree a number of global parameters (like shape, number of levels, base size) along with per-level-attributes (different angles, lengths and curve-parameters) have to be defined. Resulting trees look very natural on a large scale and due to random variations many different instances of trees can be obtained from a single parameterization. Nevertheless, since all clones on one level of recursion are based on the same set of attributes, all these clones all over the tree look quite self-similar which results in a somewhat too uniform and perfect look.

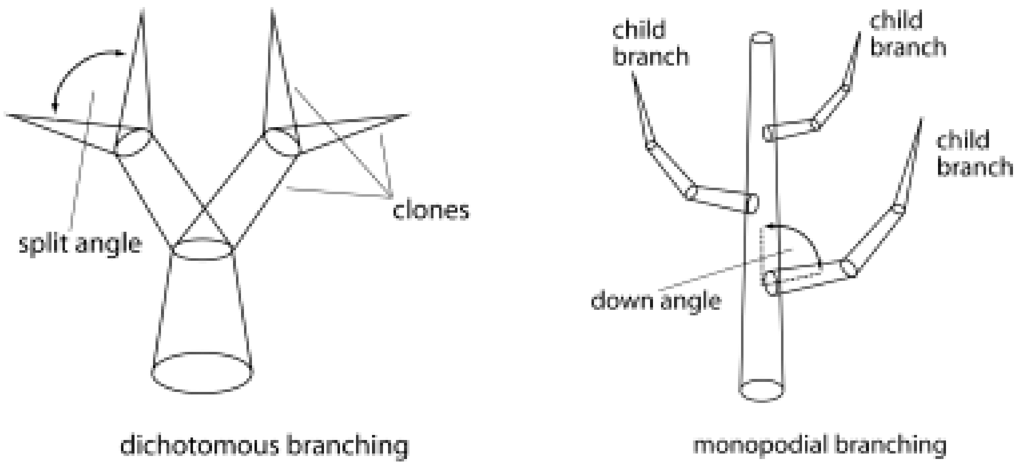


Figure 2.14: Dichotomous structure built from clones, all clones are considered to be on the same level of recursion (left). Monopodial structure, child branches are considered one recursive level below their parent (right). Figures taken from [WP95].

## 2.4.3 XFrog

An interactive modeling method is introduced by Lintermann and Deussen [LD96], [LD98], [LD99], [LD97]. A set of components containing geometric and structural information are represented as icons which can be connected to form a structure graph of the plant model. Each component has its own parameter set which can be edited, and each component can generate its own geometric representation. Each component completely describes a single part (or a collection of similar parts) of a plant while the structure of the graph defines how the components are arranged in order to form a specific plant. Table 2.1 gives a brief overview of components:

Lintermann and Deussen's model is quite complementary to Weber and Penn‘s model. The latter describes trees at a very high level of abstraction using a fixed set of parameters resembling a top-down approach. This makes it easy to design many different kinds of trees which automatically look real and to change their large scale appearance by manipulating a small number of intuitive parameters. On the other hand, it is not possible to create local variations or types of plants not hardcoded in the model. Lintermann and Deussen's method resembles a bottom-up approach. A plant is created by defining all its different parts and by connecting these components to form a graph which implicitly defines the plants global shape. This allows for the creation of much more diversified plants since arbitrary numbers of different components can be defined and combined – limited only by hardware constraints and/or the designers dedication. The drawback of this approach

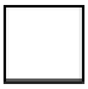
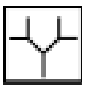
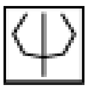
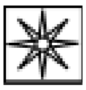
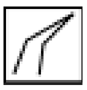
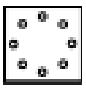
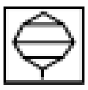
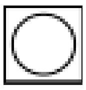
| | | | |
|---|---|---|---|
|  | Simple: produces a simple geometric object like cube, sphere or cylinder. |  | Tree: multiplies components as branches according to given distributions of position, scale and angle. |
|  | Revo: a surface of revolution |  | Hydra: arranges components on a circle with uniform angles and direction perpendicular to the direction of the parent component. |
|  | Horn: a sweep component which is used for stems, twigs, etc. |  | Wreath: arranges components similar to “Hydra” but with direction parallel to the direction of the parent component. |
|  | Leaf: used for the construction of natural leaves. |  | PhiBall: arranges components on a section of a sphere according to the golden section. |

Table 2.1: Components of Lintermann and Deussen’s model. Figures taken from [LD96].

is that the global shape can not be controlled precisely since it is the result of a potentially highly recursive process (the same problem as with pL-systems). However, in practive this seems not to be a problem, since each parameter change is immediately reflected in the plants graphical preview. So usually the user is able to find the correct parameters for a desired effect in a short period of time.

The modeling and rendering of natural scenes with thousands of plants poses a number of problems (modeling of the terrain, placement of plants in a realistic manner, reflecting the interactions of plants with each other and with the environment). In [DHL+98] Deussen et al. develop a system built around a pipeline of different tools which adress these tasks. The terrain is designed using an interactive graphical editor. Plants are placed either manually, by ecosystem simulation, or by a combination of both. The geometric complexity of the scene is reduced by *approximate instancing* which means that plants, groups of plants, or groups of organs are replaced by instances of representative objects before the scene is rendered. As a result it is possible to synthesize visually very rich scenes containing thousands of plants.

Deussen et al. also present approaches [DHR+99], [DS00] for performing non-photorealistic rendering, for example to generate pen-and-ink-like illustrations of plants.

## 2.5 Summary

In this chapter, we have discussed various techniques for surface representation – from polygonal representations to implicit surfaces, as well as multiresolution modeling and level of detail approaches. Furthermore, spline surfaces and subdivision surfaces have been surveyed which are used to create smooth surface representations from a set of control points, which control the approximate shape of the resulting shape. In Chapter 3 we will build on this knowledge to propose *generalized subdivision meshes*.

In addition to this, L-systems have been explained in great detail, which can be used to simulate and create a variety of complex botanical, or other shapes. In Chapter 4 we will introduce an extension to L-systems, which will allow the creation of complex shapes in mesh-based rendering systems.

Finally, we have surveyed selected plant models, which will serve as a basis for our own prototype implementation of a plant editor (see Chapter 6), which also builds on our proposed *generalized subdivision meshes* and *rule-based mesh growing* schemes.

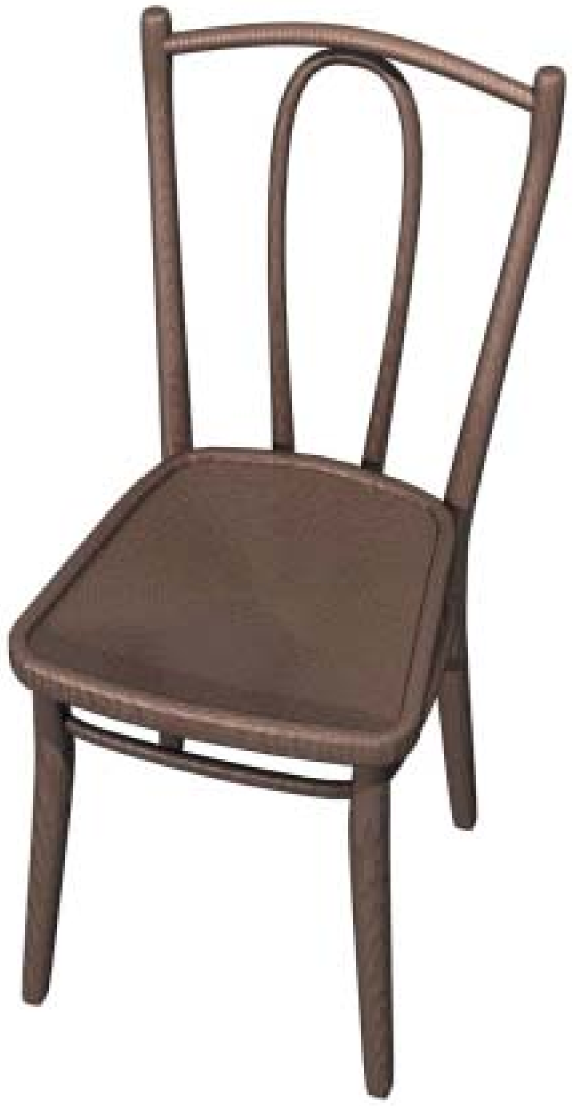


Figure 2.15: *Thonet Chair.* An example for modeling with Catmull-Clark subdivision surfaces. The model consists of 58496 faces; the scene description file (including comments, material-, light- and camera definitions) has only about 15kb; modeling effort of about 2.5 hours; photorealistic rendering takes 125 seconds on an Athlon 650MHz processor; image resolution 1200x1600 pixels.

# Chapter 3

# Generalized Subdivision Meshes

## 3.1 Introduction

As we have seen in the previous chapter, a standard surface subdivision process starts out with a mesh $M^{(0)}$ composed of vertices, edges and faces, that is the base for a sequence of refined meshes $M^{(0)}$, $M^{(1)}$, $M^{(2)}$, ... which converges to a limit surface, called the subdivision surface.

Our goal is to define a practical generalized subdivision framework in which different subdivision schemes, as well as procedural definition of geometry, can be combined. In order to realize this framework, we propose the following steps.

The process for generating submesh $M^{(n+1)}$ of a specific mesh $M^{(n)}$ in the sequence can be split up into two operations [TMW01]. The first operation, which we will call *mesh refinement*, is the logical introduction of all the new vertices in the submesh. This operation yields all the connectivity information for the vertices of the submesh without specifiying the positions of these newly introduced vertices. The second operation, which we will call *vertex placement*, is the calculation of the actual vertex positions. Standard subdivision schemes use specific rules for generating the new vertex positions, that ensure that the limit surface of the subdivision process satisfies certain continuity constraints, e.g. $G^1$ or $G^2$ continuity.

In order to break these continuity constraints at user specified locations, different rules for vertex placement have been introduced [DKT98], that maintain discontinuities at user specified edges. These rules fix the location of edge vertices in place for a user-specified number of subdivision steps. Thus this number can be viewed as a measure of edge-sharpness.

From a more general viewpoint, fractal surfaces [FFC82] can be viewed as a type of subdivision surface where the vertex placement rules at each subdivision step have been chosen to maintain only $G^0$ continuity.

In order to obtain maximum flexibility in generating subdivision surfaces, we propose to separate the two operations of mesh refinement and vertex placement, and make it possible for the user to independently specify both of these operations.

### 3.1.1 Mesh Refinement

As the *mesh refinement operation* generates the connectivity information for the submesh, it determines if the subdivision process generates quadrilateral meshes, such as Catmull-Clark subdivision, or triangular meshes such as Doo-Sabin or $\sqrt{3}$-Subdivision [Kob00]. In order to demonstrate the viability of our new approach, we implemented a variety of mesh refinement schemes (e.g. Catmull-Clark subdivision (see Figure 3.1, Loop, ...). See Section 3.4

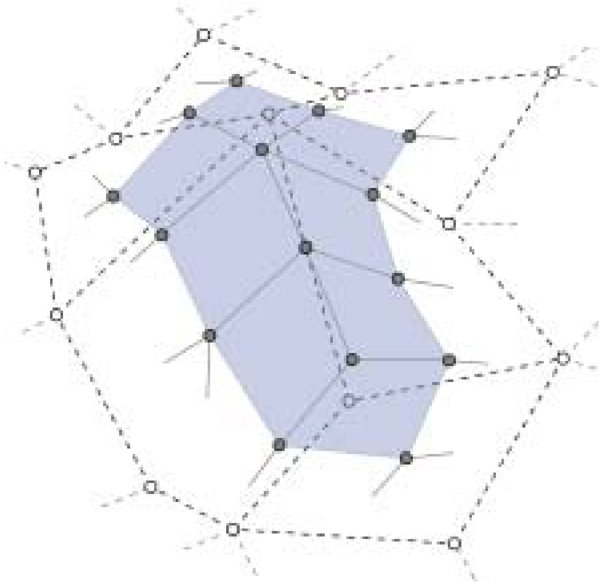


Figure 3.1: A refinement step in the Catmull-Clark subdivision scheme.

for a more detailed description of mesh refinement operators available in our framework.

### 3.1.2 Vertex Placement

Standard *vertex placement rules* consist of taking weighted averages of the vertex positions of mesh $M^{(n)}$ in order to calculate the vertex positions of mesh $M^{(n+1)}$. For standard subdivision surfaces these rules have been designed to smooth the cusps and edges of the input mesh $M^{(0)}$. Although this is desirable in a number of situations, we want to add more flexibility in the rules for vertex placement.

In order to introduce variations at any point in the subdivision process, we introduce a number of geometric properties that can be used to specify a vertex placement rule at each subdivision level. The first of these properties, the *local normal vector*, is an approximation of the surface normal of mesh $M^{(n+1)}$ at a given vertex. Although there are multiple methods for estimating this local normal vector, for simplicity we use the weighted average of the normals of all faces meeting at the vertex under consideration. The second property is the *local scale factor* of the surface, a scalar indicating the average face size at each vertex of a mesh in the sequence. Again this can be estimated with various methods. We choose the average diagonal length of all faces meeting at the vertex as a measure that can be easily computed. Both these parameters are provided in order to facilitate multi-resolution specification of displacements.

By using these two properties, it is possible to specify a vertex placement rule by an equation similar to a procedural texturing rule. Instead of a color at each position in space, we generate a displacement vector for each vertex in a mesh. If these displacement vectors are chosen to be colinear with the local normal vectors at each vertex position, the resulting vertex placement rule can be viewed as a generalized form of displacement mapping [CT84], [CCC87].

As an example (see Figure 3.2), if random displacements in the direction of the local normal vector are added to the vertices of a surface, and the size of the displacements is proportional to the local scale factor, the resulting surface will be a fractal with the standard $1/f$ frequency characteristic. This example however, uses the same rule at each level of the subdivision process.

### 3.1.3 Alternating Between Different Vertex Placement Rules

By specifying different vertex placement rules at different resolution levels, it now becomes possible to model a desired surface in a true multi-resolution fashion. At each scale of the model different variations can be introduced in

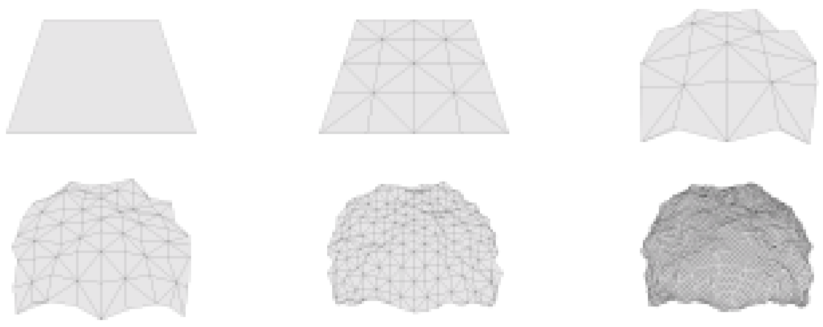


Figure 3.2: A few subdivision steps using a fractal displacement rule with $1/f$ characteristic.

order to approximate the desired result. This is similar to normal meshes [GVSS00] and displaced subdivision surfaces [LMH00], but our scheme is a generalization as there is no limitation on the type of rules that can be used at each level of subdivision. A drawback is, that we cannot automatically generate the rules at each level to approximate a given shape.

The two properties of the local normal vector and local scale factor are provided, in order to facilitate simple and easily specifyable changes. It is also possible to add variations to the vertex placement rules without regard to these properties. Using the fractal surface as an example again, instead of moving the vertices in the direction of the local normal vector, all vertex movements could be performed into the same global direction. In this way it is possible to generate a fractal height field.

The process so far makes it possible to modify vertex positions at each level of subdivision either in a globally chosen direction, or locally in the direction of an estimated normal vector. Sometimes it may be necessary to change the position of a vertex locally not only in the direction of the normal vector, but with respect to a local coordinate frame. For this purpose, a concept similar to frame-mapping [Kaj85] can be employed (see Figure 3.3).

This allows the modification of the vertex position at each subdivision level, both in the direction of the local normal vector, and along the local tangent plane.

As long as the modified subdivision rules are only used a finite number of times, with the rest of the rules being applications of the standard smoothing rules, the algorithms for evaluation of subdivision surfaces at any point [Sta98] can still be used. An example for such a model is the chair in Color Plate 7.1: at a certain subdivision level random vertex displacements were added to simulate the folds in the cushion, but it is still possible to calculate

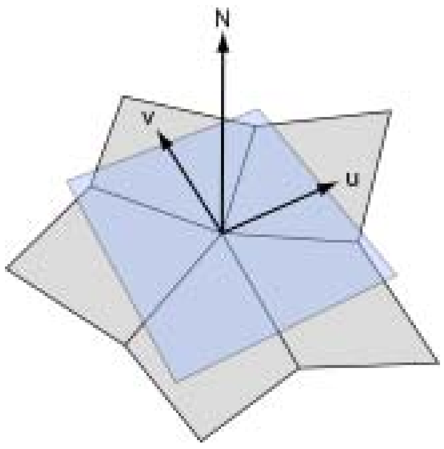


Figure 3.3: The local coordinate frame at a vertex.

the exact limit surface, as all subsequent subdivision steps are just standard Catmull-Clark steps.

If the introduced vertex displacements are always bounded by the local scale factor, the resulting surface is a fractal surface which can be approximated by terminating the subdivision process after a finite number of steps (for smooth surfaces, additional constraints have to be met [CDM91]. The resulting error in vertex positions is on the order of the local scale factor. In this case the resulting surface is only $G^0$ continous, and there is no good way of approximating the normal vector of the surface. Such surfaces are however still valuable modeling primitives, as there are a number of natural phenomena, e.g. terrains and wrinkled tissues, which can be approximated by such $1/f$ fractal surfaces.

## 3.2 Mesh Representation

The foundation of our *generalized mesh subdivision* approach is a data structure which holds a single mesh and which also stores a number of additional per-vertex, per-edge, and per-face data values (see Section 5.4 for implementation details). Each level of subdivision is stored independently in such a structure and a *meta data structure* encloses the ordered sequence of subdivision levels.

So formally, a *generalized subdivision mesh* $G = (L, O)$, where $L$ is an ordered sequence of meshes $M^{(0)}, M^{(1)}, ..., M^{(n)}$ with each mesh representing a distinct level of subdivision, and $O$ is an ordered sequence of *mesh refinement* and *vertex placement* operators $\omega_1, \omega_2, ..., \omega_m$, where the operator $\omega_i$

defines the transition $M^{(i-1)} \rightarrow M^{(i)}$.

In the following we will give a detailed description of all additional data values which can be stored per vertex, edge, and face, and which are used in different contexts, from *generalized subdivision* to *photorealistic rendering.*

### 3.2.1 Vertex Coordinates

This per-vertex property is mandatory and defines a three-dimensional position for each vertex. Vertex coordinates are always defined in the local coordinate frame of the mesh. During rendering they are transformed on-the-fly by applying the current *local to global* transformation matrix which is defined by the transformation nodes which are encountered during scene graph traversal. The stored vertex coordinates are not changed throughout this process - this makes it possible to use the same mesh instance in multiple locations in a scene without the necessity of explicitely copying the mesh data structure.

### 3.2.2 Texture Coordinates

This per-vertex property is optional and defines two-dimensional texture coordinates. Texture coordinates are used for texture mapping in rendering, but also as parameters for mesh operators (for example to parameterize a vertex-placement expression), as well as to set up a local coordinate frame.

### 3.2.3 Normals

Normals are optional per-vertex properties and define the normal vector at each vertex. The normal property is used for lighting calculations in rendering to determine incident light directions which are commonly needed for the evaluation of lighting models. If no normals are specified, they are automatically estimated from the adjacent vertex positions. Only in very rare circumstances (for example to achieve some special lighting effects or if normal vectors are part of a procedurally generated mesh) this property is defined manually.

### 3.2.4 Sharpness

*Sharpness* is an optional per-edge property and is used for subdivision operators to create creases of variable sharpness. It is a floating point value $0.0 >= s$, where a value of $s = 0.0$ stands for *no sharpness* at all (smooth edge).

### 3.2.5 Sheet Numbers

Sheet numbers are optional per-face properties and are solely used for rendering. By defining a sheet number for each face, multiple faces may be grouped together by assigning identical sheet numbers. Most frequently this is used to switch between different surface materials in rendering. As an example, imagine a leaf with a stem modeled as a single mesh. The faces which are part of the leaf area, could be assigned sheet number 0, and faces which are part of the stem could be assigned sheet number 1. At rendering time a surface map consisting of two surface materials - a texture map for the leaf and a simple green surface material - is assigned to the mesh. As a result, all faces which are part of sheet 0 are colored using the texture map and all faces which are part of sheet 1 are colored using the simple green surface material.

## 3.3 Mesh Properties Stored in the Traversal Environment

In order to create complex geometry it is essential to enable replacement rules to depend on the environment they control. This results in a feedback loop in which replacement rules govern the creation of geometry while geometry changes the rules it obeys to.

Each operator $\omega$ therefore has access to a traversal state which stores a number of properties associated with the part of the mesh it is currently applied to – typically a single vertex, or a single face. Of course not every value is applicable to each operator. If, as an example, an operator $\omega_x$ is designed to be applied to faces, there exists no well-defined single vertex index, but a set of indices. Nevertheless we try to assign *some* meaningful value – even to ill-defined parameters – if it is possible:

As we will see in a number of *mesh operator* examples in the following sections, user specified expressions may take advantage of a number of predefined mesh properties which reflect the state of the current mesh at the current vertex or face. In the following we will give a detailed description of all available predefined mesh properties $\pi_i$, that can be used for the purpose of specifiying context-sensitive expressions. $\Pi$ denotes the union of all mesh operators $\pi_i$.

All mesh properties $\pi_i$ are accessible through a typed variable binding environment that provides variables of the following types: integer, floating point, two-dimensional, and three-dimensional points and vectors. Although typed parameters are not strictly necessary, they provide some convinience for implementing parameterized rules.

### 3.3.1 Vertex Position

One of the most important mesh properties is the *vertex position* $\pi_{vp}$. It holds the three-dimensional coordinates of the current vertex and can be used in a number of different ways. Just to name a few, it may be used to create pseudo-random values depending on a three-dimensional position (e.g. Perlin noise or turbulence). Of course it may also be used in a more direct way, by specifying a condition which directly depends on the spatial position, e.g. to switch between different expressions depending on the position of the vertex. This could be useful in a scenario, where e.g. a tree is modeled, and depending on the height, different displacement functions for modeling fine surface detail (like bark) should be used:

```
...
bark1 = ...;
bark2 = ...;
bark = (VERTEX_POSITION.z < 2.0) ? bark1 : bark2;
...
```

The *vertex position* is well-defined for all vertex placement mesh operators (*add normal height*, *add global vector*, *add local vector*) since they all operate on single vertices.

### 3.3.2 Texture Coordinates

Another very useful property are the *texture coordinates* $\pi_{tc}$ of the current vertex. This property is a two-dimensional point, that can be used in similar ways as the *vertex position*. One example would be to use the two-dimensional coordinates to create ridges along a stem (see Color Plates 7.9 and 7.10 as an example for creating highly detailed surface structures by using, among other properties, two-dimensional texture coordinates).

Of course this property is well-defined in all cases where the *vertex position* is well-defined, except the case, when no texture-coordinates have been assigned to the mesh vertices in the first place. This is possible, because texture coordinates are an optional mesh property. In such a case, the *texture coordinates* property will be initialized to $(0.0, 0.0)$.

### 3.3.3 Normal Vector

The *normal vector* property $\pi_{nv}$ holds the local normal vector. For operators which work on a per-face basis, we use the face normal which is well-defined (see Figure 3.4). For vertices, we either can use the user specified normal (if per-vertex normals have been assigned to the mesh (see Section 3.2.3)) or the normal vector has to be approximated. For simplicity we use the weighted average of the normals of all faces meeting at the vertex under consideration (see Figure 3.4). Although this is an arbitrary solution, it is straightforward to implement, fast, and works well with our application.

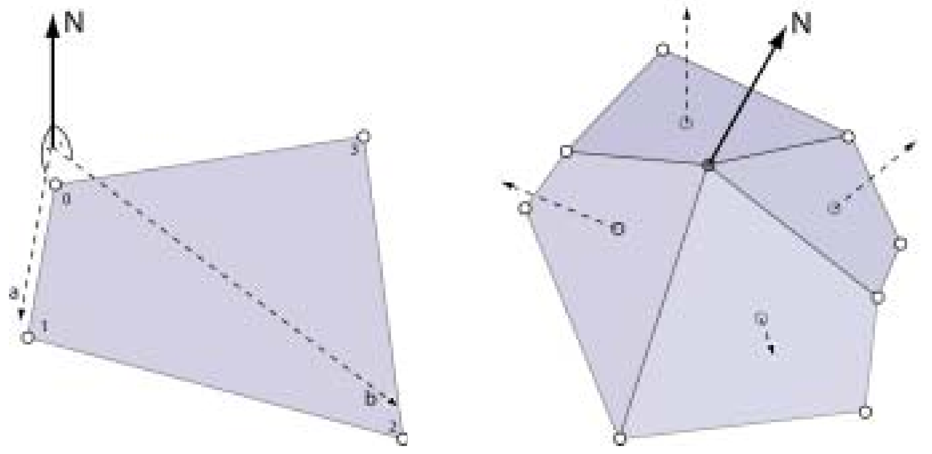


Figure 3.4: (left) Face normals are calculated using the cross product of two vectors in the face plane. If the face vertices are locally numbered $v_0$, $v_1$, ..., $v_{n-1}$ then let $a = v_1 - v_0$ and $b = v_2 - v_0$ and the normal $N = a \times b$. (right) Vertex normals are approximated by the weighted average of the normals of all faces meeting at the vertex under consideration. If the adjacent face normals are locally numbered $n_0$, $n_1$, ..., $n_{n-1}$ then the vertex normal $N = \frac{1}{n} \sum_{i=0}^{n-1} n_i$.

### 3.3.4 Vertex Index

The *vertex index* $\pi_{vi}$ is supplied for the sake of completeness and holds the vertices internal index in the mesh data structure. This index is of course well-defined for all operators which are evaluated on a per-vertex basis, since every single vertex has an unique internal index. For operators which are evaluated on a per-face basis, the index of the “first” (as stored in the mesh data structure) vertex of the face is taken, which of course is an arbitrary and ad-hoc solution.

The property should not be used for standard modeling, since it depends on too many internal interactions of data structures and algorithms, that in a sense it is not “well-defined”. It is useful only for models which are created with the sole intention to exhibit some properties of the underlying internal state of the mesh data structure (that means mainly for testing or optimization purposes, or just out of curiousity).

### 3.3.5 Local Scale

The *local scale* property $\pi_{ls}$ holds a scalar indicating the average face size at the current position. This value can be estimated with various methods. With respect to faces, we use the average length of all vertex interconnections in the face as an estimate for the faces diagonal length. With respect to vertices we choose the average diagonal length of all faces meeting at the vertex. This property is provided in order to facilitate multi-resolution specification of displacements.

### 3.3.6 Minimum Local Scale

The *minimum local scale* property $\pi_{mls}$ again holds a scalar, which is indicating the minimum face size at the current position. It is estimated similarely to the *local scale* property, but instead of the average length of all vertex interconnections the minimum length is chosen for operators working on a per-face basis, and the minimum diagonal length of all neighbouring faces is chosen for operators working on a per-vertex basis.

### 3.3.7 Maximum Local Scale

The *maximum local scale* property $\pi_{mls}$ is calculated analogous to the *minimum local scale*, but uses the maximum instead of the minimum values.

### 3.3.8 Sheet Index

The *sheet index* property $\pi_{si}$ holds the sheet index (see Section 3.2.5. For operators working on a per-face basis this is well-defined if sheet numbers have been assigned to faces. If no sheet numbers have been assigned, which is possible, since sheet numbers are an optional mesh property, the value is set to 0.

For operators working on a per-vertex basis, the *sheet index* property is not defined. In this case the value is arbitrarely set to the sheet number of the (internally) first neighbouring face (or 0 if no sheet numbers have been assigned).

### 3.3.9 Face Index

The *face index* $\pi_{fi}$ is supplied for the sake of completeness and holds the faces internal index in the mesh data structure. This index is of course well-defined for all operators which are evaluated on a per-face basis, since every single face has an unique internal index. For operators which are evaluated on a per-vertex basis, the index of the "first" (as stored in the mesh data structure) neighbouring face of the vertex is taken, which of course is an arbitrary assignment.

The property should not be used for standard modeling, since it depends on too many internal interactions of data structures and algorithms, that in a sense it is not "well-defined". It is useful only for models which are created with the sole intention to exhibit some properties of the underlying internal state of the mesh data structure.

## 3.4 Mesh Operators

All *vertex placement* methods have one thing in common – they are applied to every single vertex of a mesh, thus in a sense to the whole mesh. As a result we denote them *mesh operators* in our framework. So formally spoken, if we have a mesh $M = (V, F)$, where $V$ is an ordered sequence of vertices $v_1, v_2, ..., v_n$, and $\Pi_i$ is the set of *mesh properties* $\pi_1, \pi_2, ..., \pi_m$ applicable at vertex $v_i$, then a *mesh operator* $\omega$ is an arbitrary function, that takes a vertex position $v_i$ and maps this given vertex position (possibly using properties from $\Pi_i$) to a new vertex position $v_i'$:

$$v_i' = \mu(\Pi_i, v_i) \tag{3.1}$$

In the following, a detailed description of *mesh operators* which are available in our generalized framework is given.

### 3.4.1 Add Normal Height

The *add normal height* operator takes one argument which defines the magnitude of displacement. When the operator is applied to a mesh, the argument function is evaluated for each vertex, yielding the distance which the specific vertex is shifted along its local normal vector (see Figure 3.5). A simple example:

```
f = perlin_noise(VERTEX_POSITION) * LOCAL_SCALE;
op = ADD_NORMAL_HEIGHT(f);
```

First, a function $f$ is defined which uses a predefined Perlin noise function [Per85] which takes the current vertex position and returns a pseudo random scalar value in the range $[0, 1]$ which is subsequently scaled by the local scale approximation. Next, an *add normal height* operator *op* which uses the displacement function $f$ is defined. The mesh operator *op* now can be used in a *mesh operator sequence* (see Section 3.5). A similar operator has been used to create the image sequence in Figure 3.2.

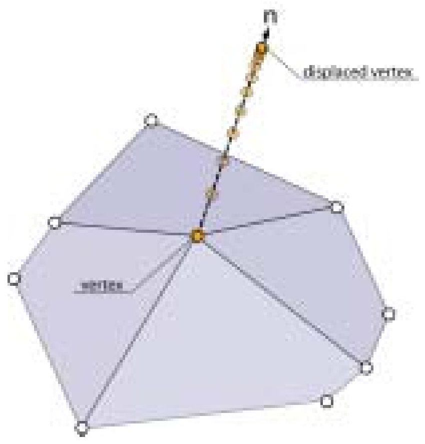


Figure 3.5: The *add normal height* operator is used to displace a vertex along its normal. The distance is evaluated from a user-specified expression.

### 3.4.2 Add Global Vector

The *add global vector* operator also takes one argument which defines a global displacement vector. When the operator is applied to a mesh, the argument function is evaluated for each vertex yielding a three-dimensional vector. This vector is used to shift the vertex in the global coordinate frame (see Figure 3.6. A simple example:

```
f = Vec3D(0.0, 0.0, perlin_noise(VERTEX_POSITION) * LOCAL_SCALE);
op = ADD_GLOBAL_VECTOR(f);
```

First, a function $f$ is defined which yields a three-dimensional vector whose $x$ and $y$ components are constantly set to 0.0 and whose $z$ component is initialized to the random value which has already been used and described in the previous section. Next, an *add global vector* operator *op* which uses the displacement function $f$ is defined. The mesh operator *op* now can be used in a *mesh operator sequence* (see Section 3.5). This sort of operator, where each vertex is shifted in the same global direction (*up* in our example) could be used to create e.g. height fields.

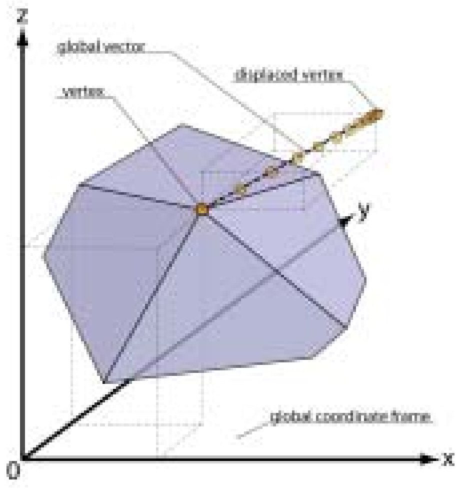


Figure 3.6: The *add global vector* operator shifts a vertex along a given vector which is specified in the global coordinate frame. The vector is evaluated from a user specified expression.

### 3.4.3 Add Local Vector

The *add local vector* operator is very similar to the *add global vector* operator. It also takes one argument function which is evaluated at each vertex position, and yields a three-dimensional vector. In contrast to *add global vector* the displacement vector is used to shift the vertex in the *local* coordinate frame (see Figure 3.7). An *add local vector* operator has (among others) been used to create the detailed surface geometry of the image of a forking branch in Color Plates 7.9 and 7.10.

### 3.4.4 Flat Quad Subdivision

The *flat quad subdivision* operator is a pure mesh refinement operator. It splits each face into quadrilateral subfaces in a Catmull-Clark-like way (see Figure 3.8). If the operator is applied to a mesh $M^{(n)}$, then all the vertex positions of $M^{(n)}$ remain unchanged in $M^{(n+1)}$, newly created *face points* are

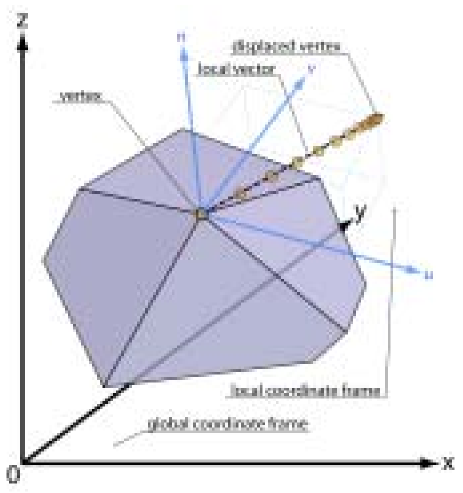


Figure 3.7: The *add local vector* operator shifts a vertex along a given vector in the vertex' own local coordinate frame. The local coordinate frame is made up of the meshes local u,v-coordinate system and the surface normal at the vertex.

placed at the face center, newly created *edge points* are placed at the edge mid-point. The mesh is refined but the shape remains unchanged, hence the term *flat* subdivision.

The *flat quad subdivision* operator is useful in situations where small scale surface detail should be added to a shape, but the shapes tesselation is still too coarse. With the *flat quad subdivision* operator it is possible to refine the mesh without changing its shape.

### 3.4.5 Catmull-Clark Subdivision

The *Catmull-Clark subdivision operator* implements the standard Catmull-Clark subdivision scheme and the special rules for generating semi-sharp creases, proposed by deRose et al. [DKT98]. It performs mesh refinement as well as vertex placement operations.

```
op = CATMULL_CLARK_SUBDIVISION();
```

### 3.4.6 Adaptive Catmull-Clark Subdivision

The *adaptive Catmull-Clark subdivision operator* is similar to the standard *Catmull-Clark subdivision operator* but takes two additional parameters *limit size* and *limit curvature*, which are used to adaptively subdivide faces. This

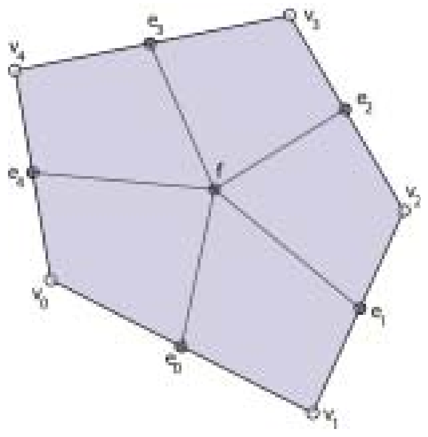


Figure 3.8: *Flat quad subdivision* operator: faces are refined in a Catmull-Clark-like fashion. *Vertex points* remain unchanged, *face points* are placed at the face center, and *edge points* are placed at the edge midpoints. As a result, the mesh is refined, but the shape remains unchanged.

means that a single face is only subdivided if its approximate diagonal length is larger than the *limit size* and the *approximate curvature* is larger than the given *limit curvature.* As a result, fewer faces are generated, since subdivision is only performed on parts of a mesh where further subdivision improves the visual quality of the resulting mesh.

```
limit_size = 0.1;
limit_curvature = 0.95;
op = ADAPTIVE_CC_SUBDIVISION(limit_size, limit_curvature);
```

### 3.4.7 Loop Subdivision

The *Loop subdivision operator* implements the standard Loop subdivision scheme. It performs mesh refinement as well as vertex placement operations.

```
op = LOOP_SUBDIVISION();
```

## 3.5 Mesh Operator Sequences

In order to define which *mesh refinement* and *vertex placement* operators should be applied to a mesh, and in which order, and how often, we introduce so-called *mesh operator sequences.* A *mesh operator sequence* is simply a linear sequence of *mesh operators* (see Section 3.4) which perform distinct *mesh refinement* and/or *vertex placement* operations.

Standard subdivision schemes like Catmull-Clark perform both *mesh refinement* and *vertex placement*. Alternatively, there also exist pure *vertex placement* and pure *mesh refinement* operators. A simple example:

```
base_mesh = create_my_mesh();
sequence = MESH_OPERATOR_SEQUENCE(
                CATMULL_CLARK, 2,
                ADD_NORMAL_HEIGHT(some_expression), 1,
                ADAPTIVE_CATMULL_CLARK, 1
                );
result_mesh = [base_mesh apply: sequence];
```

In the above example, a mesh is created and stored in the variable *base_mesh*. Next, a *mesh operator sequence* is defined which states that two Catmull-Clark subdivision steps, one *add normal height* step, and finally an *adaptive Catmull-Clark* step (see Section 3.4.6) should be performed. In the last line of code, the sequence is applied to the base mesh and stored in the variable *result_mesh*. The result mesh could now be incorporated in an arbitrary scene.

## 3.6 Summary

In this chapter, we introduced a generalized framework for mesh subdivision in which the standard application of a subdivision operator is split into two distinct operations – *mesh refinement* and *vertex placement* – which can be independently specified. Furthermore, we introduced a number of vertex placement operators, like *add normal height*, *add global vector*, and *add local vector*, as well as different refinement operators, e.g. for Catmull-Clark-like refinement.

User specified vertex placement expressions can be defined using arbitrary mathematical expressions, and in addition to this, they may take advantage of a variety of mesh properties, which store useful information associated with the current vertex (which is currently placed), and which are accessible through a typed variable binding environment. Finally we reviewed all predefined *vertex placement* and *mesh refinement* operators in the context of individual examples to demonstrate the practical applicability of our approach.

# Chapter 4

# Rule-Based Mesh Growing

Although the generalized subdivision framework, which has been introduced in the previous chapter, is a very powerful modeling tool, the resulting meshes will always have the same mesh-connectivity except at a small number of extraordinary vertices that were already present in the original mesh. As an example, if Catmull-Clark subdivision is used, each mesh in the subdivision sequence will always be composed of quadrilaterals. Introducing local variations can then only be performed by locally shifting single vertices in the subdivision mesh. Unfortunately, this can lead to arbitrarily large distortions in the adjacent quadrilaterals.

If some vertices are shifted by large vectors (considerably longer than the local scale factor), and all other vertices are left unchanged, the four quadrilaterals meeting in that one vertex will be severely distorted (see Figure 4.1). The resulting texture coordinate space of the affected vertices is severely stretched, which can lead to problems in such applications as texture mapping and finite-element methods like radiosity. In order to overcome this deficiency, we introduce rule-based mesh growing, which is an extension of parameterized Lindenmayer systems. But first, we shortly reiterate on some properties of pL-systems which are relevant to our own extension.

## 4.1 Parametric L-Systems

L-systems are defined by a number of symbols that represent components of a plant, and a set of rules, which define a string of replacement symbols for each of the available symbols. In order to simulate a biological system, a start symbol is taken, and in each replacement step, all symbols are transformed in parallel according to the given rules. Thus the start symbol can be thought of as a seed, and the ruleset encodes the growth of the plant.

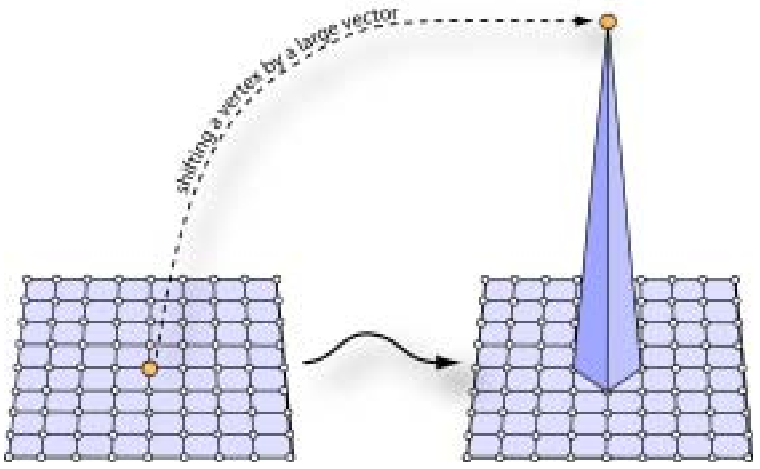


Figure 4.1: Shifting a vertex by a large vector gives rise to severly distorted faces.

In order to generate three-dimensional models, L-systems have been extended by three significant concepts:

- *Parameterized symbols*: for placing the parts of a plant in space and generating parts of different sizes it is necessary to associate parameters with each symbol, that encode the properties of each part of a plant.
- *Parameterized rule expansion*: in order to modify the parameters, it is necessary to calculate new values for the parameters at each rule expansion step. These calculations are associated with each expansion rule.
- *Encoding of a hierarchical structure*: L-systems only operate on one-dimensional strings. In order to represent hierarchical structures, such as trees, it is necessary to introduce grouping symbols. With these symbols it is possible to encode branching structures.

## 4.2 Mesh-Based pL-Systems

In order to use parameterized L-systems in the context of a mesh-based modeling system, we introduce *mesh-based pL-systems*, by associating each parameterized symbol of the system with a face in a mesh. Thus the right-hand side of each production rule is not a linear sequence of symbols, but a template mesh with each face representing a symbol.

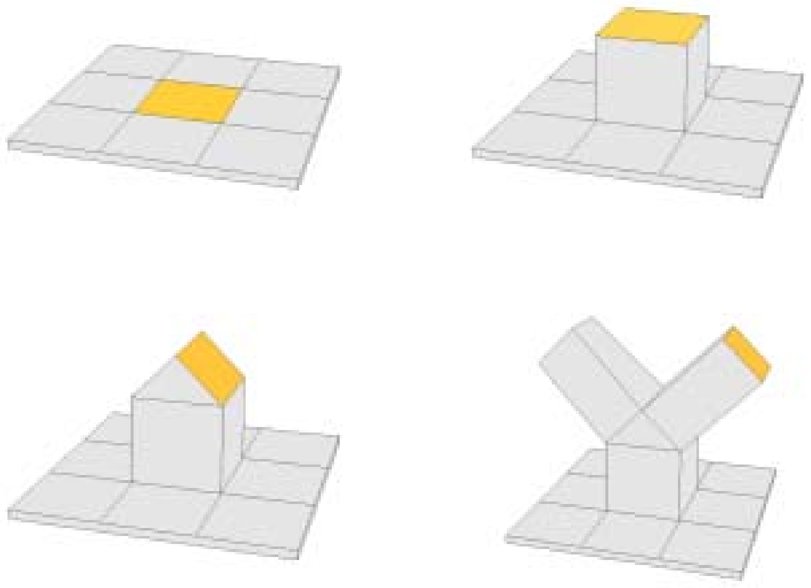


Figure 4.2: Mesh growing example. The top-left mesh is the starting mesh, where the center face is associated with a rule, that defines a cube-shaped mesh (stem). In the top-right mesh, the first mesh has been expanded (according to the associated rule) by the cube-shaped mesh. The top face of the cube-shaped mesh, again is associated with a rule, that defines a tent-shaped mesh (branch). In the bottom-left mesh the result of this next mesh-growing step can be seen. Finally, the two top faces of the tent-shaped mesh are again associated with the rule that defines a cube-shaped mesh (the left-top face can not be seen, since it is at the back of the mesh). The result of this next replacement step can be seen in the bottom-right image.

Thereby the topological structure of an object generated with such a mesh-based pL-system is automatically encoded in the connectivity information of the mesh, and we do not need to introduce grouping symbols in order to encode the hierarchical structure, like this is necessary in pL-Systems which operate on one-dimensional strings.

It is also very easy to avoid producing degenerate meshes that contain T-vertices, or malformed faces: if the template meshes which are contained in the rules are well-formed, they will not introduce any degeneracies into the growing mesh, and as a result, the final mesh will be well-formed, too.

Figure 4.2 demonstrates how a mesh can be grown from simple building blocks (templates). Each rule in such a system consists of a symbol associated with a face and a replacement mesh, where each face is again associated with a symbol. In our example in Figure 4.2 the replacement geometry is either a cube-shaped mesh (in the first and third expansion step) or a tent-shaped mesh (in the second expansion step).

Summing up, a mesh-based pL-system consists of the following parts (we forego a completely formal specification, since this would be needlessly complex):

- *an initial mesh*: a symbol is associated with each face of this initial mesh.
- *a set of rules*: each of these rules consists of a symbol on the left side, and a replacement mesh on the right side. A symbol is again associated with each face of the replacement mesh.
- *parameters*: in order to parameterize the L-system, an environment of variable bindings (parameters) is maintained.
- *calculations*: each rule can be augmented with arbitrary calculations that modify the parameters.
- *conditionals*: each rule can be augmented by conditionals that allow the definition of alternative replacement geometries based on the result of arbitrary calculations.

Mesh growing consists of taking the initial mesh, and applying all replacement rules in parallel. Each face that is associated with the left-hand symbol of a replacement rule is replaced by the mesh specified on the right-hand side of the rule, subject to the calculations and conditions that are part of the rule. Symbols that do not appear as the left-hand side of any rule are terminal symbols and denote faces that will not be changed anymore. After all rules have been applied in parallel, a new mesh is generated, again with symbols associated with each face. Successive rule expansion steps are applied until only terminal symbols are left in the mesh. Figure 4.3 gives a simple example of a ruleset.

## 4.3 Attaching Meshes in a Rule

In order to connect the mesh in a rule to the growing mesh it is necessary to transform the mesh in the rule in such a way that both meshes can be joined without creating degenerate geometry. This can be achieved by specifiying a suitable transformation between both meshes. Defining this transformation

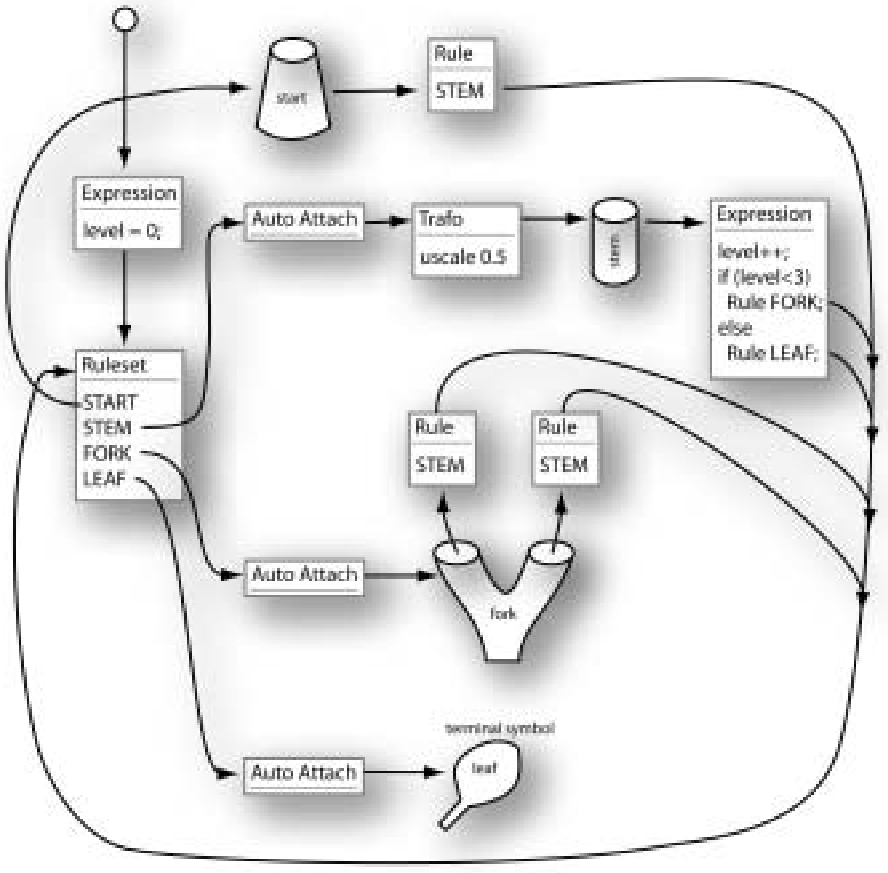


Figure 4.3: A simple, yet meaningful example ruleset. Notice how the ruleset-node and the building block meshes form a directed cyclic graph (DCG) by using the expansion faces as back-references to the ruleset node. This example also demonstrates, that arbitrary transformation nodes may be inserted into the DCG.

by hand is possible, but straightforward only for the most simple of configurations. For arbitrary meshes it is a sumptuous and error-prone job. In order to relieve the designer from this burden, we introduce a *auto-attach* operator, which constructs a transformation which will scale, translate and rotate the replacement mesh in the rule to fit onto the current expansion face (see Section 4.5.2 for a detailed description of the *auto-attach* operator).

Currently we have no provision for avoiding intersection of the geometry of neighbouring replacement meshes. But, in practice it turns out that this does not pose any problem, since it is very easy to design the rules in such a way, that no neighbouring faces will be replaced and no two replacement meshes will intersect. In order to solve the self-intersection in a general, global way, *open pL-systems* have to be used, that store space occupancy in an environment [MP96].

Coming back to our initial problem of defining transformations for the placement of replacement geoemtry, we see, that no matter whether such a transformation has been defined by hand or automatically – ultimately the replacement mesh has to be attached to the growing mesh.

An ideal solution would merge both connecting faces *without* deforming any of the two meshes and without introducing any degenerated geometry. Unfortunately this result can only be achieved in very ideal, and therefore rare, circumstances. Both connecting faces must have the same number of vertices, they must be placed in a common plane and corresponding vertices must be perfectly aligned. Given these preconditions, both meshes can be joined by merging corresponding vertices of both connecting faces and by finally removing the connecting faces.

We presume that rules are designed in such a way, that connecting faces always have the same number of vertices. This poses no great constraint to the designer but avoids the necessity of creating complicated interconnecting geometry. Another problem is to find corresponding vertices in the connecting faces. We can not simply use the face-interal vertex numbering, because there is no correlation between this numbering and the concrete geometrical configuration of faces (see Figure 4.4).

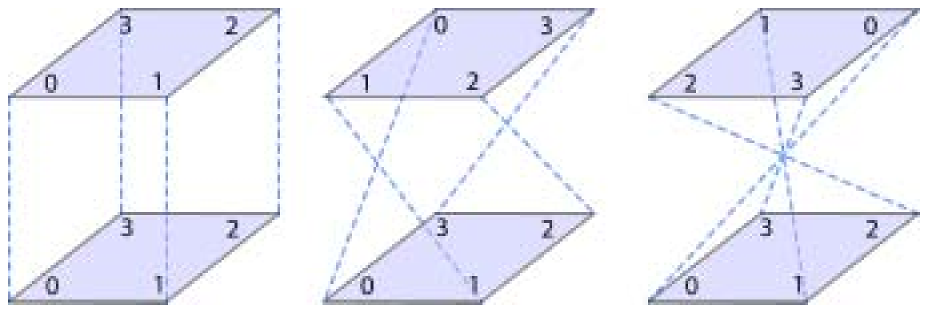


Figure 4.4: Face-interal vertex numbering is not useful to define corresponding vertices, because there exists no correlation between the numbering and the concrete geometrical configuration, therefore severly degenerated connectivity information could be introduced.

In order to find the corresponding vertices of two arbitrary orientated faces, we choose of all permutations of pairwise mapped vertices the one configuration which yields a minimum summed length of distances between mapped vertices.

### 4.3.1 Direct Merge

Provided that both faces share a common plane we can now directly merge both meshes by replacing the connecting face vertices of the replacement mesh by the corresponding vertices of the growing mesh, and by removing both connecting faces (these would otherwise be *inside* the resulting mesh and introduce topological problems). Provided that both faces are congruent, no geometrical deformations are introduced. This scenario resembles the ideal situation described above, plus we have solved the problem of finding *corresponding* vertices.

But what happens if both connecting faces are not congruent. We can still find corresponding vertices by using our method described above, but if we directly merge both meshes, the faces adjacent to the replacement meshs connecting face will be distorted. The magnitude will depend on how well both connecting faces match. From our experience, this poses no problem for models of natural objects, since the deformations simply introduce some *random variations* to the model which makes it look even more natural. Again, it is left to the designer to provide appropriate connecting faces.

If both faces are not parallel and/or if they are located at some distance, this will introduce additional distortions to adjacent faces. Nevertheless, from our experience we can say, that this too poses no problem for a majority of applications - in most cases it even possitively affects the final geometry (see Figure 4.5).

If, after all, deformations can not be tolerated for a certain application and if it is also not possible to provide appropriate rules to prevent deformations, one has to find other means of connecting meshes, namely *indirect merging.*

### 4.3.2 Indirect Merge

If the replacement mesh *must not* be deformed, the only solution for merging is to introduce additional geometry. This means, that the replacement meshes connecting face vertices are not replaced by the growing meshes connecting face vertices, but that in between each pair of corresponding edges of the connecting faces additional quadrilaterals will be inserted (see Figure 4.5). Of course both connecting faces still have to be eliminated, since they would again be placed *inside* the connected mesh and introduce topological problems. Although the expansion mesh geometry will remain unchanged, the newly created connection-faces themselves could now be sources of unwanted problems, namely self-intersections of the surface in the vicinity of the transition. Self-intersections will occur, if the replacement mesh and the growing mesh intersect. Notice that this was not a problem for *direct merging* as long as only the connecting face and adjacent faces penetrated the growing mesh, since the replacement meshs connecting vertices where *replaced* by the corresponding growing mesh vertices, which effectively eliminated the part of the expansion mesh which intersected the growing mesh.

In order to prevent this unwanted behaviour, the replacement mesh has to be positioned at a *safe distance* from the growing mesh, thus eliminating the possibility of parts of the replacement mesh penetrating the growing mesh. We assume one half of the maximum diagonal length of the connecting face plus a small $\Delta$ as a *safe distance*. This efficiently prevents any penetration and due to the additional $\Delta$ it also prevents the creation of degenerate connection-faces (which could happen if two corresponding edges incidentally coincided).

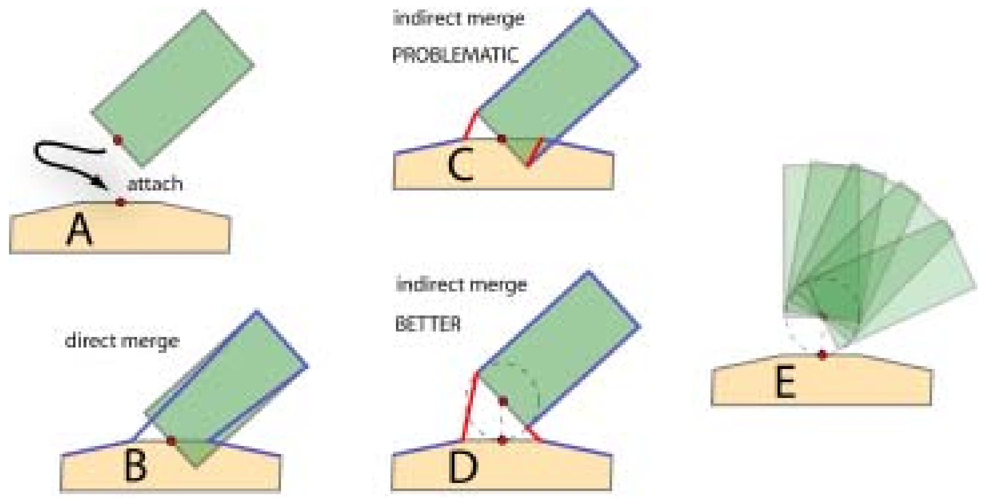


Figure 4.5: *Direct merge* vs. *indirect merge* (A) initial situation - the green replacement mesh has to be attached to the bottom mesh. (B) direct merge - both meshes are connected along the edges of the connecting faces, the connecting faces can be removed. It has to be noticed that deformations of adjacent faces of the replacement mesh might be introduced (see blue contour line of the resulting mesh). (C) indirect merge - to prevent deformations, additional faces (red) are introduced which connect corresponding edges of both connecting faces. Notice that if the replacement mesh is positioned by aligning the face centers of the connecting faces, self-intersections might be introduced. (D) to prevent self-intersections, the replacement mesh is shifted along the face-normal, the distance we use is half the maximum diagonal length of the connecting face. (E) no matter how the replacement mesh is positioned, no self-intersections or other degenerated geometry will be introduced.

## 4.4 Extensions for Rule-Based Mesh Growing

In order to accommodate both the mesh representation and the traversal environment structures, which have been introduced in the previous chapter, to rule-based mesh growing, we introduce two necessary extensions, namely *expansion indices* and *join indices*. Furthermore, we define an additional mesh operator *apply face operators* which bridges the gap between generalized subdivision meshes and rule-based mesh growing.

### 4.4.1 Expansion Indices

With respect to our mesh representation defined in Section 3.2 we define an additional per-face property named *expansion index*. The expansion index is an integer value which simply specifies the index of the appropriate expansion rule for this face.

### 4.4.2 Join Indices

The *join index* is the second extension to the mesh representation defined in Section 3.2. It determines which faces should be connected (joined) throughout the mesh growing process as specified in Section 4.5.5.

### 4.4.3 Apply Face Operators

The *apply face operators* operator serves as a link between *generalized subdivision meshes* and *rule-based mesh growing*. It takes a mesh growing *ruleset* as a parameter and applies the productions which are defined in the ruleset to the current mesh. In the following section a detailed description of *face operators* is given.

## 4.5 Face Operators

In the style of *mesh operators* introduced in Chapter 3, which perform vertex placement operations on submeshes, we introduce the notion of *face operators*. Face operators are applied to single faces, and cover for example the direct and indirect *merging operators* defined in the previous sections.

If a given submesh $M^{(i)} = (V^{(i)}, F^{(i)})$, where $V^{(i)}$ is an ordered sequence of vertices $v_1^{(i)}, v_2^{(i)}, ..., v_n^{(i)}$, and $F^{(i)}$ is the set of faces $f_1^{(i)}, f_2^{(i)}, ..., f_m^{(i)}$ of the mesh, and $R = (V_R, F_R)$ is a replacement mesh, then a *face operator*

$$\phi(M^{(i)}, r, R) = M' \tag{4.1}$$

is said to replace the $r$-th face of submesh $M^{(i)}$ by the given replacement mesh $R$. For certain operators (see Section 4.3.2), a join geometry $J = (V_J, F_J)$ is created, which may be empty for arbitrary face operators $\phi$. The mesh $M' = (V', F')$ is the result mesh after applying $\phi$, where

$$V' = \left(V^{(i)} \cup V_J \cup V_R\right) - \left(\left(V^{(i)} \cap V_J\right) \cup (V_J \cap V_R) \cup \left(V^{(i)} \cap V_R\right)\right) \tag{4.2}$$

and

$$F' = \left(F^{(i)} \cup F_J \cup F_R\right) - \left(\left(F^{(i)} \cap F_J\right) \cup (F_J \cap F_R) \cup \left(F^{(i)} \cap F_R\right)\right) \tag{4.3}$$

Finally, more than one replacement operation may be performed, the number limited only by the number of faces in the given mesh $M^{(i)}$. In practice, the parallel rewriting process commonly defined for L-systems, is serialized, which means that a given sequence of *face operators* e.g. $(\phi_1, \phi_2, \phi_3)$ is also applied sequentially (not in parallel). Each operator is applied to the result mesh of its predecessor. E.g.

$$M^{(i+1)} = \phi_3(\phi_2(\phi_1(M^{(i)}, r_1, R_1), r_2, R_2), r_3, R_3) \tag{4.4}$$

This poses no problems, since each replacement rule is associated only with single faces, so each possible permutation of the replacement order, will yield the same result mesh.

In the following sections we will give a detailed description of all available face operators and we will also demonstrate their application using examples.

## 4.5.1 Attach

The *attach* operator is either applied implicitely (performing a *direct merge* – if no face operator is defined in a rule – or it can be specified explicitely if *indirect merging* should be performed.

## 4.5.2 Auto-Attach

The *auto-attach* operator automatically creates a suitable transformation node between two meshes that should be joined together. This operator is a variable transformation node which adapts to the current environment each time it is traversed. This means, that it takes a look at the current expansion face which is stored in the traversal environment and also gathers information about the connecting face of the subsequent expansion mesh. Using this data, a transformation is constructed which will scale, translate and rotate the subsequent expansion mesh to fit onto the current expansion face (see Figure 4.6). Another advantage of using Auto-Attach nodes is, that meshes can be altered without the risc of breaking subsequent transformations or the need of re-calculating transformations by hand. A formal description of this operator is given in Appendix 9.2.

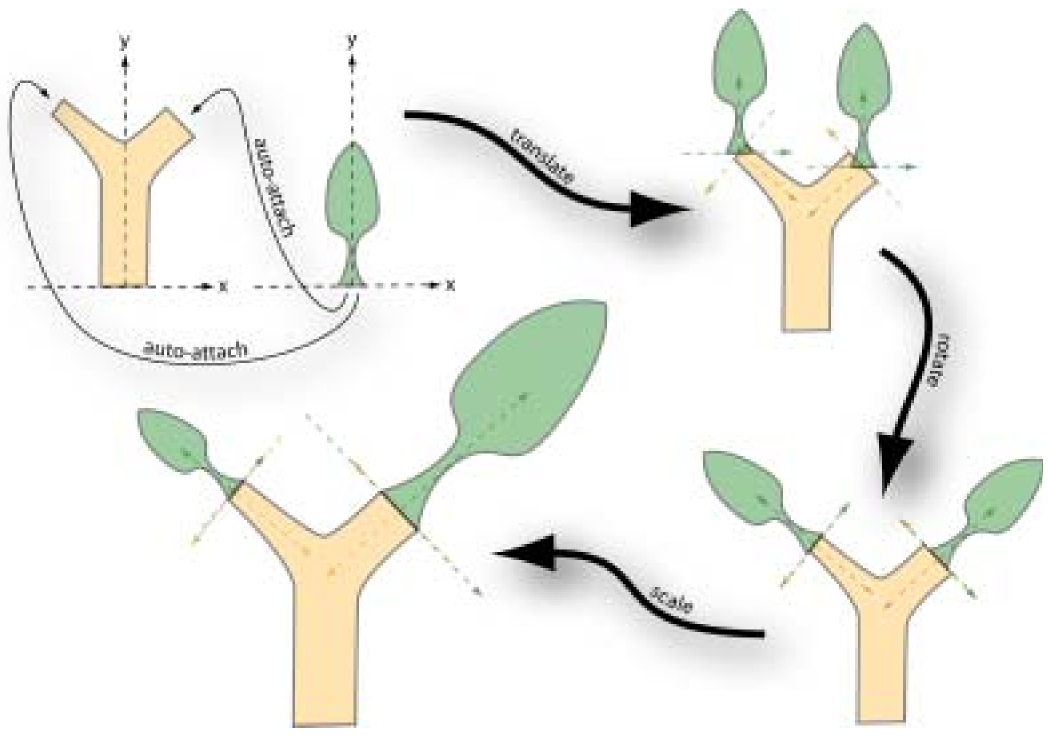


Figure 4.6: Auto-Attach: in the initial configuration (top left), a leaf-shaped mesh should be attached twice to a simple branching structure. First of all, the auto-attach operator translates the leaf-shaped mesh to an appropriate position (so that the face centers of both connecting faces are aligned). In the next step, the leaves are rotated, such that the tangent planes of the connecting faces are in parallel. Finally, the leaves are scaled accordingly.

### 4.5.3 Tropism-Attach

*Tropism-attach* operators are, like *auto-attach* nodes, variable transformation nodes which adapt to the current environment, but unlike their counterparts, *tropism-attach* nodes do not transform subsequent expansion meshes to fit perfectly onto the respective expansion face, but tweak the transformation towards a user-specified center of the tropism force ($\boldsymbol{t} \in \mathbb{R}$). The magnitude depends on a user-specified tropism strength ($0.0 \leq s \leq 1.0$), where a strength of $s = 0.0$ means no tweaking at all and $s = 1.0$ results in a complete transformation towards $\boldsymbol{t}$. In practice values of $0.0 < s < 0.3$ are used because greater values easily result in transformations which in turn result in degenerated meshes.

### 4.5.4 Face-Split

The *face-split* operator is used to split single faces into a regular grid of smaller faces. This is useful for creating architectural models, especially claddings. The most important parameters are the number of rows and columns ($rows > 0$, $columns > 0$) of the regular grid and the size of the border ($0.0 < bordersize < 1.0$). A border is necessary to avoid the creation of T-vertices at edges between the split-face and its neighbouring faces. Beyond these basic parameters a handful of additional parameters can be used to assign sheet numbers and expansion indices to sub-faces. For an example see Figure 4.7.

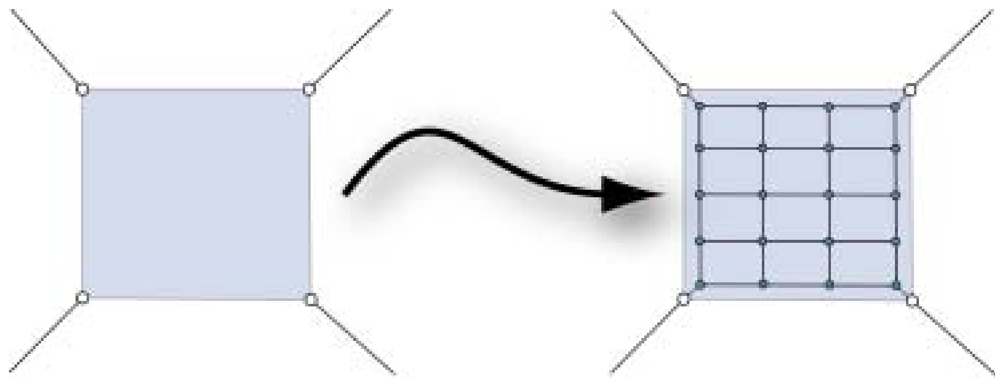


Figure 4.7: Custom face split with 4 rows, 3 columns, and border 0.15.

### 4.5.5 Joins and Join-Geometry

Up to now, the topology of a mesh can not be more complex than the combined complexity of the expansion meshes used to generate this mesh. This means, that the only way to add, for instance, a hole is to attach a mesh with a hole. It is not possible to change the topology of the growing mesh itself. Of course this is a severe limitation which prevents the efficient creation of a whole class of models like ladders, scaffolds, fences and similar objects. In

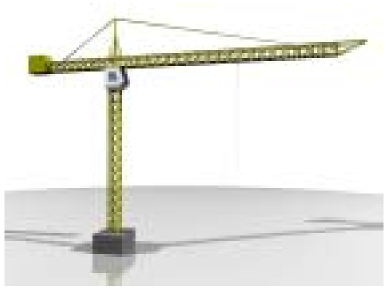


Figure 4.8: Notice that the whole object is one single mesh, and *not* a CSG object or the like. The complex strut pattern is the result of a combined process using rule-based mesh growing and join operators. Join indices have been specified procedurally. The model consists of 9246 faces; model generation takes about 0.1 seconds; photorealistic rendering takes 17 seconds; image resolution of 1600x1200 pixels; all values measured on an Athlon 650MHz processor.

order to overcome these problems we introduce the notion of a *join operator* which allows for joining pairs of faces in the growing mesh. A *join operator* can be assigned to a face and will create a numerical *join id* for this face. Faces with identical *join id* will be connected (see Appendix 9.3 for details). The value of the *join id* is either assigned statically in the scene description file (which is appropriate only for simple objects where only few faces are joined), or the value can be an arithmetic expression which will be evaluated each time the *join operator* node is traversed throughout the mesh growing process, the latter being the only way to assign *join identifiers* in highly recursive scene graphs where the same replacement mesh is attached multiple times, but each time a unique *join identifier* should be assigned. Figure 4.9 shows the creation of a simple object using *join operators* and Color Plate 4.8 depicts a complex crane model, where join operators have been used to specify the complex strut geometry.

## 4.6 Combining Both Techniques

Both presented techniques, *generalized subdivision meshes* and *rule-based mesh growing* can be combined by using a rule-based growing step as an even more generalized subdivision rule. The *apply face operators* operator defined in Section 4.4 has been defined in order to permit the application of a rule-based mesh growing step as an operator in a *mesh operator sequence.*

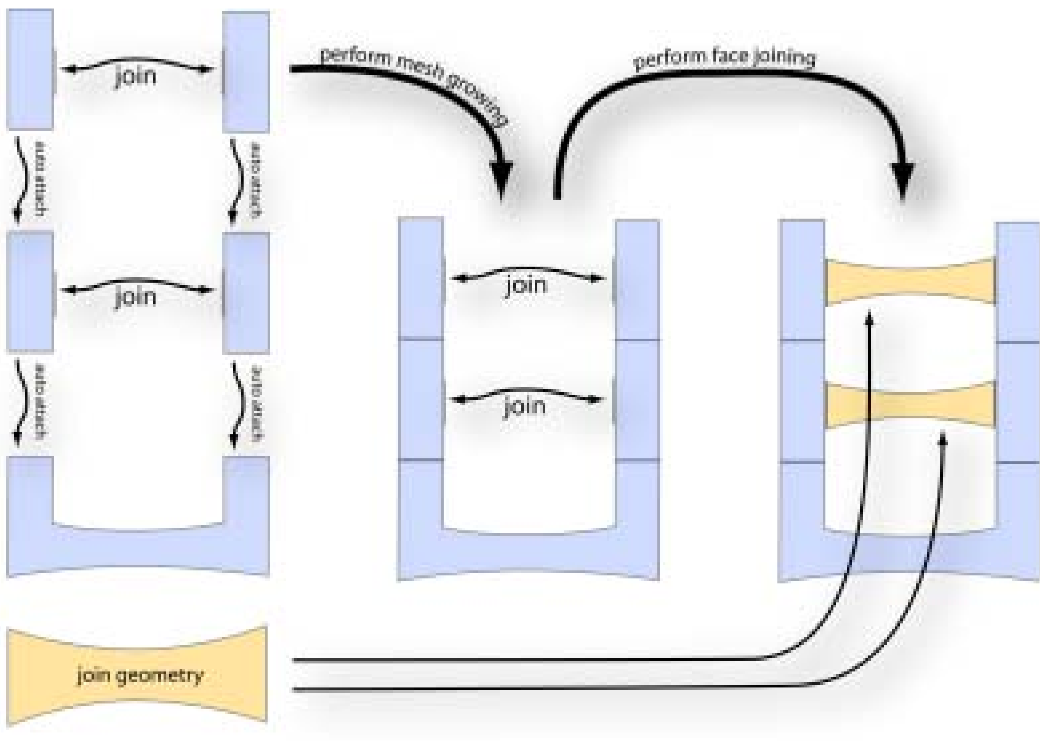


Figure 4.9: Joins

Strictly speaking, no actual subdivision takes place, but since new geometry is introduced there is some similarity between a normal subdivision step and such a mesh growing step. Thus it is possible to alternate between subdivision steps that use texturing functions for vertex placement, and mesh growing steps that expand the geometry in places where it is necessary to add more detail.

Using this combined scheme we can now generate complex geometry that uses the advantages of both of these schemes. As an example, Color Plates 7.9 and 7.10 show a forking branch of a tree.

The branching structure of the tree was modeled using a mesh growing step. Afterwards a number of smoothing steps were used to make structure look more natural. The resulting structure, although convincing in its overall form, still lacks detail. After assigning suitable texture coordinates to the vertices in the original mesh, the method of DeRose et al. [DKT98] can be used to generate texture coordinates at each subdivision level.

In Figure 4.10 these texture coordinates were used to modulate the displacement in the vertex placement step of the following subdivision steps.

The left image shows the unmodified groove structure, in the right image, some random displacement of the ridges was added to obtain a more natural look. The actual texture coordinates can be found in Figure 4.11.

## 4.7 Summary

In this chapter we proposed an extension to pL-systems called *mesh-based pL-systems* which allows the definition of L-system-like rulesets to operate directly on meshes. In our scheme, rules specifying replacement geometry are associated with single faces in a mesh. Each face which has an associated rule is replaced by the so-defined replacement geometry, which in turn may also contain faces with associated rules. Thus complex meshes can be represented by compact rulesets. In combination with *generalized subdivision meshes* (see Section 3) a powerful method for generating and representing complex and detailed geometrical shapes in a truly multiresolution fashion is available. In the next chapter we will point out selected issues concerning the implementation of our proposed framework in a general purpose rendering system. After this, we will use this implementation in the more practical setting of an prototype plant editor, in order to show its applicability to real-world applications.

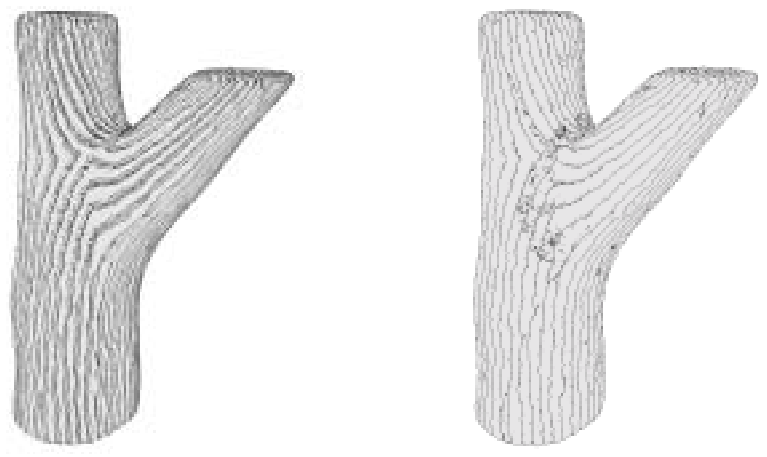


Figure 4.10: Texture coordinates are used to modulate the displacement of vertices to create a groove structure (left), some random displacement of the ridges was added to obtain a more natural look (right).

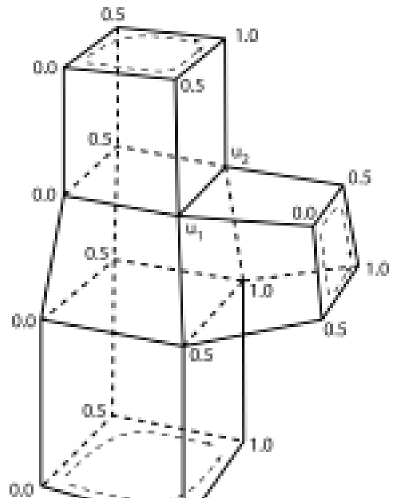


Figure 4.11: $u$-coordinates at a forking branch: $u_1 = \frac{0.0+0.5}{2}$, $u_2 = \frac{0.5+1.0}{2}$.

# Chapter 5

# Implementation

## 5.1 ART - Advanced Rendering Toolkit

The *Advanced Rendering Toolkit* (ART for short) [Tea01] is a set of *Objective C* Libraries that provide a wide range of functionality suitable for graphics applications. The ART libraries do not deal with the user interface, they provide classes and methods starting with primitive graphics objects like vectors, points and matrices up to classes that make it possible to define complete three-dimensional scenes and a number of different methods to manipulate and render these scenes. The Advanced Rendering Toolkit is distributed under the terms of the *GNU Library General Public License* [Fou00].

Scenes are modeled using a scene description language that is based on Objective C. A scene description contains the complete information about geometry, surface characteristics, and illumination in a scene. All this information is organized in a scene graph, which is a *directed acyclic graph* (DAG). Each object which can be stored in the scene graph is derived (either directly or indirectly) from class *ArNode* which implements all necessary scene graph related methods.

One of the highlights in ART is its support for a number of different color models which currently include RGB, CIEXYZ, Spectrum8, Spectrum16, Spectrum45, Spectrum450, polarized light and fluorescence. Spectral colortypes are modelled as 8, 16, 45 and 450-band spectra respectively and cover the visible spectrum from 380nm to 830nm. To the best of our knowledge, ART is the only general purpose rendering system which supports combined rendering of polarization and fluorescence effects ([WTP01]).

In consideration of the fact, that the Advanced Rendering Toolkit shows a clean and modular design and is therefore easily extensible, it was decided to implement our proposed *generalized subdivision meshes* and *rule-based mesh growing* scheme in ART. See Section 5.3 for details on how we incorporated *directed cyclic graphs* which are necessary to represent L-system-like structures into a rendering system actually based on *directed acyclic scene graphs.*

## 5.2 Scene Graphs, Scene Graph Traversal, and Traversal Environment

As stated above, ART is a scene graph based rendering system. A scene graph contains the complete description of an entire scene. This includes geometry, transformations, surface and material attributes, illumination information, camera definitions, and so on. For all possible entities classes have been defined, which encapsulate all information necessary to completely describe each of these different entities.

A particular scene is defined by creating instances of these classes to represent concrete objects and attributes, and by organizing these objects in a directed graph. In order to render a scene, the renderer traverses the scene graph and at each node performs various actions according to the type of object (e.g. calculating ray-object intersections, evaluating a solid texturing function, etc.). Furthermore, all the attributes encountered throughout scene graph traversal are collected in a so-called *traversal environment*, thus the traversal environment acts as a kind of container, which at each time holds the currently valid set of attributes. For example, if an attribute defining a wooden surface texture is encountered it is stored in the traversal environment and applied to all subsequent geometric shapes, until a new surface attribute is encountered which then replaces the previous surface attribute. See Figure 5.1 for an example scene graph.

## 5.3 Directed Cyclic Scene Graphs

The main task of implementing our proposed schemes is to give ART the ability to process directed cyclic scene graphs. As a matter of fact, the Advanced Rendering Toolkit is based on directed acyclic scene graphs. If we decided to simply extend the existing scene graph framework by DCG properties we would have rendered useless some highly optimized and proven scene graph traversal code. At least we would have had to rewrite key parts of the existing code, which would have resulted in potentially unpredictable

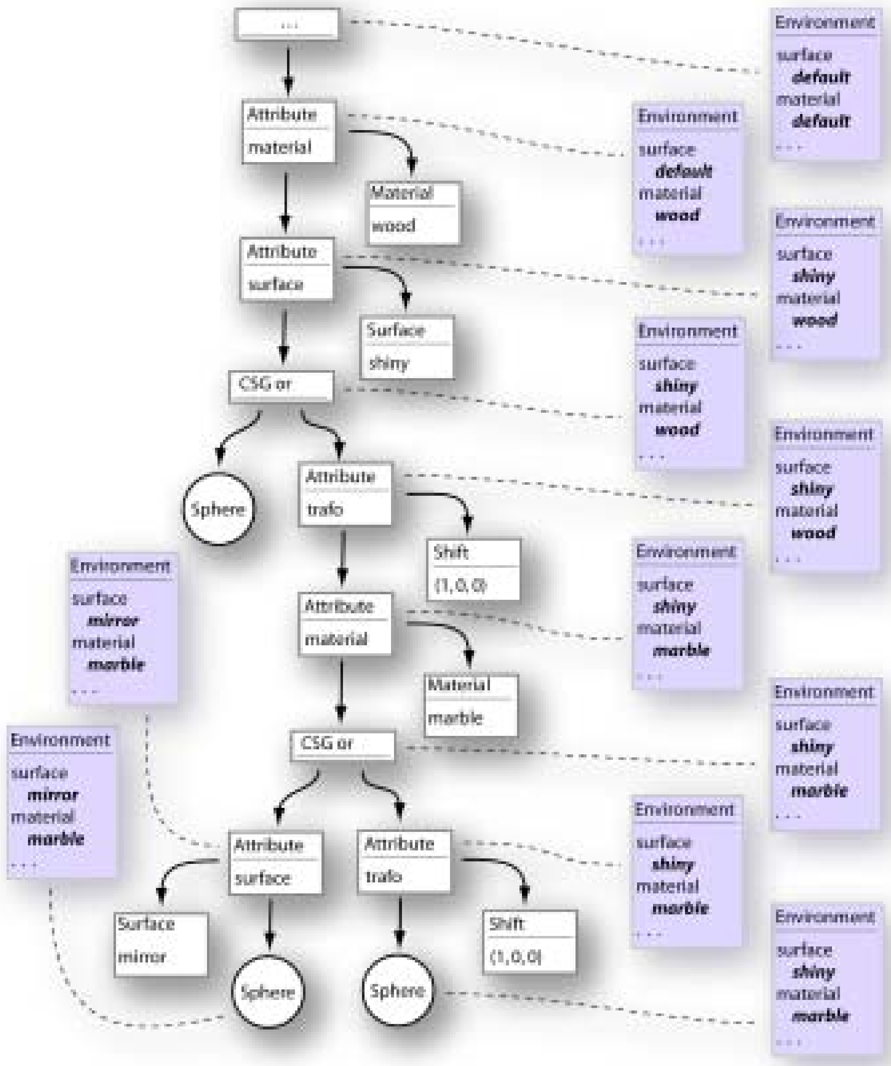


Figure 5.1: Scene graph, scene graph traversal and traversal environment. This part of a scene graph (center) defines a simple CSG object, which consists of three single spheres with different surface and material properties. The colored frames at the left and right side denote the traversal environment as it appears at each node when the scene graph is traversed. Only surface and material attributes are shown in order to keep this illustration simple, in fact various additional attributes are stored in the traversal environment too (e.g. current global-to-local transformation matrix, ...).

side-effects and a need for extensive regression testing. In order to avoid this, we introduce so-called *reference nodes*.

### 5.3.1 Named Nodes and Reference Nodes

The concept of reference nodes is similar to *pointers* in the C programming language. Instead of pointing to a certain place in memory, reference nodes point to a certain node in the scene graph. This is achieved due to a feature called *named nodes*, which makes it possible to assign unique names to arbitrary nodes. In turn, a reference node is assigned the name of the node it should point to. As a result, the combined use of *named nodes* and *reference nodes* make it possible to represent directed cyclic graphs within the framework of a directed acyclic graph. As a matter of fact, existing traversal code needs not to be adjusted and simply stops traversing the scene graph on reaching a reference node, like it was an acyclic graph. Only new parts of the scene traversal framework which actually make use of cyclicity need to interpret reference nodes like pointers to other nodes - so we effectively decoupled existing code (specific to acyclic scene graphs) from newly introduced cyclic features (i.e. rule-based mesh growing).

### 5.3.2 Value Nodes

Rule-based mesh growing is based on parametric L-systems and therefore depends on the availability of parameters which are used to pass information among different rules and nodes, in order to perform calculations like increment and decrement of branching angles and indices, and logical evaluations of conditions.

In standard pL-systems, a set of parameters $a_1, a_2, ...a_n \in \mathbb{R}$ is associated with each specific rule (module). Although floating point parameters are sufficient to model all necessary types of other parameters, there are additional types for integers, two-dimensional and three-dimensional points, and vectors - as a convinient extension for modeling purposes.

Throughout the ART framework, *value nodes* are typed nodes which hold a specific value (e.g. integer value node, $value = 17$). In addition a comprehensive collection of arithmetic operator nodes is available which allows the construction of arbitrary arithmetic expression trees. Since such a framework is also useful for applications other than *generalized subdivision meshes* and *rule-based mesh growing*, e.g. shading language contructs and user-specified solid texturing functions [THP98], [Pea85], [Per85], [Wor96], a basic implementation of *value nodes* has already been part of the ART framework.

We extended the existing implementation by the possibility of assigning names to value nodes, creating a notion of variable-like constructs. Named value nodes are stored in the traversal environment when such a node is beeing traversed, allowing for later reuse of a once defined *variable*. In order to provide a more convinient tool for the specification and manipulation of *variables* and *expressions* we implemented a parser for C/C++ like expressions [Str97] which transforms a user-specified string (containing an expression) into a valid, partial scene graph, comprised of the value and arithmetic operator nodes. In addition to primitive types *long* and *double* we introduced additional types for *two-dimensional points* (Pnt2D), *two-dimensional vectors* (Vec2D), *three-dimensional points* (Pnt3D), and *three-dimensional vectors* (Vec3D) which can be used exactly like other primitive types (see Table 5.1. The following example should give some insight:

```
level++;
angle += 22.5;
angle = (angle < 150.0) ? angle : 150;
selection = (level < 3) ? 1 : (angle > 90) ? 2 : 3;
```

Although value nodes are typed internally for performance and implementation reasons, the user does not have to deal with types and can specify fully untyped expressions. The parser automatically derives type information necessary for creating typed value nodes which are used internally. This can be seen in the above example where the variable *level* is treated as an integer value and the variable *angle* as floating point. Even if floating point values and integer values are mixed up like in line 3, the parser will insert appropriate *casts* where necessary. Another exampe demonstrates how points and vectors are fully integrated as primitive types:

```
x = 1.0;
y = 2.0;
z = 7.5;
v = Vec3D(0.1, 0.1, 1.5);
p = Pnt3D(x,y,z) + v;
u = atan(p.y, p.x);
```

Of course, also a comprehensive set of trigonometrical and other functions can be used.

### 5.3.3 Assignment Nodes

As we have seen in the above examples, value nodes can not only be created with a static value, but also new values can be assigned, which is absolutely

| **Expression Summary** | |
|---|---|
| sequence | *expr* , *expr* |
| conditional expression | *expr* ? *expr* : *expr* |
| subtract and assign | *value* −= *expr* |
| add and assign | *value* + = *expr* |
| modulo and assign | *value* %= *expr* |
| xmodulo and assign | *value* %%= *expr* |
| divide and assign | *value* /= *expr* |
| xdivide and assign | *value* //= *expr* |
| multiply and assign | *value* *= *expr* |
| simple assignment | *value* = *expr* |
| logical inclusive or | *expr* \|\| *expr* |
| logical and | *expr* && *expr* |
| not equal | *expr* != *expr* |
| equal | *expr* == *expr* |
| greater than or equal | *expr* >= *expr* |
| greater than | *expr* > *expr* |
| less than or equal | *expr* <= *expr* |
| less than | *expr* < *expr* |
| subtract | *expr* – *expr* |
| add | *expr* + *expr* |
| modulo | *expr* % *expr* |
| xmodulo | *expr* %% *expr* |
| divide | *expr* / *expr* |
| xdivide | *expr* // *expr* |
| multiply | *expr* * *expr* |
| cast | (*type*) *expr* |
| unary plus | + *expr* |
| unary minus | – *expr* |
| not | ! *expr* |
| post increment | *value* ++ |
| post decrement | *value* −− |
| member selection | *expr.member* |
| function call | *symbol* ( *sequence* ) |
| 2-dim point constructor | Pnt2D ( *expr* , *expr* ) |
| 2-dim vector constructor | Vec2D ( *expr* , *expr* ) |
| 3-dim point constructor | Pnt3D ( *expr* , *expr* ) |
| 3-dim vector constructor | Vec3D ( *expr* , *expr* ) |

Table 5.1: This table gives an overview of expressions that can be used for *generalized subdivision meshes* and *rule-based mesh growing*.

essential for constructing meaningful rulesets. Whenever values should be assigned to existing value nodes ($=, + =, - =, * =, / =, \% =, \%\% =$) a so-called *assignment node* is used. An assignment node holds a named reference to a value node (left-hand side of assignment) and an expression tree (right hand side of assignment). Whenever such an assignment node is traversed, it evaluates the given expression, and assigns it to the named value, which is stored in the traversal environment for subsequent evaluation. Each named value stored in the environment is in fact a stack, onto which the value of a new assignment is pushed on assignment node traversal, and which pops off the latest value after all child nodes have been traversed.

### 5.3.4 Rules

A single rule $lhs : rhs$ specifies, that the *left-hand side* which specifies a single *symbol* should be replaced by the given *right-hand side*, which consists of a valid partial scene graph comprised of a set of *transformation nodes*, *mesh nodes*, *reference nodes*, *value nodes* and *assignment nodes.* Transformation nodes denote either static or variable (depending on value nodes) transformations (translate, scale, rotate, ...). Mesh nodes contain a static mesh description which serves as a building block for mesh growing.

With regard to the scene graph framework presented so far, a single rule can be represented by simply assigning a name to the root node of a partial scene graph. The partial scene graph then represents the *right-hand side* and the assigned name serves as *left-hand side. Reference nodes* can be used to refer to a specific rule by referencing the required *name* (left-hand side). See Figure 4.3 for an example ruleset.

## 5.4 Winged-Edge Mesh Representation

We use a winged-edge data structure for mesh representation. This data structure has been originally proposed in the nineteen-seventies by Baumgart [Bau72], [Bau75], and has since then stood the test of time. Although there exist countless different implementations, the basic concept is always the same: lists for vertices, edges, and faces are maintained, and each vertex, edge, and face stores indices for adjacent elements which point back into the appropriate lists. Which elements store which indices mostly depends on which queries a specific application needs to perform. The more adjacency information is stored, the *richer* the data structure is. Of course more adjacency information means simpler and more performant queries, but also a higher memory footprint. Some frequently used queries are, for example:

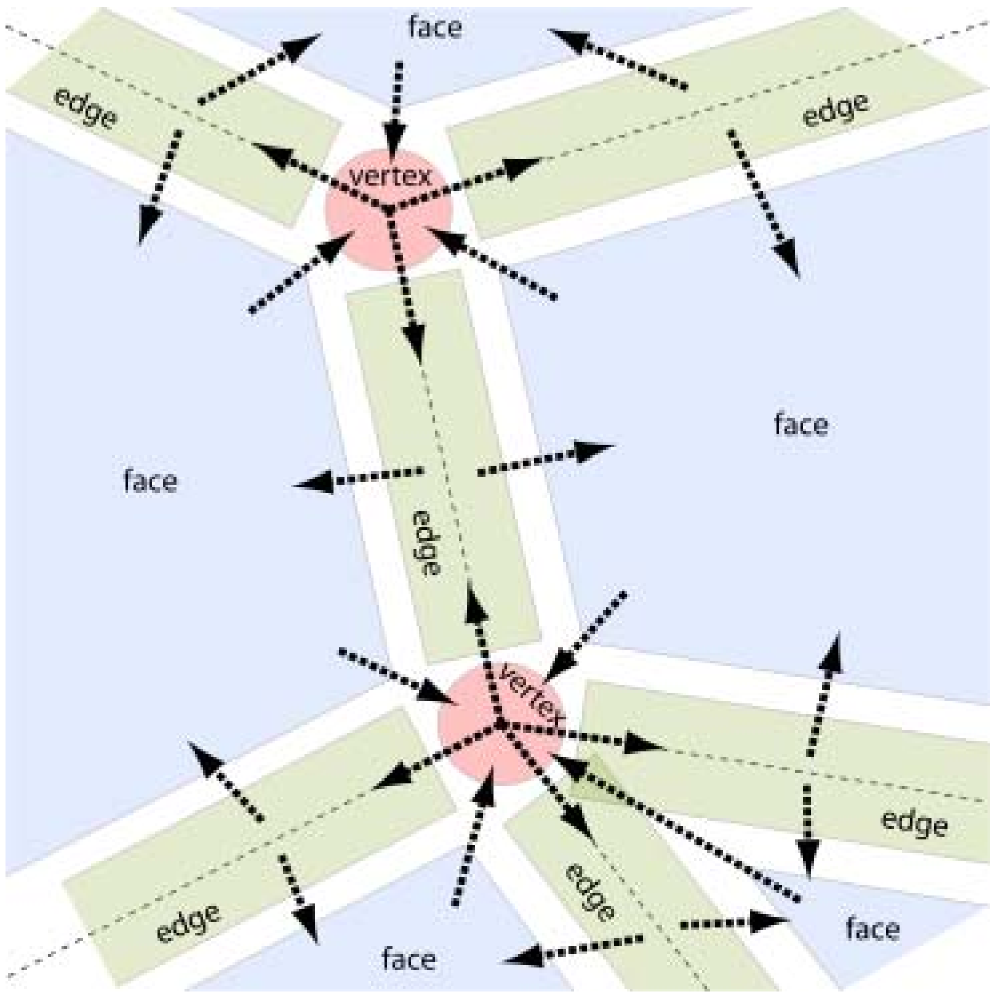


Figure 5.2: Winged-edge mesh representation: there are direct references from faces to vertices, from vertices to edges, and from edges to faces.

- find all edges for a given face in (counter)-clockwise order
- find all vertices for a given face
- find all faces adjacent to a given vertex
- find all edges adjacent to a given vertex
- find all vertices for a given edge
- find all adjacent faces for a given edge
- find the next edge for a given edge with respect to a given face

- ...

A winged-edge data structure has already been implemented in the Advanced Rendering Toolkit. In order to use it for *generalized subdivision meshes* and *rule-based mesh growing* we had to adjust it to some additional requirements that originate from our proposed schemes.

The original implementation is an extremely memory and runtime efficient data structure implemented purely in C using different *structs* for *vertices*, *edges*, and *faces*, and for reasons of performance, *macros* to query and navigate the data structure. In general, it was optimized towards representing predefined static meshes, but it was not well suited for frequent modification of connectivity information, which is necessary for rule-based mesh growing. Nevertheless we decided to go along with the existing data structure, although the necessary adaptions for mesh growing turned out to be quite involved. On the other hand we now have a very performant implementation.

The basis of the mesh data structure are tables of simple data structures for faces (*fa* - face array), vertices (*va* - vertex array), and edges (*ea* - edge array) (see Table 5.2). Each face (*ArmFace*), vertex (*ArmVertex*), or edge (*ArmEdge*) is uniquely identified by the index in its respective table. References between faces, vertices, and edges therefore use the respective index to indicate the target.

The references between these data structures are built from two parts: the index of the referenced entity, and the side of the referenced entity, which the reference points to. As an example a pentagon has 5 edges, thus these edges are locally numbered 0 to 4, in counter clock-wise order, and a reference to edge 3 of the pentagon contains both, the index of the pentagon and a 3 indicating that the third edge of the pentagon is referenced.

There are direct references from faces to vertices, from vertices to edges, and from edges to faces. The references from an edge to its two adjacent faces are stored directly in the edge data structure. Since faces can have different numbers of vertices, and vertices can have different numbers of adjacent edges, the references from faces to vertices (*fvra* - face vertex reference array), and from vertices to edges (*vera* - vertex edge reference array) are stored in two extra tables. The faces and vertices only contain the base index of the respective references in these reference tables (*vrbi* - vertex reference base index, *erbi* - edge reference base index). The number of face sides is stored in *num_v* (number of vertices), the number of vertex sides is stored in *num_e* (number of edges).

```
struct ArmFace                    struct ArmVertex
{                                 {
    UInt8       type;                 UInt8       type;
    UInt8       num_v;                UInt8       num_e;
    long        vrbi;                 long        erbi;
};                                };


struct ArMesh                     struct ArmEdge
{                                 {
    ArmFace *       fa;               UInt8       type;
    ArmVertexRef *  fvra;             UInt8       flags;
    ArmVertex *     va;               ArmFaceRef  fr[2];
    ArmEdgeRef *    vera;         };
    ArmEdge *       ea;

    long num_f;
    long num_fvr;
    long num_v;
    long num_ver;
    long num_e;   /* = num_ie + num_be */
    long num_ie;  /* inside edges */
    long num_be;  /* border edges */
};
```

Table 5.2: Data structures for faces, edges, and vertices, and the mesh data structure.

# Chapter 6

# An Interactive Editor

## 6.1 Overview

In order to use and test our proposed *generalized subdivision meshes* and *rule-based mesh growing* schemes in a real-world application, we created a prototype implementation of an interactive editor for the creation of simple plants. The editor is based on ideas from both Weber and Penns plant model [WP95] and the user interface of Deussen's XFrog editor [LD96]. In addition to this, we created an interface for *semi-automatic parameter extraction from photographs* on top of the editor framework. The editor was implemented using the Java programming language [AGH00] and the Java 3D API [SRD00], because this allowed a rapid prototyping approach concerning our ideas for the editor. Since our approach proved to be feasible, we currently work on porting the editor to a native C/C++ and OpenGL implementation which is more performant and which can be more easily integrated with the actual rendering system (Advanced Rendering Toolkit). This also renders unnecessary the now mandatory convertation of plant model data which is generated by the editor to the environment of the rendering system. The decision was taken because although we performance-tuned the Java implementation [Shi00], the actual performance nevertheless drops for about 30% to 60% compared to a native C/C++ and OpenGL implementation. This is mainly due to the fact that Java is compiled to an intermediate, plattform-independent bytecode representation which has to be interpreted or compiled just-in-time by a Java Virtual Machine which inescapably introduces some performance penalty. Another point is the lack of explicit memory management in Java (a garbage collector decides which objects are not needed anymore and frees the associated memory). This is a big advantage on the one hand - it really allowed us to use Java as a rapid prototyping environment, but simultaneously it leads to yet another performance penalty, particularly when using large datastructures which consist of a high number of small chunks, which are frequently subject to change (mesh data structures). Finally, a tight integration with the existing non-Java rendering system proved to be difficult. Altough the *Java Native Interface* [Lia99] provides means of interfacing Java code with native C/C++ code, this is a quite tedious undertaking and would have vitiated the big advantage of Java's platform independence.

In summary, we do not regret having used the Java platform as a rapid prototyping environment, since we could really try out many different ideas with a minimum of coding and virtually no debugging effort, but finally we decided against Java for the real production implementation mainly because of non-Java-specific issues concerning the desired interfacing with existing software, and also a number of performance issues (but which in fact would have not been sufficient alone to decide against the Java implementation).

## 6.2 Simple Plant Model

Our simple plant model describes the structure of plants by using only two different components: *stem* and *leaf.* Stem components are constructed by connecting a number of quadrilateral cross-sections. Additionally, stem components may split off a number of stems and/or leaves. Leaf components are arbitrary shaped components which can not split off any components.

Various parameters are defined as distribution functions along the components length. Throughout this paper, distribution functions are continuous functions defined in the interval $[0.0, 1.0]$ which is linearely mapped to the real length of the respective component.

Instances of components are connected to form a *directed cyclic graph* (DCG) which completely describes the appearance of a specific plant and which is actually used internally to create a ruleset for *rule-based mesh growing.* From this description a skeleton mesh is generated which is displayed to give immediate feedback to the designer. The skeleton model is rendered in real-time and can be rotated and examined from arbitrary directions. If the directed cyclic graph (ruleset) or single parameters are changed throughout the design process, the skeleton mesh is recreated on-the-fly to maintain a consistent visual feedback for the designer [Shn98], [PRS+94].

Finally, the skeleton mesh, which of course is only a very crude approximation of the plants final shape, can be used as a basemesh in our *generalized subdivision meshes* scheme to apply mesh operators for smoothing transitions and adding small scale detail.

### 6.2.1 The Stem Component

Stem components are defined by a number of parameters: *length*, *width*, a function $diameter\_distribution(x)$, and a set of branching patterns $\{bp_1, \ldots, bp_n\}$. All parameters are fully explained in Tables 6.1 and 6.2. The

| Parameter | Description |
|---|---|
| *length* | Stem length. |
| *width* | Maximum diameter of the stem. |
| $diameter\_distribution(x)$ | Defines the actual diameter at position $x \in [0.0, 1.0]$ along the stem as a fraction of *width*. |
| $\{bp_1, \ldots, bp_n\}$ | Each branching pattern completely defines the distribution of a specific component along the length of the stem, including branching angles and branch scaling. |

Table 6.1: Stem component

| Parameter | Description |
|---|---|
| $number\_of\_branches$ | Number of components to branch off along the length of the stem. |
| $branching\_distribution(x)$ | Defines the positions of branches along the stems length. |
| $angular\_distribution(x)$ | Defines the branching angle for each position along the stem. |
| $length\_distribution(x)$ | Defines the scaling of each branch along the stem. |

Table 6.2: Branching pattern ($x \in [0.0, 1.0]$).

skeleton mesh of a stem component is constructed by connecting two or more quadrilateral cross-sections. A stem with constant or linear diameter distribution ($diameter\_distribution(x) = const$ or $diameter\_distribution(x) = a + bx$) and no branches can be constructed by connecting only two quadrilateral cross-sections - one positioned at the bottom and one at the top of the stem.

A more complex diameter distribution requires the interconnection of more than two cross-sections. The number and positions of cross-sections is carefully chosen so that the resulting shape differs by less than a predefined fraction $\epsilon$ from the idealized shape defined by the given diameter distribution.

A value of $\epsilon = 0.1$ has proven to minimize the number of faces while it is still maintaining a near perfect resemblance of the idealized stem shape. An example for a non-linear diameter distribution and the resulting skeleton mesh is given in Figure 6.1.

If one or more branching patterns are associated with a stem component the skeleton mesh has to be further refined by inserting additional cross-sections at branching positions. This is necessary to maintain a well-formed mesh structure while simultaneously providing connecting faces where branching geometry can be attached. If the number of branches is too large or if two or more branching patterns are combined (see Figure 6.3) the problem of overlapping branches may arise. Since branching meshes are ultimately attached to single faces of the stems skeleton mesh (rule-based mesh growing) and distinct faces must not overlap in order to guarantee a well-formed mesh geometry, branches may have to be repositioned. This is done in such a way that (a) no branches overlap and (b) the accumulated difference of original and adjusted branching positions is kept to an absolute minimum. Configurations may arise where branches are distributed too densely for the optimization process to find a valid solution. In such a case, all branches which can not be placed without violating any rules are rejected. Fortunately, this is not a serious problem, since too densely packed branches would yield unrealistic results anyway. Finally, for each branch the associated component is forced to generate its own skeleton mesh which is scaled and orientated according to the $length\_distribution(x)$ and $angular\_distribution(x)$ functions and attached to the stem mesh according to its (potentially repositioned) branching position.

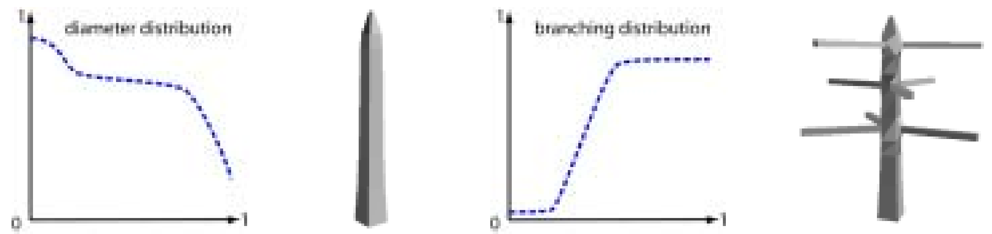


Figure 6.1: Left: a non-linear diameter distribution and the resulting skeleton mesh. Note that values in the interval $[0.0, 1.0]$ are used to scale the *width* parameter along the stem. right: a simple branching distribution and the resulting skeleton mesh. Branching positions $pos_1$ to $pos_{number\_of\_branches}$ are distributed in such a way that for each position $i$ the following holds true: $\int_{x=pos_{i-1}}^{pos_i} branching\_distribution(x)dx = \frac{\int_{x=0.0}^{1.0} branching\_distribution(x)dx}{number\_of\_branches}$.

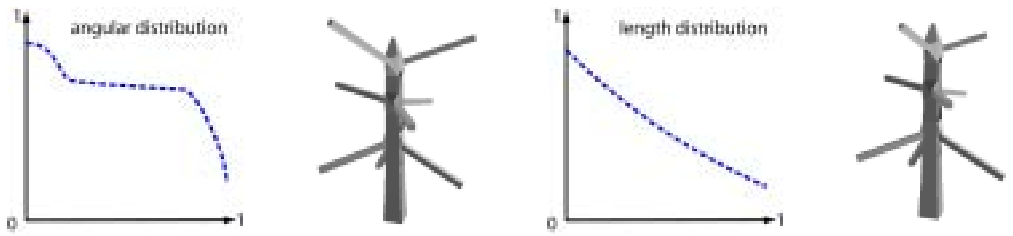


Figure 6.2: Left: angular distribution and the resulting skeleton mesh. The interval $[-1.0, +1.0]$ is mapped to $[-\frac{\pi}{2}, +\frac{\pi}{2}]$, where an angle of 0 results in a branch created normal to the stem direction. Positive angles denote an upward rotation, whereas negative angles denote an downward rotation. right: length distribution and the resulting skeleton mesh.

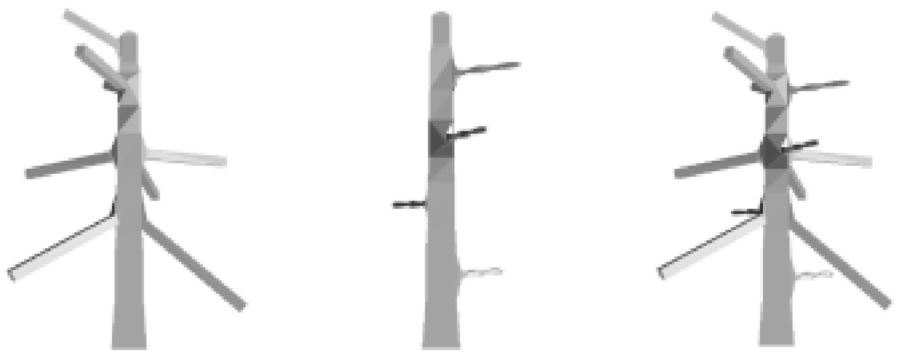


Figure 6.3: Two branching distributions (left, center) are combined (right).

### 6.2.2 The Leaf Component

A leaf component is defined by a set of control points which specify the leafs outline, and an axial and lateral cross-section. A skeleton mesh is created by simply connecting the control points after they have been re-positioned according to the user-specified cross-sections. No special problems arise with the generation of leaves. See section 6.3.2 for a detailed description of the user interface which is used to model leaf components from photographs.

## 6.3 Semi-Automatic Parameter Extraction From Photographs

On top of our simple plant model we created a graphical user interface. It is similar to [LD96] since this kind of interface seems to be a natural fit for the interactive modeling of complex plant models and also for defining rulesets for our proposed *rule-based mesh growing* scheme. The user can arrange stem- and leaf components and connect them to form a *directed cyclic graph* which defines the structural properties of a specific plant. Beyond that, for each component an additional dialog is available which can be used to extract parameters from photographs to quickly find an approximate parameterization. This feature turned out to significantly speed up the process of parameterizing components, since instead of defining a large number of parameters manually, it is now possible to define the same number of parameters by simply drawing some lines and shapes on a photograph. Even for non-experienced users, the number of iterations necessary to create a satisfying model is reduced significantly.

In order to parameterize a specific component the user has to select the desired component in the DCG and to open the dialog for parameter extraction. Extracted parameters are instanteniously transfered to the associated component and the real-time preview skeleton mesh is updated accordingly, again ensuring immediate visual feedback and reducing the number of modeling iterations. After opening a parameter-extraction-dialog, it is first necessary to load a photograph. Due to the powerful libraries provided for the Java platform, all common image storage formats can be imported. Next, a number of component specific modeling actions can be performed which will be explained in the following sections, where the actual component interfaces will be explained in detail.

### 6.3.1 Stem Dialog

Three different modeling actions are available in the stem dialog, which in fact is sufficient to extract the much higher number of actual parameters which have been defined in Section 6.2.1:

| Action | Description |
|---|---|
| Axis | The axis of a stem is specified by simply drawing a line from the bottom to the top of the stem. This information is used to extract the length parameter and also to define a mapping from pixel coordinates to model space. |
| Shape | The shape of a stem is defined by drawing the outline of the stem. From this information the *diameter distribution function* is generated. |
| Branch | A branch is defined by drawing a line from the beginning to the end of the branch. For many stems it is sufficient to mark the bottom-most and the top-most branch. This especially holds true if branches are positioned in a regular order. If the resulting branching pattern is not sufficient for a specific plant, an arbitrary number of additional branches may be marked. The position, orientation and length of all lines is used to derive appropriate *branching-*, *angular-* and *length distribution functions.* |

Figure 7.6 gives an example of the usage of the stem dialog. The first two images show the same photograph of a birch tree. In the left image the main stem, its outline and some child branches have been marked by the user. In the center image a branch and its child branches (twigs) have been marked. The right image finally shows the resulting rendered tree. Figure 6.5 only shows two of three stem dialogs which where used to create the rendered image. In Color Plates 7.13 and 7.14 two different plants are shown that have been "cloned" by using our Java prototype editor.

### 6.3.2 Leaf Dialog

Photographs which are well suited to be utilized in the leaf dialog should be taken in the normal direction to the leafs surface and from a close distance. The first is important to catch the undistorted outline of the leaf, while the latter is recommended to create a high resolution image of the leaf. This is of importance because the photograph is not only used to define the shape of the leaf, but also to automatically extract a texture map for the leaf mesh (see Figure 6.4), which is applied in the photorealistic rendering step. Four different modeling actions are available in the leaf dialog:

| Action | Description |
|---|---|
| Axis | The axis of a leaf is defined by drawing a line from the beginning of the stem of the leaf, up to the tip of the leaf. This information is used to define a mapping from pixel coordinates to model space. |
| Shape | The shape of a leaf is defined by marking a number of points along the leafs outline. It is best to mark extremal points which best describe the leafs shape. |
| Axial cross-section | A separate widget is used to draw an axial cross-section of the leaf. |
| Lateral cross-section | A separate widget is used to draw a lateral cross-section of the leaf. |

Currently the user has to define the faces which create the leafs shape manually by connecting previously defined vertices (a future implementation will take care of this by automatically creating an appropriate triangulation for the user specified vertices. The user-specified cross-sections are finally used to reposition the initially flat leaf geometry in order to resemble the requested shape.

### 6.3.3 Summary

The plant model and the associated editor undoubtedly demonstrate the practicability and applicability of our *generalized subdivision meshes* and *rule-based mesh growing* schemes. Although the plant model and the editor are far from complete and can only be seen as a kind of feasibility study – a number of impressive shapes could be created, which legitimates the assumption that the theoretical framework we proposed, in fact has the potential to simplify and enhance the modeling of complex shapes – which, after all, was our primary intention.

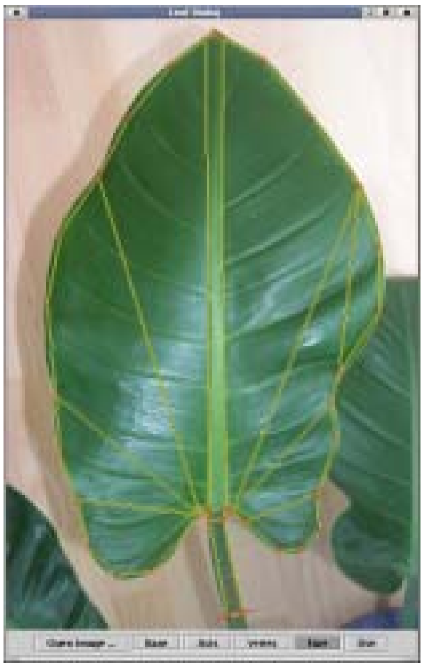
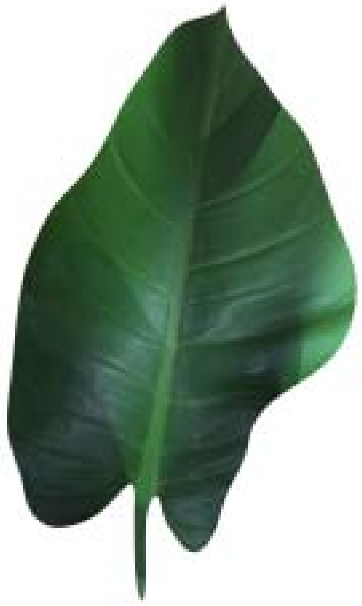



Figure 6.4: User interface for leaves (left) and rendered image (right).

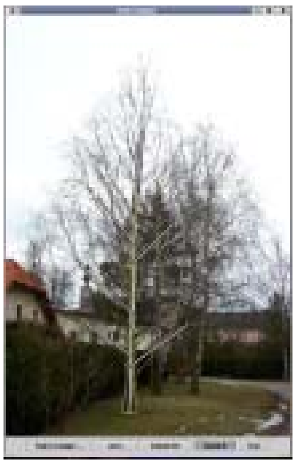
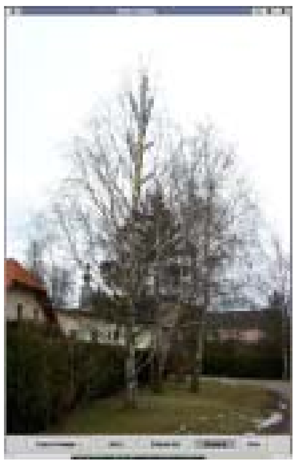
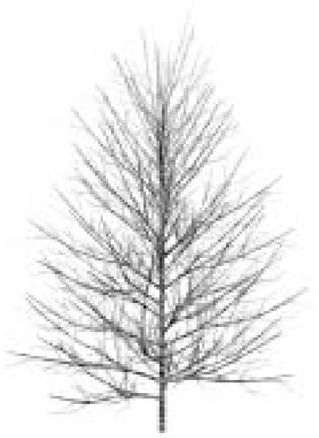




Figure 6.5: User interface for extracting stem parameters (left, center) and rendered image (right).

# Chapter 7

# Results

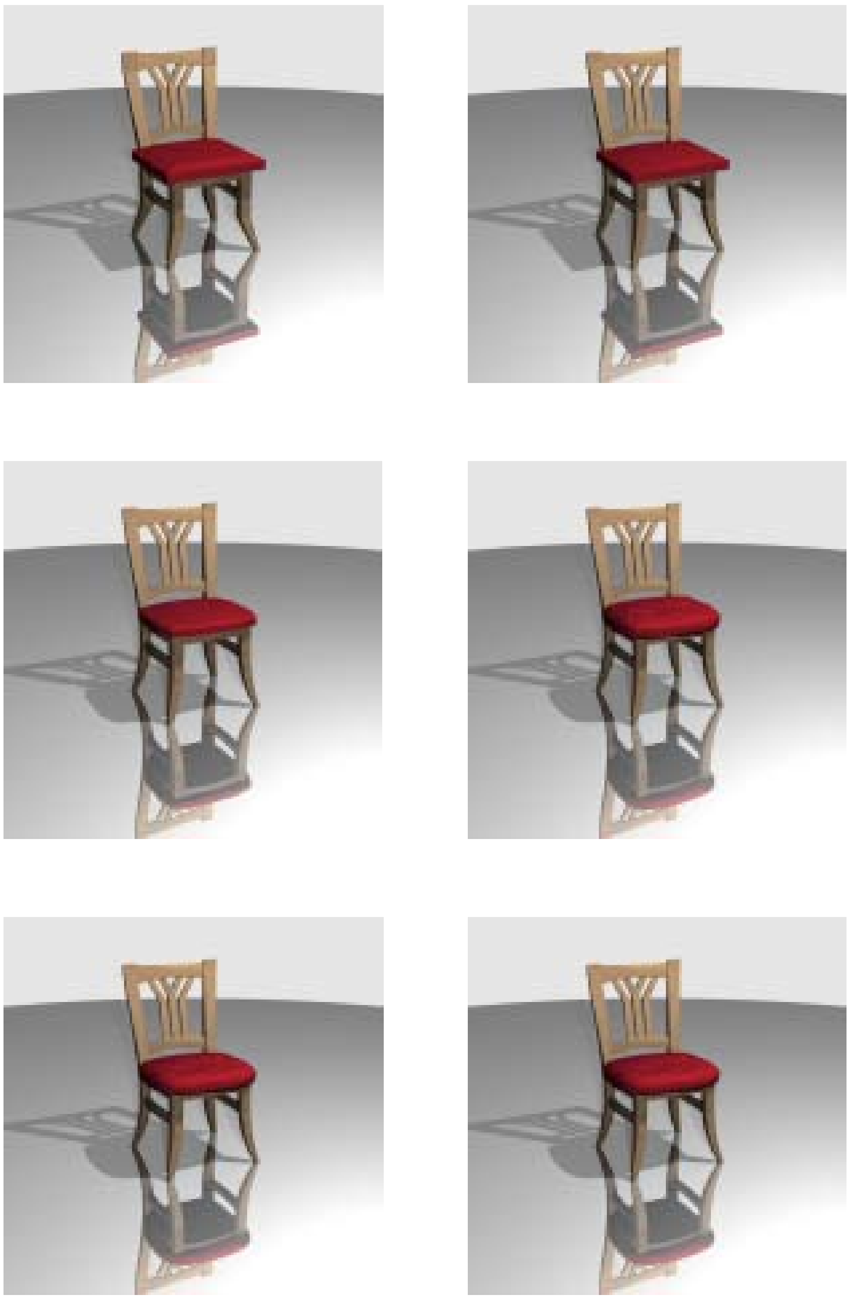


Figure 7.1: Images of a chair rendered at different levels of generalized subdivision. Observe how random variations are introduced in submesh 4 (center row, right) by the application of an *add normal height* operator, and how these variations are smoothed by subsequent Catmull-Clark subdivision steps.

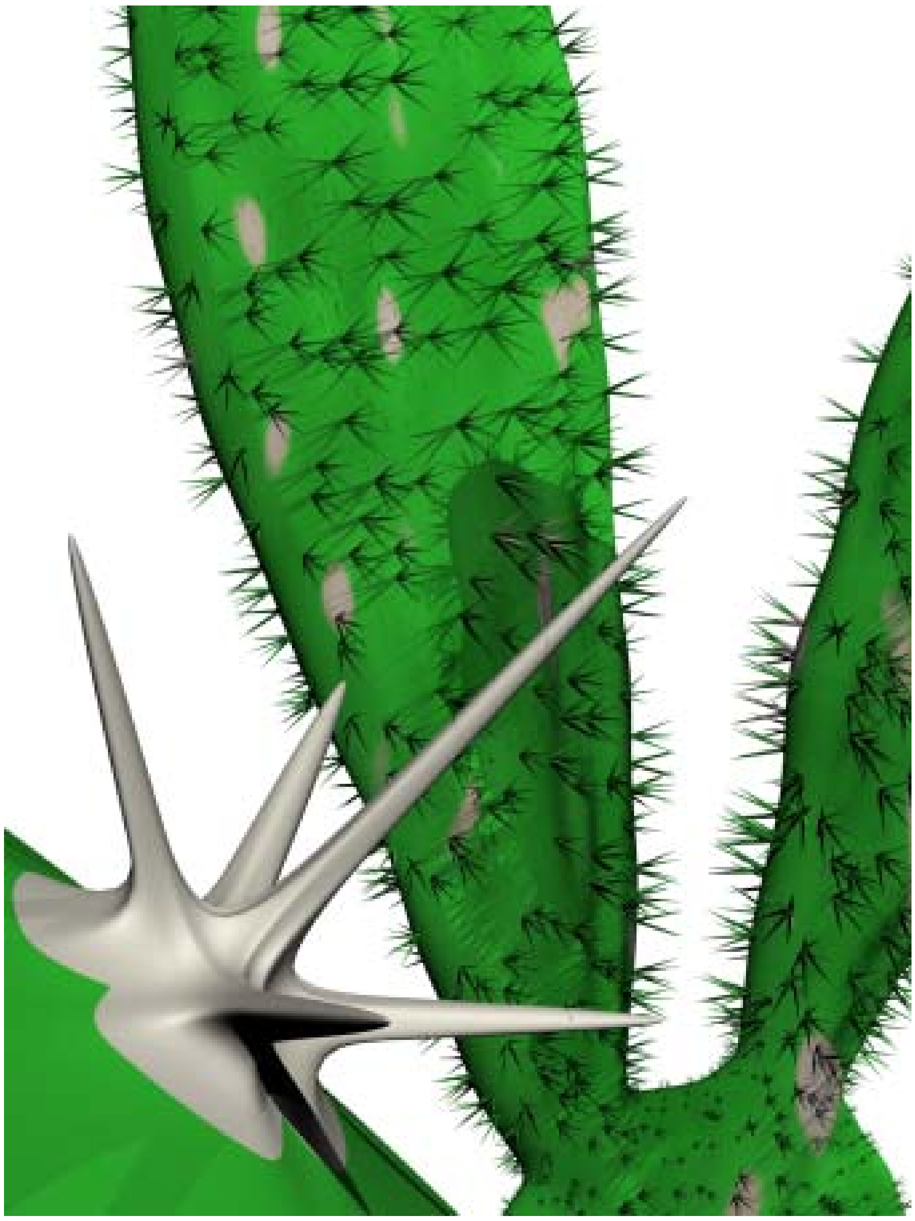


Figure 7.2: Levels of detail. The camera has been placed directly in front of a prickle which as a result is shown in very high detail. The same geometry has been used for the large number of prickles visible in the distance, but a much smaller number of subdivision steps has been applied. Individual prickles have been attached to the main cactus body by means of rule-based mesh growing and *auto attach* operators. The distribution of prickles over the surface has been procedurally defined.

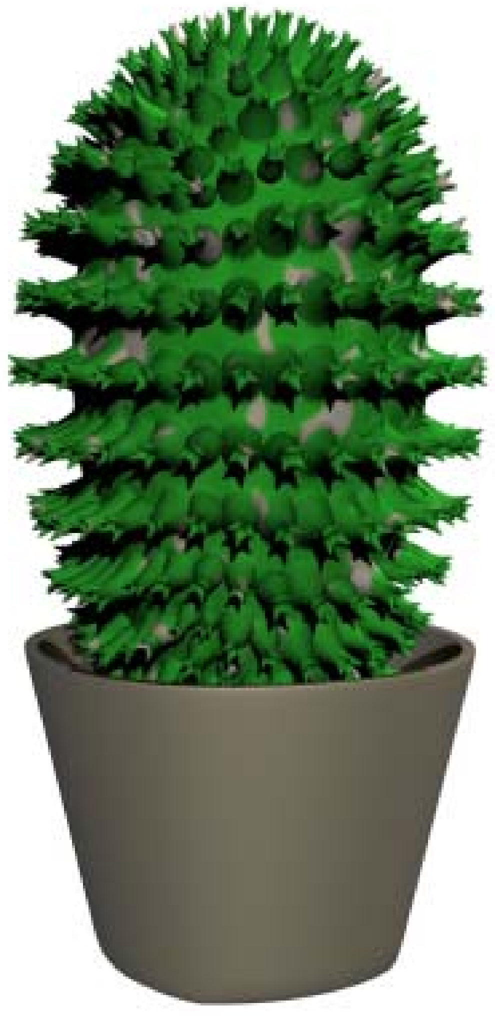


Figure 7.3: Another cactus model. After the global shape has been sufficiently subdivided using Catmull-Clark subdivision rules, a rule-based mesh growing step is performed which attaches a number of small features to the surface. Note that no individual transformations for the attached geometry had to be defined, since the *auto attach* operator automatically creates suitable transformations.

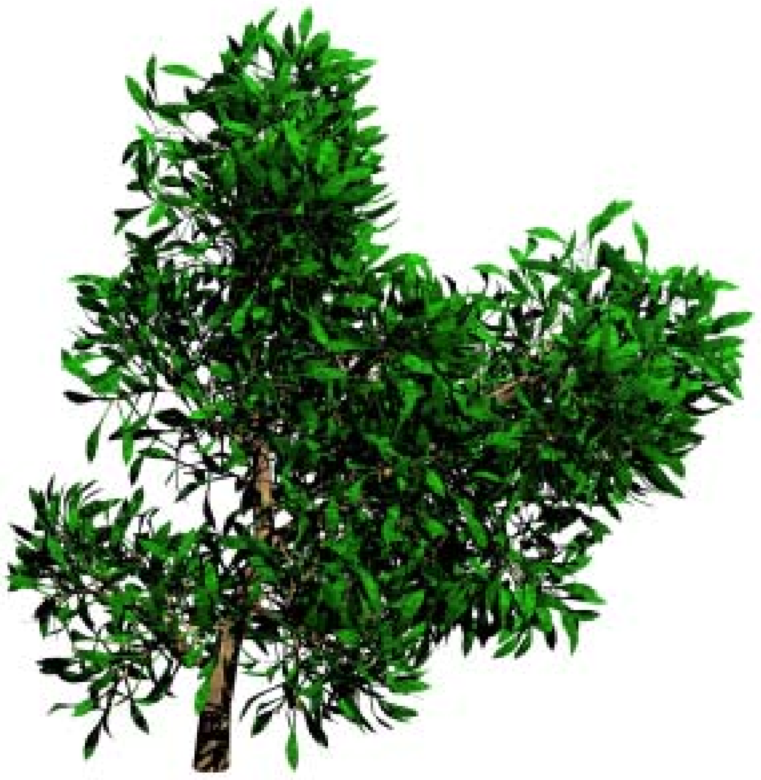


Figure 7.4: The model of this bush-like plant has been created by rule-based mesh growing and subsequent Catmull-Clark subdivision steps. The ruleset has been inspired by the Weber and Penn plant model.

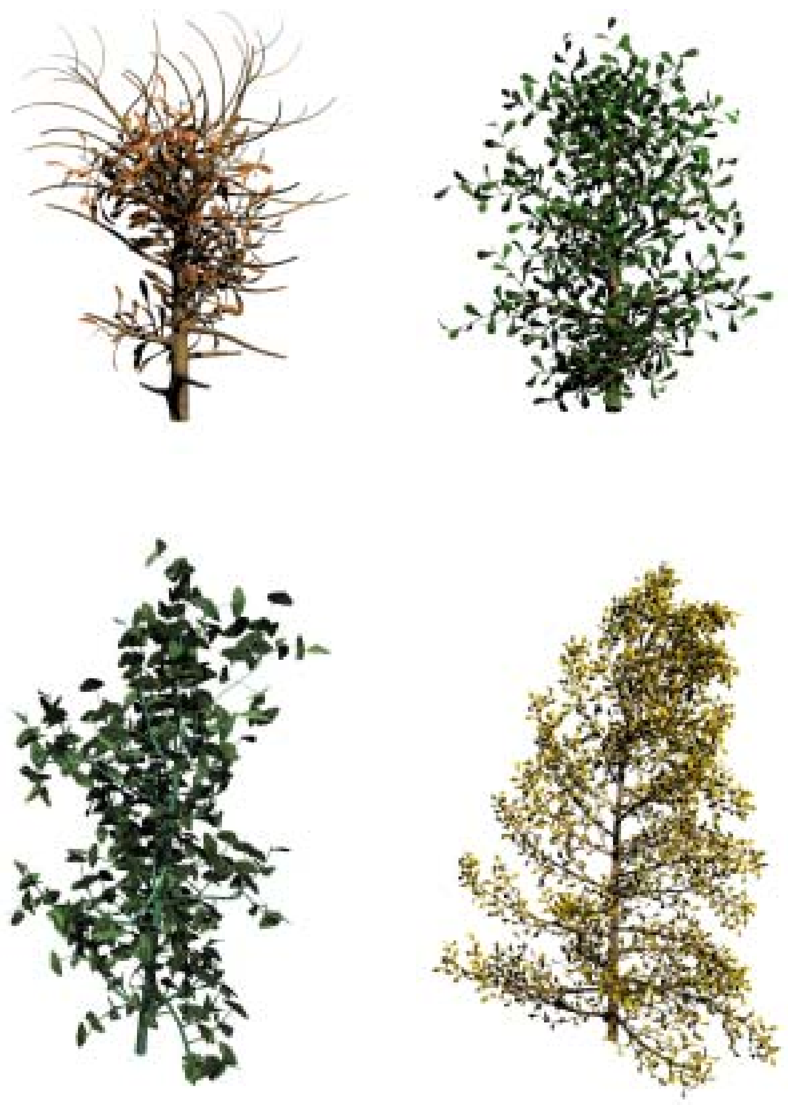


Figure 7.5: A number of different trees which have been created by rule-based mesh growing and subsequent Catmull-Clark subdivision steps. The ruleset has been inspired by the Weber and Penn plant model.

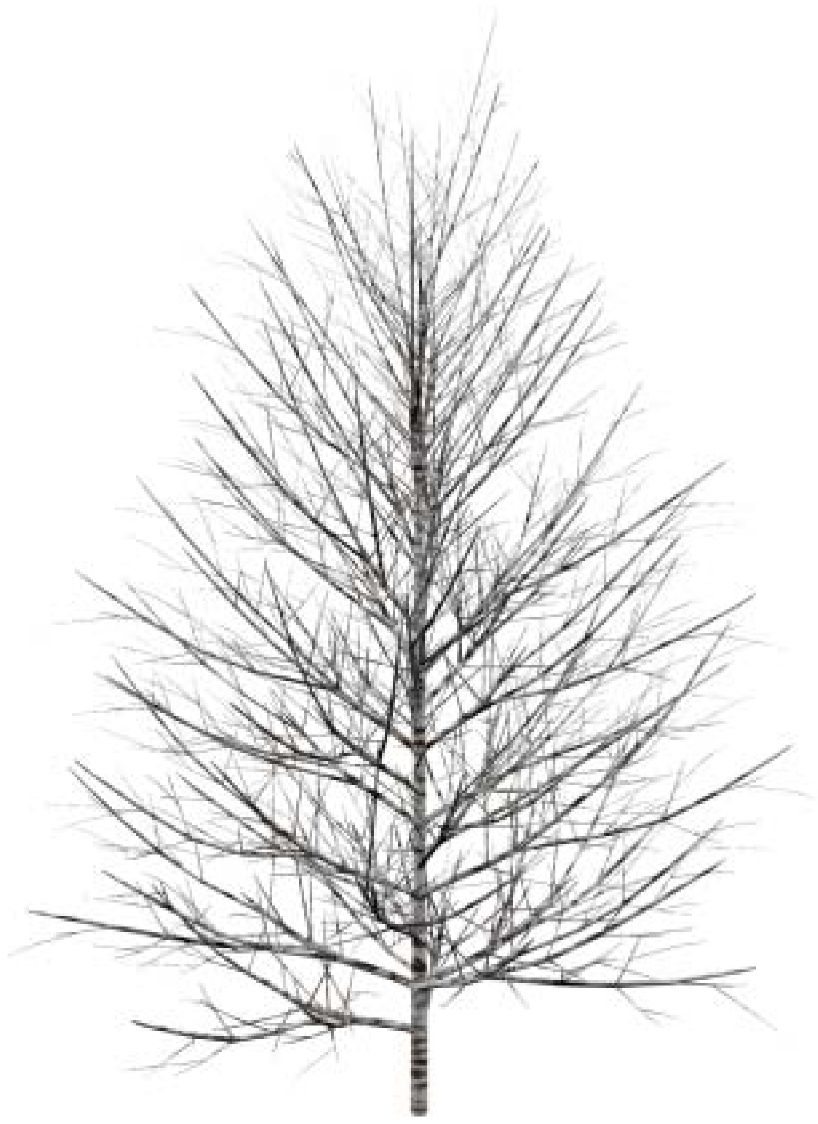


Figure 7.6: This model of a birch has been created using our prototype implementation of a plant editor.

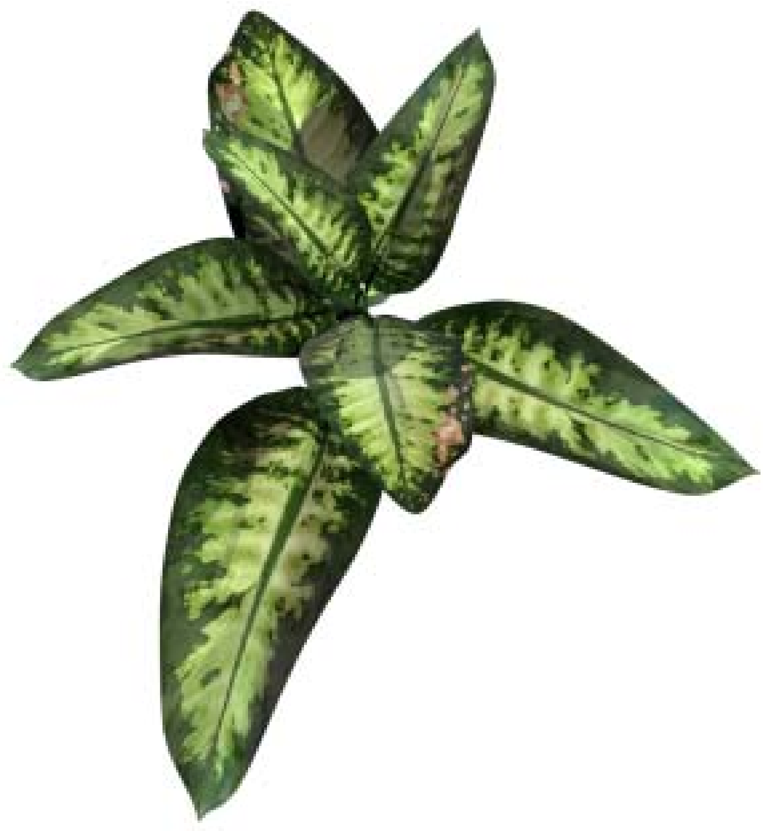


Figure 7.7: Ficus Elastica. The model has been created using our prototype implementation of a plant editor. Leaf textures have been automatically extracted from a photograph of this plant. The base mesh consists of only 273 faces. This crude approximation is smoothed using Catmull-Clark subdivision.

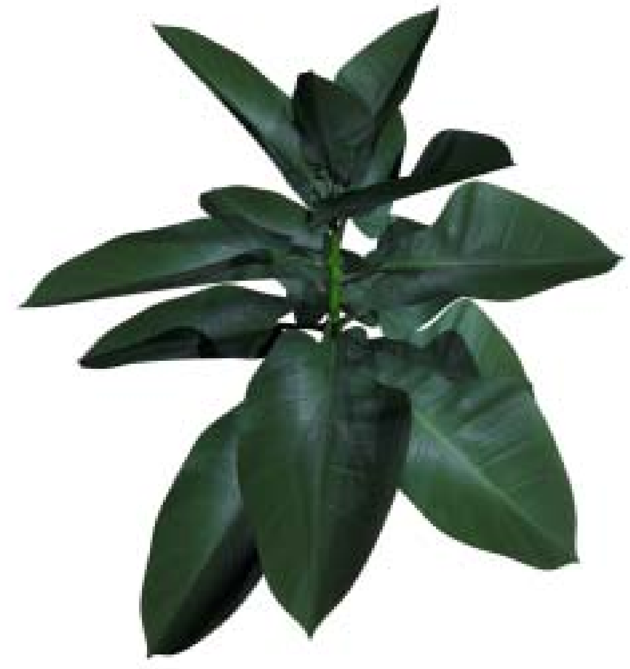


Figure 7.8: Rubbertree. The model has been created using our prototype implementation of a plant editor. Leaf textures have been automatically extracted from a photograph of this plant. The base mesh consists of only 681 faces. This crude approximation is smoothed using Catmull-Clark subdivision.

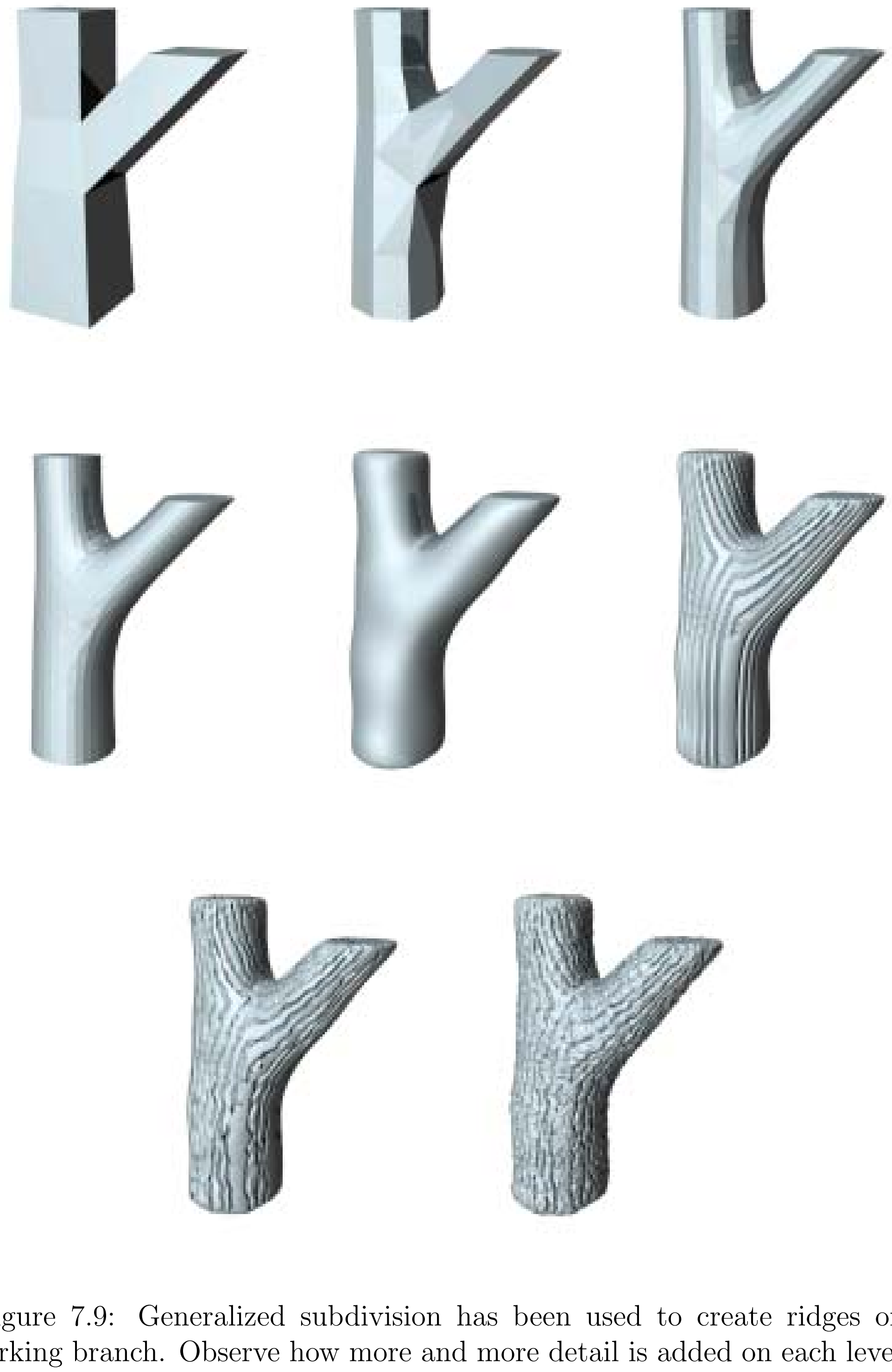


Figure 7.9: Generalized subdivision has been used to create ridges on a forking branch. Observe how more and more detail is added on each level of generalized subdivision.

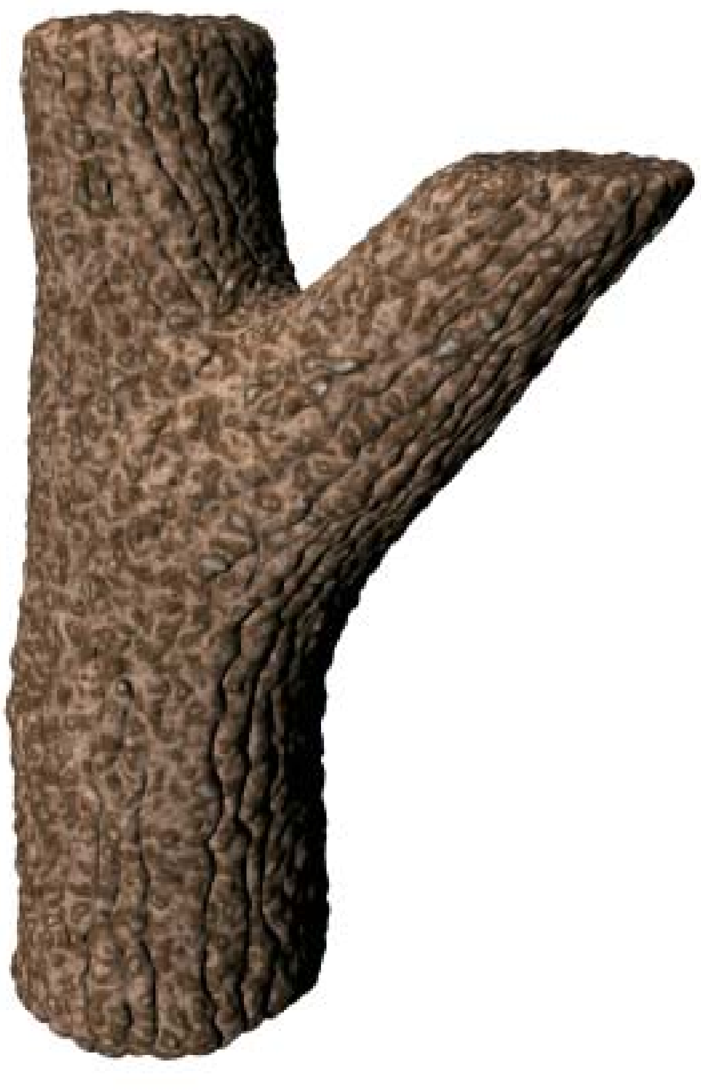


Figure 7.10: Ridges on a forking branch.

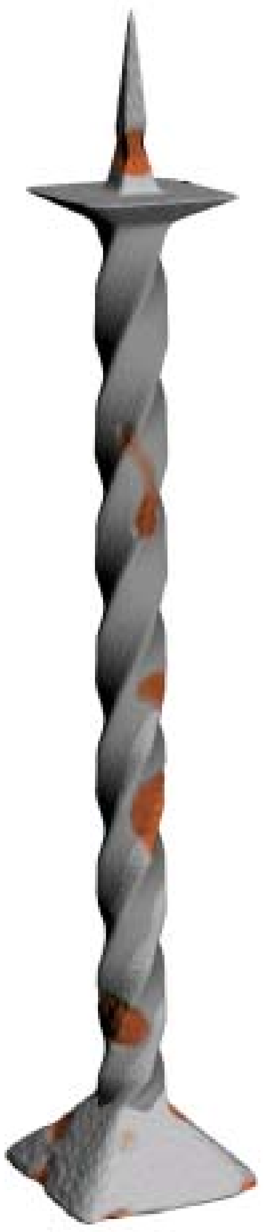


Figure 7.11: Wrought iron candle holder with rust stains. The basic geometry has been created using rule-based mesh growing. At the same time, specific vertex placement operators where used to create the deformations caused by the rust stains. The position and shape of the rust stains was defined procedurally. The scene description file has only about 2.5kb (including comments, surface-, material-, light-, and camera definitions).

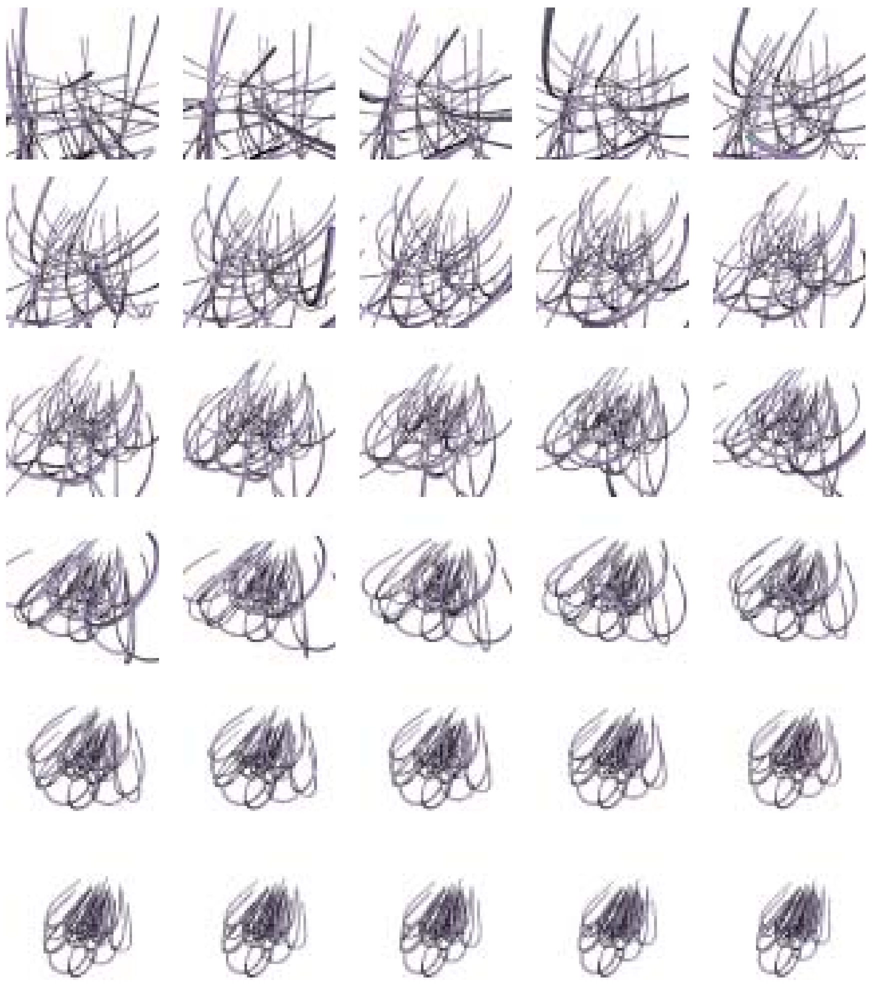


Figure 7.12: Tropism attach. A three-dimensional grid is created by rule-based mesh growing. Each section is attached using a *tropism attach* operator. Finally, a series of images has been rendered using higher and higher values for the tropism strength.

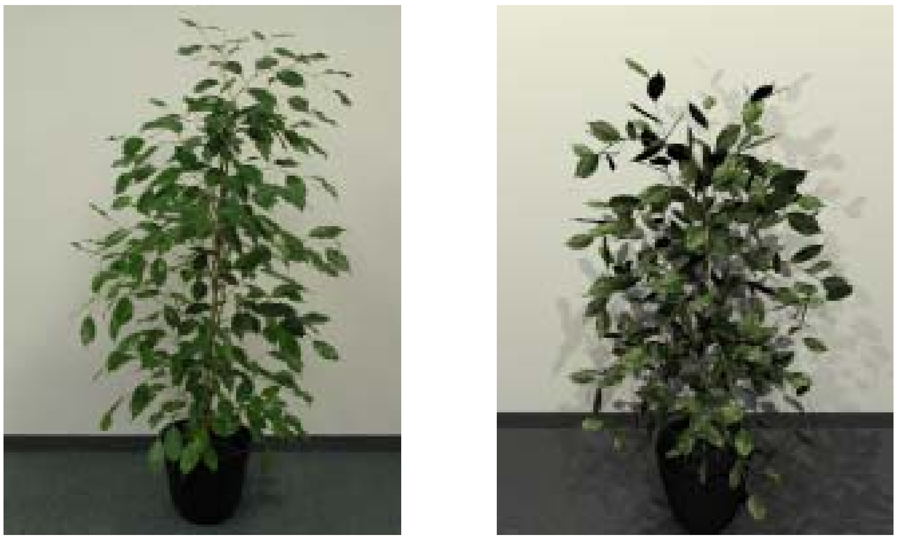


Figure 7.13: Ficus (photograph and rendered image). The model has been created by using our approach for semi-automatic parameter extraction from photographs.

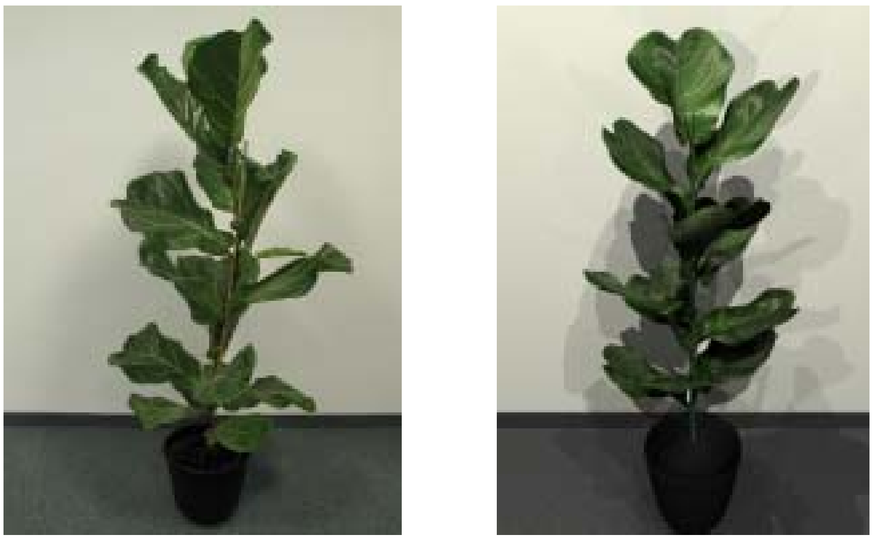


Figure 7.14: Ficus lyrata (photograph and rendered image). The model has been created by using our approach for semi-automatic parameter extraction from photographs.

# Chapter 8

# Conclusion and Future Work

A general approach for procedural mesh definition has been presented, which combines multiple techniques derived from subdivision surfaces to parametric Lindenmayer systems (pL-systems). More exactly we proposed two different mechanisms for mesh modification: *generalized subdivision meshes* and *rule-based mesh growing*.

In generalized subdivision meshes different subdivision rules can be applied at each level of subdivision. Furthermore we split the standard application of a subdivision operator in two distinct operations: *mesh refinement* and *vertex placement*; mesh refinement is the logical introduction of newly created vertices and the generation of the associated, possible changed, connectivity information, without calculating the actual vertex positions; vertex placement is the process of calculating new vertex positions in a mesh.

Rule-based mesh growing is an extension to pL-systems which works not on single symbols, but on connected symbols, where each symbol is associated with faces in a mesh. Thus the right-hand side of a production rule is a template mesh, which implicitely defines a group of connected symbols. Mesh growing replaces single faces by instances of template meshes according to a given ruleset. We also call this extension to pL-systems *mesh-based pL-systems*.

This combined approach yields exceptional modeling power, that we used to efficiently define highly complex geometry, such as trees and plants. By implementing our proposed framework in a general purpose rendering system using *directed cyclic graphs*, it is now possible to define highly complex scenes using as little as a few kilobytes. Although the actual renderer used for producing the images in this thesis generated the complete meshes at each subdivison level, the proposed approach could be used to generate all necessary geometry on-the-fly. We are currently working on a rendering system for both interactive and realistic rendering, that only generates these parts of the geometry that are visible, and disposes the mesh parts that have already been rendered.

On top of the basic implementation of *generalized subdivision meshes* and *rule-based mesh growing* we implemented a prototype of an interactive plant editor in order to demonstrate the applicability of our approach to real-world applications. Most of the plant models depicted in this thesis have been created with the help of this editor. Although far from complete and more or less intended as a kind of feasibility study, it undoubtedly demonstrates the practical relevance of our approach.

Figure 8.1 gives an overview of our frameworks principal architecture: based on the general-purpose rendering system ART (Advanced Rendering Toolkit) our proposed *generalized mesh subdivision* scheme has been implemented, including a number of mesh refinement and vertex placement operators (e.g. add normal height, add local vector, add global vector, flat quadrilateral subdivision, Catmull-Clark subdivision and Loop-subdivision). Based on this, our proposed *rule-based mesh growing* mechanism has been implemented, which allows the definition of rulesets and template meshes and also provides a number of operators (e.g. attach, auto attach, tropism attach, join and face split).

As a final conclusion – we have created a thorough framework for the modeling and creation of very complex geometry. We have brought together previously separate techniques like subdivision surfaces, procedural generation of geometry, and L-systems – and combined them in a unified framework. One of the most immediate challenges awaiting us, will be to adapt and create techniques for efficient rendering of such complex geometry. Then we will be able to use this as a cornerstone for the creation of much more sophisticated botanical and architectural modeling frameworks.

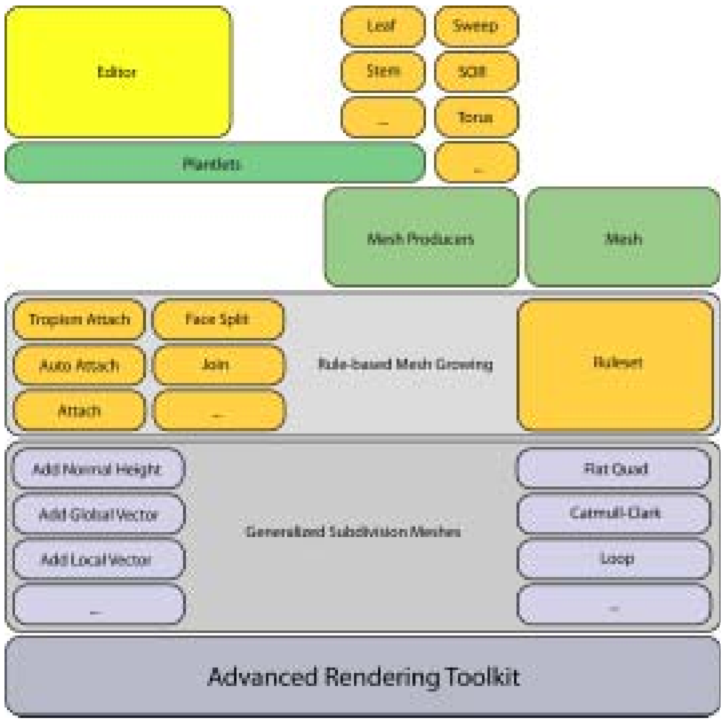


Figure 8.1: Architectural overview - relating generalized subdivision meshes, rule-based mesh growing and the plant editor to the Advanced Rendering Toolkit.

# Chapter 9

# Appendix

## 9.1 Scene Description Example

In the following, an example for a ART scene description file utilizing rule-based mesh growing and generalized mesh subdivision is given. Although the mesh growing and subdivision specific code is rather short, the whole file is quite long. This is for two reasons – extensive comments have been included throughout the file, and a large part of the description is dealing with general definitions for surface materials, lights, camera, and so on. Nevertheless we have included the complete scene description for completeness.

A simple branching, tree-like shape is created by alternately attaching simple meshes representing sections of a stem (red) and a bifurcation (green). A parameter is used to count the number of branching levels, and after a specific limit level has been reached, a mesh which has no more symbols attached to it, is used to terminate the recursive mesh growing process (blue) (see Figure 9.1 for the rendered image).

```
/* ===========================================================================
    Copyright (c) Institute of Computer Graphics and Algorithms,
                  Vienna University of Technology.

    This file is part of the ART libraries. These libraries are free; you can
    redistribute them and/or modify them under the terms of the GNU Library
    General Public License as published by the Free Software Foundation;
    either version 2 of the License, or (at your option) any later version.

    The ART libraries are distributed in the hope that they will be useful,
    but WITHOUT ANY WARRANTY;  without even the implied warranty of FITNESS
    FOR A PARTICULAR PURPOSE or MERCHANTABILITY.  See the GNU Library General
    Public License for more details.

    The precise terms of the license are stated in the file 'LICENSE' which
    is distributed with the ART libraries. If you do not find it, write to:
    Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
   ===========================================================================
    NAME:       MeshGrowingExample
    TYPE:       art source
    PROJECT:    Advanced Rendering Toolkit
    CONTENT:    mesh growing example
   ===========================================================================
    AUTHORS:    sm     Stefan Maierhofer
   ===========================================================================
    HISTORY:

    12-Dec-2001 18:56:00  sm      created
========================================================================== */

ArObj createStemMesh();
ArObj createForkMesh();
ArObj createTerminalMesh();

/* ---------------------------------------------------------------------------
        Define START mesh.
------------------------------------------------------------------------- */
ArObj createGrowingMesh()
{
    /* -----------------------------------------------------------------------
        Vertices.
    --------------------------------------------------------------------- */
    ConstPnt3DArray point_array =
    {
        PNT3D( -2.00, -2.00, 0.00 ),
        PNT3D(  2.00, -2.00, 0.00 ),
        PNT3D(  2.00,  2.00, 0.00 ),
        PNT3D( -2.00,  2.00, 0.00 ),
        PNT3D( -1.00, -1.00, 0.20 ),
        PNT3D(  1.00, -1.00, 0.20 ),
        PNT3D(  1.00,  1.00, 0.20 ),
        PNT3D( -1.00,  1.00, 0.20 ),
        PNT3D_END
    };

    ArObj vertices = VERTEX_SET( point_array, 0, 0, 0, 0 );

    /* -----------------------------------------------------------------------
        Faces.
    --------------------------------------------------------------------- */
    ConstIndexArray point_index_array =
    {
        FACE, 0,3,2,1,
        FACE, 0,1,5,4,
        FACE, 1,2,6,5,
        FACE, 2,3,7,6,
        FACE, 3,0,4,7,
        FACE, 4,5,6,7,
        FACE_END
    };

    /* -----------------------------------------------------------------------
        Assign surface material 0 to all faces.
    --------------------------------------------------------------------- */
    ConstIndexArray sheet_index_array =
        { 0, 0, 0, 0, 0, 0, INDEX_END };

    /* -----------------------------------------------------------------------
        Assign "stem" rule to top face.
    --------------------------------------------------------------------- */
```

```
ConstIndexArray expansion_index_array =
    { 0, 0, 0, 0, 0, arsymbol("stem"), INDEX_END };

/* ----------------------------------------------------------------------
    THE ACTUAL MESH GROWING AND SUBDIV DESCRIPTION STARTS HERE !!!
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
    Init variables first.
---------------------------------------------------------------------- */
ArObj init = VARIABLES("level = 0;");

/* ----------------------------------------------------------------------
    Define RULESET.
---------------------------------------------------------------------- */
ArnNamedNodeSet * rules =
    RULESET(
        /* --------------------------------------------------------------
            Replace "stem" faces by stem mesh.
        -------------------------------------------------------------- */
        "stem",
            AUTO_ATTACH(createStemMesh()),

        /* --------------------------------------------------------------
            Replace "fork" faces by either a fork mesh or apply
            rule "terminal" depending on recursion level.
        -------------------------------------------------------------- */
        "fork",
            [
                VARIABLES("level += 1;")
                inScope:
                NODE_SELECTION(
                    VALUE("(level < 7) ? 0 : 1"),
                    AUTO_ATTACH(createForkMesh()),
                    NODE_REFERENCE("terminal"),
                    0
                )
            ],

        /* --------------------------------------------------------------
            Terminal mesh - has no symbols assigned to its faces.
        -------------------------------------------------------------- */
        "terminal",
            AUTO_ATTACH(createTerminalMesh()),

        0
    );

/* ----------------------------------------------------------------------
    First, apply mesh growing ruleset to this mesh.
    Second, perform 2 levels of Catmull-Clark subdivision.
---------------------------------------------------------------------- */
ArMeshOperatorSequence sequence =
    MESH_OPERATORS(
        FACE_OPERATORS(rules), 1,
        SUBDIV_CATMULL_CLARK, 2,
    0);

/* ----------------------------------------------------------------------
    HERE THE MESH GROWING AND SUBDIV DESCRIPTION ENDS !!!

    All the rest is just standard scene description code.
    ------------------------------------------------------------------------ */

    ArObj faces =
        MESH(
            SOLID_SHAPE,
            PER_FACE_SHEET | PER_FACE_EXPANSIONS | PER_VERTEX_NORMALS,
            point_index_array,
            sheet_index_array,
            0, 0, 0, 0,
            expansion_index_array,
            sequence
        );

    ArObj mesh = [ faces apply : vertices ];

    return [init inScope: mesh];
}

/* ---------------------------------------------------------------------------
    Creates mesh growing object.
---------------------------------------------------------------------------- */
ArObj create_Object()
{
    /* -----------------------------------------------------------------------
        Specify a table of surfaces. Each face will be mapped to a specific
        surface depending on its sheet number.
    ------------------------------------------------------------------------ */
    ArObj meshSurface =
        SELECTED_SURFACE(
            HIT_SHEET_INDEX,
            SURFACE_TABLE(
                INTERPOLATED_NORMAL_SURFACE(    //  sheet 0
                    LAMBERT_REFLECTOR( COLOUR_RGB(0.6, 0.6, 0.6) ) ),
                INTERPOLATED_NORMAL_SURFACE(    //  sheet 1
                    LAMBERT_REFLECTOR( COLOUR_RGB(1.0, 0.0, 0.0) ) ),
                INTERPOLATED_NORMAL_SURFACE(    //  sheet 2
                    LAMBERT_REFLECTOR( COLOUR_RGB(0.0, 1.0, 0.0) ) ),
                INTERPOLATED_NORMAL_SURFACE(    //  sheet 3
                    LAMBERT_REFLECTOR( COLOUR_RGB(0.0, 0.0, 1.0) ) ),
                0)
            );

    /* -----------------------------------------------------------------------
        Create the START mesh and apply the surface defined above.
    ------------------------------------------------------------------------ */
    ArObj mesh = [createGrowingMesh() apply : meshSurface ];

    /* -----------------------------------------------------------------------
        Create a white background.
    ------------------------------------------------------------------------ */
    ArObj sky =
    [
        INFINITE_SPHERE apply: LAMBERT_EMITTER(COLOUR_RGB(1.0,1.0,1.0), 1.0 )
    ];

    /* -----------------------------------------------------------------------
        Create a number of lights.
    ------------------------------------------------------------------------ */
    ArObj lights =
        UNION
        (
            SPHERE_LIGHT(PNT3D( 150, -300,   50), 0.1, COLOUR_GRAY(0.5) ),
            SPHERE_LIGHT(PNT3D(-300, -300,  100), 0.1, COLOUR_GRAY(0.5) ),
            SPHERE_LIGHT(PNT3D( 300, -300,  200), 0.1, COLOUR_GRAY(0.5) ),
            SPHERE_LIGHT(PNT3D( 300,  300,  300), 0.1, COLOUR_GRAY(0.5) ),
            SPHERE_LIGHT(PNT3D(-300,  300,  400), 0.1, COLOUR_GRAY(0.5) ),
            0
        );

    return [ mesh or: sky or: lights ];
}

/* ---------------------------------------------------------------------------
    Creates camera.
--------------------------------------------------------------------------- */
ArObj create_Camera()
{
    return
        [ CAMERA
            imageSize:  IVEC2D( 320, 200 )
            ray:        RAY3D( PNT3D(  15.0, -30.0,  10.0 ),
                               VEC3D( -15.0,  30.0,  -2.5 ) )
            zoom:       1.0
        ];
}

/* ---------------------------------------------------------------------------
    Creates scene.
--------------------------------------------------------------------------- */
ArObj create_MeshGrowingExample()
{
    return
        [ SCENE
            object:     create_Object()
            camera:     create_Camera()
            renderer:   STANDARD_MIDPOINT_RAYTRACER
        ];
}

/* ---------------------------------------------------------------------------
    Create stem mesh.
--------------------------------------------------------------------------- */
ArObj createStemMesh()
{
    ConstPnt3DArray point_array =
    {
        PNT3D( -1.00, -1.00, 0.00 ),
        PNT3D(  1.00, -1.00, 0.00 ),
        PNT3D(  1.00,  1.00, 0.00 ),
        PNT3D( -1.00,  1.00, 0.00 ),
        PNT3D( -1.00, -0.80, 4.00 ),
        PNT3D(  1.00, -0.80, 4.00 ),
        PNT3D(  1.00,  0.80, 4.00 ),
        PNT3D( -1.00,  0.80, 4.00 ),
        PNT3D_END
    };

    ArObj vertices = VERTEX_SET( point_array, 0, 0, 0, 0 );

    ConstIndexArray point_index_array =
    {
        FACE, 0,3,2,1,
        FACE, 0,1,5,4,
        FACE, 1,2,6,5,
        FACE, 2,3,7,6,
        FACE, 3,0,4,7,
        FACE, 4,5,6,7,
        FACE_END
    };

    /* ----------------------------------------------------------------------
        Assign surface material 1 to all faces.
    ---------------------------------------------------------------------- */
    ConstIndexArray sheet_index_array =
        { 1, 1, 1, 1, 1, 1, INDEX_END };

    /* ----------------------------------------------------------------------
        Assign "fork" rule to top face.
    ---------------------------------------------------------------------- */
    ConstIndexArray expansion_index_array =
        { 0, 0, 0, 0, 0, arsymbol("fork"), INDEX_END };

    ArObj faces =
        MESH(
            SOLID_SHAPE,
            PER_FACE_SHEETS | PER_FACE_EXPANSIONS | PER_VERTEX_NORMALS,
            point_index_array,
            sheet_index_array,
            0, 0, 0, 0,
            expansion_index_array,
            NO_MESH_OPERATORS
        );

    ArObj mesh = [ faces apply  : vertices ];

    return mesh;
}

/* --------------------------------------------------------------------------
    Create fork mesh.
-------------------------------------------------------------------------- */
ArObj createForkMesh()
{
    ConstPnt3DArray point_array =
    {
        PNT3D( -1.00, -1.00, 0.00 ),
        PNT3D(  1.00, -1.00, 0.00 ),
        PNT3D(  1.00,  1.00, 0.00 ),
        PNT3D( -1.00,  1.00, 0.00 ),
        PNT3D( -0.05, -1.00, 1.50 ),
        PNT3D(  0.05, -1.00, 1.50 ),
        PNT3D(  0.05,  1.00, 1.50 ),
        PNT3D( -0.05,  1.00, 1.50 ),
        PNT3D_END
    };

    ArObj vertices = VERTEX_SET( point_array, 0, 0, 0, 0 );

    ConstIndexArray point_index_array =
    {
        FACE, 0,3,2,1,
        FACE, 0,1,5,4,
        FACE, 1,2,6,5,
        FACE, 2,3,7,6,
        FACE, 3,0,4,7,
        FACE, 4,5,6,7,
        FACE_END
    };

    /* ----------------------------------------------------------------------
        Assign surface material 2 to all faces.
    ---------------------------------------------------------------------- */
    ConstIndexArray sheet_index_array =
        { 2, 2, 2, 2, 2, 2, INDEX_END };

    /* ----------------------------------------------------------------------
        Assign "stem" rule to the left and right faces
    ---------------------------------------------------------------------- */
    ConstIndexArray expansion_index_array =
        { 0, 0, arsymbol("stem"), 0, arsymbol("stem"), 0, INDEX_END };

    ArObj faces =
        MESH(
            SOLID_SHAPE,
            PER_FACE_SHEETS | PER_FACE_EXPANSIONS | PER_VERTEX_NORMALS,
            point_index_array,
            sheet_index_array,
            0, 0, 0, 0,
            expansion_index_array,
            NO_MESH_OPERATORS
        );

    ArObj mesh = [ faces apply  : vertices ];

    return mesh;
}

/* --------------------------------------------------------------------------
    Create terminal mesh.
-------------------------------------------------------------------------- */
ArObj createTerminalMesh()
{
    ConstPnt3DArray point_array =
    {
        PNT3D( -1.00, -1.00, 0.00 ),
        PNT3D(  1.00, -1.00, 0.00 ),
        PNT3D(  1.00,  1.00, 0.00 ),
        PNT3D( -1.00,  1.00, 0.00 ),
        PNT3D( -0.10, -0.10, 2.00 ),
        PNT3D(  0.10, -0.10, 2.00 ),
        PNT3D(  0.10,  0.10, 2.00 ),
        PNT3D( -0.10,  0.10, 2.00 ),
        PNT3D_END
    };

    ArObj vertices = VERTEX_SET( point_array, 0, 0, 0, 0 );

    ConstIndexArray point_index_array =
    {
        FACE, 0,3,2,1,
        FACE, 0,1,5,4,
        FACE, 1,2,6,5,
        FACE, 2,3,7,6,
        FACE, 3,0,4,7,
        FACE, 4,5,6,7,
        FACE_END
    };
    /* ---------------------------------------------------------------------
        Assign surface material 3 to all faces.
    --------------------------------------------------------------------- */
    ConstIndexArray sheet_index_array =
        { 3, 3, 3, 3, 3, 3, INDEX_END };

    ArObj faces =
        MESH(
            SOLID_SHAPE,
            PER_FACE_SHEETS,
            point_index_array,
            sheet_index_array,
            0, 0, 0, 0, 0,
            NO_MESH_OPERATORS
        );

    ArObj mesh = [ faces apply  : vertices ];

    return mesh;
}
```

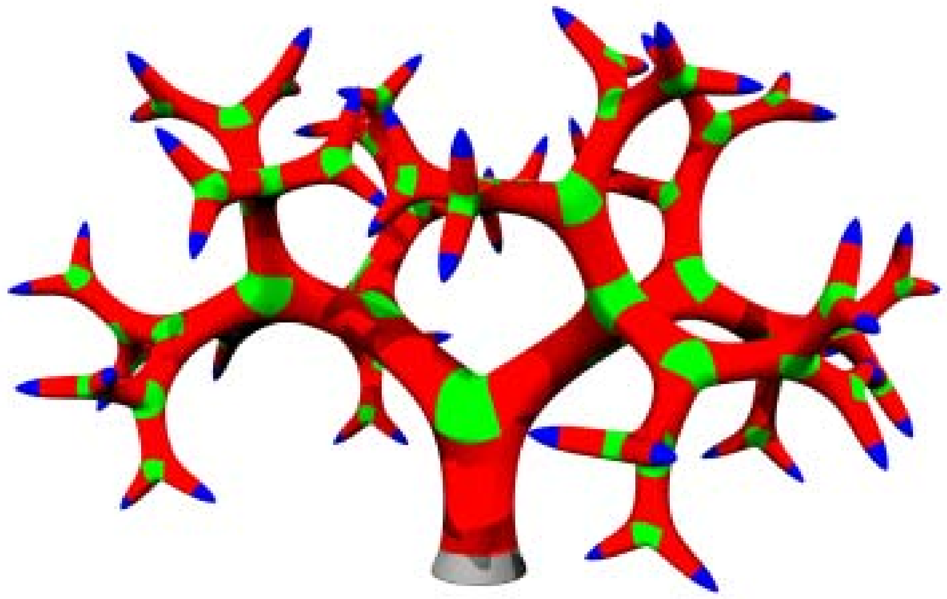


Figure 9.1: Example for mesh growing and subsequent Catmull-Clark subdivision.

## 9.2 Auto-Attach

Given two polygons $A = [\boldsymbol{v}_i^A]$ and $B = [\boldsymbol{v}_j^B]$ with $\boldsymbol{v}_i^A \in \mathbb{R}^3$, $\boldsymbol{v}_j^B \in \mathbb{R}^3$ and $0 \leq i < n$, $0 \leq j < n$ and surface normals denoted as $\vec{n}_{\ldots}$.

Find a transformation $\boldsymbol{M}$ with $B^{\otimes} = \boldsymbol{M} \times B$ such that

$$-\frac{\vec{n}_A}{|\vec{n}_A|} = \frac{\vec{n}_{B^{\otimes}}}{|\vec{n}_{B^{\otimes}}|}$$

and the sum

$$\sum_{0}^{n-1} \left|\boldsymbol{v}_n^{B^{\otimes}} - \boldsymbol{v}_n^A\right|^2$$

is minimized.

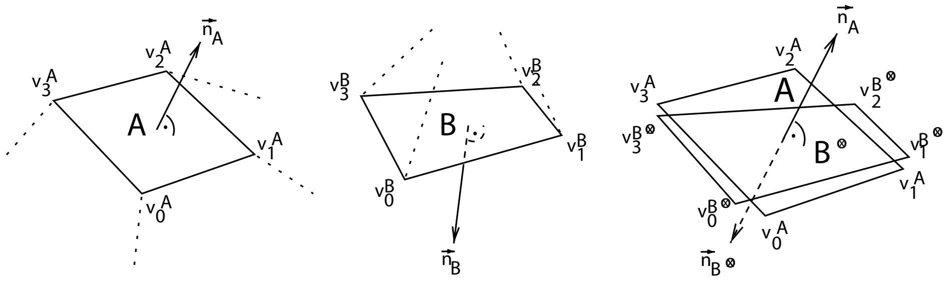


Figure 9.2: Polygons $A$ is the current expansion face and polygon $B$ is the connecting face of the respective expansion mesh. In order to connect both meshes by joining faces $A$ and $B$, it is necessary to transform the expansion mesh so that the transformed polygon $B^{\otimes}$ is in parallel to face $A$ and the summed distance of respective vertices in both poylgons is minimized. An Auto-Attach node creates such a transformation.

## 9.3 Join Algorithm

```
DEFINE JoinReference
{
    faceref
    joinid
}

JoinReference joinfaces[] = EMPTY_SET
```

```
TRAVERSE node IN CSG
{
    IF (node INSTANCEOF JoinOperator)
    {
        ref = NEW JoinReference
        ref->faceref = node->evaluateFaceRef()
        ref->joinid = node->evaluateJoinId()
        ADD ref TO joinfaces
    }
}

FOR EACH ref IN joinfaces
{
    ref2 = NEAREST_NEIGHBOUR OF ref IN joinfaces
             WITH ref->joinid
    IF (ref2 != NULL)
    {
        CONNECT FACES ref->faceref, ref2->faceref
        REMOVE ref, ref2 FROM joinfaces
    }
    ELSE
    {
        REMOVE ref FROM joinfaces
    }
}
```

# Bibliography

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition.* Addison Wesley, June 2000.

[Bau72] Bruce G. Baumgart. Winged Edge Polyhedron Representation. Technical report, Stanford University, Palo Alto, CA, 1972.

[Bau75] Bruce G. Baumgart. A Polyhedron Representation for Computer Vision. In *Proceedings of the National Computer Conference*, pages 589–596, 1975.

[Blo85] Jules Bloomenthal. Modeling the Mighty Maple. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 305–311, 1985.

[BLZ00] Henning Biermann, Adi Levin, and Denis Zorin. Piecewise Smooth Subdivision Surfaces with Normal Control. *Proceedings of SIGGRAPH 2000*, pages 113–120, 2000.

[CC78] Edwin E. Catmull and James Clark. Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes. In *Computer Aided Design*, volume 10, pages 350–355, 1978.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin E. Catmull. The REYES Image Rendering Architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[CDM91] A. Cavaretta, W. Dahmen, and C. Micchelli. Subdivision for Computer Aided Geometric Design. *Memoirs Amer. Math. Soc.*, 93, 1991.

[CT84] Robert L. Cook and S. Trees. Shade Trees. In *Computer Graphics*, volume 18(3), pages 223–231, 1984.

[DHL$^+$98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic Modeling and Rendering of Plant Ecosystems. *Computer Graphics*, 32(Annual Conference Series):275–286, 1998.

[DHR$^+$99] Oliver Deussen, Jörg Hamel, Andreas Raab, Stefan Schlechtweg, and Thomas Strothotte. An Illustration Technique Using Hardware-based Intersections and Skeletons. In *Graphics Interface*, pages 175–182, 1999.

[DKT98] Tony DeRose, Michael Kass, and Tien Truong. Subdivision Surfaces in Character Animation. *Proceedings of SIGGRAPH 98*, pages 85–94, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[DLG90] Nira Dyn, David Levin, and John A. Gregory. A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control. *ACM Transactions on Graphics*, 9(2):160–169, 1990.

[Dre01] DreamWorks. Shrek, http://www.shrek.com, 2001.

[dREF$^+$88] P. de Reffye, C. Edelin, J. Franson, M. Jaeger, and C. Puech. Plant Models Faithful to Botanical Structure and Development. In *Computer Graphics*, volume 22(4), pages 151–158, August 1988.

[DS78] D. Doo and M. Sabin. Analysis of the Behaviour of Recursive Division Surfaces Near Extraordinary Points. In *Computer Aided Design*, volume 10, pages 356–360, 1978.

[DS00] Oliver Deussen and Thomas Strothotte. Computer-generated Pen-and-Ink Illustration of Trees. *Computer Graphics*, pages 13–18, 2000.

[FFC82] A. Fournier, D. Fussell, and L. Carpenter. Computer Rendering of Stochastic Models. *Communications of the ACM*, pages 255–258, June 1982.

[Fou00] Free Software Foundation. GNU Library General Public License, http://www.gnu.org/copyleft/lesser.html, 2000.

[FPB92] Deborah R. Fowler, Przemyslaw Prusinkiewicz, and Johannes Battjes. A Collision-based Model of Spiral Phyllotaxis. *Computer Graphics*, 26(2):361–368, 1992.

[FPdB90] F. David Fracchia, Przemyslaw Prusinkiewicz, and Martin J. M. de Boer. Animation of the Development of Multicellular Structures. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation '90 (Second workshop on Computer Animation)*, pages 3–19. Springer-Verlag, 1990.

[FST92] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Symposium on Interactive 3D Graphics*, pages 11–20, 1992.

[Gar99] Michael Garland. *Quadric-Based Polygonal Surface Simplification.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1999.

[GT95] Michael Gervautz and Christoph Traxler. Representation and Realistic Rendering of Natural Phenomena with Cyclic CSG Graphs. In *Visual Computer*, volume 12, pages 62–74, 1995.

[GVSS00] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schröder. Normal meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 95–102. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[HB97] Donald Hearn and M. Pauline Baker. *Computer Graphics, C Version, Second Edition.* Prentice Hall, 1997.

[Hew97] Terry Hewitt. *The Complete Book of Cacti and Succulents.* DK Publishing, Inc., 1997.

[HHMP96] Mark S. Hammel, James Hanan, Radomír Měch Mech, and Przemyslaw Prusinkiewicz. L-Systems: From the Theory to Visual Models of Plants. *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, 1996.

[Hop98] Hugues H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 35–42, 1998.

[HP96] Mark S. Hammel and Przemyslaw Prusinkiewicz. Visualization of Developmental Processes by Extrusion in Space-Time. *Graphics Interface '96*, pages 246–258, May 1996. ISBN 0-9695338-5-3.

[HPS91] D. Hepting, Przemyslaw Prusinkiewicz, and D. Saupe. Rendering Methods for Iterated Function Systems. *Proceedings IFIP Fractaks '90*, pages 183–224, 1991.

[Joy96] Kenneth I. Joy. Doo-Sabin Surfaces, 1996. On-Line Geometric Modeling Notes, http://graphics.cs.ucdavis.edu/CAGDNotes/CAGD-Notes.html.

[Kaj85] James T. Kajiya. Anisotropic Reflection Models. In *ACM Computer Graphics (SIGGRAPH '85)*, volume 19(3), pages 15–21, July 1985.

[KDS98] Leif Kobbelt, Katja Daubert, and Hans-Peter Seidel. Ray Tracing of Subdivision Surfaces. *Eurographics Rendering Workshop 1998*, pages 69–80, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[Kob96] Leif Kobbelt. Interpolatory Subdivision on Open Quadrilateral Nets With Arbitrary Topology. *Proceedings of Eurographics 96*, pages 409–420, 1996.

[Kob98] Leif Kobbelt. A Subdivision Scheme for Smooth Interpolation of Quad-Mesh Data, 1998. Tutorial, University of Erlangen.

[Kob00] Leif Kobbelt. $\sqrt{3}$ Subdivision. *Proceedings of SIGGRAPH 2000*, pages 103–112, 2000.

[LD96] Bernd Lintermann and Oliver Deussen. *Interactive Modelling and Animation of Branching Botanical Structures.* Eurographics Computer Animation and Simulation EGCAS96. Springer-Verlag, 1996.

[LD97] Bernd Lintermann and Oliver Deussen. Erzeugung komplexer botanischer Objekte in der Computergraphik. *Informatik Spektrum 20/4*, October 1997. Springer Verlag.

[LD98] Bernd Lintermann and Oliver Deussen. A Modelling Method and User Interface for Creating Plants. *Computer Graphics Forum*, 17(1):73–??, 1998.

[LD99] Bernd Lintermann and Oliver Deussen. Interactive Modeling of Plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, January/February 1999.

[Lia99] Sheng Liang. *The Java Native Interface, Programmer's Guide and Specification.* Addison Wesley, June 1999.

[Lin68] A. Lindenmayer. Mathematical Models for Cellular Interaction in Development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[LMH00] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced Subdivision Surfaces. *Proceedings of SIGGRAPH 2000*, pages 85–94, 2000.

[Loo87] Charles Loop. Smooth Subdivision Surfaces Based on Triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[LP89] Aristid Lindenmayer and Przemyslaw Prusinkiewicz. Developmental Models of Multicellular Organisms: A Computer Graphics Perspective. In Christopher G. Langton, editor, *Artificial Life*, pages 221–249. Addison-Wesley, Redwood City, CA, 1989.

[MP96] Radomír Mech and Przemyslaw Prusinkiewicz. Visual Models of Plants Interacting With Their Environment. *Proceedings of SIGGRAPH 96*, pages 397–410, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[PBPS99] Joanna L. Power, A. J. Brush, Przemyslaw Prusinkiewicz, and David H. Salesin. Interactive Arrangement of Botanical L-System Models. *Symposium on Interactive 3D Graphics*, pages 175–182, 1999.

[Pea85] Darwyn R. Peachey. Solid Texturing of Complex Surfaces. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.

[Per85] Ken Perlin. An Image Synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.

[Pet00] Jörg Peters. Patching Catmull-Clark Meshes. *Proceedings of SIGGRAPH 2000*, 25:371–384, 2000.

[PH89] Przemyslaw Prusinkiewicz and J. Haman. *Lindenmayer Systems, Fractals, and Plants.* Springer, Berlin, 1989.

[PH91] Przemyslaw Prusinkiewicz and Mark S. Hammel. Automata, Languages and Iterated Function Systems. *Fractal Models in 3-D Computer Graphics and Imaging*, pages 115–143, 1991. ACM SIGGRAPH '91 (Course Notes).

[PH92] Przemyslaw Prusinkiewicz and Mark S. Hammel. Escape–time Visualization Method for Language–restricted Iterated Function Systems. In *Graphics Interface '92*, pages 213–223, May 1992.

[PH94] Przemyslaw Prusinkiewicz and Mark S. Hammel. Language-restricted Iterated Function Systems. *New Directions for Fractal Modeling in Computer Graphics*, 1994.

[PHHM96] Przemyslaw Prusinkiewicz, Mark S. Hammel, James Hanan, and Radomír Měch Mech. From the Theory to Visual Model of Plants. *Proceeding of the 2 nd CSIRO Symposium on Computational Challenges in Life Sciences*, 1996. Document is available at the internet address http://www.cpsc.ucalgary.ca/Redirect/bmv/papers/l-sys.csiro96.html.

[PHM93] Przemyslaw Prusinkiewicz, Mark S. Hammel, and Eric Mjolsness. Animation of Plant Development. *Proceedings of SIGGRAPH 93*, pages 351–360, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[PHMH91] Przemyslaw Prusinkiewicz, Mark S. Hammel, Radomír Měch Mech, and James Hanan. The Artificial Life of Plants. *SIGGRAPH '95 course notes on Artificial Life*, pages 1–38, 1991.

[Pix00] Pixar. Toy Story 2, http://www.pixar.com, 2000.

[PJM94] Przemyslaw Prusinkiewicz, Mark James, and Radomír Mech. Synthetic Topiary. *Proceedings of SIGGRAPH 94*, pages 351–358, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[PK96] Przemyslaw Prusinkiewicz and Lila Kari. Subapical bracketed L-systems. *Grammars and their Application to Computer Science*, 1073:550–564, 1996.

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

[PLH88] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental Models of Herbaceous Plants for Computer Imagery Purposes. *Computer Graphics*, 22(4):141–150, 1988.

[PRS$^+$94] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human-Computer Interaction.* Addison Wesley, 1994.

[Pru86] Przemyslaw Prusinkiewicz. Graphical Applications of L-Systems. *Proceedings of Graphics Interface '86 — Vision Interface '86*, pages 247–253, 1986.

[Pru93a] Przemyslaw Prusinkiewicz. Modeling and Visualization of Biological Structures. *Proceedings of Graphics Interface*, pages 128–137, May 1993.

[Pru93b] Przemyslaw Prusinkiewicz. Modelling and Visualization of Biological Structures. In *Graphics Interface '93*, pages 128–137, May 1993.

[Pru94] Przemyslaw Prusinkiewicz. Visual Models of Morphogenesis. *Artificial Life*, 1:61–74, 1994.

[PT97] L. Piegl and W. Tiller. *The NURBS Book.* Springer Verlag, New York, 1997.

[Shi00] Jacl Shirazi. *Java Performance Tuning.* O'Reilly, September 2000.

[Shn98] Ben Shneiderman. *Designing the User Interface - Strategies for Effictive Human-Computer Interaction.* Addison Wesley, March 1998.

[Smi84] A. R. Smith. Plants, Fractals and Formal Languages. *Proceedings of SIGGRAPH 1984*, pages 1–10, July 1984.

[SRD00] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java 3D API Specification, Second Edition.* Addison Wesley, June 2000.

[Sta98] Jos Stam. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. *Proceedings of SIGGRAPH 98*, pages 395–404, 1998.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison Wesley, 1997.

[SZSS98] Thomas W. Sederberg, Jianmin Zheng, David Sewell, and Malcolm Sabin. Non-Uniform Recursive Subdivision Surfaces. *Computer Graphics*, 32(Annual Conference Series):387–394, August 1998.

[Tea01] Robert F. Tobler and et al. ART - Advanced Rendering Toolkit, http://www.artoolkit.org, 2001.

[TG95a] Christoph Traxler and Michael Gervautz. Calculation of Tight Bounding Volumes for Cyclic CSG-Graphs. Technical report, Vienna University of Technology, Institute of Computer Graphics, 1995.

[TG95b] Christoph Traxler and Michael Gervautz. Efficient ray tracing of complex natural scenes. Technical report, Vienna University of Technology, Institute of Computer Graphics, 1995.

[TG96] C. Traxler and M. Gervautz. Using Genetic Algorithms to Improve the Visual Quality of Fractal Plants Generated with CSG-PL Systems. *Winter School of Computer Graphics 1996*, February 1996. Held at University of West Bohemia, Plzen, Czech Republic, 12-16 February 1996.

[THP98] Robert F. Tobler, Stefan Hynst, and Werner Purgathofer. Linearly Combining Shading Models for Texturing in Global Illumination Algorithms. In *Proceedings of the Winter School of Computer Graphics*, February 1998. Plzen, Czech Republic.

[TMW01] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. A Multiresolution Mesh Generation Approach for Procedural Definition of Complex Geometry. Technical report, Research Center for Virtual Reality and Visualization, Vienna, 2001.

[Tra97] Christoph Traxler. Modeling and Realistic Rendering of Natural Scenes with Cyclic CSG-Graphs. Master's thesis, University of Technology Vienna, Institute of Computer Graphics, 1997.

[Tra98] Christoph Traxler. Project Page - Representation and Realistic Rendering of Natural Scenes with Directed Cyclic Graphs, http://www.cg.tuwien.ac.at/research/rendering/csg-graphs/index.html, 1998.

[Tri01] Columbia Tristar. Final Fantasy, http://www.finalfantasy.com, 2001.

[Wat93] Alan Watt. *3D Computer Graphics, Second Edition.* Addison-Wesley, 1993.

[Wil01] Alexander Wilkie. *Photon Tracing for Complex Environments.* PhD thesis, University of Technology Vienna, Institute of Computer Graphics, April 2001.

[Wor96] Steven Worley. A Cellular Texture Basis Function. In *Computer Graphics (SIGGRAPH 1996 Proceedings)*, pages 291–294, 1996.

[WP95] Jason Weber and Joseph Penn. Creation and Rendering of Realistic Trees. In Robert Cook, editor, *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 119–128. Addison Wesley, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[WTP01] Alexander Wilkie, Robert F. Tobler, and Werner Purgathofer. Combined Rendering of Polarization and Fluorescence Effects. Technical report, University of Technology Vienna, Institute of Computer Graphics, Vienna, 2001.

[ZSD+99] Denis Zorin, Peter Schröder, Tony DeRose, Jos Stam, Leif Kobbelt, and Joe Warren. Subdivision for Modeling and Animation. In *Computer Graphics (SIGGRAPH '99 Course Notes)*, 1999.

[ZSS96] Denis Zorin, Peter Schröder, and Wim Sweldens. Interpolating Subdivision for Meshes with Arbitrary Topology. *Computer Graphics*, 30(Annual Conference Series):189–192, 1996.