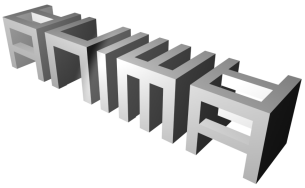

Students' Project ANIMA

Rule Laboratory for Visualisation and Animation

Final Report

Winter 1999/2000 – Summer 2001

Universität Bremen

Fachbereich 3

Final Report of the students' project

ANIMA – Rule Laboratory for Visualisation and Animation

Department of Computer Science, University of Bremen

Winter 1999 — Summer 2001

Contributors:

Lutz Albrecht, Sarah Behrens, Maksym Bortin, Jianfeng Chen, Björn Cordes, Lars Fischer, André René Gefken, Karsten Hölscher, Renate Klempien-Hinrichs, Antonio Krampe, Hans-Jörg Kreowski, Reiner Leins, Thomas Meyer, Thomas Oliver Moll, Michael Prahl, Rafael Trautmann, Carolina von Totth

March 2002, University of Bremen, Germany

Published by Research Group Theoretical Computer Science

For further copies contact:

Hans-Jörg Kreowski
Renate Klempien-Hinrichs

University of Bremen
Department of Computer Science
P.O. Box 330 440
D-28334 Bremen
Germany

{kreo,rena}@informatik.uni-bremen.de

(http://www.informatik.uni-bremen.de/anima)

Preface

This volume presents the final report of the students' project ANIMA that took place from the winter semester 1999/2000 to the summer semester 2001 at the University of Bremen. Such projects are integral parts of the computer science studies, covering about a quarter of the graduate phase. On average, about 15 to 20 students participate in a project. The aim of a project is usually the development of some software system as a case study to learn by research and in team work about the principles of system engineering and project management, the theoretical foundations and some application domain.

ANIMA had 19 participants at the beginning; before the end, four students left for various personal reasons. It was supervised by Frank Drewes in the first quarter, by Renate Klempien-Hinrichs in the last quarter, and by Hans-Jörg Kreowski all the time. ANIMA's vision and mission was to develop a rule laboratory for visualization and animation. The work was divided into three main topics:

- the study of a spectrum of syntactic and rule-based methods for picture and scene generation,
- the sample modelling of visual phenomena and processes of various kinds by means of these methods, and
- the development of AnimaLab as a system to support the modelling and to make the modelled phenomena and processes visible.

Part I of this volume reports on ANIMA's activities in the first two lines of investigation. The chapters of this part are written in a style between seminar notes and scientific papers. Part II is a kind of system handbook that explains the functionality and use of the components of AnimaLab. Moreover, there is a third part which is written in German and is a kind of protocol of the on-goings, decision processes, ups and downs, highlights and difficulties, pros and cons of ANIMA. Although Part III has a single author, it is based on a number of statements provided by most of the participants of ANIMA. This part is concluded by a short note from the supervisor's point of view.

Bremen, March 2002 Students' project ANIMA

Contents

Part II. AnimaLab

Part III. ANIMA-**Projekt**

Part I

ANIMA **Methods**

1 Introduction

Renate Klempien-Hinrichs and Hans-Jörg Kreowski

The aim of ANIMA was to provide an experimental rule laboratory for visualization and animation. The two-year project had three intertwined lines of investigation and development:

1. the conception, specification, and implementation of the system AnimaLab,
2. the study of a spectrum of syntactic and rule-based methods for picture and scene generation,
3. the sample modelling of visual and visualizable phenomena and processes in nature, technology, mathematics, and imagination.

AnimaLab can be used to model, visualize and animate 3D scenes that are composed of a usually large set of basic 3D objects like polyhedra and that are specified by a combination of rule-based methods. The system is open and distributed. It is composed of three types of components: object generators, arrangers, and viewers. An object generator is able to generate 3D objects and sections of scenes by means of a particular rule-based method, an arranger allows one to put various elements of various object generators together into a 3D scene and its development in time, and a viewer displays the scenes. The components communicate via an object network interface. A detailed discussion of AnimaLab can be found in the second part of this report as a kind of handbook introducing those components that were actually developed in the project.

Figure 1. Syntactic picture generation

The first part of this report is devoted to rule-based methods for picture and scene generation and to some examples of visual modelling. A syntactic or rule-based method for picture generation is any concept that provides a way to associate a 2D picture, a 3D scene or an arbitrary geometric object, a set or sequence or stream of such pictures, scenes or objects with a finite description of some kind (see Figure 1). The description may be a Chomsky grammar or an L system, some array or picture grammar, a cellular automaton or some other type of specification. Usually, such a syntactic entity generates a

language or sequence of configurations that are either already pictures, scenes or geometric objects, or can be interpreted as such. In ANIMA, a wide range of such methods was studied:

- chain code picture languages (see, e.g., Maurer, Rozenberg and Welzl [MRW82] and Dassow and Hinz [DH89]),
- turtle picture languages (see, e.g., Prusinkiewicz and Lindenmayer [PL90]),
- cellular automata (see, e.g., Toffoli and Margolus [TM87]),
- iterated function systems and networked iterated function systems (see, e.g., Barnsley [Bar88] and Peitgen, Jürgens and Saupe [PJS92]),
- collage grammars (see, e.g., Habel, Kreowski, and Taubenberger [HKT93] and Drewes and Kreowski [DK99]),
- random context picture grammars (see, e.g., Ewert and van der Walt [EvdW98, EvdW99]),
- two-dimensional languages (see, e.g., Giammarresi and Restivo [GR97]),
- ants and turmites (see, e.g., Langton [Lan86]),
- tilings (see, e.g., Grünbaum and Shephard [GS89]).

Five of these methods are represented and discussed in this report.

3D chain code picture languages are sets of line drawings specified by chain codes. A chain code is a word over the alphabet $\{e, s, w, n, u, d\}$. The drawn picture is obtained by reading such a word from left to right and, continually, drawing a straight line of unit length in the direction given by each symbol read: e for east, s for south, w for west, n for north, u for up, and d for down.

Networked iterated function systems consist of directed graphs the edges of which are labelled with (affine) transformations on some Euclidean space. If each node is associated with a set of points, one gets a new set of points for each node by means of the so-called Hutchinson operator. This operator takes the transformation of each edge, transforms the set of points of the source node pointwise, and adds the result to the new set of points of the target node. Obviously, the Hutchinson operator can be iterated ad infinitum starting from arbitrary sets of points for the nodes of the underlying network.

Collage grammars provide start collages and sets of rules, where a rule consists of a nonterminal label and a collage. A collage consists of a set of parts, i.e. geometric objects, a sequence of pin points, and a set of hyperedges each of which is labelled with a nonterminal and has a sequence of attachment points. Beginning with a start collage, one can derive a collage from a collage by applying a rule. Given a collage with some hyperedge, a rule such that the label of the hyperedge equals the nonterminal of the rule, and an affine transformation transforming the pin points of the collage of the rule into the attachment points of the hyperedge, then the derived collage is obtained by replacing the given hyperedge by the transformed collage of the given rule.

Turmites are Turing machines that work on a three-dimensional cuboid of cells instead of a one-dimensional tape. If the content of each cell is a colour, the cuboid is a 3D scene. A turmite can change the colour of a cell in such a

scene according to the current colour of the cell and its own state, and move to a neighbour cell.

3D tilings are sets of 3D tiles (being sets of points in the three-dimensional Euclidean space) that fill the space without gaps or overlaps. An interesting special case are 3D tilings that are specified by finite sets of prototiles and a finite set of transformations for each prototile.

Networked iterated function systems, collage grammars and 3D tilings are the main topics of three chapters in the first part of this report. The first two are also used as basic methods of object generators described in the second part of the report. The 3D chain code picture languages and the turmites are described in the corresponding chapters on the object generators.

While all the methods are illustrated by typical examples in the respective chapters, there are two separate chapters in the first part of the report that demonstrate the use of collage grammars to model visual phenomena: plants on one hand, and city areas on the other hand. In addition, a mountainous landscape is modelled by the so-called midpoint-displacement method.

Moreover, the first part of the report starts with two chapters on affine transformations. The first chapter is devoted to the mathematical background of 3D graphics. In particular, it recalls how the transformations mainly used in ANIMA, like translation, scaling, rotation, reflection, and shearing, can be handled by means of matrix multiplication. In addition, the lighting of 3D scenes is discussed. The second chapter deals with a particular gap between some of the syntactic methods and open-source graphical interfaces. While the methods employ affine transformations given by quadratic matrices and translation vectors, the interfaces support only special transformations like scaling, turning, reflection and translation. Hence it is necessary to decompose arbitrary matrices into such special transformations.

2 Basics of Three-Dimensional Graphics

Rafael Trautmann and Thomas Meyer

1 Introduction

Within the framework of the students' project ANIMA at the University of Bremen we mainly dealt with the syntactic generation of three-dimensional pictures. These generated objects should of course be displayed so that they can be viewed in an easy and comfortable way.

To be able to work effectively with three-dimensional objects a decent knowledge about three-dimensional computer graphics is needed. The goal of this document is to offer a short compilation of the basics about this topic. It is mainly focused on the parts that are needed for ANIMA, but we hope it is universal enough to be useful even outside of the project.

The following section explains the basics that should be considered common knowledge for someone working with three-dimensional objects and their transformations. This includes building objects and manipulating them with transformations. The last section deals with more complex concepts of three-dimensional computer graphics, i.e. lighting.

As already said, this article is only an introduction into the field of three-dimensional computer graphics. Further informations about this topic can be found in Foley, van Dam, Feiner and Hughes [FvDFH00], Policarpo and Watt [WP99], and for more practical use see Akeley and Segal [SA99].

2 3D Objects and Transformations

In this section, the basic notions and notations of 3D objects and their transformations are recalled.

2.1 Preliminaries

Point

A point is a basic element of an axiomatic geometry. Informally, it is a geometrical element that has no dimensions. In Cartesian space, this is an element that can be located by a single n-tuple of coordinates.

Line

A line is an undefined primitive concept of Euclidean geometry. In Cartesian geometry this is a straight one-dimensional geometrical figure of infinite length and no thickness. Any pair of points uniquely determines a line such that the segment between the given points is the shortest path between those points.

Plane

A plane is a geometrical figure with the property that the line joining any two of its points lies completely on its surface. Informally, a plane is a flat surface.

Normal

In mathematics a vector perpendicular to a plane is called the normal of this plane. Same in 3D graphics, only here a normal is assigned to a (not necessarily coplanar) polygon or vertex. If a polygon has only one normal this normal is called face or surface normal. When each vertex has its own normal, they are called vertex normals.

3D Coordinate Systems

A coordinate system is a system for locating points by their coordinates with respect to some set of reference points, lines, directions, etc. Commonly known is the Cartesian (rectangular) coordinate system, which has a set of mutually perpendicular axes that intersect in the origin.

There are two common orientations for three-dimensional cartesian coordinate systems in computer graphics: the left- and righthanded coordinate systems. In the righthanded coordinate system the z-axis points out of the image, while in the lefthanded it points into the image.

2.2 3D Objects

A *vertex* is a point in three-dimensional space that is used to define a polygon.

A *polygon* is specified by a series of three or more vertices. To avoid problems with the rendering engine, polygons are required to be coplanar. There are specialized kinds of polygons like triangles and quads. A triangle is built of exactly three vertices, a quad of four. Triangles, the most common form of polygons, are easy to create and use because they are always coplanar.

Through composition of two or more polygons, *objects* can be created. A rectangle can be built using two triangles. Such a rectangle can be treated as a side of a box. Six of them can be composed to a box and so on.

Building round objects is a bit more tricky. It is impossible to build a circle with straight lines, but an n-gon (e.g. a hexagon) can be built. If n has

a high enough value, such an n-gon will look like a circle. The same is true in three dimensions. It is possible to build a polyhedron that will look like a sphere.

2.3 Transformations in 3D

Handling Transformations

After the creation of one or more objects it may be useful to be able to change them. This is possible with the aid of transformations and translations. They can be used to move and rotate objects, to create a scene or even to move objects and/or the virtual camera to create an animation. A set of vertices can be transformed into another set of vertices by a linear transformation. Both sets of vertices remain in the same coordinate system. Matrix notation is usually used to describe the linear transformations. The de facto convention in computer graphics is to have the vertex or vector as a column matrix, preceded by the transformation matrix T. Using a matrix notation, a point V is transformed under translation, scaling, and rotation as

$$V' = V + D,\ V' = S \circ V,\ V' = R \circ V$$

where D is a translation vector, S is a scaling, R is a rotation matrix and V' is the resulting point. These three transformations are the three most commonly used ones. Scaling and rotation are matrix multiplications while translation is an addition. It would be much easier to work with transformations if all of them could be treated in the same way, and combined as well. This can be achieved by using the homogeneous coordinates, which increases the dimensionality of the space. In a homogeneous system a vertex $V(x, y, z)$ is represented as $V(x', y', z', w)$ for some scale factor $w \neq 0$. The 3D Cartesian coordinate representation is then:

$$x = x'/w,\ y = y'/w,\ z = z'/w$$

If we consider w = 1 then the matrix representation of a point is

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Then the matrix representations of linear transformations are as follows.

Translation

Translation can now be treated as matrix multiplication like the other two transformations, and becomes $V' = T \circ V$ with

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \circ \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

This specification implies that the object is translated in three dimensions by applying a displacement T_x, T_y and T_z to each vertex that defines the object.

Scaling

Scaling is defined by $V' = S \circ V$ with

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where S_x, S_y and S_z are scaling factors. For uniform scaling the factors are equal, i.e. $S_x = S_y = S_z$. Otherwise scaling occurs along the axes for which the scaling factor is not equal to 1.

Rotation

To rotate an object in 3D space, an axis of rotation must be specified. It may have any spatial orientation, but it is easiest to consider rotations that are parallel to one of the coordinate axes. The transformation matrices for anti-clockwise (looking along the unit vector towards the origin) rotation about the X, Y and Z axes respectively are

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Shearing

Shearing is a transformation in which all the points in one line or plane remain fixed while all other points move parallel to the fixed line or plane. This transformation causes some trouble in computer graphics, because after shearing an object its normalvectors are no longer correct and need to be recalculated. On that score shearing is usually not used in computer graphics. A more detailed explanation of shearing can be found in Section 3.4 of Chapter 3.

Inverse Transformation

Sometimes the inverse of a transformation is needed. Let T be a translation matrix, S a scaling matrix and R a rotation matrix. T^{-1} can be obtained by negating T_x, T_y and T_z. Replacing S_x, S_y and S_z by their reciprocals gives S^{-1}, and negating the angle of rotation gives R^{-1}.

Composition

Any set of rotations, scalings and translations can be composed by matrix multiplication to give a final transformation matrix. For example if

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = M1 \circ \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \text{ and } \begin{pmatrix} x'' \\ y'' \\ z'' \\ 1 \end{pmatrix} = M2 \circ \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

then the transformation matrices can be multiplied: $M3 = M2 \circ M1$, and one gets

$$\begin{pmatrix} x'' \\ y'' \\ z'' \\ 1 \end{pmatrix} = M3 \circ \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Translations and scalings are commutative, but rotations are not, i.e.

$$S1 \circ S2 = S2 \circ S1,\ T1 \circ T2 = T2 \circ T1,\ T1 \circ S1 = S1 \circ T1$$

but

$$R1 \circ R2 \neq R2 \circ R1,$$

in general. An arbitrary composition of translations, scalings and rotations is of the form:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & T_x \\ a_{21} & a_{22} & a_{23} & T_y \\ a_{31} & a_{32} & a_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The 3×3 upper left sub matrix represents the final rotation and scaling while the fourth column gives the final translation. The final matrix can of course be inverted to obtain a matrix which combines all the inverse transformations.

3 Illumination and Shading

When an object (that is, its surface) is shaded, its material characteristics, position and orientation, as well as light sources are taken into account. Illumination models are used to determine a surface colour at a given point. Shading models are used to determine when and on which point an illumination model is applied.

3.1 Illumination Models

Ambient Light

In the possibly simplest illumination model each object is displayed using an intensity intrinsic to it. There is no external light source, no reflection, only self-luminous objects. The illumination equation that expresses this model is

$$I = k_i \tag{1}$$

where I is the resulting illumination and the coefficient k_i is the intrinsic intensity of the object.
If there is instead of self-luminosity a diffuse, nondirectional lightsource which is the result of multiple reflections of light from many surfaces one speaks of *ambient light*. Assuming that ambient light impinges equally on all objects, the illumination equation is

$$I = I_a k_a \tag{2}$$

where I_a is the intensity of the ambient light (constant for all objects) and k_a is the *ambient-reflection-coefficient*, ranging from 0 to 1. The ambient reflection is a material property which does not correspond directly to any physical property of real materials. Ambient light is used to account for all the complex ways in which light can reach an object.

Diffuse Reflection

So far, objects are uniformly illuminated across their surface by ambient light. If there is a *point light source*, the rays of which emanate uniformly in all directions from a single point, the brightness of an object varies from one part to another depending on the direction of and distance to the light source.

Lambertian Reflection Diffuse reflection is also called Lambertian reflection. The brightness of a point on a surface depends on the angle θ between the directions to the light source $\boldsymbol{L}$ and the surface normal $\boldsymbol{N}$ at that point (Figure 1).

So the diffuse illumination equation is

$$I = I_p k_d \cos(\theta). \tag{3}$$

I_p is the intensity of the point light source; the *diffuse-reflection-coefficient* of a material k_d is a constant between 0 and 1 and is a material property like k_a. The angle θ must be between 0 and 90 degrees to have any direct effect on the point, because otherwise the light source is on the other side of the material. Assuming that the vectors $\boldsymbol{N}$ and $\boldsymbol{L}$ are normalized (i.e. they are of unit length), the equation can be rewritten using the dot product:

$$I = I_p k_d (\boldsymbol{N} \cdot \boldsymbol{L}) \tag{4}$$

Adding the ambient light gives the following equation:

$$I = I_a k_a + I_p k_d (\boldsymbol{N} \cdot \boldsymbol{L}) \tag{5}$$

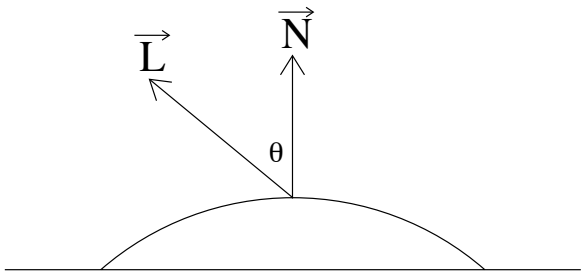

Figure 1. Diffuse reflection

Light-Source Attenuation Since the distance to the light source affects the brightness of a surface, the light-source-attenuation f_{att} is introduced.

$$I = I_a k_a + f_{att} I_p k_d (\boldsymbol{N} \cdot \boldsymbol{L}) \tag{6}$$

where f_{att} is computed in the following way:

$$f_{att} = min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right) \tag{7}$$

where $d_L$is the distance to the light source and c_1, c_2 and c_3 are constants associated with the light source. The constant c_1 keeps the denominator from becoming too small when the light is close, and the expression is clamped to a maximum of 1 to ensure that it always attenuates.

Coloured Lights and Surfaces

Until now the illumination equation can only be used for monochromatic lights. For coloured lights and surfaces one equation is written for each component of the colour model. For example the tuple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green and blue colour component in the RGB system. So the equation for the red component is:

$$I_R = I_{aR} k_a O_{dR} + f_{att} I_{pR} k_d O_{dR} (\boldsymbol{N} \cdot \boldsymbol{L}) \tag{8}$$

The equations for I_G and I_B look similar. In order not to restrict this equation to one colour model, the wavelength-dependent terms of the illumination equation are indicated with a subscript λ. So the equation becomes

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\boldsymbol{N} \cdot \boldsymbol{L}) \tag{9}$$

Specular Reflection

On shiny surfaces (i.e. apple) a highlight can be observed when lit. This highlight has the same colour as the light, while the rest of the surface is

illuminated by its diffuse colour. When moving around an object the highlight also moves. It does so because the shiny surfaces reflect light unequally in different directions. On a perfectly shiny surface, light is reflected only in the direction of reflection $\boldsymbol{R}$, which is $\boldsymbol{L}$ mirrored about $\boldsymbol{N}$. Thus the specularly reflected light can only be seen when the angle α, between $\boldsymbol{R}$ and the direction to the viewport $\boldsymbol{V}$, is 0 (see Figure 2).

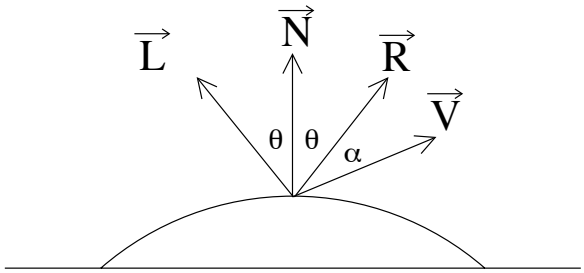

Figure 2. Specular reflection

The Phong Illumination Model This illumination model is for non-perfect reflectors, which assumes that maximum specular reflection occurs when α is 0, and decreases sharply as α increases. This rapid decrease is approximated by $\cos^n \alpha$, where n is the *specular-reflection-exponent* of the material, which typically varies from 1 to several hundred. A value of 1 provides a broad, gentle decrease, whereas higher values simulate a sharp, focused highlight. For a perfect reflector n would be infinite. The amount of incident light reflected specularly depends on the angle of incidence θ. If $W(\theta)$ is the fraction of specularly reflected light and if the direction of reflection $\boldsymbol{R}$ and the the direction to the viewer $\boldsymbol{V}$ are normalized, then $\cos\alpha = \boldsymbol{R} \cdot \boldsymbol{V}$ and the illumination equation is

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} \cos\theta + W(\theta)(\boldsymbol{R} \cdot \boldsymbol{V})^n]. \tag{10}$$

$W(\theta)$ is typically set to a constant k_s, the *specular-reflection-coefficient* of the material, which ranges from 0 to 1. So far the specular component of this illumination model is independent from the material. To change this and give the specular highlight a colour, the equation can be rewritten as follows:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} [k_d O_{d\lambda} \cos\theta + k_s O_{s\lambda} (\boldsymbol{R} \cdot \boldsymbol{V})^n] \tag{11}$$

where $O_{s\lambda}$ is the *specular colour* of the object.

Multiple Light Sources

If there are m light sources, then the terms for each light source are added:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} f_{att_i} I_{p\lambda i} [k_d O_{d\lambda} (\boldsymbol{N} \cdot \boldsymbol{L}_i) + k_s O_{s\lambda} (\boldsymbol{R}_i \cdot \boldsymbol{V})^n] \quad (12)$$

3.2 Shading Models for Polygons

Since computing the normal for each visible point of the surface and applying an illumination model is very expensive, there are several techniques to reduce the costs of shading for polygons. While explaining this, there is one thing to note:

- In most cases triangles, not general polygons, are rendered, because the three points defining a triangle are always coplanar and also define a plane. So every point in this special polygon has the same normal, which is called *surface normal* or *face normal*.

Constant Shading

Constant shading is also known as *flat shading*. When it is applied the entire polygon is shaded with the same intensity. This is valid if one of the following assumptions is true:

- The light source is positioned at infinity, so the dot product of the normal and the (normalized) vector towards the light source is constant across the polygon.
- The viewer is at infinity, so the dot product of the normal and the (normalized) vector towards the viewer is constant across the polygon.
- The polygon represents the actual surface being modelled, and is not an approximation to a curved surface.

Interpolated Shading

Interpolated shading computes the intensity for the vertices and linearly interpolates these across the polygon. These vertices have the same normal (the face normal), but the vectors towards the viewer and the light sources are different.

Gouraud Shading

When using one of the previous shading models there is an intensity discontinuity in the computed image. This is because adjoining polygons may have different normals and so the intensity varies from polygon to polygon. There is a large change in intensity along the border of neighbouring polygons. In

the case of interpolated shading the intensity changes smoothly across the polygon.

Gouraud shading, also called *intensity interpolation shading* or *colour interpolation shading*, extends interpolated shading by interpolating illumination values that take into account the surface being approximated. To do so, the normal for each vertex has to be known. These are called *vertex normals*. A vertex normal can be approximated by averaging the surface normals of all polygons sharing the vertex (see Figure 3).

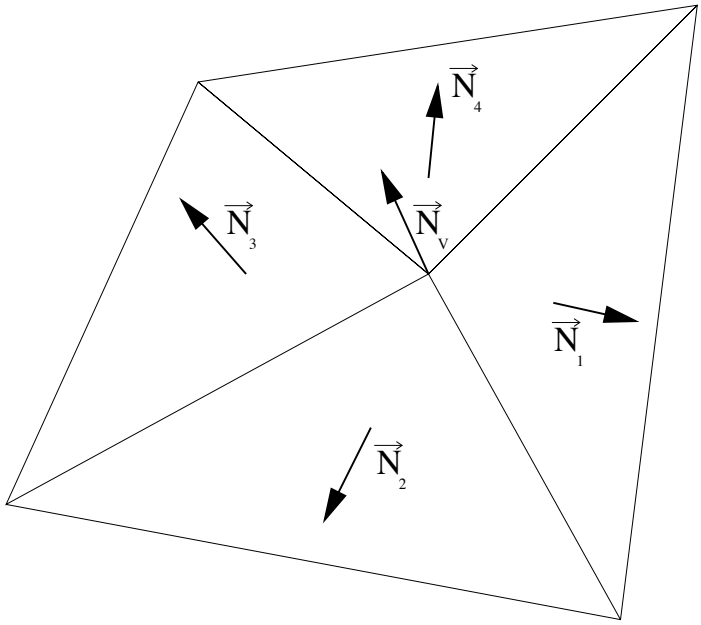

Figure 3. From surface normals to a vertex normal

The *vertex intensities* are then found by applying the illumination model with the computed vertex normals. These intensities are then linearly interpolated across the polygon.

Phong Shading

So far, it is possible to prevent the intensity from making large changes across a polygon and across adjointed polygons. But when a light source (let us call it a spot light) only illuminates a small area, say the center of a polygon, in a way that the border is not illuminated, this is not possible. When looking at Gouraud shading, the intensity at the vertices would be evaluated and then

interpolated across the polygon, but since there is no light at the borders, there is no light in the center.

Phong shading, also known as *normal vector interpolation shading*, extends Gouraud shading by not just interpolating the intensity at the vertices, but by interpolating the normals and applying the illumination model at each point, with the computed normal.

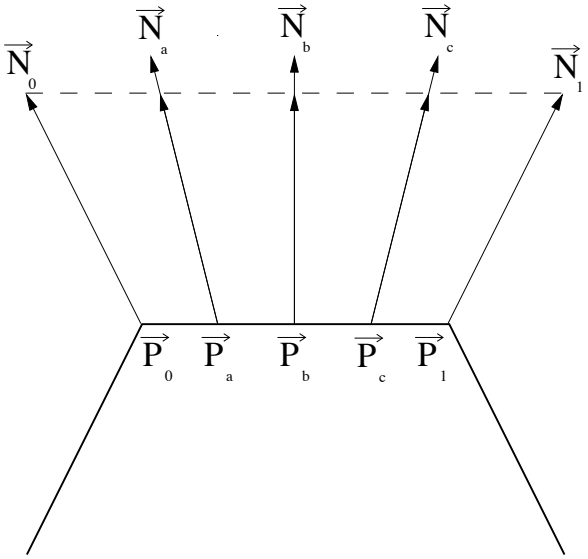

Figure 4. Normal vector interpolation

3 Matrix Decomposition

Karsten Hölscher

1 Introduction

Java3D, the basis of *Collage3D*, uses matrices to represent transformations on the given parts. A part is regarded as the set of its points. Thus a transformation is applied by multiplying all of these points from the left with the corresponding matrix respectively. If there is more than one transformation to be applied, the matrices are combined by matrix multiplication in the same order as the transformations from the left. This composite matrix is then used to store and further represent the transformations done on the part. The information of the actual transformations are lost. There are four primitive affine transformations supported by the underlying collage grammars: translation, scaling, shearing and rotation. To export the geometric data of a generated scene to a file format that does not support matrices, a decomposition of the final transformation matrix into a series of basic transformations yielding the same result is needed. This is subject of the following considerations.

Recovery of primitive transformations from a given composite matrix in a meaningful way is not as easy as it may seem at first glance.

In general, it is impossible to recover the original transformations that lead to a given composite matrix. Just imagine to move an object around in the scene using a sequence of translations (for example move it two units up, one unit down and then four units to the right). It is impossible to find out what translations were applied to the object, since the final position (represented by the composite matrix) could be reached using any of an infinite number of completely different translation sequences. Since the composite matrix just contains informations about the final position, not the movement steps that lead there, we call this effect *absorption*. Fortunately absorption is not important for the Collage3D export since only transformations that yield the same result are of interest. Thus a translation recovered from the composite matrix that moves the object to the same position suffices.

Another problem is the order of the applied transformations. It is impossible to detect for example whether the object was moved and then stretched, or the other way round. This is no real problem, since any order yielding the same result is acceptable. Thus a canonical order can be assumed, like translation of rotation of scaling of a given part.

Most transformations do not operate on isolated partitions of the composite matrix. On the contrary, they change all columns when applied (for example scaling affects translation). This effect can be called *interaction*. Interaction is the main difficulty in any simple approach to decompose a given transformation matrix because every entry is affected by more than one primitive. In fact, all pairs of primitives may interact.

The considerations above clearly show that every simple and naive approach to decompose a matrix will fail in general. Thus another way is needed to achieve that goal. One promising approach is to factor the composite matrix into a rotation matrix and additional ones. This will be discussed in the following section.

2 Decomposition Methods

A transformation matrix in common computer graphic systems is a 4×4 matrix with a perspective partition in the last row and a translation partition in the last column. Since a perspective transformation is not affine the last row can be ignored. The upper three entries of the translation partition represent a translation vector, so the last column can be ignored as well, but it has to be stored as translation vector for later use. Thus only a 3×3 matrix needs to be decomposed.

Before the discussion of different methods is started, a few facts on rotation matrices are recalled.

Let $A \in \mathbb{R}^{m \times m}$ be a rotation matrix. Then A has the following properties:

$$A^T \circ A = I \tag{1}$$

$$|A| = 1 \tag{2}$$

Here $I \in \mathbb{R}^{m \times m}$ denotes the identity matrix.

Property (1) is that of orthogonality (each column is a unit length vector perpendicular to the others) and property (2) makes the orthogonality special. Given any orthogonal matrix A, the determinant of A must be either 1 or -1.

A positive determinant makes A a rotational matrix, while a negative one indicates the presence of a reflection in the matrix, which can easily be factored out. Given any matrix A with $|A| = -1$, regard $A' = -IA$. Then

$$|A'| = |-IA| = |-I||A| = -1|A| = 1.$$

Thus A' represents a rotational matrix.

Now the methods that yield a decomposition into an orthogonal rotational matrix and additional ones are described. Three different methods describing orthogonal decompositions of a given square and non-singular matrix can be found:

1. *QR decomposition* yields $M = QR$ with Q and R both orthogonal and lower triangular.
2. *Singular value decomposition* yields $M = UKV^T$ with U and V both orthogonal and K diagonal and positive.
3. *Polar decomposition* yields $M = QS$ with Q orthogonal and S symmetric and positive definite.

The least attractive method is singular value decomposition because the factors cannot be determined uniquely, it is complicated to implement, and the known algorithms are inefficient.

QR decomposition is a better choice since the factors can be uniquely determined and the algorithms known for this method are efficient. But the orthogonal factor depends on the coordinate basis used, and this can lead to problems. If M is constructed by rotating and then scaling, the QR decomposition will not capture the rotation! It will recover the original factors only if M is constructed using scaling first and rotation afterwards.

Polar decomposition will determine unique factors, a coordinate independent orthogonal matrix, and it is simple and efficient to compute. A more theoretical aspect using this method is the fact that Q is the closest possible orthogonal matrix to M. That is, Q minimizes $||Q-M||_F^2$ using the Frobenius matrix norm

$$||A||_F = \sum_{i,j} a_{ij}$$

as a measure of closeness[1]. A disadvantage of this method is the fact that it does not directly represent shearing (Shoemake and Duff [SD92]).

The above considerations show that the decomposition method depends on the situation. Since the QR decomposition suffices, it is the method implemented in Collage3D in order to export objects and their transformations. But the polar decomposition seems to be an interesting alternative, so there will be a short introduction of this approach in the paper.

3 A QR Decomposition Variant

3.1 Decomposing the Orthogonal Part

Let $M \in \mathbb{R}^{3\times 3}$ be an arbitrary, non-singular matrix. Then the rows of M, denoted u, v, w, are linearly independent. Thus the Schmidt orthogonalization (see, e.g., Meyberg and Vachenauer [MV90]) can be applied in order to retrieve three orthonormal vectors u', v', w': [2]

[1] A more detailed discussion of the different decomposition methods and a proof of the closeness claim can be found in [SD92].

[2] Here $v \cdot w$ denotes the scalar product of v and w.

$$u' = \frac{u}{|u|}$$

$$v' = \frac{v - (v \cdot u')u'}{|v - (v \cdot u')u'|}$$

$$w' = \frac{w - (w \cdot u')u' - (w \cdot v')v'}{|w - (w \cdot u')u' - (w \cdot v')v'|}$$

Now u', v', w' are unit length vectors and pairwise perpendicular. A matrix M' that consists of the rows u', v', w' is therefore an orthogonal matrix.

3.2 Extraction of the Scale

The scaling transformations are easy to determine. The scale of each vector u', v', w' is its length before the normalization, i.e. the denominators of the fractions in Section 3.1:

$$s_1 = |u|$$
$$s_2 = |v - (v \cdot u')u'|$$
$$s_3 = |w - (w \cdot u')u' - (w \cdot v')v'|$$

Here s_1, s_2, s_3 denote the scaling along the x-axis, y-axis and z-axis, respectively.

As stated in Section 2 the determinant $|M'|$ can be 1 or -1. In case of a negative determinant the matrix contains a reflection. In order to decompose that reflection, the points are reflected in all planes, that means all the scale values are negated and $M'' = -IM'$ is calculated for further use (see Thomas [Tho91] for more details).

3.3 Extraction of the Rotation Parts

Now M'' is an orthogonal 3×3 matrix being a pure rotation matrix. The specific rotations about the different axes x, y and z have to be determined. As shown in [Tho91], this can be done by assuming a certain order of rotations.

Let α be the rotation angle about the x-axis, β the angle about the y-axis and γ the one about the z-axis. Regard the composite matrix obtained by rotating about the x-, y- and z-axis, in this order:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{pmatrix} \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\beta\cos\gamma & \cos\beta\sin\gamma & -\sin\beta \\ \cos\gamma\sin\alpha\sin\beta - \cos\alpha\sin\gamma & \cos\alpha\cos\gamma + \sin\alpha\sin\beta\sin\gamma & \cos\beta\sin\alpha \\ \sin\alpha\sin\gamma + \cos\alpha\cos\gamma\sin\beta & \cos\alpha\sin\beta\sin\gamma - \cos\gamma\sin\alpha & \cos\alpha\cos\beta \end{pmatrix}$$

The angle β can easily be obtained from the upper right entry of R.

$$\beta = \arcsin(-R_{13})$$

Now two cases have to be distinguished:

1. $\cos\beta = 0$.
   In this case the matrix contains a rotation about the y-axis using an angle $\beta = \pm\frac{\pi}{4}$. Remember that the previously assumed rotation sequence is applied to the canonical basis vectors e_x, e_y and e_z. The first rotation about the x-axis does only change the image vectors of e_y and e_z. Now the rotation about the y-axis using an angle $\beta = \pm\frac{\pi}{4}$ is applied. Thus the image vector of e_x coincides with the z-axis. Observe that the last rotation about this axis could have already been done in the previous rotation about the x-axis. Thus the rotation about the z-axis can be ignored and the desired angle can be added to the rotation angle about the x-axis. So $\gamma = 0$, which implies $\sin\gamma = 0$ and $\cos\gamma = 1$.
   Furthermore $\sin\beta = \pm 1$, yielding a simplified matrix.

$$R = \begin{pmatrix} 0 & 0 & \pm 1 \\ \pm\sin\alpha & \cos\alpha & 0 \\ \pm\cos\alpha & -\sin\alpha & 0 \end{pmatrix}$$

   Now α can obtained from the entry R_{32}: $\alpha = -\arcsin R_{32}$.
2. $\cos\beta \neq 0$.
   In order to retrieve the rotation angles from the rotation matrix recall that this matrix represents the image of the canonical basis after the rotation has been applied. Thus every row of the rotation matrix represents the image vector of one of the canonical base vectors in cartesian coordinates. As known from linear algebra, it is possible to transform these cartesian coordinates to polar coordinates, which explicitly contain the rotation angle.
   A rotation about the x-axis affects the z-coordinates of the vectors e_y and e_z. Thus these coordinates are projected onto the plane and converted to polar coordinates.
   The rotation angle about the z-axis affects the x- and y-coordinate of the vector e_y, thus these coordinates are also projected onto the plane and transformed.
   As known from linear algebra the angle of the polar coordinates can then be calculated by $\gamma = \arctan\frac{R_{11}}{R_{12}}$ if $R_{12} \neq 0$.
   This formula does not suffice, since the denominator of that fraction can be 0, and the arcustangens function always yields a resulting angle residing in the first quadrant. But the correct angle can be determined

from the signs of the nominator and denominator. So there is a defined function in programming languages called $\arctan_2$. It is defined as follows:

$$\arctan_2(x_1, x_2) = \begin{cases} \arctan \frac{x_1}{x_2} & , x_1 > 0, x_2 \neq 0 \\ -\arctan \frac{x_1}{x_2} & , x_1 > 0, x_2 \neq 0 \\ 0 & , x_1 = 0, x_2 > 0 \\ \pi & , x_1 = 0, x_2 < 0 \\ \frac{\pi}{2} & , x_2 = 0, x_1 > 0 \\ -\frac{\pi}{2} & , x_2 = 0, x_1 < 0 \\ 0 & , x_1 = 0, x_2 = 0 \end{cases}$$

The function also takes care of the range, that is,

$$-\pi \leq \arctan_2(x_1, x_2) \leq \pi.$$

Thus $\arctan_2(x_1, x_2)$ calculates the angle of the polar coordinates of the point (x_1, x_2). So this function is applied to the previously mentioned coordinates respectively in order to retrieve the rotation angles.

3.4 Shearing

Basics of Shearing

In order to understand the extraction of the shearing transformations, the basics of shearing are recalled.

To define a shearing the tensor product of two vectors is needed. Given two vectors v, w the *tensor product* $v \otimes w$ is defined as

$$v \otimes w = v^T * w = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} * \begin{pmatrix} w_1 & w_2 & w_3 \end{pmatrix} = \begin{pmatrix} v_1 w_1 & v_1 w_2 & v_1 w_3 \\ v_2 w_1 & v_2 w_2 & v_2 w_3 \\ v_3 w_1 & v_3 w_2 & v_3 w_3 \end{pmatrix}.$$

A shearing transformation in three dimensions is a non-uniform scaling along any pair of axes proportional to the shear-axis.

To formally define a shearing a unit vector v normal to a plane Q, a unit vector w residing in that plane and an angle ϕ is needed.

Regard any given point P. To apply the shearing, project P orthogonally onto a point P' in the shearing plane S. Now P is slid parallel to the vector w to a point P'', so that the angle between P'', P and P' equals ϕ. This is shown in Figure 1. Then the shearing H is defined as $H = I + \tan\phi(v \otimes w)$ (see, e.g., Goldman [Gol91]). Thus the resulting matrix is given as

$$H = \begin{pmatrix} 1 + v_1 w_1 \tan\phi & v_2 w_1 \tan\phi & v_3 w_1 \tan\phi \\ v_2 w_1 \tan\phi & 1 + v_2 w_2 \tan\phi & v_3 w_2 \tan\phi \\ v_3 w_1 \tan\phi & v_2 w_3 \tan\phi & 1 + v_3 w_3 \tan\phi \end{pmatrix}.$$

Observe that the determinant of H is 1, thus a shearing is a transformation that preserves volume.

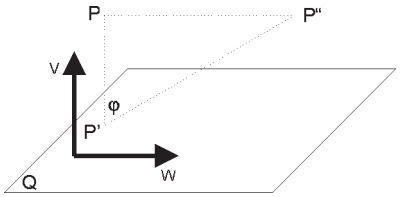

Figure 1. Shearing

If the shearing plane is one of the planes spanned by two of the coordinate axes, special transformation matrices can be found.

To shear the points along the y-axis parallel to the plane spanned by the z-axis and the x-axis into the direction of the x-axis, we choose the unit vectors $e_y = (0, 1, 0)$ and $e_x = (1, 0, 0)$. In this case the shear amount given by $\tan\phi$ will hence be called sh_{xy} (for shearing into the direction of the x-axis along the y-axis).

The resulting transformation matrix H_{xy} then calculates to

$$H_{xy} = I + sh_{xy}(0,1,0) \otimes \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ sh_{xy} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

A shearing along the y-axis into the direction of the z-axis is also possible, the resulting matrix calculates analogously to the one above to

$$H_{zy} = I + sh_{zy}(0,1,0) \otimes \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & sh_{zy} \\ 0 & 0 & 1 \end{pmatrix}.$$

These transformations do not interact, thus we can summarize them to a matrix representing the shearing along the y-axis parallel to the x,z-plane:

$$H_y = \begin{pmatrix} 1 & 0 & 0 \\ sh_{xy} & 1 & sh_{zy} \\ 0 & 0 & 1 \end{pmatrix}$$

The transformation matrices for shearing along the other two coordinate axes calculate analogously to:

$$H_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ sh_{xz} & sh_{yz} & 1 \end{pmatrix}$$

$$H_x = \begin{pmatrix} 1 & sh_{yx} & sh_{zx} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Extraction of the Shearing

Recall that the rows of a 3×3 matrix A represent the image of the transformation mapping under the canonical basis. The only way to change the angles between these image vectors using affine transformations is to apply a shearing. In order to decompose a shearing between two vectors, the angle between them is needed. As known from linear algebra, the angle between two vectors can be obtained calculating their scalar product.

The scalar product of two vectors v_1, v_2 yields $v_1 v_2 = |v_1||v_2| \cos\alpha$, with α being the angle between v_1 and v_2.

Now assume that the first row (the image vector of the unit vector e_x), is only scaled and rotated, if anything. As shown in Section 3.1, the calculated vector u' has the length one. Thus the scalar product of u' and v yields $u'v = |u'||v| \cos\alpha = |v| \cos\alpha$.

So the result of the scalar product represents the shearing between the vectors u' and v multiplied with the length of v.

The a priori approach of dividing the scalar product by the length of v will not work in general, since a shearing may change the length of a vector as shown in the following example.

Regard a simple shearing H_1 along the y-axis into the direction of the x-axis with a shearing amount of one. Applying H_1 to the canonical base vector $e_y = (0, 1, 0)$ yields an image vector $H_1(e_y) = (1, 1, 0))$. Thus the length of the transformed vector changed from one to the square root of two.

This means that the shearing has to be removed before calculating the length. This is done by making the vector perpendicular to the other one using the method of Schmidt, as shown in Section 3.1.

The length of the resulting vector can now be calculated and the scalar product is divided by this length. This yields the actual shearing between the two vectors. As stated before, it is assumed that the first row is not sheared at all, so the shearings of the second row into the direction of the x-axis and the third row into the direction of the x-axis and the y-axis are calculated.

3.5 Summary

The previous considerations are based on isolated parts of the decomposition problem, but in fact the calculations are not separately done. A short summary of the neccessary decomposition steps may serve as a basis for implementation or even as an algorithm for manual decomposition.

Given any arbitrary non-singular matrix $M \in \mathbb{R}^4 \times \mathbb{R}^4$, the rows are denoted `r1`, `r2`, `r3` and `r4`. An entry is `rij` with `i,j`$\in \{1, 2, 3, 4\}$.

```
// extract the translations
translate_x := r41;
translate_y := r42;
translate_z := r43;
```

```
// clear the perspective and
// the translation partition:
for (i := 1; i < 4; i++){
  r4i := 0;
  ri4 := 0;
}
r44 := 1;

scale_x     := |r1|;
r1 := normalize(r1);
shear_xy := scalarProduct(r1,r2);
r2 := r2 - shear_xy*r1;
scale_y := |r2|;
shear_xy := shear_xy/scale_y;
r2 := normalize(r2);
shear_xz := scalarProduct(r1,r3);
r3 := r3 - shear_xz*r2;
shear_yz := scalarProduct(r2,r3);
scale_z := |r3|;
r3 := normalize(r3);
shear_xz := shear_xz/scale_z;
shear_yz := shear_yz/scale_z;
if (determinant(R) < 0){
  R := -1*R;
  scale_x := -1*scale_x;
  scale_y := -1*scale_y;
  scale_z := -1*scale_z;
}
rotate_y := asin(-r13);
if (rotate_y == 0){
  rotate_x := asin(r32);
  rotate_z := 0;
} else {
  rotate_x := atan2(r23,r33);
  rotate_z := atan2(r12,r11);
}
```

This decomposition method yields a sequence of transformations, that has to be applied to the unit matrix using a special order. Scaling has to be the first transformation to be used, then the shearings followed by rotations. Finally the translation has to be applied. The resulting matrix from this sequence of transformations should then equal the matrix before the decomposition except for numerical rounding errors.

3.6 Verification of the Result

Since a transformation matrix is linear, it suffices to check the result for the canonical basis. The factors of the transformations to be applied can be

deduced from the previous sections. The necessary calculation is simple but elaborate, so it is left as an exercise to the reader.

On the other hand a close look at the underlying method by Schmidt shows that this orthogonalization simply yields exactly the needed factors.

4 Polar Decomposition

Let M be a non-singular matrix and $X_0 = M$. Now construct the sequence

$$X_{k+1} = \frac{1}{2}(X_k + (X_k^{-1})^T)$$

with $k = 0, 1, 2, \ldots$[3] This sequence is quadratically convergent to Q. As usual the iteration will be aborted once a convergence tolerance δ is reached:

$$||X_{k+1} - X_k||_1 \leq \delta ||X_{k+1}||_1$$

The matrix norm used in this criterion is the induced vector l_1-norm $||A||_1 = \max_j \sum_i |a_{ij}|$.

Since the iteration above will be implemented on a computer in our case, the convergence tolerance can easily be obtained. Let ϵ be the machine precision such that $1 + \epsilon$ is the smallest number greater than 1. Then choose the machine precision ϵ as the convergence tolerance, since it is not possible to get any closer to the actual matrix Q in a machine environment anyway.

Now the matrix S can be obtained from the formula $M = QS$, using

$$S = \frac{1}{2}(Q^T M + M^T Q).$$

This formula is used instead of the a priori approach $S = Q^T M$ in order to minimize rounding errors in a numerical environment.

4.1 Extraction of the Rotation Parts

This section is meant to describe the extraction of each rotational part from the computed matrix Q. Since Q is orthogonal, the decomposition method presented in Section 3.3 suffices.

4.2 Extraction of Scale and Shearing

As mentioned in Section 2, the polar decomposition does not directly represent shearing. As shown in [SD92], the factor S of the polar decomposition $M = QS$ is diagonal in selected rotated coordinate systems. In other words,

[3] This method is inspired by Newton's iterative method for computing the square root of a number c: $x_{k+1} = \frac{1}{2}(x_k + \frac{c}{x_k})$ with $k = 0, 1, 2, \ldots$ and $c > 0$.

S itself can be factorized into rotation and scaling to $S = UKU^T$, with U being a rotation matrix and K being diagonal. Since this factor has the property of preserving the scaling through coordinate system changes, Shoemake and Duff propose to introduce it as a new primitive called *stretch*. A simple shearing is decomposed into rotation and stretch by polar decomposition.

If the shearing is explicitly needed it can be calculated from the above factorization. In the *VRML97* standard shearing cannot be specified explicitly. However, it can be specified implicitly as a rotation of the coordinate system, non-uniform scaling and reversed rotation.

5 A Decomposition Example

In order to illustrate the previous considerations an example decomposition will be shown. A composite matrix will be constructed from an arbitrarily chosen sequence of single elementary transformations. Every two transformation steps are illustrated by a screenshot from a *VRML*-Viewer. The resulting composite matrix will then be decomposed using the previously explained QR decomposition method.

5.1 Construction of a Composite Matrix

The part that has to be transformed is a unit cube centered in the origin of the three-dimensional space as shown in Figure 2.

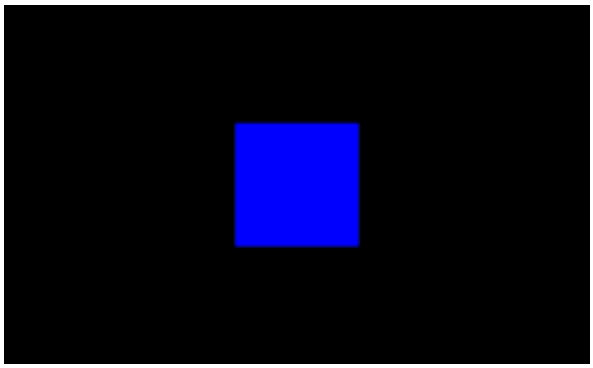

Figure 2. Cube centered in the origin

First the cube is moved 0.75 units to the right. Now a shearing of 1 in the direction of x along the y-axis parallel to the x,z-plane is applied. The result of these transformations is shown in Figure 3.

Now the cube is rotated about the y-axis using an angle of $\frac{\pi}{2}$ and stretched into the direction of the x-axis at an amount of 1.5. The resulting cube is shown in Figure 4.

Figure 3. Cube moved and sheared

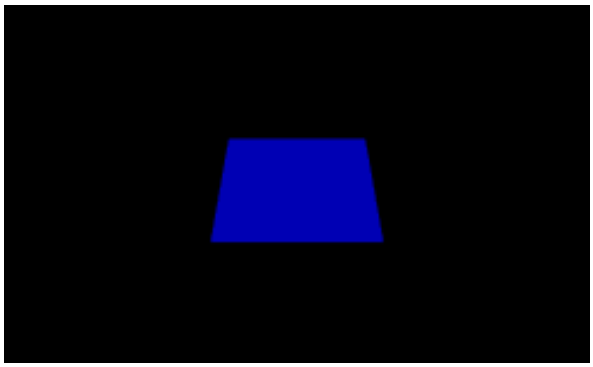

Figure 4. Cube rotated and stretched

Finally, the cube is moved 1.3 units forwards and rotated about the x-axis at $-\frac{\pi}{7}$. The transformed cube is shown in Figure 5.

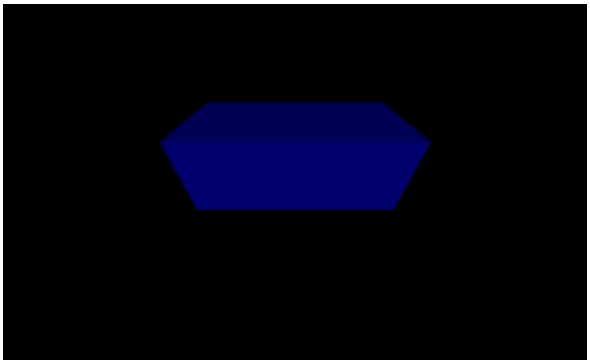

Figure 5. Cube moved and rotated

The composite matrix is:

$$M = \begin{pmatrix} 0 & -0.4339 & -0.901 & 0 \\ 0 & 0.4671 & -1.3349 & 0 \\ 1.5 & 0 & 0 & 0 \\ 0 & 0.2386 & 0.4955 & 1 \end{pmatrix}$$

5.2 Decomposition Using the QR Method

The composite matrix M will now be decomposed using the QR-method described in Section 3.

First the translation is extracted. This is simply the fourth row of M. Thus the translation values are:

$$\begin{aligned} \text{translate}_x &= 0 \\ \text{translate}_y &= 0.2386 \\ \text{translate}_z &= 0.4955 \end{aligned}$$

Set M' to the left upper 3×3 partition of M. M' consists of the rows u, v and w. Now calculate the length of u:

$$|u| = 1 \Rightarrow u' = \frac{u}{|u|} = u \Rightarrow \text{scale}_x = 1.$$

The scalar product of u' and v yields the factor for the normalization: $u'v = 1$. Thus $v' = v - u' = (0, 0.9010, -0.4339)$. Calculating $|v'| = 1$ yields:

$$\text{scale}_y = 1 \text{ and } \text{shear}_{xy} = 1.$$

Now the scalar product of w and u' is calculated in order to retrieve the normalization factor:

$$wu' = 0 \Rightarrow \text{shear}_{xz} = 0 \quad \text{and} \quad w' = w.$$

Then $w'v' = 0 \Rightarrow \text{shear}_{yz} = 0$. Moreover, $|w'| = 1.5 \Rightarrow \text{scale}_z = 1.5$ and $w'' = \frac{w''}{1.5} = (1, 0, 0)$.

Now consider the matrix R consisting of the rows u', v' and w'':

$$R = \begin{pmatrix} 0 & -0.4339 & -0.901 \\ 0 & 0.901 & -0.4339 \\ 1 & 0 & 0 \end{pmatrix}$$

R is a pure rotational matrix because $|R| = 1$.

Finally the rotation angles are decomposed using the entries of R:

$$\beta = \arcsin 0.901 = 1.1221 \Rightarrow \text{rotate}_y = 1.1221.$$

As $\cos\beta = 0.4338 \neq 0$, this implies:

$$\alpha = \arctan_2(-0.4339, 0) = -\frac{\pi}{2} \Rightarrow \text{rotate}_x = -1.5708$$
$$\gamma = \arctan_2(-0.4339, 0) = -\frac{\pi}{2} \Rightarrow \text{rotate}_z = -1.5708$$

Thus the decomposition yields:

$\text{scale}_x = 1$	$\text{shear}_{xy} = 1$
$\text{scale}_y = 1$	$\text{shear}_{xz} = 0$
$\text{scale}_z = 1.5$	$\text{shear}_{yz} = 0$
$\text{rotate}_x = -1.5708$	$\text{translate}_x = 0$
$\text{rotate}_y = 1.1221$	$\text{translate}_y = 0.2386$
$\text{rotate}_z = -1.5708$	$\text{translate}_z = 0.4955$

Applying these transformations to the original cube yields the same transformed cube as shown in Figure 5. The transformation matrix of that cube is:

$$C = \begin{pmatrix} 0 & -0.4338 & -0.901 \\ 0 & 0.4672 & -1.3348 \\ 1.5 & 0 & 0 \end{pmatrix}$$

Due to rounding errors in this example, C does not equal M, but this is a common numerical situation and its effect depends on the chosen precision.

4 Networked Iterated Function Systems

Sarah Behrens, Maxim Bortin, and Jianfeng Chen

1 Introduction

In this chapter we are going to introduce another image generating device, the *networked iterated function systems*. To refresh some of the basic concepts of the linear algebra we start with a short section about affine transformations. For further details about transformations in the three dimensional space we refer to the Chapter 2. The concept of the networked iterated function systems is a generalization of the *iterated function systems*, therefore we are going to recall the concept of those systems in the Section 3. Iterated function systems are a well known device for the generation of fractals. In the Section 4 we will then describe the networked iterated function systems, draft some of their properties and show a few examples. Finally we briefly introduce the idea of composition in Section 4.7.

2 Affine Transformations

An *affine transformation* $f : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ is defined by $f(x) = A \cdot x + b$ where A is a $d \times d$ - matrix and $b \in \mathbb{R}^d$ a translation vector. We will consider the affine transformations where the matrix A has the property: $\det A \neq 0$. The set of all affine transformations from $\mathbb{R}^d$ to $\mathbb{R}^d$ will be denoted by $AFF(\mathbb{R}^d)$.

In this paper we are going to treat examples in $\mathbb{R}^2$. Transformations in $AFF(\mathbb{R}^2)$ can be of the following kinds or compositions of these kinds:

- scaling
- shearing
- rotation
- translation

Applied to an image (generally a compact set of points in the $\mathbb{R}^d$, here in the $\mathbb{R}^2$) they have an important property: a polytope with n corners always remains a polytope with n corners, but angles are not maintained. Without the shearing we would obtain a set of similarity transformations for those angles need to be maintained.

Subsequently we need a 2×2-matrix in the plane and a translation vector $(a, b) \in \mathbb{R}^2$. To handle the different transformations more easily we write our four coefficients as follows:

$$\begin{pmatrix} r \cdot \cos\phi & -s \cdot \sin\psi \\ r \cdot \sin\phi & s \cdot \cos\psi \end{pmatrix}$$

- s is the scaling factor in the y-direction.
- r is the scaling factor in the x-direction.
- If ψ equals ϕ, they represent the angle we want the image to be turned about.
- Giving them different values makes it possible to model reflections with respect to any axis.

In the examples we denote scaling with **scale(r,s)** which corresponds to the following matrix:

$$\begin{pmatrix} r \cdot \cos 0 & -s \cdot \sin 0 \\ r \cdot \sin 0 & s \cdot \cos 0 \end{pmatrix} = \begin{pmatrix} r & 0 \\ 0 & s \end{pmatrix}$$

Rotations are denoted by **rotate(α)**. It corresponds to the matrix:

$$\begin{pmatrix} 1 \cdot \cos\alpha & -1 \cdot \sin\alpha \\ 1 \cdot \sin\alpha & 1 \cdot \cos\alpha \end{pmatrix}$$

Finally, translations are denoted by **shift(a,b)**.

3 Iterated Function Systems

Iterated function systems (see, e.g., Peitgen et al. [PJS92] and Hutchinson [Hut81]) are devices to generate images. They encode complex images by just a few transformations. Therefore they are often used in connection with image compression, but this will not be discussed here (see, e.g., Fisher [Fis95]).

An iterated function system consists of a set of affine transformations. Its so-called Hutchinson operator, applied to an image, yields a refined image by applying all transformations and overlaying the results. By iteration, one gets an infinite sequence of images for each start image.

3.1 Basic Concepts

Definition 3.1.1
1. An image Y is an *affine refinement* of an image X if there is a set F of affine transformations with

$$Y = \bigcup_{f \in F} f(X).$$

2. An *iterated function system* (IFS for short) is a finite set F of affine transformations (on $\mathbb{R}^d$ for some $d \in \mathbb{N}$).

3. Let F be an IFS and X an image. Then the *Hutchinson operator* of F (applied to X) yields the image

$$HO_F(X) = \bigcup_{f \in F} f(X).$$

4. By iteration, the Hutchinson operator yields an infinite sequence of images for each initial image $B \subseteq \mathbb{R}^d$ defined by

$$B_0 = B \text{ and } B_{i+1} = HO_F(B_i) \text{ for all } i \in \mathbb{N}.$$

This sequence is called *Hutchinson sequence of F and B* and is denoted by $H_F(B)$ with $H_F(B)_n = B_n$ for all $n \in \mathbb{N}$. In examples, $H_F(B)_n$ is shortly denoted by $F(n)$.

Observation 3.1.2
Let F be an IFS, $B \subseteq \mathbb{R}^d$ an image and $H_F(B)$ the corresponding Hutchinson sequence. Then, for all $i, j \in \mathbb{N}$ with $i < j$, $H_F(B)_j$ is an affine refinement of $H_F(B)_i$.

Proof
It is easy to see that $H_F(B)_j$ is an affine refinement of $H_F(B)_i$ with respect to the set F^{j-i} (where $F^1 = F$ and $F^{n+1} = F^n \circ F$). □

3.2 Example

This example shows the simple IFS **square** which consists of two affine transformations f_1 and f_2 and starts with a square. Figure 1 illustrates this.

The transformation f_1 is the identity and f_2 scales by the factor 1.1 and shifts by 0.1 in the x and the y directions. Figures 2 and 3 show the iteration steps 0 to 6 of **square**.

3.3 Contracting IFSs

This section introduces the special case of contracting IFSs, the transformations of which move each two points closer together.

As precisely formulated in Observation 3.1, an IFS may provide images with more and more details by means of more and more applications of the Hutchinson operator. In the case of contracting IFSs, the situation is even better. The Hutchinson sequence is then a Cauchy sequence such that it converges yielding a single fractal image as the limit, which is called attractor. The attractor turns out to be independent of the initial image and to be self-affine meaning that it is an affine refinement of itself with respect to the Hutchinson operator.

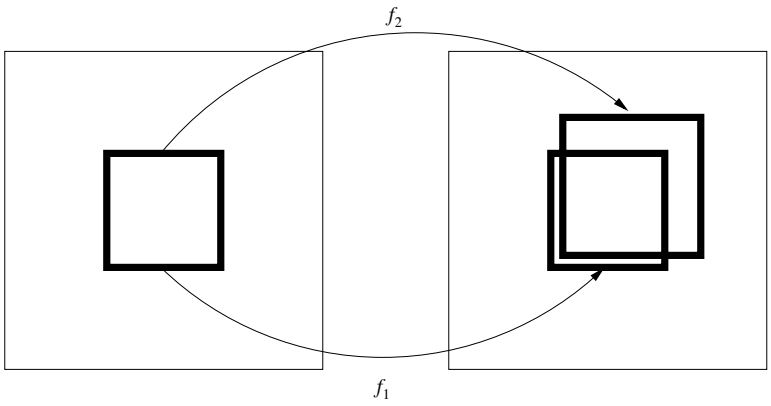

Figure 1. The construction of **square**

Figure 2. **square**(0), **square**(1) and **square**(2)

Figure 3. **square**(3), **square**(4), **square**(5) and **square**(6)

Definition 3.3.1
1. A transformation $f : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ is *contracting* if there exists a constant $c < 1$ such that the following holds for each two points P_1 and P_2:

$$\Delta(f(P_1), f(P_2)) \leq c \cdot \Delta(P_1, P_2),$$

where $\Delta(P_i, P_j)$ is the distance between two points P_i and P_j using the Euclidean metric.

2. A *contracting IFS* consists of contracting transformations.

3. An image X is called *self-affine* if

$$\bigcup_{f \in F} f(X) = X,$$

where F is a set of contracting affine transformations.

Theorem 3.3.2
Let F be a contracting IFS and B and B' be two non-empty images. Then there is an image attr(F), called ***attractor of F****, which is the limit of $H_F(B)$ as well as of $H_F(B')$. Moreover, attr(F) is self-affine.*

3.4 Example

The most popular example in the literature is the Sierpinsky gasket, that can be encoded as the attractor of an IFS. This IFS (**sierpinski**) consists of three transformations f_1, f_2, f_3, that scale by the factor of one half and shift the image in three different positions as shown in Figure 4. The first four pictures in the corresponding Hutchinson sequence are shown in Figure 5.

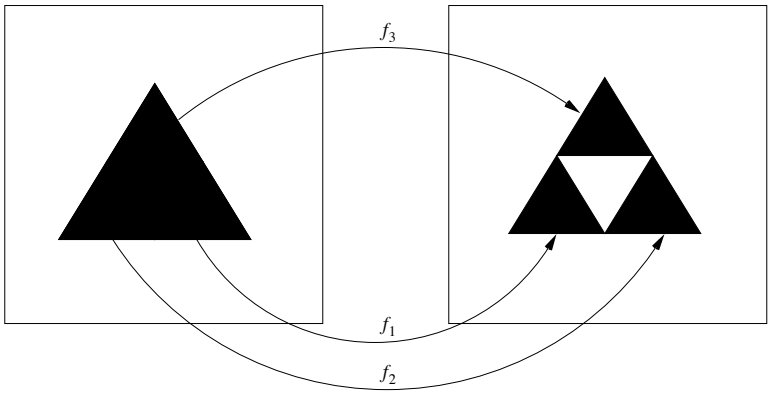

Figure 4. The construction of the Sierpinski gasket

These transformations are contracting if we use the Euclidean metric. This can be seen in the following way. The distance between two points in $\mathbb{R}^2$ is defined as follows:

$$\Delta((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_2 - y_1)^2}$$

Figure 5. Iteration steps 0, 1, 2 and 3 of the Sierpinski gasket

First we are going to take a look at $f_1(x, y) = (0.5x, 0.5y)$.

$$\begin{aligned}
\Delta(f_1(x_1, y_1), f_1(x_2, y_2)) &= \Delta((0.5x_1, 0.5y_1), (0.5x_2, 0.5y_2)) \\
&= \sqrt{(0.5x_1 - 0.5x_2)^2 + (0.5y_2 - 0.5y_1)^2} \\
&= \sqrt{0.25(x_1 - x_2)^2 + 0.25(y_2 - y_1)^2} \\
&= \sqrt{0.25((x_1 - x_2)^2 + (y_2 - y_1)^2)} \\
&= 0.5 \cdot \sqrt{(x_1 - x_2)^2 + (y_2 - y_1)^2} \\
&< \sqrt{(x_1 - x_2)^2 + (y_2 - y_1)^2} \\
&= \Delta((x_1, y_1), (x_2, y_2))
\end{aligned}$$

Analogously we could show this for f_2 and f_3. f_2 and f_3 differ from f_1 just in a translation. Translations do not have any effect on the distance between the points.

If we take a look at any two points of the triangle, we can immediately see that all three transformations move them closer together. In consequence this IFS has an attractor, the Sierpinski gasket. This means that the image sequence produced by this IFS always converges against the Sierpinski gasket independently from the start image. Usually we start with a triangle, but we could also take any other image as start and we can see that if the Hutchinson operator is applied often enough we obtain always the same image that resembles the attractor of this IFS, the Sierpinski gasket.

The image sequence in Figure 6 shows the iteration steps 1, 3 and 7 starting with a triangle. In comparison to Figure 6, the same steps starting with a square are shown in Figure 7.

4 Networked Iterated Function Systems

An iterated function system generates an image sequence in which members are affine refinements of their predecessors. To overcome this restriction, networked iterated function systems (see, e.g., Peitgen et al. [PJS92], Fisher

Figure 6. sierpinski(1), **sierpinski**(3) and **sierpinski**(7) starting with a triangle

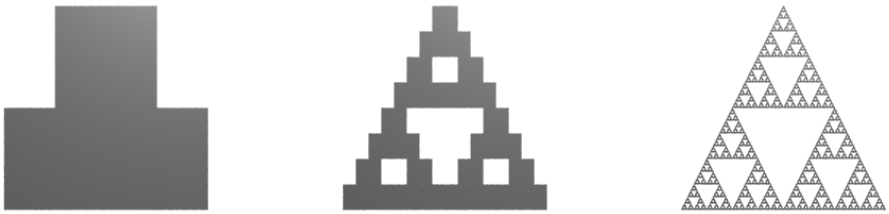

Figure 7. sierpinski(1), **sierpinski**(3) and **sierpinski**(7) starting with a square

[Fis95]) are introduced that generalize IFSs. To illustrate the idea, another well-known fractal, the Hilbert curve, is considered. This area-filling curve was presented by David Hilbert in 1891 (see Hilbert [Hil91]). Figure 8 shows iteration steps 0, 1, 2 and 3 of the Hilbert curve.

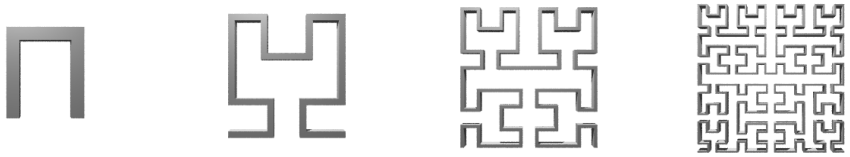

Figure 8. Iteration steps 0, 1, 2 and 3 of the Hilbert curve

It is not possible to generate this image sequence by an IFS because the images are not affine refinements of their predecessors. Figure 9 makes this clear. While the four bold lines in the right side image can be covered nicely by affine transformations of the left side image, the three dotted lines cannot be obtained by transforming the left side image.

To cover this we need a new concept. Therefore networked iterated function systems are introduced as an instrument to make more complex and maybe more interesting images. Networked iterated function systems are de-

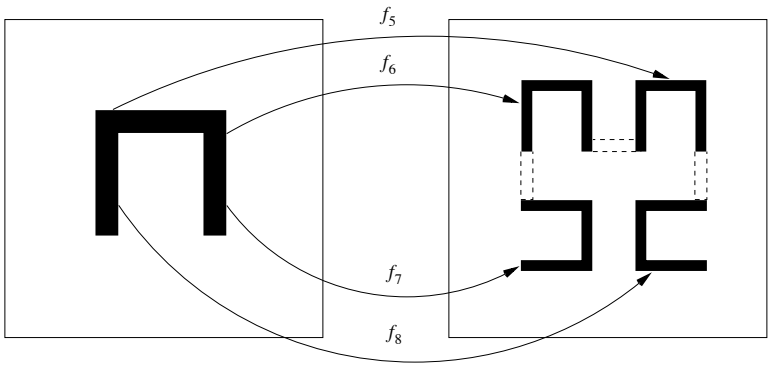

Figure 9. The construction of the Hilbert curve

fined as directed graphs with a distinguished output node, the edges of which are labeled by affine transformations.

Such a system induces a generalized Hutchinson operator that transforms and refines an image vector comprising an image for each node of the graph. To obtain the refined image of a node, one applies the transformation of each incoming edge to the image of the source node and overlays all resulting images. Starting with an arbitrary image vector, one gets an infinite sequence of image vectors by iterated application of the Hutchinson operator. As in the simpler case of iterated function systems, this sequence converges if all involved transformations are contracting. If one restricts the image vector sequence to the output node, the networked iterated function system describes a sequence of images which converges against a single attractor image in the case of contracting transformations.

4.1 Basic Concepts

Definition 4.1.1
1. A *networked iterated function system* (NIFS for short) is a system

$$N = (V, E, s, t, f, v_0),$$

where

- V is the finite set of *vertices*,
- E is the finite set of *edges*,
- $s : E \longrightarrow V$ and $t : E \longrightarrow V$ are mappings that associate a *source s(e)* and a *target t(e)* to each edge $e \in E$,
- $f : E \longrightarrow AFF(\mathbb{R}^d)$ is a function that maps an edge to an affine transformation,
- $v_0 \in V$ is called *output*.

2. Let $N = (V, E, s, t, f, v_0)$ be an NIFS and $X : V \longrightarrow 2^{\mathbb{R}^d}$ be a vector of images (with index set V). Then the *Hutchinson operator of N* (applied to X) yields a vector of images $HO_N(X) : V \longrightarrow 2^{\mathbb{R}^d}$ given by

$$HO_N(X)(v) = \bigcup_{e \in E, t(e)=v} f(e)(X(s(e))).$$

3. By iteration, the Hutchinson operator yields an infinite sequence of image vectors for each initial image vector $B : V \longrightarrow 2^{\mathbb{R}^d}$ defined by

$$B_0 = B \quad \text{and} \quad B_{i+1} = HO_N(B_i) \quad \text{for all} \quad i \in \mathbb{N}.$$

This sequence is called *Hutchinson sequence of N and B* and denoted by $H_N(B)$ with $H_N(B)_n = B_n$ for all $n \in \mathbb{N}$.

4. If $H_N(B)$ is restricted to the output component, one gets the *image sequence N(B)* of N with respect to B, i.e.

$$N(B)_n = H_N(B)_n(v_0) \quad \text{for all} \quad n \in \mathbb{N}.$$

$N(B)_n$ may be denoted by *N(n)* if B is clear from the context.

Analogously to the concept of contracting IFSs in Section 3.3, one can define contracting NIFSs.

Definition 4.1.2
An NIFS $N = (V, E, s, t, f, v_0)$ is called *contracting* if, for all $e \in E$, $f(e)$ is contracting.

Theorem 4.1.3
Let $N = (V, E, s, t, f, v_0)$ be a contracting NIFS. Then there is a unique vector of images which is the limit of the Hutchinson sequence $H_N(B)$ for each initial vector B of non-empty and compact images.

The proof is omitted because it is a straightforward generalization of the corresponding one for IFSs. The limit in the theorem is called *attractor* of the given NIFS N and denoted by *attr(N)*. The output component $attr(N)(v_0)$ of the attractor is called *the picture of N* and denoted by *pict(N)*.

4.2 Example

We can model the Hilbert curve by an NIFS with two vertices **Node a** and **Node b**. The initial image of **Node a** is a line and the initial image of **Node b** consists of three lines as shown in Figure 10. The line from **Node a** is transformed to the three needed positions into **Node b** (f_2, f_3, f_4) which applies the four transformations (f_5, f_6, f_7, f_8) we already saw in Figure 9. Furthermore the line from **Node a** must be scaled by one half which is done by f_1. This NIFS is shown in Figure 10.

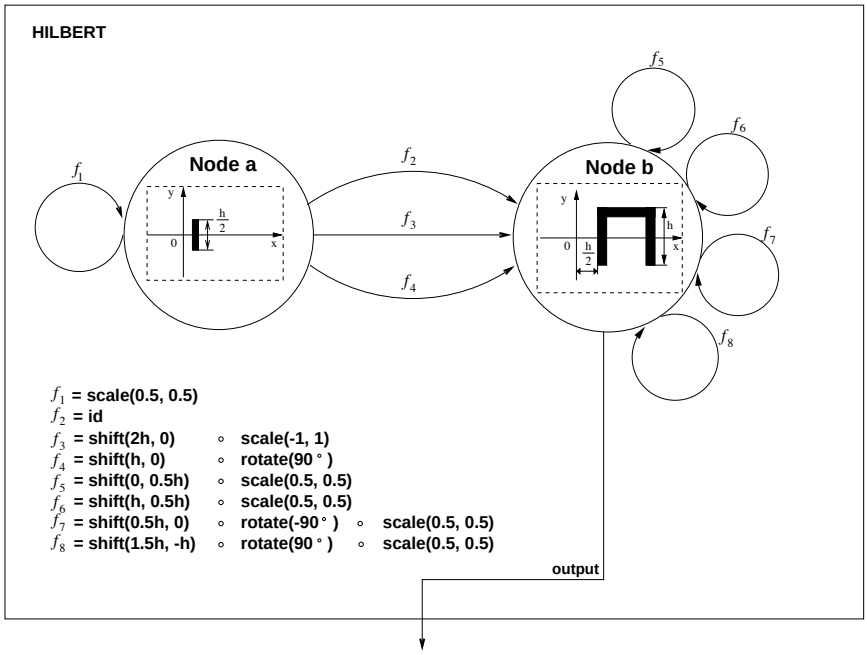

Figure 10. The Hilbert curve as an NIFS

4.3 IFS as a Special Case of NIFS

It turns out that each IFS can be seen as a special NIFS with a single vertex and its transformations as edges.

Let F be an IFS. Then F induces the NIFS

$$N(F) = (\{v_0\}, F, s_F, t_F, in, v_0),$$

with $s_F(e) = t_F(e) = v_0$ for all $e \in F$ and $in : F \longrightarrow AFF(\mathbb{R}^d)$ is the inclusion, i.e. $in(f) = f$ for all $f \in F$.

It is easy to see that the Hutchinson sequence of the IFS and some initial image coincides with the image sequence of the induced NIFS.

Observation 4.3.1
Let F be an IFS, $N(F)$ its induced NIFS and B an initial image. Then $N(F)(B) = H_F(B)$.

Proof (by complete induction)
n=0. $N(F)(B)_0 = H_{N(F)}(B)_0(v_0) = B_0 = H_F(B)_0$

$\mathbf{n \to n+1}$.

$$
\begin{aligned}
N(F)(B)_{n+1} &= H_{N(F)}(B)_{n+1}(v_0) && \text{(by def. of image seq.)}\\
&= HO_{N(F)}(B_n)(v_0) && \text{(by def. of Hutch. seq.)}\\
&= \bigcup_{e\in F, t_F(e)=v_0} in(e)(B_n(v_0)) && \text{(by def. of Hutch. op.)}\\
&= \bigcup_{f\in F} f(B_n(v_0)) && \text{(by def. of } in\text{)}\\
&= \bigcup_{f\in F} f(HO_{N(F)}(B_{n-1})(v_0)) && \text{(by def. of image seq.)}\\
&= \bigcup_{f\in F} f(H_{N(F)}(B)_n(v_0)) && \text{(by def. of Hutch. seq.)}\\
&= \bigcup_{f\in F} f(N(F)(B)_n)\\
&= \bigcup_{f\in F} f(H_F(B)_n) && \text{(induction hypothesis)}\\
&= H_F(B)_{n+1} && \text{(by def. of Hutch. seq. for IFS)}
\end{aligned}
$$

□

4.4 Example

The Sierpinski gasket as NIFS in this sense looks as shown in Figure 11.

4.5 About Cycles in the NIFS Graph

Does an NIFS always yield a non-empty image sequence starting with a non-empty image vector? This section deals with this question and we will see that in this context cycles play an important role. Two theorems are formulated considering NIFS under the aspect of the existence of cycles. The first theorem states that an acyclic NIFS yields an empty image sequence after a specific number of iterations (starting with a non-empty image vector). The existence of cycles in an NIFS guarantees that at least one component in each image vector of the image sequence is non-empty, which is stated by the second theorem.

First of all, we need to define some concepts. If an NIFS N is acyclic then there must be at least one vertex v_s with $\{e \mid t(e) = v_s\} = \emptyset$ and at least one vertex v_e with $\{e \mid s(e) = v_e\} = \emptyset$. This follows from the finiteness of V. We will call such vertex v_s a *start vertex* and v_e an *end vertex*. An isolated vertex is both a start vertex and an end vertex.

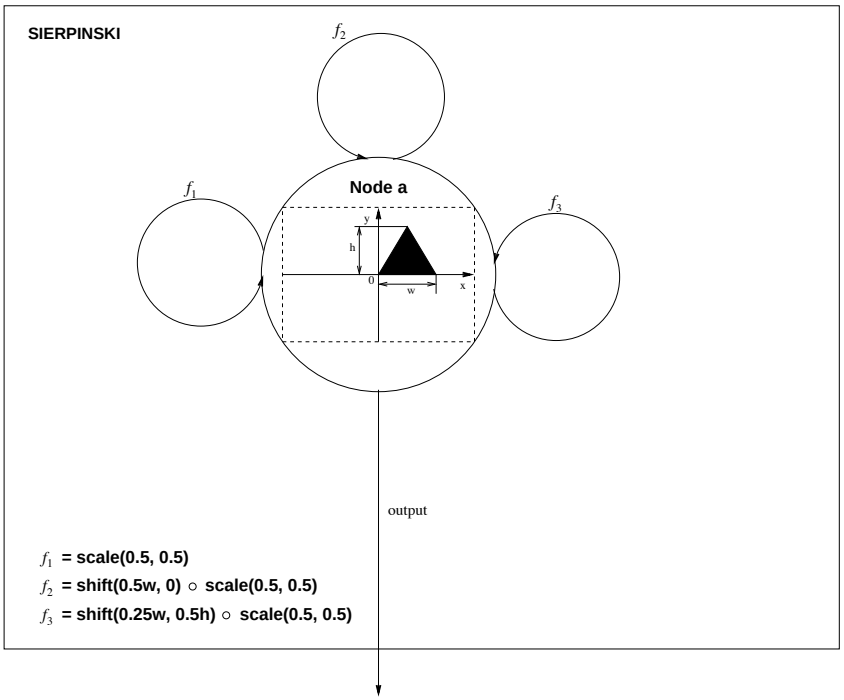

Figure 11. The Sierpinski gasket as an NIFS

The set of all paths $p = v_1, ..., v_k$, $k \leq |V|$ with start vertex v_1 and end vertex v_k is denoted by P_N. It is clear that P_N is finite. For a directed path $p = v_1, ..., v_k$ we denote the number of vertices on p by *nov(p)=k*. For the set P_N of an acyclic NIFS N we set

$$lp(N) = \max\{nov(p) \mid p \in P_N\}$$

representing the number of vertices on the longest path in N.

Theorem 4.5.1
Let N be an acyclic NIFS with an initial image vector B with $B(v) \neq \emptyset$ for all $v \in V$. Let $n_0 = lp(N)$. Then the images of the Hutchinson sequence are empty up to an initial section, i.e. $HO_N(B)_n(v) = \emptyset$ for all $n > n_0$ and $v \in V$.

For the proof of this theorem we need the following claim.

Claim
Let N be an acyclic NIFS that consists of one directed path p. Suppose there exists a vertex $v \in V$ with $B_n(v) \neq \emptyset$, $n \in \mathbb{N}$. Then follows that $n < nov(p)$.

This will be shown by the proof of the equivalent proposition below.

Proposition
Let N be an acyclic NIFS that consists of one directed path $p = v_1, ..., v_k$ with $B_0(v_i) \neq \emptyset$ for all i, $1 \leq i \leq k$ and $n \geq nov(p)$, $n \in \mathbb{N}$. Then $B_n(v_i) = \emptyset$ for all i, $1 \leq i \leq k$.

Proof (by complete induction)
k=1. $p = v_1$, $E = \emptyset$, $nov(p) = 1$. Then

$$B_1(v_1) = HO_N(B_0)(v_1) = \bigcup_{e \in E, t(e) = v_1} f(e)(B_0(s(e))) = \emptyset$$

$\Rightarrow B_n(v_1) = \emptyset$, $n \geq 1$.

k $\rightarrow$ k + 1. $p = v_1, ..., v_{k+1}$.
According to the induction hypothesis, $B_k(v_i) = \emptyset$, for all i, $1 \leq i \leq k$.

$$B_{k+1}(v_{k+1}) = HO_N(B_k)(v_{k+1}) = f(e)(B_k(v_k)) = f(e)(\emptyset) = \emptyset$$

where $s(e) = v_k$, $t(e) = v_{k+1}$. Hence $B_{k+1}(v_i) = \emptyset \Rightarrow B_n(v_i) = \emptyset$ (for all i, $1 \leq i \leq k+1$, $n \geq k+1 = nov(p)$). □

Proof of the theorem
Suppose there exists a vertex $v \in V$ with $B_n(v) \neq \emptyset$, $n \geq n_0$. There must exist at least one path $p \in P_N$ with $v \in p$. From the claim follows $n_0 \leq n < nov(p) \leq lp(N) \Rightarrow n_0 < lp(N)$. That is a contradiction to the definition of n_0. □

Theorem 4.5.2
Let N be an NIFS with at least one directed cycle $c = v_0, ..., v_{k-1}$, $v_i \in V$ and $B_0(v) \neq \emptyset$ for all $v \in V$. Then for each vertex $v \in c$ holds $B_i(v) \neq \emptyset$ for all $i \in \mathbb{N}_0$.

Proof (by complete induction)
n=0. $B_0(v) \neq \emptyset$ for all $v \in V$ is the precondition.

n $\rightarrow$ n + 1. For each vertex $v_j \in c$ $(0 \leq j \leq k-1)$ exists $e' \in E$ with $s(e') = v_{(j-1 \bmod k)}$ and $t(e') = v_{(j \bmod k)}$. That implies for all $v \in c$:

$$\begin{aligned} B_{n+1}(v) &= HO_N(B_n)(v) \\ &= f(e')(B_n(s(e'))) \cup \bigcup_{e \in E - e', t(e) = v} f(e)(B_n(s(e))) \neq \emptyset \end{aligned}$$

because $f(e')(B_n(s(e'))) \neq \emptyset$ follows from the induction hypothesis. □

As a consequence an NIFS N should contain directed cycles to avoid that the image vector becomes empty eventually. In the previous examples, the Sierpinski gasket and the Hilbert curve, both NIFSs had cycles consisting of single edges. Therefore they yield an image vector sequence all images of which are not empty. The example in Section 4.6 illustrates the effect of cycles that are not just loops. We again consider the Sierpinski gasket as NIFS, but now with three vertices that have different start images. The vertices exchange their images mutually.

4.6 Example

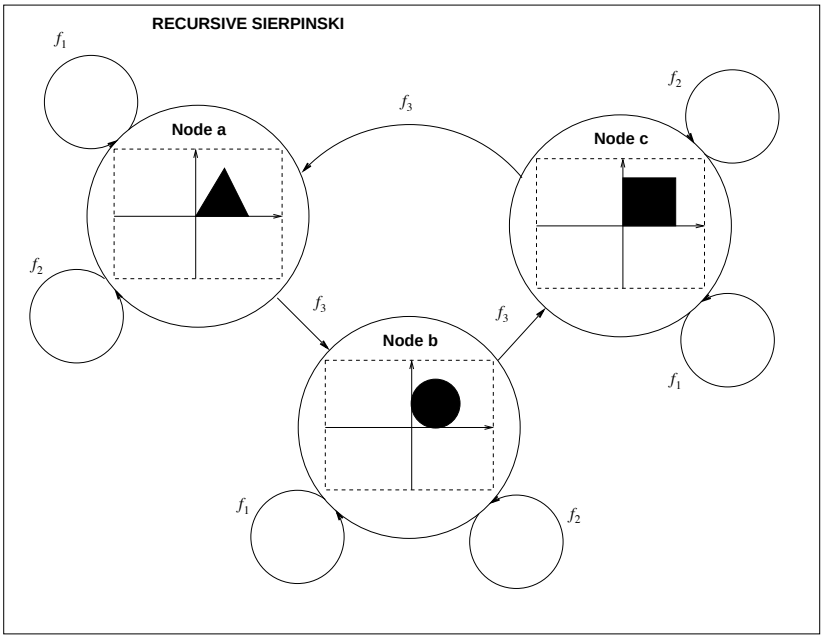

Figure 12. The Sierpinski gasket as an NIFS with three vertices

The NIFS in Figure 12 yields the three images (shown in Figure 13) after the first application of the Hutchinson operator in the order **Node a**, **Node b** and **Node c** as output. The second and the third iteration steps with the same order of outputs are shown in Figures 14 and 15.

Figure 13. The first iteration

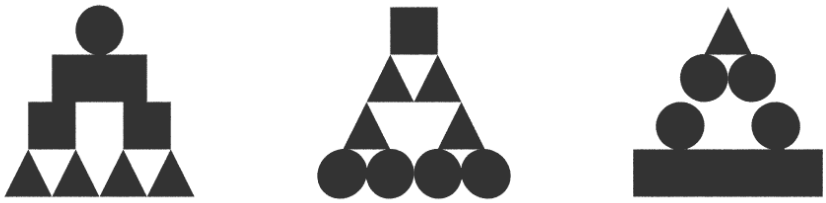

Figure 14. The second iteration

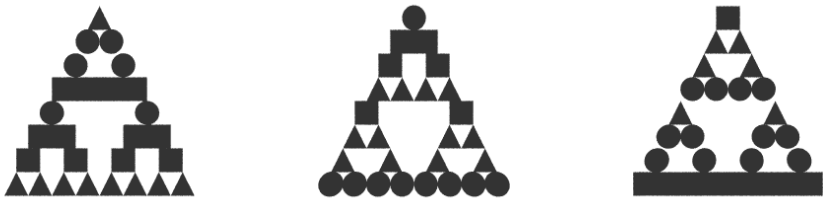

Figure 15. The third iteration

4.7 Composition of NIFSs

Our idea is to model a concept that allows to use the output *N(n)* (see Section 4.1) of an NIFS N for the initial image vector of another NIFS.

As shown in the theorem of Section 4.1, the attractor of a contracting NIFS does not depend on the initial vector. Therefore it does not make sense to strive for the attractor to achieve an "interesting" image with our idea of composition. For example we can use **hilbert(3)** as initial image $B_0(v_0)$ for **sierpinski(3)**. The result is shown in Figure 16.

5 Conclusion

In this chapter we have seen that the well-known concept of IFS can be generalized to the concept of NIFS. With NIFS we can generate more complex image sequences, the members of which do not need to be affine refinements of their predecessors. We saw that cycles play an important role if we are interested in a non-empty image as attractor of a contracting NIFS.

The concept of composition is more interesting under the practical aspect than under the theoretical. This idea was just intended to construct funnier images.

Future work could be the development of more complex and good-looking examples of NIFSs, for example NIFSs with several cycles that are not just loops, or further composition examples. Another idea is to extend the concept of NIFSs to coloured images so that the transformations also change colours.

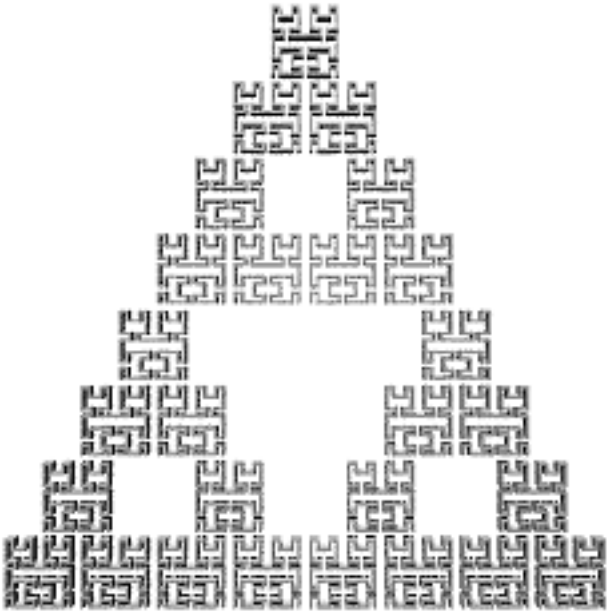

Figure 16. The third iteration of the Sierpinski gasket starting with **hilbert(3)**

5 Tree-based Collage Generation*

Frank Drewes

1 Introduction

In this chapter picture generation is studied from the point of view of formal language theory. A general framework for the generation of picture languages, called *tree-based picture generation*, is discussed. Roughly speaking, a tree-based picture generator is a tree generator together with an algebra that interprets trees as expressions denoting pictures. (Here, the word 'tree' is used as a synonym for 'term'. Readers who feel uncomfortable with this terminology may substitute one for the other wherever it occurs in the chapter. The term 'tree' is chosen here in order to comply with the terminology found in the literature on tree grammars and tree transducers.)

In principle, the tree generator which is part of a tree-based picture generator can be any device that defines a tree language. However, in this chapter we shall concentrate on regular and context-free tree grammars and on top-down tree transducers, as introduced by Rounds and Thatcher [Rou69, Rou70a, Rou70b, Tha70]. A concrete class of tree-based picture generators is obtained by selecting a class of tree generators and a picture algebra. For the latter, a notion of pictures and a set of operations on them is needed. The tree generator can be perceived as a syntactic device that generates derivation trees (which, as such, have no particular meaning) while the algebra is the semantic part which associates with every derivation tree the corresponding picture.

The idea to associate a tree grammar or tree transducer with an algebra that maps trees into a semantic domain was already mentioned in [Eng80]. For graphs generated by node or hyperedge replacement this has been worked out in [Eng94] (see also [CE95, DE98, Dre98a]). For the field of picture generation the idea seems to be new, however.

In this chapter collage grammars [HK91] are translated into tree-based picture generators, and the equivalence of the traditional device with its

* This chapter on collage grammars is an excerpt from [Dre00], where collage grammars [HK91], mutually recursive function systems (or hierarchical iterated function systems; cf. [Bar88, CD93, PJS92]), context-free chain-code grammars [MRW82], and 0L-systems with turtle interpretation [PL90] are translated into tree-based picture generators. Proofs are omitted here and can be found in the original publication.

tree-based counterpart is shown. This establishes a sound formal basis for future work as well as for earlier work in [DKL02, Dre96a, Dre96b, Dre98b], where tree-based definitions of picture-generating devices were already used.

The tree-based formulation of picture-generating devices turns out to be useful for several reasons. First, it gives some insight into the conceptual similarities and differences between devices that can be translated into this framework. Furthermore, it makes it easy to generalise (or restrict) picture-generating devices—and to distinguish between generalisations on the syntactic and on the semantic level. However, the most useful advantages are probably the proof-technical ones. As a common framework for different methods of picture generation the tree-based formalism simplifies the comparison of different classes with respect to their generative power, for example. Moreover, proofs can be written in a tree-oriented way (cf. [Dre96b]), which is often convenient as it allows to benefit from a variety of known results from the theory of tree grammars and tree transducers. In addition, proofs of similar statements for closely related sorts of devices (like collage grammars and IFSs in [Dre96b]) need only be written once, which is hard to achieve using the traditional definitions because they use quite different basic notions.

The chapter is structured as follows. The next section contains the preliminaries. In Section 3 the required definitions and results concerning tree generators are recalled and the notion of tree-based picture generation is defined formally. In Section 4 it is shown that collage grammars have an equivalent definition in terms of tree-based picture generators. Furthermore, some of the possible generalisations are mentioned. Section 5 contains a short conclusion.

All pictures shown in the examples of the following sections have been produced with TREEBAG [Dre98b], a software system which is based on the ideas presented here.

2 Preliminaries

2.1 Basic Mathematical Notation

The sets of all natural numbers (including 0) and of all real numbers are denoted by $\mathbb{N}$ and $\mathbb{R}$, respectively, and $\mathbb{N}_+$ denotes $\mathbb{N}\setminus\{0\}$. For every $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. For a set S, $\wp(S)$ denotes the powerset of S and $|S|$ denotes its cardinality. The set of all sequences (also called strings or words) over a set S is denoted by S^*. Furthermore, $S^+ = S^* \setminus \{\lambda\}$, where λ denotes the empty sequence. Concatenation of sequences is denoted by juxtaposition. For every $n \in \mathbb{N}$ the set of all sequences of length n in S^* is denoted by S^n. The length of a sequence w is denoted by $|w|$. If $f\colon S \to T$ is a function then the canonical extensions of f to $\wp(S)$ and to S^* are denoted by f, too. Thus, $f(S') = \{f(s) \mid s \in S'\}$ for all $S' \subseteq S$ and $f(s_1 \cdots s_n) = f(s_1) \cdots f(s_n)$ for all $s_1, \ldots, s_n \in S$.

For a relation $r \subseteq S \times T$ and $s \in S$, $r(s)$ denotes the set $\{t \in T \mid (s,t) \in r\}$. Furthermore, $r(S')$ denotes the set $\bigcup_{s \in S'} r(s)$ for $S' \subseteq S$. By convention, if $r(s)$ is a singleton $\{t\}$ then one may write $r(s) = t$. In particular, if $|r(s)| \leq 1$ for every $s \in S$ then r is considered as a partial function. The composition of r with another binary relation $r' \subseteq T \times U$ is given by $r' \circ r = \{(s,u) \in S \times U \mid (s,t) \in r \text{ and } (t,u) \in r' \text{ for some } t \in T\}$ (which applies to functions as well, as functions are special binary relations). The n-fold composition of a relation $r \subseteq S \times S$ with itself is denoted by r^n and its reflexive and transitive closure is denoted by r^*.

2.2 Signatures and Trees

A *ranked symbol* is a pair (f,n) consisting of a symbol f and a number $n \in \mathbb{N}$, its *rank*. A ranked symbol (f,n) is denoted as $f^{(n)}$ or simply f, and is usually just called a symbol. Notice, however, that $f^{(m)}$ and $f^{(n)}$ are different for $m \neq n$, even if both may be denoted by f. A *signature* is a (possibly infinite) set Σ of symbols.

A (labelled and ordered) *tree* is a pair consisting of a root symbol $f^{(n)}$ and n *direct subtrees* $t_1, \ldots, t_n$, which are trees. Such a tree is denoted by $f[t_1, \ldots, t_n]$. In case $n = 0$ one may abbreviate $f[]$ as f. Notice that, by this convention, every symbol of rank 0 is identified with the single-node tree whose root is labelled with this symbol. A signature or tree is said to be *monadic* if no symbols of rank 2 or greater occur in it. The *height* of a tree t is the maximum length of a path from the root to a leaf. Thus, if t is a single symbol of rank 0 then its height is 0. Otherwise, if m is the maximum height of its direct subtrees then the height of t is $m+1$.

If T is a set of trees and Σ a signature then the set $\mathrm{T}_\Sigma(T)$ of *trees over Σ with subtrees in T* is defined to be the smallest set of trees containing T and, for every $f^{(n)} \in \Sigma$ and all $t_1, \ldots, t_n \in \mathrm{T}_\Sigma(T)$, the tree $f[t_1, \ldots, t_n]$. The set $\mathrm{T}_\Sigma(\emptyset)$ of *trees over Σ* is denoted by T_Σ. Furthermore, $\Sigma(T) = \{f[t_1, \ldots, t_n] \mid f^{(n)} \in \Sigma \text{ and } t_1, \ldots, t_n \in T\}$. Symbols of rank 2 may be used as infix symbols associating to the right. Thus, $t_1 + t_2 + t_3$ denotes the tree $+[t_1, +[t_2, t_3]]$, for example. Furthermore, if f has rank 1 then $f^i[t]$ denotes t for $i = 0$ and $f[f^{i-1}[t]]$ for $i \geq 1$.

A set L of trees is called a *tree language* if $L \subseteq \mathrm{T}_\Sigma$ for some finite signature Σ.

The *yield* of a tree t is the string $\mathit{yield}(t)$ obtained by reading its leaves from left to right. In addition, there is one special symbol ε which, when it occurs as a leaf, denotes the empty string. Formally, $\mathit{yield}(\varepsilon) = \lambda$, $\mathit{yield}(f) = f$ for all symbols $f^{(0)} \neq \varepsilon$, and $\mathit{yield}(f[t_1, \ldots, t_n]) = \mathit{yield}(t_1) \cdots \mathit{yield}(t_n)$ for all trees $f[t_1, \ldots, t_n]$ with $n \geq 1$.

2.3 Substitution and Rewriting

For the rest of this chapter let $X = \{x_1, x_2, \ldots\}$ be a signature of pairwise distinct symbols called *variables* and denote by X_n its subset $\{x_1, \ldots, x_n\}$, for every $n \in \mathbb{N}$. In order to avoid confusion, variables are considered as special symbols that always have the rank 0 and are not allowed to occur in ordinary signatures. Thus, for a signature Σ, Σ and X are disjoint and $\mathrm{T}_\Sigma(X)$ is the set of all trees over $\Sigma \cup X$.

For $Y \subseteq X$ a tree t is said to be *linear in* Y if every $y \in Y$ occurs at most once in t and is called *nondeleting in* Y if every $y \in Y$ occurs at least once in t.

For trees $t, t_1, \ldots, t_n$, $t[\![t_1 \cdots t_n]\!]$ denotes the substitution of the tree t_i for the variable x_i in t ($i \in [n]$). More precisely, if $t = x_i$ for some $i \in [n]$ then $t[\![t_1 \cdots t_n]\!] = t_i$ and if $t = f[s_1, \ldots, s_k]$ with $f \notin X_n$ then $t[\![t_1 \cdots t_n]\!] = f[s_1[\![t_1 \cdots t_n]\!], \ldots, s_k[\![t_1 \cdots t_n]\!]]$. A (left-linear) *rewrite rule* is a pair $\rho = (l, r)$ of trees, called the *left-* and *right-hand side*, respectively, such that l is linear in X and every variable in r occurs in l, too. Such a rule is usually denoted by $l \to r$. Consider some $n \in \mathbb{N}$ such that X_n contains all variables that occur in l. Then, ρ determines the binary relation $\to_\rho$ on trees such that $t \to_\rho t'$ if t can be written as $t_0[\![l[\![t_1 \cdots t_n]\!]]\!]$ for a tree t_0 containing x_1 exactly once, and t' equals $t_0[\![r[\![t_1 \cdots t_n]\!]]\!]$. If R is a set of rewrite rules, $\to_R$ denotes the union of all $\to_\rho$ such that $\rho \in R$. As usual, $t \to_R t'$ is called a *derivation step* and a sequence $t_0 \to_R t_1 \to_R \cdots \to_R t_k$ ($k \geq 0$) of derivation steps is a *derivation*.

2.4 Algebras

If Σ is a signature, a *Σ-algebra* (which is just called an algebra if Σ is of minor importance) is a pair $\mathcal{A} = (\mathbb{A}, (f_\mathcal{A})_{f \in \Sigma})$, where $\mathbb{A}$ is a set, the *domain* of $\mathcal{A}$, and for every $f^{(n)} \in \Sigma$, $f_\mathcal{A} \colon \mathbb{A}^n \to \mathbb{A}$ is an n-ary operation on $\mathbb{A}$, the *interpretation* of f in $\mathcal{A}$. Operations of arity 0 are called *constants*. The *value* $val_\mathcal{A}(t)$ of a tree $t \in \mathrm{T}_\Sigma$ with respect to a Σ-algebra $\mathcal{A}$ is defined as usual: if $t = f[t_1, \ldots, t_n]$ then $val_\mathcal{A}(t) = f_\mathcal{A}(val_\mathcal{A}(t_1), \ldots, val_\mathcal{A}(t_n))$. If the algebra in question is clear from the context then $val(t)$ abbreviates $val_\mathcal{A}(t)$.

In order to simplify things, when defining concrete algebras in subsequent sections we shall always consider Σ-algebras $\mathcal{A}$ such that the symbols in Σ are operations (where ranks coincide with arities) and $f_\mathcal{A} = f$ for all $f \in \Sigma$. Thus, every operation is denoted by itself. Due to this convention, it suffices to select a domain and a set of operations in order to define an algebra.

3 Tree and Picture Generators

In this section the classes of tree generators considered in this chapter are recalled and the way in which they can produce pictures is made precise.

A *tree generator* is either a tree grammar or a tree transducer. A *tree grammar* is any sort of grammatical device g that directly generates a tree

language $\mathrm{L}(g)$ over an output signature Σ'. A *tree transducer* is a device τ that computes a relation $\tau \subseteq \mathrm{T}_\Sigma \times \mathrm{T}_{\Sigma'}$, where Σ and Σ' are the input and output signatures, respectively. For such a tree transducer the range is considered as the generated language: $\mathrm{L}(\tau) = \tau(\mathrm{T}_\Sigma)$.

A tree generator can be turned into a picture-generating device by viewing the generated trees as expressions that denote pictures, i.e., by associating it with an algebra whose domain is a set of pictures. Let $\mathcal{P}$ be such a Σ-algebra and let g be a tree generator whose output signature is a subset of Σ. Then the pair $(g, \mathcal{P})$ is called a *tree-based picture generator*. It generates the picture language $\mathcal{L}_\mathcal{P}(g) = val_\mathcal{P}(\mathrm{L}(g))$. If there is no reason to expect confusion one may identify $(g, \mathcal{P})$ with g and denote $\mathcal{L}_\mathcal{P}(g)$ by $\mathcal{L}(g)$.

Notice that the terminology compiled above neither requires nor yields any specific notion of pictures. Strictly speaking, one could even generate objects that hardly anyone would agree to call pictures—like strings, numbers, or truth values. Here, however, we shall concentrate on the investigation of picture generators that really deal with pictures and thus deserve this name.

From a conceptual point of view maybe the most central characteristic of the tree-based approach to picture generation is that syntax and semantics are considered separately as far as possible. The syntactic aspects are dealt with on the level of tree generators while the semantics of the generated trees is determined by the chosen algebra. Intuitively, the tree generator yields the set of valid derivation trees of pictures, but it does not fix an interpretation. Conversely, the algebra determines the meaning of trees, but it does not say how these trees are to be generated.

The main classes of tree generators considered in the following are regular and context-free tree grammars, and top-down tree transducers. These notions were first studied by Rounds and Thatcher in [Rou69, Rou70a, Rou70b, Tha70]; see [GS84, GS97] for an introduction to the field and for many further references. In order to obtain the main result of this chapter regular tree grammars will turn out to be sufficient. Context-free tree grammars and top-down tree transducers will only occur in examples showing possible generalisations.

Definition 1 (*context-free* and *regular tree grammar*). A *context-free tree grammar* is a tuple $g = (N, \Sigma, P, A_0)$ consisting of a finite signature N of *nonterminals*, a finite *output signature* Σ which is disjoint with N, a finite set $P \subseteq N(X) \times \mathrm{T}_{\Sigma \cup N}(X)$ of *productions*, and an *initial nonterminal* $A_0 \in N$ of rank 0. The *context-free tree language* generated by g is

$$\mathrm{L}(g) = \{t \in \mathrm{T}_\Sigma \mid A_0 \overset{*}{\underset{g}{\to}} t\},$$

where $\to_g$ denotes $\to_P$.[1] If all symbols in N are of rank 0 then g is called a *regular tree grammar* and $\mathrm{L}(g)$ is a *regular tree language*.

[1] This definition corresponds to the outside-in or unrestricted derivation mode studied in the literature (cf. [ES77, ES78]). The inside-out mode will not be considered in this chapter.

Notice that the productions of a regular tree grammar are of the form $A \to t$ where $A \in N$ and $t \in \mathrm{T}_\Sigma(N)$. Thus, nonterminals occur only at the leaves. By contrast, in a context-free tree grammar variables can occur anywhere in a derived tree. Thus, the monadic cases correspond to regular and context-free string grammars, respectively (identifying a monadic tree with a string in the obvious way).

It is sometimes useful to assume that all productions of a regular tree grammar $g = (N, \Sigma, P, A_0)$ have the form $A \to f[A_1, \ldots, A_k]$, where $f^{(k)} \in \Sigma$ and $A, A_1, \ldots, A_k \in N$ (in other words, $P \subseteq N \times \Sigma(N)$). Such a regular tree grammar is said to be *in normal form*. The following lemma is well known and can be proved by standard techniques, using basically the same construction as in the string case.

Lemma 1. *Every regular tree grammar g can effectively be transformed into a regular tree grammar g' in normal form such that* $\mathrm{L}(g) = \mathrm{L}(g')$.

There is a close relationship between regular tree grammars and sets of derivation trees of context-free Chomsky grammars. Every regular tree language is a projection of a derivation-tree language of a context-free grammar. Conversely, the set of derivation trees of a context-free grammar is a regular tree language. These quite obvious facts amount to the following characterization of context-free languages (see, e.g., [GS97, Prop. 14.2 and 14.3]).

Lemma 2. *A string language L is context-free if and only if $L = yield(L')$ for some regular tree language L'.*

It was indicated above that the regular tree grammar is quite a direct generalisation of the regular string grammar. Likewise, the concept of a top-down tree transducer is a natural generalisation of generalised sequential machines (cf. the title of [Tha70]). Instead of processing a string in a left-to-right manner, an input tree is transformed into an output tree starting at the root and proceeding towards the leaves.

Definition 2 (*top-down tree transducer*). Let Σ and Σ' be finite signatures and let Γ be a finite signature of symbols of rank 1 called *states*, disjoint with $\Sigma \cup \Sigma'$. A *top-down tree transducer* is a tuple $\tau = (\Sigma, \Sigma', \Gamma, R, \gamma_0)$ such that $\gamma_0 \in \Gamma$, called the *initial state*, and $R \subseteq \Gamma(\Sigma(X)) \times \mathrm{T}_{\Sigma'}(\Gamma(X))$ is a finite set of rewrite rules.

The *top-down tree transduction* computed by τ, which will also be denoted by τ, is given by

$$\tau(t) = \{t' \in \mathrm{T}_{\Sigma'} \mid \gamma_0[t] \underset{\tau}{\overset{*}{\to}} t'\}$$

for all $t \in \mathrm{T}_\Sigma$, where $\to_\tau$ denotes the rewrite relation $\to_R$.

By the definition above the left-hand side of a rule of a top-down tree transducer always has the form $\gamma[f[x_{i_1}, \ldots, x_{i_k}]]$ where $\gamma \in \Gamma$, $f^{(k)} \in \Sigma$, and $x_{i_1}, \ldots, x_{i_k}$ are variables (which, by the definition of rewrite rules in

the previous section, are pairwise distinct). From now on it will be assumed without loss of generality that $i_j = j$ for all $j \in [k]$, i.e., the variables used are $x_1, \ldots, x_k$ and they are numbered from left to right.

In order to simplify the denotation of trees, and in particular of rules, the following conventions will be employed throughout the rest of this chapter.

1. A tree of the form $\gamma[t]$, where γ is a state, is denoted by γt.
2. The left-hand side $\gamma f[x_1, \ldots, x_n]$ of a rule is denoted by γf.
3. In the case of a top-down tree transducer whose input signature is monadic the variable x_1, which is the only one that occurs in the rules, is omitted also in the right-hand sides. Thus, in this case the rule

$$\gamma f \to g[h[\gamma' x_1, \gamma'' x_1], \gamma' x_1]$$

would be written $\gamma f \to g[h[\gamma', \gamma''], \gamma']$, for example. (This is slightly ambiguous when there is a symbol $\gamma^{(0)} \in \Sigma'$ such that $\gamma^{(1)} \in \Gamma$. Therefore, such a situation should be avoided.)

A rule with left-hand side γf is also called a γf-rule, or just f-rule if only f is of interest. A top-down tree transducer as in the definition is *total* if R contains at least one γf-rule for every $\gamma \in \Gamma$ and $f \in \Sigma$, and is *deterministic* if it contains at most one γf-rule for every such pair. It is *linear* if all right-hand sides of rules in R are linear in X and *nondeleting* if the right-hand side of every $f^{(k)}$-rule in R is nondeleting in X_k. Finally, it is called *producing* if there are no rules with right-hand sides in $\Gamma(X)$ (i.e., if each application of a rule produces at least one output symbol) and *one-producing* if all right-hand sides are elements of $\Sigma(\Gamma(X))$ (i.e., if each application of a rule produces exactly one output symbol).

The lemma below, which is the tree version of the well-known result saying that the class of regular string languages is closed under generalised sequential machine maps, will be useful.

Lemma 3 (cf. [Tha70]). *For every linear top-down tree transducer τ and every regular tree language L the image $\tau(L)$ of L under τ is regular.*

In this chapter, top-down tree transducers will mainly be used as tree generators, i.e., we are mostly interested in the set $\mathrm{L}(\tau)$ of output trees rather than in the computed input-output relation. In this situation the input trees usually play the role of control information that can be used to maintain dependencies between different branches of the output tree. As an example, consider the top-down tree transducer $\tau = (\Sigma_s, \Sigma, \{\gamma_0\}, R, \gamma_0)$ where $\Sigma_s = \{s^{(1)}, 0^{(0)}\}$, $\Sigma = \{.^{(2)}, a^{(0)}, b^{(0)}\}$, and $R = \{\gamma_0 s \to \gamma_0 \cdot \gamma_0, \gamma_0 0 \to a, \gamma_0 0 \to b\}$.[2] Then, $\mathrm{L}(\tau)$ is the set of all fully balanced binary trees in T_Σ, which is a non-regular tree language and is obtained by using the input tree as a kind of counter that determines the depth of the output tree.

[2] Recall that the occurrences of x_1 are omitted in the right-hand sides since the input signature Σ_s is monadic.

4 Collage Grammars

In this section a formulation of collage grammars as tree-based picture generators will be given and it will be shown that it is equivalent to the original definition. Collage grammars were introduced by Habel and Kreowski in [HK91] and their properties have been studied in a variety of papers (see, e.g., [HKT93, DHKT95, DK96, Dre96b]). As their name indicates, collage grammars generate picture languages which consist of so-called collages. This notion is defined next.

Consider some arbitrary dimension $d \in \mathbb{N}$, which is supposed to be fixed throughout this section. Intuitively, a collage is a finite set of geometric parts, every part being a bounded subset of $\mathbb{R}^d$. For technical reasons which will soon become clear, a finite sequence of so-called *pin points* is added. Thus, formally, a *collage* is a pair $(PART, pin)$ where $PART$ is a finite set of bounded subsets of $\mathbb{R}^d$ and $pin \in (\mathbb{R}^d)^*$ is a sequence of points. The elements of the first component are called the *parts* of the collage and the points in the sequence pin are its *pin points*. A collage $(PART, \lambda)$, i.e., one with an empty sequence of pin points, will be identified with the set $PART$ of parts in the following. A *collage language* is a set L of collages such that $pin_C = pin_{C'}$ for all $C, C' \in L$.

Remark 1. The original definition of collages given in [HK91] allows also unbounded parts and infinite sets of parts to be used in collages. The restriction to finite sets of bounded parts, which stems from more recent papers (cf. [Dre96b, DKL02]), has been adopted here because it seems natural. The results of this section do not depend on this restriction, however.

Now, the idea behind collage grammars is to augment collages with nonterminal place holders that can be replaced with other collages. These place holders are called *hyperedges*. Each of them is attached to a finite sequence of points and carries a label. A collage with such hyperedges in it is called a decorated collage. Formally, a *decorated collage* is a tuple $(PART, E, att, lab, pin)$ where $(PART, pin)$ is a collage, E is a finite set whose elements are called hyperedges, $att\colon E \to (\mathbb{R}^d)^*$ is a function that assigns to each hyperedge the sequence of attached nodes, and $lab\colon E \to M$ assigns to every hyperedge $e \in E$ a label $lab(e) \in M$ taken from a finite set M of labels. For every label A and every sequence pin of pin points $(A, pin)^\bullet$ and $(A, pin)^\circ$ denote the collages $(\emptyset, \{e\}, att, lab, pin)$ and $(\emptyset, \{e\}, att, lab, \lambda)$, respectively, where $att(e) = pin$ and $lab(e) = A$.

In the following, a collage is considered as a decorated collage whose set of hyperedges is empty. Furthermore, the attribute *decorated* will usually be dropped, speaking of collages also if, in fact, decorated collages are meant. If it is necessary to make a distinction, a collage without hyperedges will be called an *undecorated collage*. The five components of a collage C are also denoted by $PART_C$, E_C, att_C, lab_C, and pin_C, respectively. For a finite

alphabet N, $\mathcal{C}_N$ denotes the set of all collages C such that lab_C has the form $lab_C: E_C \to N$. Thus $\mathcal{C}_\emptyset$, which will be abbreviated by $\mathcal{C}$, denotes the set of all undecorated collages.

Intuitively, replacing a hyperedge by a collage C means to remove the hyperedge and insert C in its place. For this, C must be transformed in such a way that its pin points match the attached points of the hyperedge. Thus, strictly speaking, rather than inserting C one inserts a suitably transformed copy of C. For this, one has to agree upon an admissible set of transformations. Although in principle any set of transformations could be chosen, the most interesting one (and the one that has been considered in all the papers about collage grammars until now) is the set of affine transformations.

In order to define hyperedge replacement formally, some basic operations on collages are needed. Let C be a collage. For a set $E \subseteq E_C$ of hyperedges, $C - E$ denotes their deletion from C, i.e.,

$$C - E = (PART_C, E_C \setminus E, att, lab, pin_C)$$

where att and lab are the restrictions of att_C and lab_C to $E_C \setminus E$. If C' is a second collage then

$$C + C' = (PART_C \cup PART_{C'}, E, att, lab, pin_C)$$

where E is the disjoint union of E_C and $E_{C'}$,[3] and for all $e \in E$

$$att(e) = \begin{cases} att_C(e) & \text{if } e \in E_C \\ att_{C'}(e) & \text{otherwise} \end{cases} \quad \text{and} \quad lab(e) = \begin{cases} lab_C(e) & \text{if } e \in E_C \\ lab_{C'}(e) & \text{otherwise.} \end{cases}$$

Note that a is associative, but it does not commute in general because $C + C'$ inherits the pin points of C. For collages $C, C_1, \ldots, C_k$ the sum $C + C_1 + \cdots + C_k$ is abbreviated as $C + \sum_{i=1}^{k} C_i$. Finally, if a is an affine transformation on $\mathbb{R}^d$ then $a(C) = (a(PART_C), E_C, att, lab_C, a(pin_C))$ where $att(e) = a(att_C(e))$ for all $e \in E_C$.

Consider a collage C, pairwise distinct hyperedges $e_1, \ldots, e_k \in E_C$, and collages $C_1, \ldots, C_k$ such that for every $i \in [k]$ there is a unique affine transformation a_i that maps pin_{C_i} to $att_C(e_i)$. In this case $C[e_1/C_1, \ldots, e_k/C_k]$ denotes the collage obtained from C by replacing each of the hyperedges e_i with the transformed collage $a_i(C_i)$, i.e.,

$$C[e_1/C_1, \ldots, e_n/C_k] = (C - E) + \sum_{i=1}^{k} a_i(C_i).$$

In the following the notation $C[e_1/C_1, \ldots, e_k/C_k]$ is always meant to imply that the conditions are satisfied, i.e., that for every $i \in [k]$ there is a unique affine transformations mapping pin_{C_i} to $att_C(e_i)$.

Now, collage grammars and their generated languages can be defined.

[3] If E_C and $E_{C'}$ are not disjoint, an implicit renaming is assumed to take place.

Definition 3 (*collage grammar*). A *collage grammar* (in $\mathbb{R}^d$) is a system $G = (N, P, Z)$, where N is a finite set of hyperedge labels, $P \subseteq N \times \mathcal{C}_N$ is a finite set of *productions*, and $Z \in \mathcal{C}_N$ is the *start collage*. For collages C, C' we write $C \Longrightarrow_P C'$ (C *derives* C' *using* P) if there are hyperedges $e_1, \ldots, e_k \in E_C$ and productions $(A_1, R_1), \ldots, (A_k, R_k) \in P$ such that $C' = C[e_1/R_1, \ldots, e_k/R_k]$.

The *collage language generated by* G is $L(G) = \{C \in \mathcal{C} \mid Z \Longrightarrow_P^* C\}$.

Example 1. As an example, consider the collage grammar

$$G = (\{S, R\}, \{(S, C_S), (S, C'_S), (R, C_R), (R, C'_R)\}, (S, pin_{C_S})^\circ)$$

in $\mathbb{R}^2$, with right-hand sides of productions as depicted in Figure 1. Here, hyperedges are shown as labelled boxes which are connected to their attached

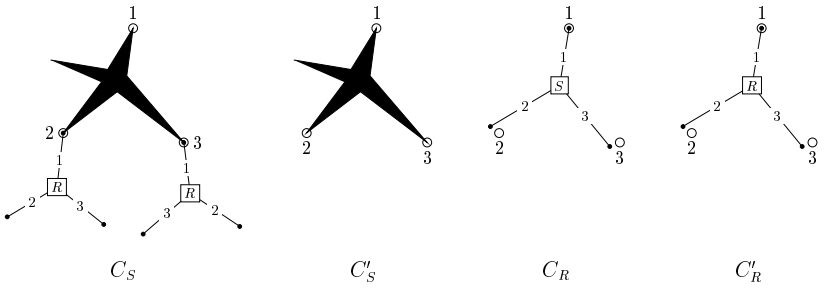

Figure 1. The right-hand sides of the productions used in Example 1

points by lines, where the order on the attached points is indicated by numbers. Pin points are indicated by circles and are also numbered. Three collages from the language generated by this collage grammar are shown in Figure 2.

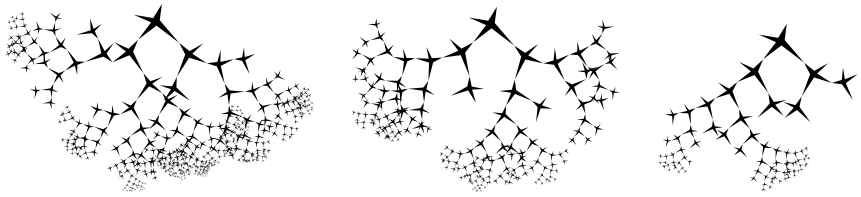

Figure 2. Some collages generated by the collage grammar of Example 1 (scaled by the factor 1/3)

In the following, the languages generated by collage grammars are called *context-free collage languages*, a terminology that will be justified by the context-freeness lemma, soon. As a direct consequence of the definitions above, all collages in a context-free collage language share their pin points—the reason being that they all inherit the pin points of the start collage of the generating grammar. Thus, a collage grammar indeed generates a collage language as defined earlier in this section. Furthermore, as the pin points of the start collage never play any role in the derivation process it is clear that they can be chosen arbitrarily. In other words, if L is a context-free collage language then $\{(PART_C, pin) \mid C \in L\}$ is context-free, too, for every sequence pin of pin points. Clearly, when studying context-free collage languages this means that it does not make sense to distinguish between languages that differ only in their sequences of pin points. Therefore, in the following two collage languages L and L' are considered to be equivalent, which is written $L \equiv L'$, if $\{PART_C \mid C \in L\} = \{PART_C \mid C \in L'\}$.

An inconvenient property of collage grammars is that, intuitively, a production (A, R) need not necessarily be applicable to an A-labelled hyperedge e in a derived collage C, due to the fact that there need not always be a *unique* affine transformation a satisfying $a(pin_R) = att_C(e)$. There may in fact not be any such transformation, or there may be infinitely many of them. In order to prevent this unpleasant situation the following notion of proper collage grammars is useful.

Definition 4 (*proper collage grammar*, cf. [HK91]). A collage grammar $G = (N, P, Z)$ is *proper* if for every $A \in N$ there is a sequence $pin(A)$ of pin points such that

i. $pin_R = pin(A)$ for all productions $(A, R) \in P$,
ii. for every production $(A, R) \in P$ and each hyperedge $e \in E_R$ there is a unique affine transformation a such that $a(pin(lab_R(e))) = att_R(e)$, and
iii. Z has the form $(A_0, pin(A_0))^\bullet$ for some $A_0 \in N$.

The following result from [HK91] states that every collage grammar can effectively be transformed into a proper collage grammar without affecting the generated language (up to equivalence).

Lemma 4 (cf. [HK91]). *For every collage grammar G one can effectively construct a proper collage grammar G' such that $L(G') \equiv L(G)$.*

For the purpose of the present chapter it is useful to prove a slightly strengthened variant of the lemma. Call a collage grammar $G = (N, P, Z)$ *uniformly proper* if it is proper and there is a sequence pin_G of pin points such that $pin(A) = pin_G$ for all $A \in N$.

Lemma 5. *For every collage grammar G one can effectively construct a uniformly proper collage grammar G' such that $L(G') \equiv L(G)$.*

In the following, whenever a uniformly proper collage grammar $G = (N, P, Z)$ is considered, $(A, pin_G)^\bullet$ will be abbreviated by $A^\bullet$ for all $A \in N$.

The results of this section will mainly turn out to be consequences of the so-called context-freeness lemma for collage grammars.

Lemma 6 (context-freeness lemma, cf. [HK91]). *Let $G = (N, P, Z)$ be a uniformly proper collage grammar and consider some $A \in N$, $C \in \mathcal{C}_N$, and $n \in \mathbb{N}$. There is a derivation $A^\bullet \Longrightarrow^{n+1} C$ if and only if $C = R[e_1/C_1, \ldots, e_k/C_k]$ for some production $(A, R) \in P$ and collages $C_1, \ldots, C_k$ such that $E_R = \{e_1, \ldots, e_k\}$ and $lab_R(e_i)^\bullet \Longrightarrow^{n_i} C_i$ for every $i \in [k]$, where $\sum_{i \in [k]} n_i = n$.*

Notice that the expression $R[e_1/C_1, \ldots, e_k/C_k]$ in the lemma is never undefined, due to the fact that G is assumed to be proper.

In order to translate the concept of collage grammars into the tree-based framework one has to make two choices. On the syntactic level a class of tree generators must be chosen while on the semantic level an appropriate algebra $\mathcal{P}_c$ must be found. Consider the semantic aspect first. The objects dealt with should of course be collages. However, since hyperedges and pin points merely play a technical role in the definition of the derivation process they turn out to be superfluous. Therefore, let $\mathbb{P}_c$ be the set of all undecorated collages in $\mathbb{R}^d$ without pin points, i.e., $\mathbb{P}_c$ consists of all finite sets of bounded subsets of $\mathbb{R}^d$. Bearing in mind the context-freeness lemma it is not hard to determine the required set of operations. For this, if $a_1, \ldots, a_k$ are affine transformations on $\mathbb{R}^d$, let $\langle\!\langle a_1 \cdots a_k \rangle\!\rangle$ denote the k-ary function $F\colon \wp(\mathbb{R}^d)^k \to \wp(\mathbb{R}^d)$ such that $F(p_1, \ldots, p_k) = a_1(p_1) \cup \cdots \cup a_k(p_k)$ for all $p_1, \ldots, p_k \subseteq \mathbb{R}^d$. F is extended to $\mathbb{P}_c^k$ in the canonical way, i.e., $F(C_1, \ldots, C_k) = a_1(C_1) \cup \cdots \cup a_k(C_k)$ for $C_1, \ldots, C_k \in \mathbb{P}_c$, where $a_i(C_i) = \{a_i(p) \mid p \in C_i\}$.

Now, the *collage algebra* $\mathcal{P}_c$ is the Σ_c-algebra with domain $\mathbb{P}_c$ such that Σ_c consists of

- all operations $\langle\!\langle a_1 \cdots a_k \rangle\!\rangle$, where $a_1, \ldots, a_k$ are affine transformations, and
- all elements of $\mathbb{P}_c$, viewed as constants.

Finally, it is probably already clear to the reader that the regular tree grammar is the tree-generating device which must be chosen on the syntactic level (cf. the seminal paper [MW67] by Mezei and Wright). Therefore, in the following we will consider the class of tree-based picture generators of the form $(g, \mathcal{P}_c)$, where g is a regular tree grammar and $\mathcal{P}_c$ is as defined above. Such a picture generator will be called a *tree-based collage grammar*. Roughly speaking, the tree language $\mathrm{L}(g)$ generated by g corresponds to the set of derivation trees of the corresponding collage grammar and vice versa.

The following two lemmas show that the tree-based version of collage grammars is indeed equivalent to the original one.

Lemma 7. *For every collage grammar G there is a tree-based collage grammar g such that $\mathcal{L}(g) \equiv L(G)$.*

Lemma 8. *For every tree-based collage grammar g there is a collage grammar G such that $L(G) \equiv \mathcal{L}(g)$.*

Combining Lemma 7 and Lemma 8 the main theorem of this section is obtained.

Theorem 1. *A collage language L is context-free if and only if $L \equiv \mathcal{L}(g)$ for some tree-based collage grammar g.*

The tree-based formulation of collage grammars makes it possible to define generalisations (or restrictions) in an easy way. Clearly, on the syntactic level one can replace regular tree grammars by more powerful devices like, for example, context-free tree grammars. This is exploited in the following example in order to generate a collage language that cannot be generated by an ordinary collage grammar (cf. [DKL02] for a proof of the latter).

Example 2. The aim is to generate the collage language shown in Figure 3, which consists of L-shaped approximations of a tiling presented in [GS89].

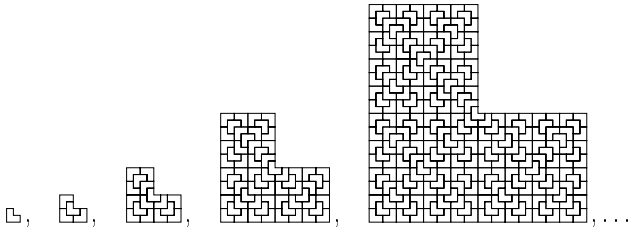

Figure 3. A collage language that can be generated using a suitable context-free tree grammar

The problems are twofold. Intuitively, each of the collages (except the first) is a composition of four suitably shifted and rotated copies of its predecessor. This indicates that one has to use an operation $F = \langle\!\langle a_1 a_2 a_3 a_4 \rangle\!\rangle$ of arity 4. However, the amount by which F has to translate three of its arguments is not fixed—it depends on the dimension of the arguments. As a consequence, an infinity of such operations would be needed in the naive approach. Furthermore, the second problem, all four arguments of F should be equal. This means that the tree generator should only generate fully balanced trees, which cannot be done by a regular tree grammars. This is possible using a context-free tree grammar, however. Moreover, in order to solve the first problem one may define F in such a way that it does not produce a larger collage from its arguments. Instead, the arguments are scaled down by the factor $1/2$ so that, all in all, the size stays the same. An additional operation S is then used to scale the resulting collage by the factor 2. The nonterminal that generates

the tree has rank 1. It produces a chain of S symbols above and a balanced tree of F symbols of the same height below itself.

To make this precise, let $\Sigma = \{F^{(4)}, S^{(1)}, C^{(0)}\} \subseteq \Sigma_c$ where C is the collage consisting of a single polygon with corners $(0,0)$, $(1,0)$, $(1,.5)$, $(.5,.5)$, $(.5,1)$, and $(0,1)$, S is a scaling by the factor 2, and F is given as follows. It scales all four arguments by the factor $1/2$. In addition, arguments two and three are rotated by 90 and -90 degrees[4] and translated by $(1,0)$ and $(0,1)$, respectively. The fourth argument is translated by $(1/4, 1/4)$ (in addition to the scaling).

Finally, let $g = (\{A^{(1)}, A_0^{(0)}\}, \Sigma, P, A_0)$ where

$$P = \{\ \begin{array}{ll} A_0 & \to A[C], \\ A[x_1] & \to x_1, \\ A[x_1] & \to S[A[F[x_1, x_1, x_1, x_1]]]\ \}. \end{array}$$

$\mathrm{L}(g)$ consists of all trees in T_Σ of the form $t_S[\![t_F]\!]$, where $t_S \in \mathrm{T}_{\{S\}}(X_1)$ is nondeleting in X_1 and $t_F \in \mathrm{T}_{\{F,C\}}$ is a fully balanced tree of the same height as t_S. Consequently, the generated collages are those indicated in Figure 3, as intended.

Another interesting possibility is to use more powerful algebras, i.e., to extend tree-based collage grammars on the semantic level. One may, for instance, consider operations $\langle\!\langle f_1 \cdots f_n \rangle\!\rangle$ where the f_i are not necessarily affine transformations. Obviously, such an extension does not provide any difficulty—one just has to choose the desired algebra. This may be interesting to notice because the original definition of collage grammars makes sense only if the considered class of transformations satisfies certain requirements. This is caused by the fact that transformations are determined implicitly by the (finitely many) attached points of hyperedges. Thus, it must be possible to determine a transformation uniquely by fixing the images of finitely many points. Furthermore, as pointed out in [HK91], in order to ensure that the grammars behave nicely one should require that the class of transformations forms a group (where composition of transformations is taken as the group operation). This is needed in order to ensure that the transformed hyperedges resulting from a replacement step still determine a unique admissible transformation. In the setting proposed here these difficulties have disappeared.

Of course, considering operations $\langle\!\langle f_1 \cdots f_n \rangle\!\rangle$ which contain non-affine transformations is not the only extension that one may think of. One could, for instance, add operations that build the intersection of two collages, or the difference. Another interesting possibility is to use coloured parts and to provide operations that allow to change the colours of the parts of an argument collage in some way.

[4] If not explicitly stated otherwise, rotation always means rotation about the origin.

5 Conclusion

A device that generates a tree language and an algebra that interprets these trees as expressions which denote pictures: these are the two components of a tree-based picture generator as discussed in this chapter. In Section 4 it was shown that certain choices of the tree-generating device and the algebra yield a tree-based equivalent of collage grammars. This provides a formal justification for the use of these tree-based definitions in [DKL02, Dre96a, Dre96b] and in the software system TREEBAG [Dre98b] which is based on these ideas (but not limited to the generation of pictures).

The tree-based definition of picture-generating devices has some advantages that may sometimes turn out to be useful. A proof-technical advantage is that constructions can often be formulated as tree transductions, as it was done in [Dre96b] (see also [Dre96a]). In these cases one can make use of known closure properties and other results from the theory of tree transductions in order to get concise proofs (cf. [DE98]).

Another point is that a construction or a proof idea may not only apply to a single type of picture-generating systems, but may be applicable to related devices as well. In such cases it is convenient to use a unified framework in order to avoid having to write proofs twice (see [Dre96a], again).

A closely related observation is that the tree-based approach is a suitable basis for the comparison of different methods of picture generation, both on the formal and on the conceptual level. For instance, on the semantic level tree-based collage grammars and IFSs are very similar as their algebras deal with pictures in $\mathbb{R}^d$ by applying transformations and taking unions.

6 3D Tilings

Michael Prahl

1 Introduction

In this paper, the notion of 3D tilings is introduced generalizing the usual concept of 2D tilings (see, e.g., Grünbaum and Shephard [GS89]). A 3D tiling is a countable set of 3D tiles that fill the three-dimensional Euclidean space $\mathbb{R}^3$ without gaps and overlaps. First of all, some basic notions are given. Then a construction method for 3D tilings is introduced and illustrated by some examples. The last two sections are concerned with a method to construct 3D tilings with collage grammars and especially with AnimaLab.

2 Basic Notions

A 3*D tiling* is a countable set $\mathcal{T}$ of 3D tiles that fill the $\mathbb{R}^3$ without gaps and overlaps, i.e.

$$T \subseteq \mathbb{R}^3 \text{ for all } T \in \mathcal{T}$$

$$\bigcup_{T \in \mathcal{T}} T = \mathbb{R}^3$$

$$T_1 \cap T_2 \subseteq dT_1 \cap dT_2 \text{ for } T_1, T_2 \in \mathcal{T} \text{ with } T_1 \neq T_2$$

where dX denotes the boundary[1] of a set X of points.

Usually, tiles are subject to further restrictions. A tile is bounded if it can be enclosed into a ball. It is closed if it contains its boundary. It is simple if it has no holes, if each point belongs to a subtetrahedron, and if the intersection of each two closed sets of points that cover the tile contains at least a face. Typical examples of simple, bounded and closed tiles are tetrahedra, pyramids, cubes, parallelopipeds, etc.

Figure 1 shows some tiles. The first tile has a hole. The second one consists of two disjoint cubes. The third tile consist of two disjoint cubes that are connected by a line. The fourth and fifth tiles consist of two cubes each, the intersection of which is a point and a line respectively. Only the last

[1] A point belongs to the boundary of $X \subseteq \mathbb{R}^3$ if it is the limit of a convergent sequence of points in X.

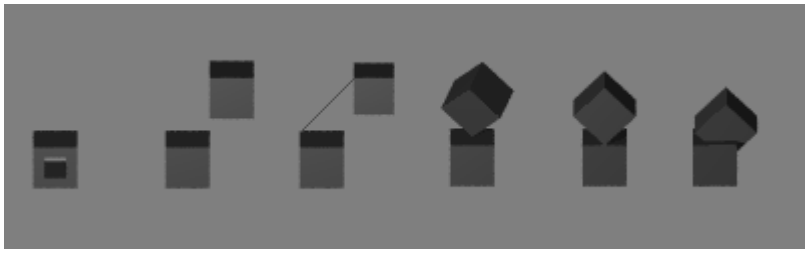

Figure 1. Some tiles

tile fulfills all the restrictions mentioned above. In the following only such bounded, closed and simple tiles are considered.

To visualize a 3D tiling, a finite subset of the tiling will be depicted. Note that the resulting picture is always incomplete and can have gaps. Although colours were not mentioned above, they will be used for better visualization.

3 Construction of 3D Tilings

One of the most elementary questions is how tilings can be specified in a reasonable way. Because there is an uncountable number of tilings, only a small, i.e. countable part of them can have a finite description. Nevertheless, they are of particular interest because they can be considered as computable. A step in this direction is the description of tilings by finite sets of tiles and transformations. The following formal definition is restricted to similarity transformations, i.e. translations, rotations, reflections and all of their compositions.

Let $\mathcal{T}$ be a 3D tiling. Let $\mathcal{P}$ be a finite set of tiles, which are called prototiles, and let $st : \mathcal{P} \longrightarrow 2^{SIM}$, where SIM denotes the set of all similarity transformations and 2^{SIM} its powerset, be a mapping that assigns a finite set of similarity transformations to each prototile. Then $\mathcal{T}$ is said to be specified by $\mathcal{P}$ and st if the following holds:

$$\mathcal{T} = \{f(T) \mid T \in \mathcal{P}, f \in st(T)^*\}$$

where $f(T) = \{f(x) \mid x \in T\}$ for all $f \in SIM$ and, given $X \subseteq SIM, X^*$ denotes the set of all sequential compositions of transformations of X, i.e.

$$X^* = \{id\} \cup \{t_1 \circ \ldots \circ t_n \mid t_i \in X, i = 1, \ldots, n, n \geq 1\}$$

In this case, $\mathcal{T}$ is also denoted by $\mathcal{T}(\mathcal{P}, st)$. $\mathcal{T} = \mathcal{T}(\mathcal{P}, st)$ means that each tile of $\mathcal{T}$ is either a prototile or can be obtained from a prototile by applying a sequence of its similarity transformations. The following examples will illustrate this construction principle.

4 Example

The Euclidean space is easily filled by cubes of the same size. The only prototile is the unit cube *cb* given by the three orthogonal unit vectors $a = (1,0,0)$, $b = (0,1,0)$ and $c = (0,0,1)$ of length 1.

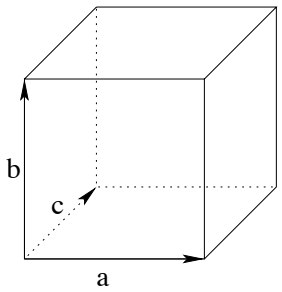

Figure 2. The protocube

The set of similarity transformations associated with this prototile is $\mathit{Shift}(a,b,c) = \{\mathit{shift}_x \mid x \in \{a,b,c,-a,-b,-c\}\}$, where shift_x is the translation by the vector x, i.e. $\mathit{shift}_x(y) = y + x$ for all $y \in \mathbb{R}^3$. Then $\mathit{CUBE} = \{f(cb) \mid f \in \mathit{Shift}(a,b,c,-a,-b,-c)^*\}$ is a 3D tiling.

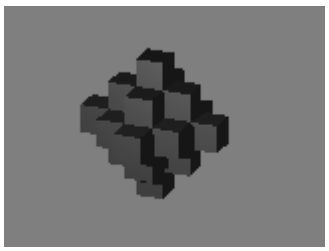

Figure 3. A section of CUBE

Analogously, one gets a tiling of the 3D space by packing parallelopipeds of the same size side by side. Given three linearly independent vectors a, b and c, they span a parallelopiped $\mathit{ppp}(a,b,c)$ (containing all linear combinations of a, b and c with scalars between 0 and 1). This prototile together with the set $Shift(a,b,c)$ of transformations specifies a tiling $\mathcal{T}(a,b,c)$.

5 Further Examples

The tiling CUBE is a tiling with only a single prototile. Tilings with two or more prototiles can be constructed in a similar way. An example with two prototiles is illustrated in Figure 4.

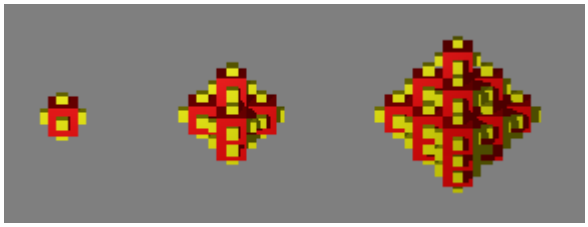

Figure 4. Three sections of the tiling

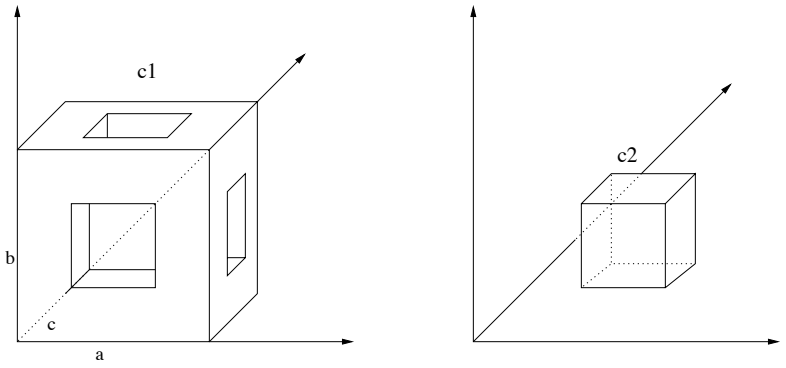

Figure 5. Two prototiles

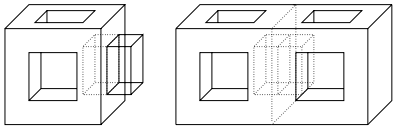

Figure 6. Shifting c1 and c2

The prototile $c1$ is made of a cube given by the three unit vectors $a = (1, 0, 0)$, $b = (0, 1, 0)$ and $c = (0, 0, 1)$. It has six indentations that are located in the middle of each side. They have a width and height of $\frac{2}{5}$ units and a

depth of $\frac{1}{5}$ units. The prototile $c2$ is a cube that is obtained from the unit cube by scaling it down by the factor $\frac{2}{5}$ and translating it by the vector $(\frac{4}{5}, \frac{3}{10}, \frac{3}{10})$. To construct the tiling shown in Figure 4 the two prototiles have to be shifted in a certain way. Figure 6 shows the prototiles $c1$ and $c2$ together. As it can be seen, $c2$ fits into the indentation such that half of it is reaching out. If $c1$ is shifted by $shift_a$, $c2$ is located between $c1$ and its shifted twin without gaps and overlaps. In the same way, the whole $\mathbb{R}^3$ can be filled by shifting $c1$ side by side and shifting $c2$ in between each two shifted $c1$ that are neighbours. The following transformations are needed.

$$st(c1) = \{shift_a, shift_{-a}, shift_b, shift_{-b}, shift_c, shift_{-c}\}$$

$$st(c2) = st(c1) \cup \{shift_{(\frac{1}{2},\frac{1}{2},0)}, shift_{(\frac{1}{2},0,\frac{1}{2})}\}$$

The tiling in Figure 8 is specified by the three different prototiles sp, sl and cu which are depicted in Figure 7. It can be constructed in a way similar to the latter tiling using the following transformations. Note that the prototile sl must sometimes be rotated to fill the $\mathbb{R}^3$ properly. $rotate_1$ rotates sl about the x-axis by 90°, while $rotate_2$ rotates sl about the y-axis by 90°.

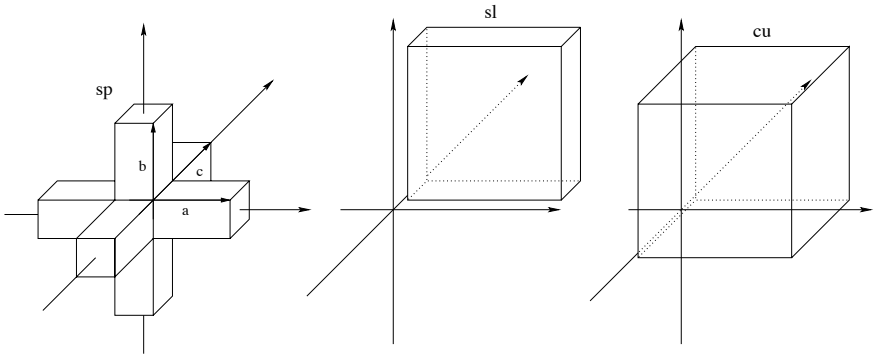

Figure 7. The prototiles of the tiling in Figure 8

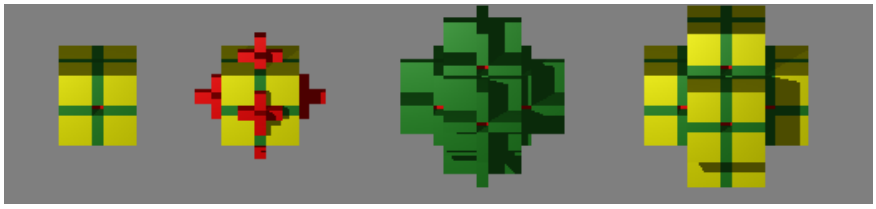

Figure 8. A tiling with three prototiles

$$st(sp) = st(cu) = \{shift_{2a}, shift_{-2a}, shift_{2b}, shift_{-2b}, shift_{2c}, shift_{-2c}\}$$

$$st(sl) =$$

$$\{shift_{2a}, shift_{-2a}, shift_{2b}, shift_{-2b}, shift_{2c}, shift_{-2c}, rotate_1, rotate_2\}$$

where

$$rotate_1(x) = x \circ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} \text{ for all } x \in \mathbb{R}^3$$

$$rotate_2(x) = x \circ \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \text{ for all } x \in \mathbb{R}^3$$

6 Constructing 3D Tilings with Collage Grammars

Another method for constructing 3D tilings is the concept of collage grammars (see, e.g., Drewes and Kreowski [DK99]). This section presents a collage grammar that generates the 3D tiling $\mathcal{T}(a, b, c)$ of Section 4.

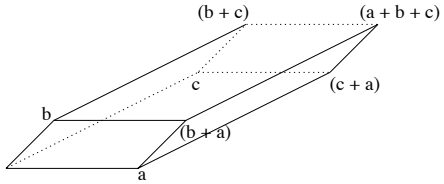

Figure 9. A parallelopiped

The grammar has a single nonterminal S. The start collage consists of a single hyperedge which is labelled with S and has the eight corners of the parallelopiped $ppp(a, b, c)$ as pin points, i.e.

$$pin_s = 0\ a\ b\ c\ (b + c)(b + a)(c + a)(a + b + c)$$

There are two rules with the left-hand side S and the right-hand sides R_t and R_n given by

$$R_t = (\{ppp(a, b, c)\}, \emptyset, \emptyset, \emptyset, pin_s)$$

$$R_n = (\{ppp(a, b, c)\}, \{f_1, f_2, f_3, f_4, f_5, f_6\}, att_n, lab_n, pin_s)$$

Both have $ppp(a, b, c)$ as single part and pin_S as pin points. While R_t is terminal, R_n has six nonterminal hyperedges that are labelled with S, i.e.

$$lab_n(x) = S \text{ for all } x \in \{f_1, f_2, f_3, f_4, f_5, f_6\}.$$

The attachment is given by shifting pin_S by $a, -a,\ b,\ -b,\ c$ and $-c$ resp., i.e.

$$att_n(f_1) = shift_{-a}(pin_s)$$

$$att_n(f_2) = shift_a(pin_s)$$

$$att_n(f_3) = shift_{-b}(pin_s)$$

$$att_n(f_4) = shift_b(pin_s)$$

$$att_n(f_5) = shift_{-c}(pin_s)$$

$$att_n(f_6) = shift_c(pin_s)$$

This means that the attachment points of the hyperedges span parallelopipeds each of which shares one of the faces with $ppp(a, b, c)$. To construct sections of $\mathcal{T}(a, b, c)$ the hyperedges have to be replaced repeatedly with properly shifted copies of R_n and terminated with R_t in the last step. For instance, starting with the start collage and replacing every hyperedge with R_n two times and terminating with R_t, the resulting picture is similar to Figure 3 (with the difference that the parts are arbitrary parallelopipeds instead of cubes).

7 Constructing 3D Tilings with AnimaLab

Corresponding to the construction method shown in Section 6 the object generator *Collage3D* can be used to generate 3D tilings. For details about *Collage3D* and the concept of the special collage grammars used there please see Chapter 16. In this section only the input files will be shown. They will generate the same 3D tiling as introduced in Section 4.

```
applications.pictures3D.collageAlgebra("simple cube algebra"):
{
 parts {
  cube = cube{1,1,1, colour[0,1,0]},
 },
 transformations {
  idmove = scale(1),
  move1 = translate(1,0,0),
  move2 = translate(-1,0,0),
  move3 = translate(0,1,0),
  move4 = translate(0,-1,0),
  move5 = translate(0,0,1),
  move6 = translate(0,0,-1)
 }
}
```

One needs an algebra for scene description. The part is the unit cube in Figure 2; the "start" of the tiling. The transformations are similar to the transformations of CUBE. An identity transformation ist also needed because the old tiling should be shown, too.

```
generators.regularTreeGrammar("simple cube grammar"):
(
 { S },
 {idmove:6,move1:1,move2:1,move3:1,move4:1,move5:1,
  move6:1,cube:0},
 {
  S -> idmove[move1[S],move2[S],move3[S],move4[S],
       move5[S],move6[S]],
  S -> cube
 },
  S
)
```

In the next step a grammar of the tiling is needed. Every part is moved as given by the transformations. The picture looks like Figure 3.

7 Case Study: City Modelling

Björn Cordes and Karsten Hölscher

1 Introduction

For modelling, a very simple structure of a city is that of a chessboard pattern. These patterns can be found in cities that have not grown in a natural way, but were planned from the very beginning. Famous examples of our time are several big cities in the United States of America, but there are also historical ones like Roman garrison cities.

As the name chessboard pattern suggests, all these structures have rectangular grids of streets with blocks of buildings in between. In the modern examples these buildings are constructed skywards in order to optimize the use of area.

In order to simplify our approach we decided to use square grids instead of rectangular ones. Every square of this grid serves as a base for exactly one building or a green. Thus our cities will not consist of blocks of houses, but every building will be completely framed by streets. The houses can be of varying shape and height — they consist of a varying number of cubes that may vary in size.

2 Modelling

The first decision to make was the selection of a suitable method of picture generation. The method of our choice for building cities is collage grammars.

The process of modelling can be conceptually divided into two steps. Each of the two steps is independent of the other one, and they can be combined with each other.

The first step deals with the expansion of areas constructed of squares. This can be used to simulate the horizontal growth of a city. The second step is used to simulate the vertical growth of a single building. Putting these steps together yields a city with the above-mentioned characteristics.

2.1 Area Expansion

There are at least two possibilities to generate a square grid using collage grammars. One is to refine recursively a given square into four smaller ones.

Since this approach always yields a square base for our cities, we decided against it. Another effect is that the gaps (representing the streets) between the squares continously shrink with every refinement step. This can be a problem when trying to refine in a non-uniform way.

The other possibility is to add recursively adjacent new squares of the same size to a given square. Thus it is possible to control the direction of the expansion, which can be useful when modelling a city with environmental constraints like rivers, mountains, or the like. Note that the base of the city generated by this method is not necessarily a square.

A first, simple approach would be to take a starting hyperedge labelled with S and a rule that adds four new hyperedges labelled with S above, below, left and right of that starting hyperedge. Iterating this rule will lead to a horizontal and not necessarily regular expansion. Now the hyperedges are replaced with either a house base or a green. A green is a plain, green square of smaller size than the hyperedge it replaces. A house base is the same in grey. House bases and greens are parts, so no further rules are needed to replace them. These considerations lead to the set of three rules shown in Figure 1.

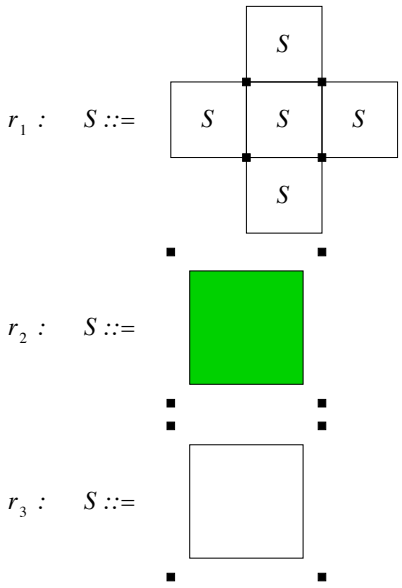

Figure 1. Area expansion I: simple approach

But this simple a priori approach leads to problems. Regard the following example: Starting with one hyperedge, apply rule r_1. This leads to five hyperedges. Now apply rule r_3 to the hyperedge in the centre and make it

a green. Then apply rule r_1 to the top hyperedge. This yields a hyperedge overlapping the previously built green. Now apply rule r_2 to that hyperedge, replacing it with a house base. Then we have a house base and a green in the same place. Since a green should not overlap a house base and vice versa, we need to find a different approach to solve this conflict.

In order to avoid overlapping, we need to introduce further nonterminal symbols. The starting hyperedge labelled with S is replaced with a hyperedge labelled with T, and eight new hyperedges labelled with S_i, $i \in \{N, E, S, W, NE, SE, SW, NW\}$ are added as shown in Figure 2.

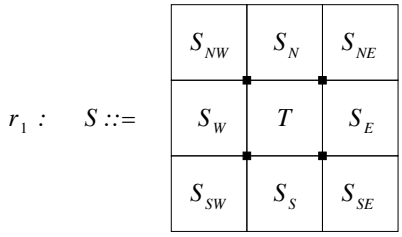

Figure 2. Area expansion II: rule r_1

These new hyperedges have to be replaced with a hyperedge labelled with T, and further hyperedges that do not overlap existing ones have to be added. This leads to the set of rules shown in Figure 3.

Now these new hyperedges have to be terminated in order to generate pictures. Therefore we establish rules r_i that terminate the area expansion by changing the labels S_i to T as shown in Figure 4 (this is formally a replacement of a hyperedge labelled with S_i with one labelled with T).

To terminate the process, rules r_{10} and r_{11} are introduced, replacing the hyperedges labelled with T with either a house base or a green as shown in Figure 5.

Figure 6 shows two collages generated through the rules above using the *Collage3D* object generator of AnimaLab.

2.2 Skyscraper Growth

Since our skyscrapers grow only in one direction, our first approach would be a simple regular collage grammar. Beginning with a starting hyperedge labelled with *Sky*, we introduce a rule s_1, replacing it with a grey cube of the same base area and a hyperedge labelled with *Sky* placed above that cube. To generate a greater variety of skyscraper shapes, we further use a rule s_2, replacing the hyperedge labelled with *Sky*, with the same cube and the hyperedge labelled with *Sky*, where the base area of the cube spanned by

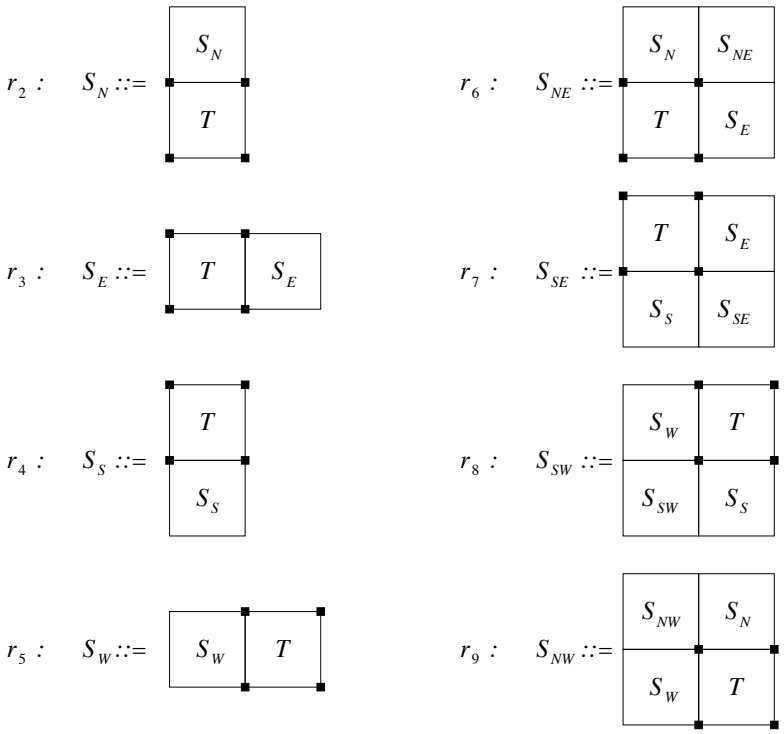

Figure 3. Area expansion II: rules r_2 to r_9

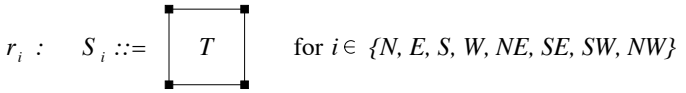

Figure 4. Area expansion II: rules r_i

the attachment points of the hyperedge is scaled down. A terminating rule s_3 replaces a hyperedge labelled with *Sky* with a grey cube of the same size.

Actually we are not satisfied with this approach, since we cannot control the growth in this case. Obviously there is no maximal height of the buildings generated by our set of rules, thus applying them may yield statically impossible buildings.

In order to introduce a maximal height of n cubes we need to introduce $n-1$ new nonterminal symbols representing the growth. The first rule s_{a_1} replaces the hyperedge labelled with *Sky* with a grey cube of the same size, and a hyperedge labelled with N_1 above it. Now we introduce rules s_{a_i} that replace hyperedges labelled with N_{i-1} with a grey cube of the same size and place the hyperedge labelled with N_i on top of it, for $i \in \{2, ..., n-1\}$. Rule

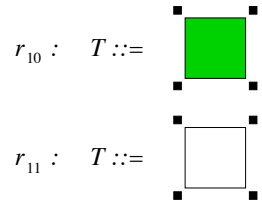

Figure 5. Area expansion II: rules r_{10} and rules r_{11}

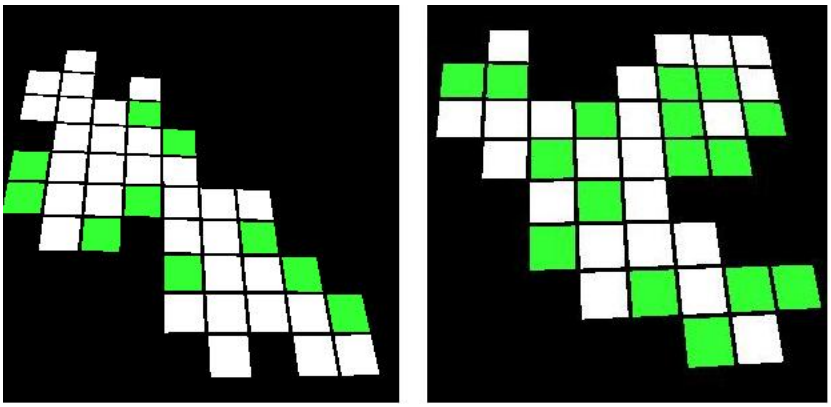

Figure 6. Two examples of area expansion

s_{b_1} replaces the hyperedge labelled with *Sky* with the same cube, and a scaled down hyperedge labelled with N_1 as mentioned above. The rules s_{b_i} replace the hyperedge labelled with N_{i-1} with the same cube and place the scaled down hyperedge labelled with N_i on top of it, for $i \in \{2, ..., n-1\}$. Rule s_{t_1} terminates the growth by replacing the hyperedge labelled with *Sky* with a grey cube of the same size. The same applies to the rules s_{t_i} that replace the hyperedges labelled with N_{i-1} with cubes for $i \in \{2, ..., n\}$. Using this set of rules, a skyscraper can consist of one up to n cubes. These cubes can all have the same size, or their base area may shrink, either regularly or not, with increasing height. These rules are shown in Figure 7.

2.3 Assembly

In the previous sections we introduced methods for building skyscrapers and city bases. But so far these methods are separate from each other, and thus we actually cannot build cities using them in their current form. In fact, only slight changes are necessary to combine both.

Combining both methods concretely means to build a skyscraper on a house base. Formally, this is achieved by uniting the two sets of rules and removing the terminating rule r_{10}. Finally we add a new rule r'_{10} that re-

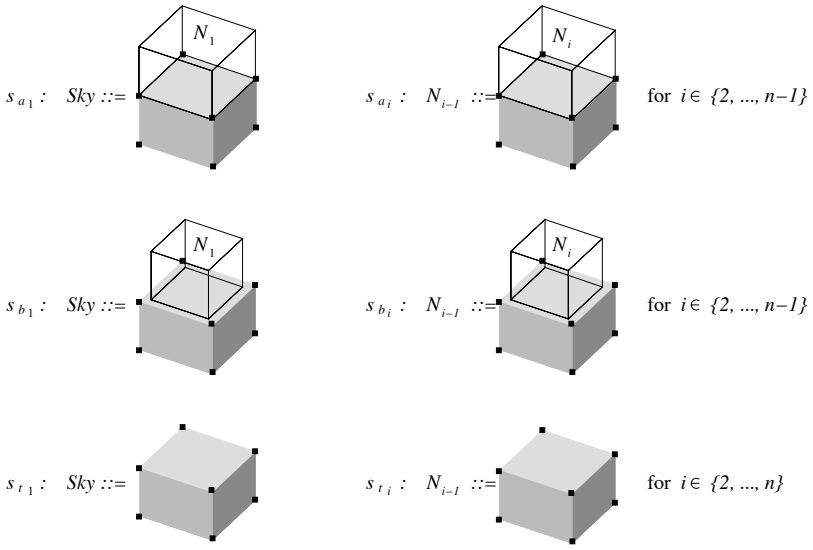

Figure 7. Skyscraper growth

places the hyperedges labelled with T with the starting hyperedge for the skyscrapers.

Figure 8 shows a sample city, generated using the *Collage3D* object generator and the combined set of rules explained above.

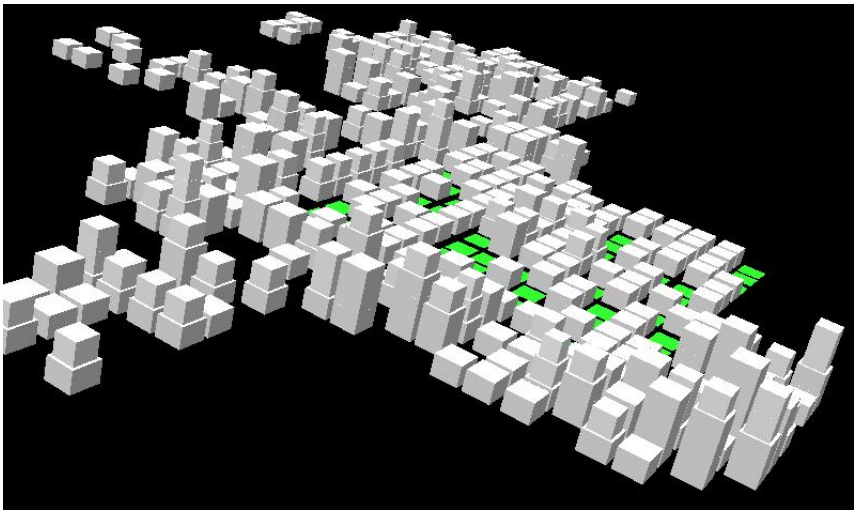

Figure 8. A sample city

3 Conclusion

The above considerations yield a set of rules for generating a very basic chessboard pattern city that does not look very authentic. From the pictures alone it is not obvious that the displayed structure represents a city.

An easy visual improvement would be to add textures to the parts representing skyscraper blocks and green areas. This can be enhanced by adding further rule sets that generate skyscrapers with different textures. In this case the skyscrapers would look different, but they would still be of the same shape. To achieve further diversity new rule sets for buildings that are of different shape than the ones generated using the above rules could be introduced. An easy approach would be to simply change the base area from a square to a non-square rectangle.

All these changes still yield a chessboard pattern city. Of course not all the cities in the world are built this way. In fact only few cities have a settlement structure as simple and regular as a chessboard pattern. So the area expansion could be changed in order to generate different settlement structures, like a medieval settlement where houses are built circularly around a special site like the church. Of course a medieval town does not consist of skyscrapers, so completely different building types would be required as well.

As shown, cities provide lots of possibilities for modelling with syntactical methods.

8 Plant Modelling

Carolina von Totth

1 Introduction

The goal of this paper is to illustrate two different approaches to plant modelling, and on that basis propose a number of guidelines that might apply to computer-aided plant generation in general. These insights were won over a two-year period in the context of a students' project: during this time, we studied different approaches to modelling — fractal as well as rule-based — and finally settled on collage grammars as a means of modelling.

There are other means, however: the work of Przemyslaw Prusinkiewicz referring to modelling with L-systems (turtle grammars) has gained wide recognition world-wide, and in the field of computer graphics L-systems are mostly used synonymously with plant modelling even to this day.

Turtle grammars are ideally suited for simulating growth and branching processes of any kind, and have been predominantly used for modelling both plant and cell growth. Still, even with a basically sound method for picture generation, a more goal-oriented modelling approach — in this case the modelling of plants — requires several alterations to the method before the results become satisfactorily life-like.

Despite the popularity of L systems, we settled on an equally versatile, though less known, picture generation method: collage grammars. There were various reasons for this decision, the most important being that the method is little known, powerful, and nobody had done actual three-dimensional modelling with it before.

Apart from collage grammars, particle systems are the second plant generation method presented in this paper. We chose particle systems over L systems because they differ more strongly from the method we used, and presenting two entirely different methods allows us a wider perspective on the field of plant generation in general.

In Section 2 of this paper an approach to plant modelling using particle systems is described, followed by a short description of collage grammars and a small modelling example. Finally, a description of the modelling guidelines mentioned above is included.

Figure 1. Bluebell: movie frame

2 Plant Generation with Particle Systems

Many of the aspects of modelling described in Section 5 have appeared in modelling-related papers over a long period of years, under more or less different denominations. One of the most important is what Reeves and Blau [RB85] call *data amplification*, a term that applies to the use of stochastic models and describes the generation of complex visual constructs from a small, user-generated data base. It involves taking that data base, which usually describes the general characteristics of the desired object (or class of objects), and expanding on it algorithmically in order to add detail.

Reeves and Blau point out that this approach requires less human design time, and that "particle systems can be used to model objects that change form over a period of time." Actually, these two arguments apply to most procedural modelling approaches, so in order to get to the advantages and disadvantages of each method, one must dig a little deeper.

Basically, when using particle systems, the emphasis lies on

1. the sheer multitude of particles that can be employed to describe an object, and
2. the dynamic nature of the method, which allows to model objects that change shape over time.

A method commonly used for modelling explosions, fireworks and similar effects, when applied to plants it can be used to animate the created models, resulting for example in a simulation of a field of grass under the influence of wind.

Here is a short definition given by Reeves in [Ree83] : "Particle systems are procedural stochastic representations controlled by several global parameters. Particle systems model an object as a cloud of primitive particles that define its volume. Over a period of time, particles are generated into the system, move and change from within the system, and die from the system. The resulting model is able to represent motion, changes of form, and dynamics that are not possible with classical surface-based representations. [...] Stochastic processes are used to generate and control the many particles within a particle system."

Clearly this approach is targeted towards the generation of large clusters of trees, and thus adresses problems that are specifically connected with the creation and rendering of large scenes containing lots of objects, while being able to ignore more plant-specific modelling problems. Reeves and Blau also hint at this, suggesting that their goal is to create a more "global view of a forest environment" [RB85], as opposed to the detailed representation of individual plants pursued by Bloomenthal [Blo85] and Prusinkiewicz and Lindenmayer [PL90] in their work.

The solutions presented by Reeves and Blau are therefore clearly tailored to generating a large forest environment. The placement problem is addressed first: trees are placed randomly over a grid that specifies the minimum distance between any pair of trees; the height at which each tree is placed is given by an elevation map. Also, texture maps can be used to describe the density of growth over a given region. The type of a tree (deciduous, coniferous) is influenced by such factors as terrain elevation.

The actual trees are generated recursively from a few randomly chosen parameters using a method not entirely unlike rule-based generators. Subbranches are generated recursively and inherit many characteristics from their parent; the recursion is stopped either by reaching a previously specified maximum depth, or when a branch has reached minimum thickness.

The colours for both trunk and foliage are specified along with the shape of the tree. The colours are chosen from predefined colour tables and assigned to specific parts of the tree depending on such parameters as age, or thickness of a branch.

Grass is generated stochastically as well; the parameters (position, orientation, circumference, density, type) influence whole clumps of grass rather than single blades. Another set of parameters (position, height, thickness, curvature, ...) influences the appearance of each single blade. The curve segment representing each blade is approximated by short line segments. Flowers can also be created by adding different-coloured (red, yellow, ...) clusters of particles to a blade.

The overall scattering of grass across the terrain is specified either randomly or by a texture map.

It is interesting to note that a lot of post-processing is being done as soon as the models have been generated: separate algorithms have been devised in

order to bend or twist tree branches. Leaves or needles are also added after the generation of the model, and then only to those branches that have no sub-branches.

One of the most prominent features of modelling with structured particle systems is that no actual geometry is generated; instead, "particle systems represent objects as clouds of primitive particles that occupy their volumes, rather than using more classical surface-based representations such as polygons, patches and quadric surfaces" (Reeves and Blau [RB85]).

In the final rendering step, each particle is drawn either as an antialiased line, or a small circle.

On the one hand the lack of conventional geometry is a huge advantage, since in order to be renderable, the size of a geometry scene is limited by the size of usable memory. On the other hand it requires a specific renderer, since ray-tracing particles would be so time consuming as to be virtually impossible. Also, as opposed to conventional particles used to model fire and other light effects, the particles used in plant generation are no longer light emitters but reflectors, making them more difficult to render.

Here again, the solutions consist of simplified lighting and shadow computations: the amount of light received by each particle depends on its depth in the tree, measured along a ray from the light source(s) to the particle; whether a particle is in shadow is determined by computing, for each tree, planes containing a light source and the top point of each adjacent tree. Particles of this tree above the plane are fully lit, while those below the plane are receiving less and less light with increasing distance from the plane (cf. Reeves and Blau [RB85]).

Even with the lighting and shadowing problems out of the way, in order for rendering to be possible, the visible surfaces have to be determined. The algorithm described by Reeves and Blau consists of two major steps: first the trees in the scene are sorted back-to-front with respect to screen depth — this only works where the bounding volumes of the trees do not intersect. During the second step, the volume of each tree is divided into 'buckets' — each bucket stands for a certain scene depth interval. After every generated particle of the current tree has been sorted into the appropriate bucket, all particles of this tree are shaded and rendered in back-to-front order, after which the data for the tree is discarded.

Therefore, both the trees and their components are rendered back-to-front, so that every object that is further ahead in eye space in drawn on top of the objects behind it.

A small problem with this approach is described in Foley and van Dam [FvDFH00]: since the data for an already rendered tree is immediately discarded, a branch of a nearer tree may obscure a branch of a tree slightly farther away, although the obscured branch is actually closer to the viewer. This happens because, although the obscured branch may be closer, the tree it belongs to is, as a whole, farther away than the one being processed and has

therefore been already completely rendered and its data discarded. However, this does not seem to be a noticeable error if the scene is large enough, and there are no close-ups in the final rendered picture (or movie).

The sorting algorithm for grass allows intersection, and is a more complex and more expensive version of the one described above. For a detailed description see Reeves and Blau [RB85].

3 Plant Generation with Collage Grammars

3.1 Collage Grammars

A collage grammar (see Kreowski and Drewes [DK99]) is a syntactic, rule-based device for picture generation. The picture-generating mechanism is based on hyperedge replacement.

A collage consists of a set of *parts* and a sequence of *pin points*. A part may be an arbitrary set of points in a Euclidian space — usually taken from some standard set of geometric objects like line segments, polygons, etc. that have simple finite descriptions and are easy to deal with on graphical user interfaces. The pin points are used to paste collages into collages. So, in three-dimensional space, a collage represents a model by the union of its parts.

To generate collages from collages by the application of productions, they are decorated with hyperedges in intermediate steps. A hyperedge is a labelled item with an ordered number of tentacles, each of which is attached to a point. A hyperedge serves as a place holder, and thus may eventually be replaced by a (transformed) decorated collage provided there exists an affine transformation of the pin points of the collage to the attachment points of the hyperedge.

This kind of hyperedge replacement establishes the rewrite steps of a collage grammar if the label of the replaced hyperedge and the replacing decorated collage form a production.

3.2 TOL Collage Grammars

Since we are using ETOL collage grammars for a simple example later on in this paper, here is a short description of how they work.

A TOL collage grammar provides a set of tables, which is a set of sets of productions. In each derivation step a table is chosen, and all hyperedges are replaced simultaneously. This way, nonterminal hyperedges are being refined and eventually terminated, resulting into pictures. In our case, the results are three-dimensional shapes.

ETOL collage grammars (the E in ETOL stands for Extended) are a slightly modified version of the above principle.

4 Example (Using Collage Grammars)

Collage grammars are a very intuitive way of generating complex shapes using a finite set of simple shapes. Affine transformations are easy to use as well — although one has to get used to shuffling parts around in three-dimensional space and it is important to keep in mind that rotation will always take place about the center of the coordinate system — but otherwise this kind of modelling does not present any great difficulties.

Two components are needed when modelling with tree-based collage grammars: an *algebra*, containing the descriptions for a set of graphical entities called *parts* as well as a set of *affine transformations* (or composed transformations), and a *tree grammar*, containing a set of rewriting rules, also known as *productions*. Productions can be applied either in parallel or sequentially, depending on the type of the grammar. For more detailed information, see Drewes [Dre00].

4.1 Modelling a Simple Fern

This one is a branch of a fern-like plant, and somewhat similar to an existing fern species.

Figure 2. Fern branch growth sequence

Simple Fern Algebra

First of all, the algebra is defined; the first half defines a polygonal construct named `leaf`, as well as two frustums, `apex` and `tip`. Since the parts used here are extremely simple shapes, the end results will look slightly angular as well. Far better results can be achieved by using more elaborate parts.

```
applications.pictures3D.collageAlgebra("Simple Fern Algebra"):
{
parts {
leaf =  polyhedron{
         points{ a=(0,0,0),      b=(.4,.3,-.2),
                 c=(.5,.9,-.1),  c1=(.23, 1.65, 0),
                 d=(0,2,-.15),
                 e=(-.5,.9,-.1), e1=(-.23, 1.65, 0),
                 f=(-.4,.3,-.2),
                 s=(0,1.2,.1),   s1=(0,1.5,0)},
         polygons{ (a,s,c,b),   (a,f,e,s), (s,s1,c1,c),
                   (s,e,e1,s1), (s1,d,c1), (s1,e1,d)},
         colour[.6, .7, .1]
                 },

apex = frustum{.02, .02,  1.5, 12, colour[.3, .55, 0]},
tip  = frustum{.02, .012, 1  , 12, colour[.3, .55, 0]}
      },
```

In the second part of the algebra the transformations are defined.

```
transformations {
nothing  = scale(1),
left     = rotate(0,0,1, 70) . rotate(0,1,0, 15),
right    = rotate(0,0,1,-70) . rotate(0,1,0,-15),
up       = translate(0,1.45,0),
bend1    = rotate(1,0,0, -4.5),
bend2    = rotate(1,0,0, -2),
last     = translate(0,.95,0),
longleaf = scale(.6,2.3,.6),
grow     = scale(1.08),
rot1     = rotate(0,1,0,  8),
rot2     = rotate(0,1,0, -8)
               }
}
```

Simple Fern Grammar

The grammar is divided in three parts: the first block contains a list of nonterminals, the second contains a list of ranked symbols. The third part, finally, is a set of rules.

```
generators.ETOLTreeGrammar("Simple Fern Grammar"):
(
  {
   S, L, R,
```

```
   L1, L2, L3, L4, L5, L6,
   R1, R2, R3, R4, R5, R6
  },
  {
   nothing:1, nothing:2,
   left:1,
   right:1,
   up:1, up:2, up:3,
   bend1:1, bend1:3,
   bend2:1, bend2:3,
   last:1,
   longleaf:1,
   grow:1,
   rot1:1, rot1:2, rot1:3,
   rot2:1, rot2:2, rot2:3,
   leaf:0,
   apex:0,
   tip:0
  },
```

For the sake of simplicity we will describe the modelling process on the basis of the grammar alone. At this point, however, it is important to keep in mind that the grammar is only a generator of trees, and that the actual yield we are interested in — namely the three-dimensional objects — is the result of these trees being interpreted in the above algebra.

The productions are grouped in two tables: the first table contains the productions that describe the growth of the plant, while the second table contains the terminating rules for each of the nonterminals of this grammar.

```
{
  {
    S  -> nothing[apex, up[bend1[rot1[left[L], right[R], S]]]],
    S  -> nothing[apex, up[bend2[rot1[left[L], right[R], S]]]],
    S  -> nothing[apex, up[bend1[rot2[left[L], right[R], S]]]],
    S  -> nothing[apex, up[bend2[rot2[left[L], right[R], S]]]],
```

The first four productions describe the overall shape of the fern leaf: an apex is placed, followed by a 'branching' nonterminal to the left side of the apex (`L`), and another one (`R`) to the right. The actual 'branching' is done by `left` and `right`: both are rotations about the z-axis, of 70 and -70 degrees respectively, concatenated with a slight twist about the y-axis for a more natural look.

Since this is a very simple example, the branching elements are mere leaflets instead of more complex structures.

Also, by rotating each leaflet, however slightly, about all three axes and combining the rotation operations differently within each production, the

arrangement of the leaflets around the stem looks more random, and thus more natural.

```
    % leaf growth rules
    L  -> grow[L1],
    L1 -> grow[L2],
    L2 -> grow[L3],
    L3 -> grow[L4],
    L4 -> grow[L5],
    L5 -> grow[L6],

    R  -> grow[R1],
    R1 -> grow[R2],
    R2 -> grow[R3],
    R3 -> grow[R4],
    R4 -> grow[R5],
    R5 -> grow[R6]
  },
  {
    S -> nothing[tip, last[bend1[longleaf[leaf]]]],
    S -> nothing[tip, last[bend2[longleaf[leaf]]]],
    L -> longleaf[leaf],
    R -> longleaf[leaf],


    % leaf growth rules
    L  -> longleaf[leaf],
    L1 -> longleaf[leaf],
    L2 -> longleaf[leaf],
    L3 -> longleaf[leaf],
    L4 -> longleaf[leaf],
    L5 -> longleaf[leaf],
    L6 -> longleaf[leaf],

    R  -> longleaf[leaf],
    R1 -> longleaf[leaf],
    R2 -> longleaf[leaf],
    R3 -> longleaf[leaf],
    R4 -> longleaf[leaf],
    R5 -> longleaf[leaf],
    R6 -> longleaf[leaf]
  }
 },
 S
)
```

Using a slightly more advanced grammar, the leaflets can be combined to form the actual plant shown in Figure 3.

Figure 3. Fern

4.2 Further Plant Models

Besides the above example, we have used collage grammars for modelling various other plant-like structures. With some of the examples we have incorporated the simulation of a growth process in such a way that the resulting models could be used in creating animations, as shown in Figure 4.

5 Aspects of Plant Modelling

On the whole, it looks as if there would be no inherently *correct* approach to plant modelling, or indeed modelling of any kind. The method is usually dictated by what is actually needed — sweeping forest scenes or a single, detailed, delicate plant. But whatever those needs might be, there are a few method-independent principles that can serve as guidelines.

Also, if one wants to build convincing three-dimensional models of plants, there are basically two ways to go about it:

Figure 4. Animation frames from flower growth simulation

1. aim for biological accuracy, or
2. try to make a plant look the way people imagine it to look.

From an aesthetic point of view, the second one is probably the better choice and the casual observer will not reject the result for ugliness or other divergence from the expectations.

So, when it comes to holding the mirror up to nature, we find that it had better be an old and blurry one. Trees, for instance, are mostly pretty ugly. Not the kind of knotted, poetic, mist-shrouded bonsai ugliness of paintings and bad horror movies, but more a kind of uninteresting scarecrow ugliness — with straight, too-thick branches shooting out at unpleasant angles and small, spindly-looking shoots growing all over the trunk, making it look like an overlong hedgehog with leaves. Besides, there may not be pretty knots and curves anywhere.

Trees like this can be found at every roadside, and no one would get the idea of questioning their provenance, or even asking themselves if something that unsightly could be really *there*. But look at a tree like that on a computer screen, and the same people will instantly say "That model is put together rather badly, don't you think," and proceed to find fault with the quality of the texture. It does not matter that they have probably seen dozens of the type just this morning on their way to work. Which only proves how selective human memory tends to be.

Figure 5. Single tree at night

So, what basic factors should one take into account if they want to generate real-looking plants? During our modelling experience, the following aspects emerged.

1. **Small-scale influence**
   There has to be some system that tells a new branch/cluster of branches (or leaves, or flowers) where on the rest of the plant to attach, when to grow or open, and similar information.
2. **Global-scale influence**
   The individual parts should probably know where *up*, *down*, *left*, *right*, *front* and *back* are in an absolute sense. Magnolia blooms, for example, always grow upwards, facing the sky — the same way the branches of a weeping willow always grow towards the ground. This would be rather hard to emulate if the system we are planning (rule-based or otherwise) would not incorporate a notion of what up and down actually *are*.
3. **Random(ness)**
   When looking at an actual plant, lots of factors seem to vary randomly: the diameter and length of branches, the shape and size of leaves and blooms, branching angles and distances and so on. There is a huge number of varying factors, out of which we must choose the most important (read: noticeable) ones, and incorporate them into our system.
   However, with all the random stuff going on, there is still one important detail: for one given plant type, the factors never change so much that

the plant becomes unrecognizable. Translated, this means they only vary over a certain given *range*, while the basic 'construction' rules remain the same. And parametric rules, where parameter values are chosen from within a certain range of numbers, are something we can both understand and reproduce.

It is plain that a real plant will differ from another plant of the same sort down to the very cell structure, which is something we cannot (and do not really want to) reproduce. We settle for varying appearances instead and hope it is enough to fool the superficial observer.

4. Little work for the user

Often enough, the only point of trying to break down the process of, for example, plant creation into simple steps that can be performed by a simple automaton over and over is to spare a human the bother of doing exactly that. Of course, the user would still have to write down the rules for a special kind of plant, but they would do it within an already established framework, and once finished, they would have access to an endless supply of varied-looking specimens of the same plant.

It sounds like a goal worth achieving (to people interested in three-dimensional plant models, anyway). Of course, the set of rules should not have to be too large, otherwise people might decide it is easier to build the models by hand.

5. Representation and rendering of complex models

While this is not a modelling problem, the representation of complex scenes with limited resources is still not satisfactorily solved. Most algorithms used today require the complete scene to be stored in memory all at once in order to render it. Since memory is limited, the number of polygons allowed in a renderable scene is not as large as one would like.

A tree-based rewriting system might allow a complex scene to be rendered gradually; also, the complexity or the models can be varied with their distance from the viewer (and thus the number of pixels allotted to them in the actual rendered frame). This way, for distant objects (trees, for example), one of the first derivations would suffice, while the close-up models would be more complex.

It would be even better if the system made it possible to extract the bounding boxes of objects *without* having to know what the actual geometry looks like.

Finally, and more specifically, there are a few other, plant-specific requirements that can only be met by a plant-oriented modelling system. Later generalization might be possible, but it might result in an impractically complex rewriting system nobody would be able to use.

6. Curved limbs and surfaces

With plants, shapes are seldom perfectly straight. Branches twist, leaves curve and sometimes curl around one another. In modelling, irregular twists and turns are harder to build, partly because the proper curves

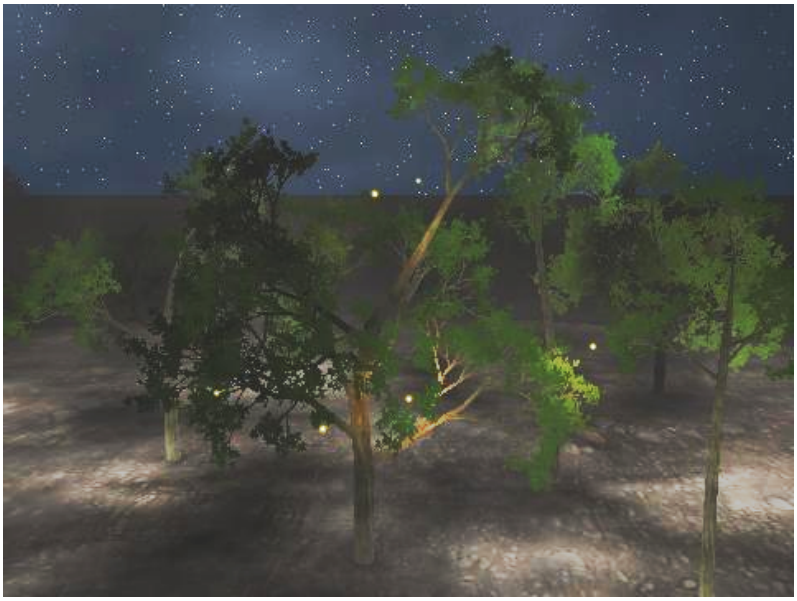

Figure 6. Trees at night

have to be computed, the joining of branches has to be done properly, and last but not least because of the collision problem adressed in the next paragraph.

7. Collision detection

Wherever three-dimensional objects grow irregularly and abundantly in a confined space (treetops, for example), the possibility of collision is given. In order to be avoided, such collisions have to be detected first. However, what happens so easily and naturally in real life becomes an expensive process where computer graphics are concerned.

8. Animation

While building a convincing plant model is a worthwhile aim in and of itself, the creation of fully animated plant models is, if achieved, even more spectacular.

There are multiple types of animation that could be incorporated in such a system — the first one that comes to mind would be growth animation. However, there are others, more general ones: the movement of plants in the wind, or the simulation of gravity in connection with growth.

9. Usability of the model

In most practical applications, modelling is only one step in a long production pipeline. It is usually done under pressure, so the generation method has to be easy to use, and the finished result has to fit into that pipeline. Particularly in special effects for feature films, commercials and similar applications, whenever plants are required, they are generated

procedurally. Nobody has the time to model a tree or any other complex plant entirely by hand.
But even with a finished model, later steps like texturing and animation have to be taken into consideration.
Texturing is a cheap method to increase the visual complexity of a model without actually having to model the details. No model, however complex and well-done, is really usable without proper texturing. However, doing the basic texturing by hand on a complex model is almost as time-consuming and undesirable as the modelling itself. Therefore it would be best if the model would incorporate its own texture coordinates along with the geometry.
Actually, this is automatically the case with models that are generated using parametric bicubic surfaces (the most popular being Bezier or NURBS: Non-Uniform Rational Bicubic Surfaces). Here, the surface would be constructed from joined 1×1 patches; since texturing implies mapping a two-dimensional image onto the surface of a tree-dimensional model, finding the (u, v) texture coordinates for every point on the parametric surface is trivial.
Therefore, apart from the abstract description of a three-dimensional shape, it is also important to consider how this shape will actually be represented in a computer, either as a polygon model, subdivision surface, or one of the aforementioned parametric surface types.
Like there are many types of representing geometry, there are also many ways of animating it. The most expensive one, both in terms of memory storage and computing time, is point-level animation (PLA). Here, several models with different shapes, but the same number of points, are being used in keyframes and morphed into each other over a set of intermediate frames. This would also be the common approach for animating things like the effect of wind on a grass field or trees.
Up to now, there is no general way of morphing models with a different number of points into each other. So, growth animation (which, apart from the change of shape, usually implies the addition of new geometry elements along the way) would have to be done in a way that nevertheless allows a smooth transition between keyframes, at the frame rate specified by the user. The only way to do this is to devise a general way of generating intermediate geometry *between* the actual derivation steps and incorporate it into the generation system.

6 Conclusion

The aspects of plant modelling presented above are valid for modelling in general, but no known modelling method implements them fully.

Both surface-based (Prusinkiewicz and Lindenmayer [PL90]) and fuzzy-object (Reeves [Ree83]) modelling methods have their advantages, and in

both cases the original method has been severely altered in order to increase the complexity of the generated images.

Therefore, the next step will be extending collage grammars in such a way that they allow curved surface creation and the incorporation of more complex stochastic controls into the picture generation process.

Also, due to the separation of generator (grammar) and interpreter (algebra), three-dimensional objects could be generated in such a way that they contain both primitive surface elements (polygons, patches) and particle clouds.

These and other issues represent the scope of future work.

9 The Midpoint-Displacement Method

Lutz Albrecht

1 Introduction

This paper presents the midpoint-displacement method, as introduced in Przemyslaw Prusinkiewicz and Mark Hammel's paper "A fractal model of mountains with rivers" [PH93]. The method allows one to generate a mountainous landscape. In the second section the method is demonstrated by some examples. In the third section the method is compared with context-sensitive L systems, and a short introduction to a possible implementation is given. The conclusion shows that there is still some work to do.

2 Midpoint-Displacement Method

The midpoint-displacement method is illustrated by an example taken from Prusinkiewicz and Hammel [PH93]. The initial triangle, which is defined by the vertices A, B and C, is subdivided into four smaller triangles by connecting the midpoints of each side of the triangle pairwise. These midpoints are displaced vertically by random values in such a way that the corresponding sides are drawn like rubber bands. These random values are called Y_i. The opposite point of the corner is used as the index. For example, in Figure 1 the displacement value of the new created point E can be named Y_C. Figure 1 shows the first step of refinement. The point E ist vertically displaced by a negative amount; the points D and F are vertically displaced by positive amounts. The illustrations are generated with Fractal Landscapes 3.0 [She96] and decorated with additional information.

This step of refinement is repeated for each triangle, until the desired number of iterations is reached. The displacement of the midpoints between neighbouring triangles must be the same. For example, in Figure 2 Y_F must be equal to Y_A. Figure 2 shows the second step of refinement.

As shown in Figure 3, the result after seven steps of refinement looks like a mountainous landscape (this time illustrated by another example).

3 Rewriting Processes

In [PH93] these steps of refinement are seen as a rewriting process and are compared with context-sensitive Lindenmayer grammars (IL-grammars).

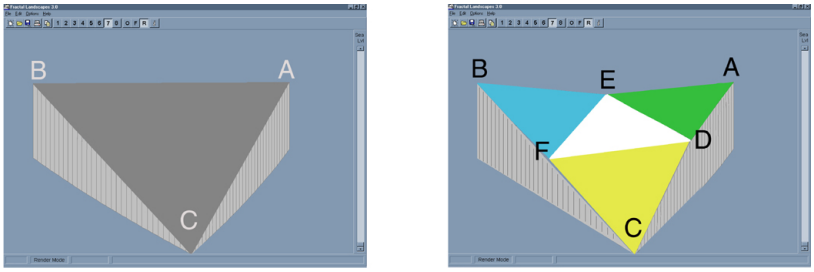

Figure 1. The initial triangle and the result after one step of refinement

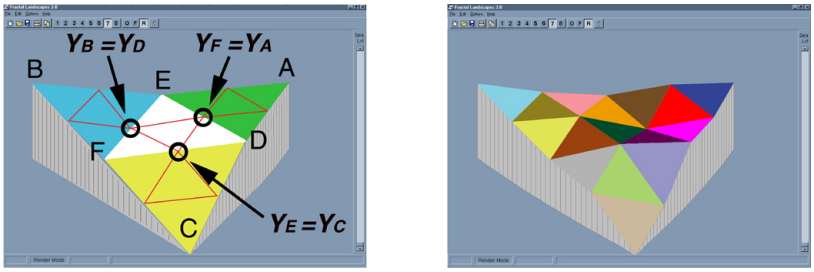

Figure 2. The second step of refinement

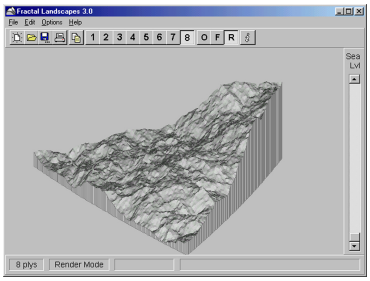

Figure 3. The result after seven steps of refinement (this time illustrated by another example)

Each time midpoints are displaced, the corresponding sides and triangle areas are automatically displaced too. So displacement depends on neighbouring sides. Figure 4 shows these dependencies (dashed lines) for the second step of refinement.

This is quite similar to the rewriting in 2L grammars, which are special cases of IL grammars (cf.,e.g.,[PL90]). Given a string $w = A_1A_2A_3...A_n$ with symbols A_i for $i = 1, ..., n$, eachA_i is replaced by a string u_i in a parallel rewriting step if the given grammar contains a rule $A_i < A_i > A_{i+1} \longrightarrow u_i$. This means that the rule requires A_{i-1} as left neighbour of A_i and A_{i+1} as right one where A_0 and A_{n+1} are optional.

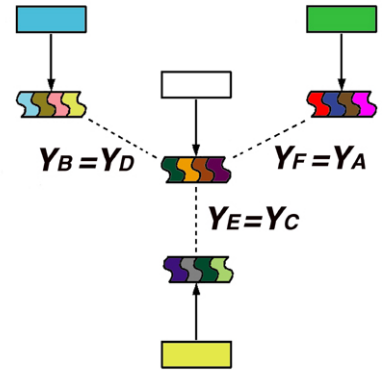

Figure 4. This figure illustrates the process of refinement. A triangle is subdivided into four smaller triangles (continuous lines) and neighbouring sides have influence on the displacement (dashed lines)

In a possible implementation of the midpoint-displacement method the context-sensitive manner can be hidden by some technique. If an edge is subdivided, a displacement for this edge is taken from a respective table whenever its neighbour edge has left an entry already. Otherwise a new random value is generated and entered into the table.

4 Conclusion

This short note about the midpoint-displacement method is based on [PH93], but the idea has its origin in [FFC82] where the method is sketched in a few lines only. The midpoint-displacement method is interesting because it provides a way to generate landscapes without time and space overhead. These landscapes have a mountainous character. The method itself is not very difficult so that it could easily be employed within AnimaLab and combined with other methods — for example, to place trees into the landscape.

Part II

AnimaLab

10 Introduction

Reiner Leins and Thomas Oliver Moll

The students' project ANIMA studied methods of syntactically generated pictures and wanted to integrate different methods of generation into one big system.

Further requirements to such a system were to give users the possibility to study the results of several types of picture grammars in one scene and to give life to these scenes so that not only still environments can be produced. The user shall be able to move around inside the scene and to watch the generation process from different points of view.

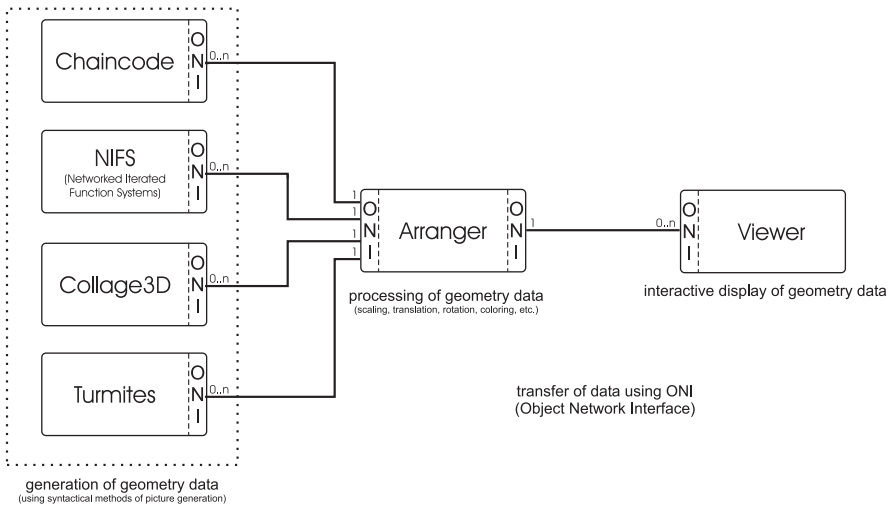

Figure 1. AnimaLab – system diagram

AnimaLab is ANIMA's implementation of these requirements. It is a distributed system designed for producing and displaying syntactically generated 3D scenes. Its network consists of components of three main types:

- The **Object Generators** are programs that produce 3D geometry data from picture grammars or other syntactic methods of picture generation.
- The **Arranger** is the front end of the system and provides a Graphical User Interface (GUI) from which the user can configure and manage the parts of the system.

- The **Viewer** is a 3D engine that reads the generated geometry data of all combined generators and visualizes them.

The data is transferred between the different components by the **Object Network Interface** (ONI).

This part of the report is organized into eight chapters corresponding to the components of AnimaLab that are actually implemented. The next chapter describes the Arranger, followed by the chapter on the Viewer. Chapter 13 concerns the Object Network Interface. Chapters 14, 15, 16, and 18 are devoted to object generators for 3D chaincode picture languages, networked iterated function systems, 3D collage grammars, and turmites, respectively. In Chapter 17, an interesting subcomponent of the *Collage3D* Object Generator is discussed that translates *Java3D* scene graphs into *VRML* scene graphs.

The chapters of this part are written in the style of a system handbook providing information about requirements, usage, command line options, configuration files, error messages, graphical user interfaces, application programming interfaces, and so on.

11 Arranger

Reiner Leins and Thomas Oliver Moll

1 Introduction

The motivation of the Arranger is to connect object generators with the Viewer and to give the user the ability of arranging the results of the object generators, controlling the data streams, and modelling a complete world of syntactically generated 3D objects and its dynamic development.

Therefore, the Arranger allows to start the different object generators on several remote computers and use the generated data streams for further processing. Also it builds up a connection to the viewer that receives all the produced 3D objects.

2 Requirements

The hardware should meet the following requirements:

- **System:** Any system where a Java VM works can be used to run the Arranger.
- **Software:** jdk or jre 1.4.1 or higher is needed.
- **Network:** TCP/IP network support is needed for connecting the Arranger to other parts of the system. the Arranger is in this case a server, so it cannot be reached through NAT (Network Address Translation).
- **Hardware:** There is no *minimum* performance, but
  - 100MBit/s Network
  - 128MB RAM
  - Pentium II class CPU with 400MHz or higher

  is recommended.

3 Command Line Options

The Arranger is started by the following command line:

```
arranger [filename]
```

where `filename` is the complete path and file name to a configuration file which will be loaded when the program starts. If no file name is given, defaults are used.

4 Graphical User Interface

Due to the complex structure of the path of the data passing the system and being processed it is necessary to use a representation that helps the user to keep the overview over components like sources, paths and destinations of the data.

The Arranger is the Graphical User Interface (GUI) of AnimaLab where the user can do the actual arranging of the components.

The main tool of the Arranger is the arranger graph shown in the diagram panel (see Figure 1). The program is controlled via the menus, and the information about the current work is shown in the status bar.

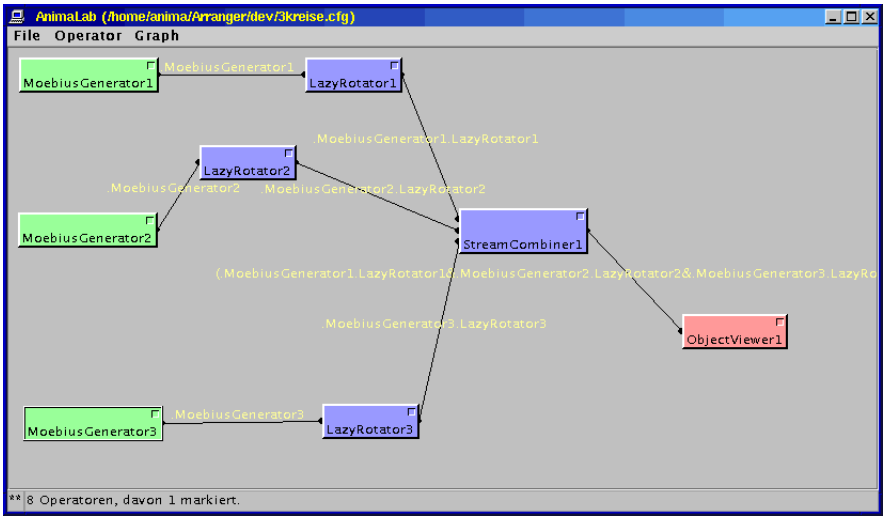

Figure 1. The main window of the Arranger with the diagram of the Arranger graph

4.1 Diagram Panel

The diagram panel shows the components of the whole system: the object generators, which are the data sources; the object operators, where the data is being processed and combined; and at least the Viewer, which is a 3D engine showing the generated scenes.

Of course, the components shown on the screen are just graphical representations (icons) of the modules working in the background (partly distributed on the net). This background is described in detail in Section 7.

To connect object generators via the Arranger to a viewer and to add manipulating object operators, the user has to arrange this icons. This builds a graph which shows the way the data is being processed.

4.2 Menus

The menus File, Operator, Graph, and Context give access to the functions provided by the GUI. This section describes the menus and their functions.

File

The items of the file menu provide the file managing functions, and the system can be run from here.

- **new:** Discards current worksheet and opens a new, empty one
- **load built-in demo:** Discards current worksheet and opens the built-in demo.
- **open:** Discards current worksheet and lets the user choose a new configuration file.
- **save:** Saves the current worksheet under its name.
- **save as:** Saves the current worksheet under a new name.
- **run operators:** Starts all operators and external parts (via network).
- **pause all:** Pauses all operators.
- **stop operators:** Stops and resets all operators.
- **quit:** Closes this programm.

Operator

This menu shows a list of all dynamically loaded operator classes (e.g. as shown in Figure 2). From this list the user can choose the new operator that is to be added.

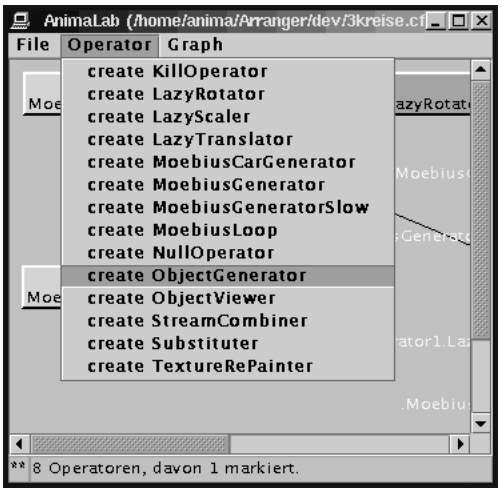

Figure 2. The operator menu shows all loaded operator classes

Graph

This menu provides a single item.

- **arrange graph:** This function *reorganizes* the arranger graph so that the icons fit on the screen.

Context Menu

- **refresh screen:** Redraw the contents of the diagram panel.
- **rearrange the graph:** Same as *arrange graph* from the graph menu.
- **show/hide properties:** Opens or closes the property windows of the marked operators.
- **delete marked operators:** Remove all marked operators from the graph.
- **create operator:** Same as the operators from the *Operator menu*.

4.3 Status Bar

The status bar shows information about the current graph (e.g. whether it is saved.

4.4 Usage

This section describes how to use the basic functions

Adding Operators

To add an operator the user has to choose the desired operator. This operator appears in the top left corner of the diagram panel or at the place where the context menu was opened. It is still unconnected and unconfigured, so a configuration window will be opened.

Configuring Operators

To configure an operator the user has to open a configuration window by choosing *show/hide* from its context menu. Every operator has its own parameters which can be viewed and edited from the configuration window (see Figure 3).

- **Name:** The unique name of this operator.
- **Color:** The RGB values that describe the color of the icon on the diagram panel.
- **xPos/yPos:** The coordinates of the icon on the diagram panel.
- **maxIn/maxOut:** The maximum amount of inlets and outlets of this operator (these values should not be changed, because they depend on the type of the operator).
- **others:** Type-specific fields and values which vary between the different operator types.

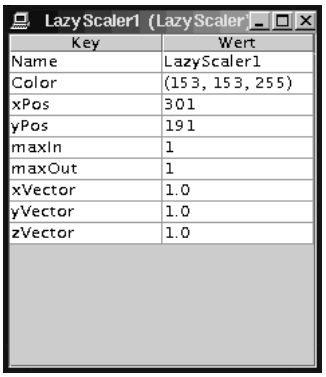

Figure 3. The configuration window of an operator

Connecting Operators

To tell the arranger that the results of an operator have to be transferred to another operator for further processing, the user has to draw a line from the data source to its destination using the right mouse button.

Removing Operators

To remove an operator, the user has to click on the icon with the right mouse button and choose *delete marked operators* from the context menu. If additional operators are marked, these will be deleted, too.

Running the System

When all parts of the system are connected correctly, the user can run, pause and stop the generation process from the file menu (see Section 4.2).

5 Configuration Files

Configuration files are used to store a configuration as arranged by the user in a file.
This file is a text file with a simple format construction. The parser for this format is not fault tolerant. So the integrity is not checked.

5.1 Format of a Configuration File

A configuration file consists of any number of sections. Every section has the same structure:

```
<TypeOfItem> <NameOfItem> {
        <Key> = <Value>;
};
```

A section describes an item of the Arranger graph which can be an `Operator` or a `Stream` (see Section 7). A section can have any number of key/value pairs. If a key is given but not understood or used by an `Operator` or `Stream` it will be ignored.

Every configuration file has to have a section with `Arranger` as its type and name. In this section the user can define default values for the `Arranger`.

The naming convention of `Stream` names is described in detail in Section 7.1 in detail.

5.2 Sample Configuration File

The following sample configuration file would instructs the Arranger to load two instances of the generator class `MoebiusGenerator`, to merge their output, and to send the data to a computer with a network name "viewer" (at Port 30001).

```
Arranger Arranger {
        pathToSSH = /usr/local/bin/ssh;
        defaultHostName = ;
        defaultUserName = ;
        defaultViewerHostName = viewer.informatik.uni-bremen.de;
        defaultViewerPort = 30000;
        defaultPort = 50000;
        defaultBufferSize = 100000;
};

MoebiusGenerator MoebiusGenerator1 {
        Color = (153, 255, 153);
        xPos = 10;
        yPos = 5;
        maxIn = 0;
        maxOut = 1;
        Radius = 25;
};

MoebiusGenerator MoebiusGenerator2 {
        Color = (153, 255, 153);
        xPos = 10;
        yPos = 47;
        maxIn = 0;
        maxOut = 1;
        Radius = 25;
};

StreamCombiner StreamCombiner1 {
        Color = (153, 153, 255);
        xPos = 256;
        yPos = 42;
```

```
        maxIn = 13;
        maxOut = 1;
        Type = waitForObject;
        TickType = 0;
};

ObjectViewer ObjectViewer1 {
        Color = (255, 153, 153);
        xPos = 388;
        yPos = 47;
        maxIn = 1;
        maxOut = 0;
        Hostname = viewer;
        Port = 30001;
};

Stream .MoebiusGenerator1 {
        Source = MoebiusGenerator1;
        Destination = StreamCombiner1;
};

Stream .MoebiusGenerator2 {
        Source = MoebiusGenerator2;
        Destination = StreamCombiner1;
};

Stream (.MoebiusGenerator1&.MoebiusGenerator2).StreamCombiner1 {
        Source = StreamCombiner1;
        Destination = ObjectViewer1;
};
```

6 Error Messages

There are no standard error messages because incorrect manipulations of the graph are ignored. The plausibility of the data in the configuration windows is not checked.

7 Background

7.1 Data Streams

This section describes the data, its behaviour and handling.

Data

All data processed by the Arranger consists of 3D descriptions of geometric objects and some control sequences. Since all data has to be interpreted by

the Viewer, it is organised as commands. These commands tell the Viewer to add 3D objects to its object library or to add instances thereof to the current scene. Further commands control the handling of the data streams (see, e.q. Section 7.4).
The commands used in arrangers data streams are transmitted by ONI. For a complete list of the commands see Section 6.1 of Chapter 13.

Special attention should be paid to the non-terminals. These commands are produced by the generators and are not intended for the Viewer. Non-terminals describe the position and size of whole generated pictures which are to be inserted in the current scene. Correctly configured, the Arranger will replace them by actual data by calling additional object generators (see Section 7.2).

Why Does the Data Stream?

The process of generating data is that slow that you can see the data "tumbling" through the net. So the Arranger treats the results of the object generators not as a static amount of data, but as a data stream where we can interfere and modify the data before they reach the viewer.

Data Stream Nomenclature

For communicating with the viewer, unique names are needed for the data streams which gives the user the ability to easily describe a group of 3D objects that has passed a certain part of the arranger graph. The name of a data stream starts with a dot[1] followed by the identifier of the object generator (e.g., ".S1" for a "Sierpinsky" object generator). Each object has its own name within a data stream (e.g., ".S1.obj1"). Every object operator which modifies a stream, extends the name by a dot and its own identifier (e.g., ".S1" modified by the object operator "mod1" results in the stream name ".S1.mod1". An object in this stream could, therefore, be called ".S1.mod1.obj1"). To be able to understand the name of a combined stream, we have to clearly show that the new stream is a combination of two other streams. This is done by the concatenation of these two names which are separated by an ampersand and enclosed with round brackets. Objects keep their full names even if its stream was combined with another stream (e.g., two streams ".S1" and ".S2" combined with one another would conclude in a stream called "(.S1&.S2).".

7.2 Arranger Graph

Since we can see the data streams as connections between components of the system, we visualize all the components and all the data streams as a graph.

[1] The leading dot indicates the root of the stream.

Edges

The data streams are represented in this graph by directed edges.

Nodes

Every component of the system which generates, consumes or modifies a data stream is represented by a node.

- Object generators: A node with one outgoing edge.
- Object viewer: A node with one incoming edge.
- Object operator: A node with several incoming and outgoing edges.
- Non-terminal substitutor: A node with an incoming and an outgoing edge that has an additional inlet-port to which object generator classes can be connected. The non-terminal is replaced by actual data provided by some object generator.
- Object generator class: A node with one outgoing edge, representing a set of streams, which is generated by a set of object generators described by this class.

7.3 Object Generators

A generator is a component of the system which generates a stream of objects.

Base Object Generator

These generators are part of the arranger and are used to generate a simple "stream" which returns just one object:

- Sphere
- Polyhedron
- Color

Möbius Generator

The Möbius generator is used to generate an animated möbius loop. As it starts it generates pyramids. The floor of these pyramids are the Möbius strip, so two full loops must be generated to have on Möbius strip. After the generation of one loop, this generator removes all pyramids step by step. This process does not have an end.

Network Connected Object Generators

Object generators can be started on different machines, so the load can be distributed over the whole network. The object generator is started by the arranger on the remote machine. This takes place with the standard UNIX "ssh" or "rsh" command. On start-up the object generators get a set of arguments with all necessary data to start the object generation and to connect to the arranger. The generated data is sent to the arranger via the Object Network Interface (ONI, see Chapter 13).

7.4 Object Operators

Parameter Modification

This operator modifies any of the parameters of the objects. The possible parameters could be for instance:

- Matrix
- Texture
- Colour

Multiplier

This operator duplicates a data stream.

Combiner

This operator merges two data streams into one. There are several ways to combine the objects of these streams:

Tick-wise For this option the object generators have to generate so-called ticks. A tick tags the ends of units of time. All objects between two ticks have to be regarded as arriving simultaneously. In this case the output stream would have just one tick for two incoming ticks.

ONI provides several types of ticks which can be distinguished by an integer value. So it is possible to tag different time units by choosing different ticks. Some of these ticks are reserved for special cases like "End-Of-Stream".

Object-wise The two streams are merged object by object. If one stream stops sending objects this operator waits until an object arrives, and all other streams must yield to the stream that stopped sending.

Buffer-wise This is the fastest way to merge streams together, because it always lets pass an object if possible. The buffers of the streams to be merged are emptied rotationally. You cannot tell anything about the order the objects of the streams are combined. If both streams are slower than the maximum processing speed of this machine, this combiner will waste a lot of CPU power.

7.5 Object Consumer

Object consumers are nodes of the arranger graph with no outgoing streams. They consume the data they get, which also means exporting it out of the Arranger to a different component process, or even into a file. The main consumer is the the Viewer.
Every arranger graph must have at least one consumer, because no stream ends in the void.

Connection to the Viewer

The Viewer has to be started before the Arranger tries to connect to it. The composer connects itself to the waiting Viewer using the ONI.

8 Application Programming Interface (API)

Extending the Arranger is done by adding further `Operator`s. All available `Operator`s are loaded at startup, so they can be used within, the program where the type (generator, operator or consumer) does not matter.

8.1 The Operator Class

A new `Operator` has to extend the `Operator` class. The following code abstract shows the methods that are typically to be adapted to the user's needs.

```
public abstract class Operator
       extends java.lang.Thread
       implements Shape {

   protected int initialized = 0;

   protected boolean thread_ok = true;
   protected boolean running = false;
   private boolean paused = true;

   public Operator() {
          color = new Color(0.6f,0.6f,1f);
          name = "";
          maxIn = 0;
          maxOut = 0;
          inPortNum=0;
          outPortNum=0;
          paused = true;
          xPos=0;
          yPos=0;
          width=32;
          height=32;
          out = new Vector();
          in = new Vector();
   }

   public Operator(String aname,
                   int acountIn,
                   int acountOut) {
          this();

          name = aname;
```

```
        maxIn = acountIn;
        maxOut = acountOut;
    }

    public Operator(String aname,
                    int acountIn,
                    int acountOut,
                    int x,
                    int y) {
        this(aname, acountIn, acountOut);

        xPos = x;
        yPos = y;
    }

    public Vector getFieldlist() {}
    public boolean setProperty(String property,
                               String value) {}
    public String getProperty(String property) {}

    public abstract void operate();

    public void initialize()
           throws OperatorNotReadyException {}
    public void cleanUp() {}
}
```

8.2 Adapting Constructors

The fields of an `Operator` are mainly configured by the `Property` methods, so the `Operator`s have to be handled with care. Some of the standard fields have to be set to default values which is normally done by the constructor `super()`. This means that usually no new constructor is needed.

8.3 Adapting Properties

All properties are accessed by the methods `setProperty` and `getProperty`. To do so own fields have to be declared previously.

Declaration of Own Fields

Declaration of own fields is done by overwriting the method `getFieldList()` so that it returns a `Vector` containing the fields of the parent class and the own fields. For example see the following code snippet.

```
public Vector getFieldlist() {
        String[] internalFieldList = {"xVector", "yVector"};
        Vector tmp = new Vector(super.getFieldlist());

        tmp.addAll(Arrays.asList(internalFieldList));

        return tmp;
}
```

Setting the Values of Own Fields

To enable setting a value, the method `setProperty()` has to be adapted so that the values given as key/value pairs are assigned to the corresponding member variables. The member variables of the parent class have to be assigned by calling the method `super.getProperty()`. For an example see the following code snippet.

```
public boolean setProperty(String property,
                           String value) {
        if (property.equalsIgnoreCase("xvector")) {
            xvector = Double.parseDouble(value);
        } else if (property.equalsIgnoreCase("yvector")) {
            yvector = Double.parseDouble(value);
        } else {
            return super.setProperty(property, value);
        }

        return true;
}
```

Readout of Own Fields

To enable reading out own fields, the method `getProperty()` has to be overwritten. This method returns all values as a `String` so it may be necessary to convert them. The method `super.getProperty()` has to be called, too. For an example see the following code snippet.

```
public String getProperty(String property) {
        if (property.equalsIgnoreCase("xvector")) {
            return String.valueOf(xvector);
        } else if (property.equalsIgnoreCase("yvector")) {
            return String.valueOf(yvector);
        } else {
            return super.getProperty(property);
        }
}
```

8.4 Adapting the Work Thread

Every `Operator` has a working loop calling the method `operate()`, which should contain the code for the smallest possible working step. The `operate()` method of an *object generator* could generate one object or process one object if it is an *object operator*.

The following code snippet shows the `operate()` method of the `StreamCombiner` class which is shipped as a standard `Operator` class.

```
public void operate() {
        StreamCommand cmd = null;
        Port outport = ((Port)out.firstElement());
        Port inport = ((Port)in.elementAt(streamnr));
        Stream outStream = outport.getStream();
        Stream inStream = inport.getStream();

        try {
            if ((inStream.available() > 0)
                || (type >= waitForObject)){
                cmd = inStream.readObject();
                cmd.addName(name);
            } else {
                streamnr++;
            }

        } catch(StreamCorruptedException e) {
            in.remove(streamnr);
        } catch(IOException e) {
            in.remove(streamnr);
        }

        if (cmd != null){
            try {
                outport.getSourceOutputStream().writeObject(cmd);
            } catch(StreamCorruptedException e) {
                resetThread_ok();
            } catch(IOException e) {
                resetThread_ok();
            }

            outport.getStream().trigger(new Date());

            if (type <= 10){
                streamnr++;
            }

            if (type == tickWise){

                if (cmd instanceof Command_tick){
```

```
                    Command_tick tick = (Command_tick)cmd;

                    if (tick.getType() >= ticktype){
                        streamnr++;
                    }
                }
            }

            if (type == tickWiseStrict){
                if (cmd instanceof Command_tick){
                    Command_tick tick = (Command_tick)cmd;

                    if (tick.getType() == ticktype){
                        streamnr++;
                    }
                }
            }
        }

        if (streamnr >= in.size()){
            streamnr = 0;
        }

        if (in.size() == 0){
            resetThread_ok();
        }
}
```

This `Operator` reads from all input streams and writes all data to a single output stream. Therefore it iterates through the list of input streams and writes the read data to the first output stream (which should be the only one).

The incoming and outgoing `Port`s are held in two `Vector`s named `in` and `out` from where they can be accessed. The `Port`s provide a method `getStream()` by which the stream can be addressed.

When the operation comes to its end the method `resetThread_ok()` should be called to gracefully stop the execution of this `Operator` thread.

To actually get data for processing, the method `readObject()` from a `Stream` is used to get the next object from this `Stream` which can be written to the output streams with the `writeObject()` methods. Everything that happens between this is freely definable.

8.5 Initializing and Clean-up

If something has to be done before an `Operator` thread can be started, it has to be done in the method `initialize()`. The member variable `initialized`

has to be set to `100` if the initialization was successful. To show the progress of initialization it can be increased step by step.

If something is to be done after the execution of an `Operator` thread to reset this `Operator` to a state where it can be started again, this has to be done in the method `cleanUp()`. The member variable `initialized` has to be set to `0` if the cleanup process was successful. To show the progress of cleaning up it can be decreased step by step.

12 Viewer

Thomas Meyer and Rafael Trautmann

1 Introduction

Within the scope of AnimaLab syntactic generation of three-dimensional scenes was not the only focus, but also the possibility of viewing a generated scene. The complex form of a 3D objects is hardly comprehensible when seeing it as a two-dimensional picture. The ability to view a three-dimensional scene from different angles and even move through it greatly increases the possibility to grasp it. A program which is able to display such scenes was needed. The choice was to either take and modify an existing program or to write a new one from scratch. Such a program must allow the modification of the displayed data during runtime over the network and take into account the special requirements of syntactically generated pictures i.e. a large amount of polygons and objects. No program that satisfied both requirements was found. So we decided to write one from scratch. The Viewer was born.

This document is primarily a manual for the Viewer. It explains in the first four chapters how to start, configure and use the Viewer. Furthermore the last chapter grants some insight into the development and used concepts of the Viewer.

2 Command Line Option and Usage

2.1 Setting the Environment Variables

The Viewer is using a couple of shared libraries. They are included, but it is necessary to set the environment variable `LD_LIBRARY_PATH` properly to enable the operating system to find them. This is usually done by the start script. If the usage of the start script is not wished then this must be done manually with the following command:
export LD_LIBRARY_PATH=<path to Viewer>/lib/:$LD_LIBRARY_PATH.

2.2 Starting the Viewer

The Viewer is started by the following command:

```
Viewer [OPTIONS] [FILENAMES]
```

where *OPTIONS* are optional and can be one or more of:

-help : print the list of the allowed options with a short description.
-W <*pixels*> : specify the width of the window in pixels.
-H <*pixels*> : specify the height of the window in pixels.
-h <*host*> : the name of the host to connect with.
-p <*port*> : the number of the port which will be used for connection. When the host is defined this is the port the Viewer will connect to. Otherwise the Viewer waits on this port for incoming connection.
-tex : turns texturing on.
-alphatest : cause the Viewer not to draw some parts of a polygon depending on the alpha value of the texture of the polygon.
-e : start the Viewer with control panel, which can be used to set and change some of the parameters like lighting and appearance of objects
-c <*filename*> : cause the Viewer to load a configuration file specified by *filename*. If this option is not used the default configuration file named default.cfg is loaded. If the default configuration file is not present the Viewer use its internal configuration. All settings in a configuration file override the internal configuration and are overriden by the options.

The options may be followed by a list of scene files as described in Section 4. The objects defined in these files will all be loaded and displayed.

2.3 Navigating Through a Scene

Once the Viewer is running, the following keyboard commands can be used to navigate through the displayed scene:

h : show or hide the helpscreen
w : move forward
s : move backward
a : move right
d : move left
r : move up
f : move down
cursor up : look up
cursor down : look down
cursor right : turn right
cursor left : turn left
p : make screenshot
ESC : quit

2.4 Editor Window

The editor window can be used to change the parameters of the displayed scence.

The upper four input fields grouped in the panel called "Light" allow to change the parametrs of the ambient light (see Section 3.1 of Chapter 2). To modify the colour of the light the amount of red, green and blue colour components must be entered into the corresponding input fields. The allowed values are ranging between 0 and 1. The entry in the fourth input field changes the intensity of the light. Allowed are values between 0 and 100.

The control elements placed on the panel called "Material" allow to manipulate the materials of the entities and primitives. These concepts are explained in Section 5. The input field "Material Id" can be used to select a defined material. After a material is chosen, its name and if present the name of its texture will be displayed in the corresponding text fields. It is now possible to change the colour properties of this material. To do this the values for the colour components must be entered into the input fields. Allowed are again values between 0 and 1. The meaning of ambient and diffuse colour is explained in Section 3.1 of Chapter 2. The emissive colour is the colour of the light that is emitted from this object. It will be added to the amount of the light reflected from the object.

The mouse cursor can be used to select a displayed object (entity) by clicking on it. After an object is selected, the button "EntMat" will select the material of this entity while the "PrimMat" button will select the material of the corresponding primitive.

The controls grouped on the "Entity" panel provide another method, beside selecting it with the mouse, to select one entity and change its material. The entity can be selected by entering the identification of the entity into the input field "Entity Id". After an entity is selected, its name and the name of the primitive it is based upon will be displayed in the corresponding text fields. The identification of the current material of this entity will be displayed in the input box "Material Id". By changing this Identification the material of this entity will be changed. The identification -1 means that the material of the primitive the entity is based upon will be used for this entity.

3 Configuration File

3.1 Commands

All commands shown here may appear in a random order. Every command occupies one line with the following syntax: $CommandName = Params$. The parameters and effects for each command are described below. Some are simple switches that are turned on and off; to set a switch to on the parameter must be 1, yes or on, everything else means off.

WindowSize

Size of the window in pixels.
Format: $width \times height$

StartPosition

Initial position of the camera.
Format: (x, y, z)

ClearColor

This background colour.
Format: (r, g, b)

FogEnabled

Switch, turns fog on/off.

FogColor

Colour of the fog.
Format: (r, g, b)

FogRange

The first value is the distance where the fog becomes visible and the second is the distance where the fog totally obscures the view.
Format: ($begin$, end)

FogDensity

Density of the fog, the greater the value the thicker the fog.
Format: $density$

AlphaTestEnabled

Switch, if enabled some parts of a polygon may not be drawn depending on the alpha value of its texture. A pixel is drawn when the current alpha value on the texture is greater than 0.2 (in a range from 0 to 1).

TexturesEnabled

Switch, turns texturing of polygons on/off.

EditorEnabled

Switch, turns the editor window on/off.

Host

The name of the host to connect with.

Port

The number of the port which will be used for connection. When the host is defined this is the port the Viewer will connect to. Otherwise the Viewer waits on this port for incoming connection.

MaxReceive

This defines how many ONI commands (see Chapter 13) should be processed between the drawing of two frames.

ScreenShotDirectory

Directory in which screenshots should be saved.

ScreenShotFileName

Common name of the sequentially numbered images (i.e name.ext will become name0000.ext to name9999.ext).

3.2 Example

```
WindowSize=640x480
StartPosition=(0, 0, 0)
ClearColor=(0,0,0)
FogEnabled=1
FogColor=(0.5, 0.5, 0.5)
FogRange=(10,50)
FogDensity =  1

AlphaTestEnabled=0
TexturesEnabled=1
EditorEnabled=

Host=127.0.0.1
Port=21
MaxReceive=

ScreenShotDirectory=./screens/
ScreenShotFileName=capture.png
```

4 Scene Files

4.1 File Structure

A scene file consists of a series of commands, where every line is seen as one command. All elements that are defined through a command are consecutively numbered, beginning with 0. The following commands may be supplied in any order, but note that only already defined elements can be referred through the index.

Vertex

The basic element for constructing geometric objects.
Syntax:
`v` x y z

Triangle

Constructed from three vertices, this is the most primitive geometric object. A number of triangles combined can create more complex structures.
Syntax:
`t` a b c [s_a t_a s_b t_b s_c t_c]
Description:
a, b, c are the corners of the triangle and s_x t_x are the texture coordinates associated with the corner/point x. If the texture coordinates are supported, all six have to be present. Otherwise, their values are undefined.

Texture

An image that can be mapped onto a triangle in a way that is defined by its texture coordinates.
Syntax:
`i` n f
Description:
n is the name this texture is bound to, while f is the image to be loaded.

Material

The overall appearance of an object, that is its colour and perhaps a texture too.
Syntax:
`a` n i a_r a_g a_b a_a d_r d_g d_b d_a s_r s_g s_b s_a e_r e_g e_b e_a *shininess*
Description:
n is the name of the material and i is the index of the texture to be used. a_λ, d_λ, s_λ, e_λ and *shininess* are the material properties (see Section 3.1 of Chapter 2).

Mesh

A collection of triangles and a material that represent an object.
Syntax:
`m` $a\ m\ i_1\ \ldots\ i_m$
Description:
a is the index of the material to be used, or -1 if the default material is to be used. m is the number and i_1 to i_m are the indices of the triangles that form this mesh.

Primitive

A complex object that may consist of multiple meshes.
Syntax:
`p` $n\ i_1\ \ldots\ i_m$
Description:
n is the primitive name and i_1 to i_m are the indices of the meshes that are part of this primitive.

Entity

Syntax:
`e` $n\ p\ a\ m_{11}\ m_{12}\ m_{13}\ m_{14}\ m_{21}\ m_{22}\ m_{23}\ m_{24}\ m_{31}\ m_{32}\ m_{33}\ m_{34}\ m_{41}\ m_{42}\ m_{43}\ m_{44}$
Description:
n is the entity name and p the primitive name that is used for this entity. a is the index of the material to be used, -1 if none. m_{11} to m_{44} give the following 4×4 matrix

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix}$$

which positions the entity in the scene. For further information about transformation matrices see Section 2.3 of Chapter 2.

Comments

Any line that does not begin with one of the above key characters is seen as a comment. Additionally a line beginning with # is not processed.

4.2 Example

This example shows two different ways to create texture and position cubes so that the resulting scene looks like a stack of crates (Figure 1).

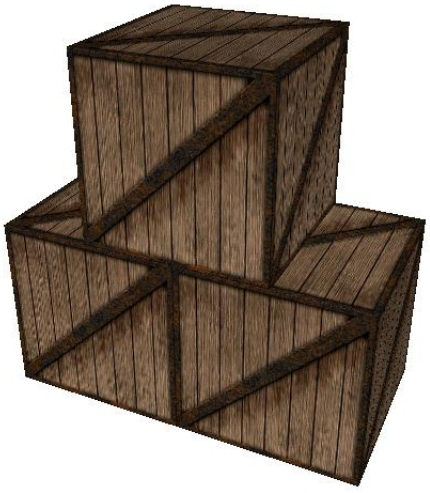

Figure 1. Stack of crates

```
# the four lower points
v -1 -1  1  # front-left  | index 0
v  1 -1  1  # front-right | index 1
v -1 -1 -1  # back-left   | index 2
v  1 -1 -1  # back-right  | index 3
# the four upper points
v -1  1  1  # front-left  | index 4
v  1  1  1  # front-right | index 5
v -1  1 -1  # back-left   | index 6
v  1  1 -1  # back-right  | index 7

# the front
t 0 4 5   0 0   0 1   1 1 # index 0
t 0 5 1   0 0   1 1   1 0 # index 1
# the back
t 3 7 6   0 0   0 1   1 1 # index 2
t 3 6 2   0 0   1 1   1 0 # index 3
# the left
t 2 6 4   0 0   0 1   1 1 # index 4
t 2 4 0   0 0   1 1   1 0 # index 5
# the right
t 1 5 7   0 0   0 1   1 1 # index 6
t 1 7 3   0 0   1 1   1 0 # index 7
# the top
t 4 6 7   0 0   0 1   1 1 # index 8
t 4 7 5   0 0   1 1   1 0 # index 9
# the bottom
t 2 0 1   0 0   0 1   1 1 # index 10
```

```
t 2 1 3   0 0   1 1   1 0 # index 11

# a texture
i tex1 textures/crate.png # index 0

# a material using the above defined texture
a crate 0 0.2 0.2 0.2 1.0 0.8 0.8 0.8 1.0 0.0 0.0 0.0
          1.0 0.0 0.0 0.0 1.0 0.0

# a mesh including all defined triangles
m 0 0 1 2 3 4 5 6 7 8 9 10 11 # index 0
# 6 meshes each representing on side of the cube
m 0 0 1   # index 1
m 0 2 3   # index 2
m 0 4 5   # index 3
m 0 6 7   # index 4
m 0 8 9   # index 5
m 0 10 11 # index 6

# 2 primitives, both create a crate.
# The first by using a single mesh, the
# second by using one mesh for each side
p prim1 0
p prim2 1 2 3 4 5 6

# 3 entities creating a stack of crates
e crate1 prim1 -1  1 0 0  1  0 1 0 0  0 0 1 0  0 0 0 1
e crate2 prim1 -1  1 0 0 -1  0 1 0 0  0 0 1 0  0 0 0 1
e crate3 prim2 -1  1 0 0  0  0 1 0 2  0 0 1 0  0 0 0 1
```

5 Background

5.1 Language and Tools

The Viewer is written from scratch using *C++*. The resulting code from a C++ compilation is very efficient compared to some other high level languages.

This program was developed under Linux using the *Gnu Compiler Collection* (GCC). GCC is an open source compiler for multiple languages, among others C++. The Viewer was not ported to any other operating system, but due to the portability of C++ it is possible with only minor source code modifications.

In order to be able to display complex scenes at a reasonable speed, the use of hardware acceleration is inevitable. Currently there are two major APIs for the development of 3D programs, *OpenGL* and *DirectX*. Since OpenGL is available for multiple platforms and is less complex than DirectX, which is

only available for Windows, the former was selected. OpenGL was originally developed in 1992 by Silicon Graphics, as a descendant of an API known as Iris GL for UNIX. It was created as an open standard. Mark Segal and Kurt Akeley describe OpenGL in [SA99] as follows:

> OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. [...] To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

Instead of writing our own image loading and saving routines, the *DevIL* (formerly OpenIL) library was used. Developer's Image Library (DevIL) is a programmer's library to develop applications with image loading capabilities that is easy for a developer to learn and use. Additionally *GLUI* is included in the Viewer. GLUI is a C++ user interface library based on GLUT [1], which provides controls such as buttons, checkboxes, radio buttons, and spinners to OpenGL applications.

5.2 Data Organization

Due to the nature of syntactic picture generation, the amount of created objects is likely to become very large, but the objects are usually transformated copies of a small number of "base objects". This behaviour leads very fast to a huge amount of data when all the transformed objects are stored. To prevent this a concept of primitives and entities was introduced. A primitive can be seen as a template how to build an object like a cube, for example. It contains all the data about the object shape, but not its position. So a primitive does not appear anywhere in the world. To create a visible object specified by a given primitive, an entity of this primitive must be created. The entity consists of a reference to a template and information about its position in the world. The information about the position is stored in a transformation matrix. Besides the position, a transformation matrix can contain information about the orientation, size and more. The transformation matrix and its capabilities are described in Section 2.3 of Chapter 2. If only one object of any type is present, this system uses more space than needed because a primitive and an entity exist. But as soon as two or more objects of any type

[1] GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL.

are placed in the scene, less space is needed to store these objects. Although each primitive has its own appearance[2], an entity may overwrite this to give each entity own colours and textures.

5.3 Data Acquisition

The Viewer supports two ways of generating a scene. The first is to load all needed data from one or more files, as described in Section 4. The second is to receive data or modifications during runtime through the Object Network Interface (ONI). ONI provides a simple-to-use interface to send object data over a network. For more detailed information about ONI see Chapter 13.

Since the Viewer is used for displaying and not for generating objects, only some part of the receive and none of the send functionality is implemented. These are the commands the Viewer receives and processes:

- oni_loadTexture
- oni_destroy
- oni_modify
- oni_create
- oni_addPrim
- oni_destory

A detailed description of these commands can be found in Section 6 of Chapter 13.

[2] All the properties that define the appearance of an object, like colour and texture, are stored in a concept called material.

13 ONI – Object Network Interface

Lars Fischer

1 Introduction

> *Oni* are mysterious japanese creatures which, unlike *Kami*, torment mankind. *Oni* can be thought of as evil spirits or demons.

The Object Network Interface (ONI) is the network interface of AnimaLab. It provides a simple-to-use interface for the different modules through which they are able to communicate with each other. The emphasis lies on transporting data on graphical objects. This data will mostly be generated by one kind of modules and displayed and modified by other modules, with the modules possibly running on different hosts.

The idea is that because both derivation in a grammar and displaying graphical objects takes a lot of computing power, these two tasks should not be run on the same computer.[1] The Object Network Interface is the glue between all the parts of AnimaLab. Its job is to establish connections between the different parts as well as to provide an easy-to-use interface for the programmers of the different parts of AnimaLab.

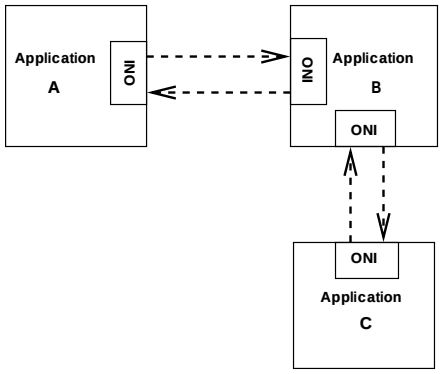

Figure 1. Three applications connected by ONI

[1] The computers that were available to the project ANIMA were only PCs, but we could obtain a lot of them in our local network.

Figure 1 shows an example with three applications which are connected by ONI. As shown, an application can actually have more than one ONI-connection. All connections of an application are handled separately. Furthermore, ONI-connections are always bi-directional.

2 Usage

2.1 C Interface

The following is an example of an ONI-session in C.

```
#include <ONIinterface.h>
```

First include the ONI library. Then define all receiving functions that will be used. In the two examples defined below all the bodies are left empty.

```
static void
   rx_NTset(connection con, void* data,char** nt_list){
}

static void
   rx_combPrim(connection con, void* data, char* name,
               char** prims, transformation*** trafos){
}
```

In the next step we construct and initiate a `callbacks` structure. Then we accept an incoming connection. It is possible to both send and receive on a connection, but here we only receive until the connection is broken. Last but not least, we clean up after we are finished, and close the connection.

```
int main(int argc, char** argv){
      struct callbacks cbf = {0};
      connection con ;
      int port = 3333;


/* initiate the callback functions */

      cbf.rx_createComp = 0;
      cbf.rx_createTex = 0;
      cbf.rx_destroy = 0;
      cbf.rx_modify = 0;
      cbf.rx_start = 0;
      cbf.rx_stop = 0;
      cbf.rx_pause = 0;
      cbf.rx_addPrim = 0;
      cbf.rx_combPrim = rx_combPrim;
```

```
      cbf.rx_tick = 0;
      cbf.rx_NTset = rx_NTset;

      if(1 < argc)
            port = atoi(argv[1]);


/* wait for an incoming connection */

      con = oni_getconnect(port, 0, cbf);

      printf("-------- got connection\n");

      printf("receiving\n");
      fflush(stdout);


/* Receive 20 messages maximum and wait for them
                                      forever. */

      while(0 <= oni_recv(con,0,20,-1)){
            printf(".");
            fflush(stdout);
      }

      printf("\n");
      fflush(stdout);

      printf("-------- closing connection\n");

      oni_closeconnect(con);
      return 0;
}
```

2.2 Java Interface

The following example illustrates the usage of the Java implementation of ONI.

In this example we simply open a connection and send two commands, `ntSet` and `create`. The first parameter of the constructor for the `Connection` is the host address, the second parameter is the port address, and the last one is an empty `Receiver` object, which is instantiated just before the constructor invocation.

```
public static void main(String[] args){
    if(args.length < 2)
          USAGE();
```

```
Receiver receiver;
Connection con;
```

Instantiate (and therefore start a new connection) a `Connection` and a `Receiver` object. The `Receiver` is an interface, which has to be implemented by the users of ONI. The constructor of `Connection` throws an exception if the connection setup has failed.

```
try{
      receiver = new RecvTest();
      con = new Connection(args[0], Integer.parseInt(args[1]),
                           receiver);
      System.out.println("[SendTester] build connection");
```

Now call two commands from the `Connection`-instance, `NTset` and `create`. Each of them throws an exception if anything goes wrong.

```
      String[] nts = {"Erde", "Mond", "Mars", "Hustensaft"};
      try{
            con.NTset(nts);
      }
      catch (Exception e){
            System.out.println(e.getMessage());
      }

      System.out.println("send NTset");

      Matrix[] trafos =
               {new Matrix(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4),
                new Matrix(3,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4),
                new Matrix(2,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4)
                              };

      Matrix[][] nt_trafos =
             {{new Matrix(1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6),
               new Matrix()},
              {new Matrix(), new Matrix(), new Matrix()}};
      System.out.println("sending create");
      try{
            con.create("pier1", "primitivling", trafos , nts,
                       nt_trafos);
            System.out.println("send create");

      }
      catch(Exception e){
            System.out.println("died while sending create: " +
```

```
                e.getMessage());
        }
```

When everything is finished, just close the connection.

```
        System.out.println("closing connection");
        con.close();

    }
```

Lastly comes the `catch` clause for any failure of the connection setup.

```
          catch(ConnectionFailedException e){
        System.out.println("Connection failed: "+e.toString());
        System.exit(1);
    }
}
```

3 Requirements

ONI is implemented based on BSD-sockets. It should therefore be used on a computer with Linux as operating system, because we have not ported it anywhere else. It might be possible to run it on any Unix-like operating system, but there might be some small patches required.

There are no special requirements for using the C implementation.

For using Java, the requirements are:

- Java 2: j2sdk 1.3.0 or higher.

3.1 Required Qualifications for the Users

AnimaLab has been developed for people who have some basic knowledge of computers and networking: they know about command lines and are comfortable with host names and port numbers. Therefore it is not necessary to make the whole process of setting up an end-to-end connection transparent to them. Instead, the users of AnimaLab must be able to determine the name of the host on which their modules should run. ONI does not implement host-finding via multicast.

The users of ONI itself are the programmers of the various parts of AnimaLab. They should be able to distinguish between opening a connection and waiting for a connection, and as well to figure out which parameters should be set or not.

4 Error Messages

4.1 Connection Setup Error Messages

Errors that occur during the setup of a connection can be caused by a lot of different conditions. But the result is always the same: a connection cannot be set up. In C the return value is `NULL` if an error occurs, in Java the `ConnectionFailedException` is thrown.

4.2 Sending Error Messages

Errors that occur while sending a command are indicated by either an exception in Java or by the return value in C. An error can be caused differently, it can be recoverable or not. Most of the time the error is simply a hangup or a sending error. The latter one can most often be ignored; the command should be repeated if it is crucial that it be executed.

4.3 Receiving Error Messages

While receiving, errors of several grades may occur. First there are the non-fatal errors, which can be caused by anything from a glitch to a badly encoded message. The second grade of errors occurs when the other side simply has hung up. This can indicate, within the context of an application, that the work is done and one may exit the process or ask the user for some other work to do. The third-grade errors while receiving are so serious that even a *close* might be impossible. The connection was broken without reason. This could mean that the communication partner has serious problems and could not finish its work properly. This type of errors is indicated by different return values of `oni_recv`. In Java there is only one exception which is thrown if any error above grade one has occurred.

5 Background

5.1 Nomenclature

The purpose of this section is to define some terms which are used within the context of this paper.

When we are talking about *applications*, we especially address those programs which are using ONI. The programmers of these applications are the *users* of ONI, which differs a bit from the usage of the term *users* by the other modules within AnimaLab. The person who is using a module of AnimaLab is called *operator* within this paper.

We use the terms *client* and *server* to differentiate between the application which waits for a connection and the one which requests this connection.

5.2 Functional Specification

The Object Network Interface should:

- avoid concurrency problems,
- offer a RPC-like semantics (at-most-once semantics),
- introduce the least complicated data-structures possible,
- implementations for C and Java,
- enable multiple connections at the same time.

5.3 Architecture

Overview

As a matter of fact ONI must be usable for Java and C programmers. To avoid the necessity to completely implement two libraries, both in Java and C, the Java interface uses the ONI library, written in C, through the Java Native Interface (JNI). The C library implements a protocol for the communication between two modules of AnimaLab; the modules or a Java class are using this library, which is itself just an interface for a module which is written in Java. Figures 2 and 3 illustrate the two different program stacks of ONI with the C interface on top of the C library in both stacks. In the C stack, this is the interface toward the application or module, and in the Java stack this is just a part within the whole Java-ONI stack.

Layers

The work of ONI is divided into three layers: the Top Layer, the Middle Layer and the Lower Layer. Figure 2 shows, for the C library, the three layers embedded between the Application Layer and the OS-Network-Stack. The Application Layer contains the modules of AnimaLab, and the OS-Network-Stack is the common TCP over IP stack. The operator is atop of the whole stack and only seeing the currently used application. Within the OSI-Model ONI is arranged between the Transport Layer and the Application Layer, without claiming to provide specific services from the Presentation and the Session Layer. In fact we are only providing a few, relatively raw services like basic data types and basic flow control functions. For information about the OSI Model the reader may wish to take a look into a computer networks textbook, for example [Tan96].

The Top Layer provides an interface for the applications, parameter checking, and delivers errors and commands to and from the applications. In Figure 2 the Top Layer is found directly below the applications. Because of the two languages for which we have to provide an interface, the Top Layer looks a little different, depending on the language in which the application is written.

The task of the Middle Layer is to marshal and unmarshal parameters. The Lower Layer of ONI composes and retrieves messages and puts or gets

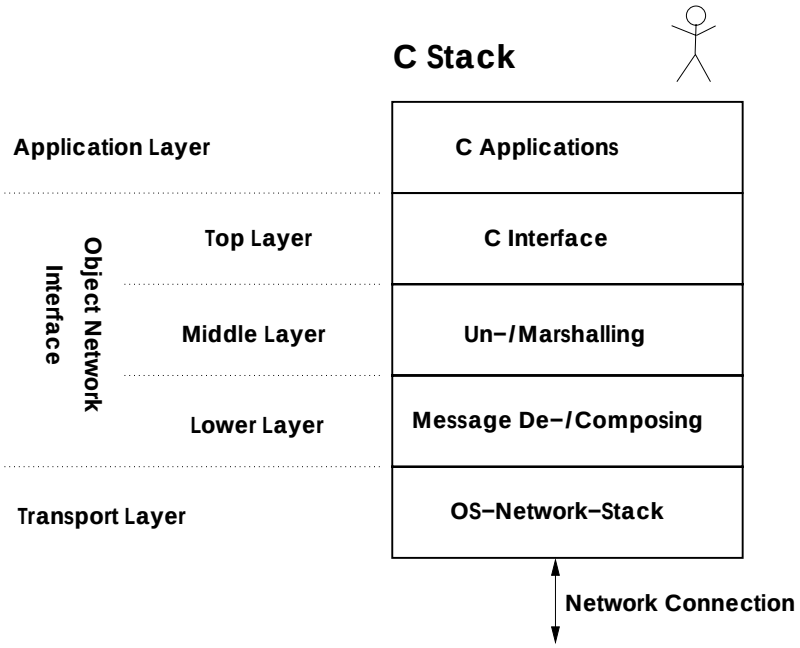

Figure 2. ONI protocol stack for C modules

them from a stream socket, which implements the network stack of the operating system. TCP provides us with a safe way to transfer data.

The whole stack should be implemented only once. This should make it easier to implement and test changes because they only have to be implemented in one language. And it is less likely that we end up with two versions which are not able to communicate with each other. Therefore the Java implementation is put into the upper layer. Figure 3 shows the whole ONI stack for Java, the Java parts are found above the C parts that are known from Figure 2 above.

Because of this, the Java implementation provides only an encapsulation of the C implementation. The task of the Java stack is to provide a Java-specific interface and implement functionality by using the C implementation. This has to be done in two steps. First it has to build its interface towards Java, and second it has to channel parameters and data to and from C functions.

C Implementation

This section describes the functionally distinct parts of ONI. These parts mostly match the different layers of ONI, but they are not that strictly divided into different modules. ONI consists of three modules: `ONIparser`, `ONIinterface` and `debug`.

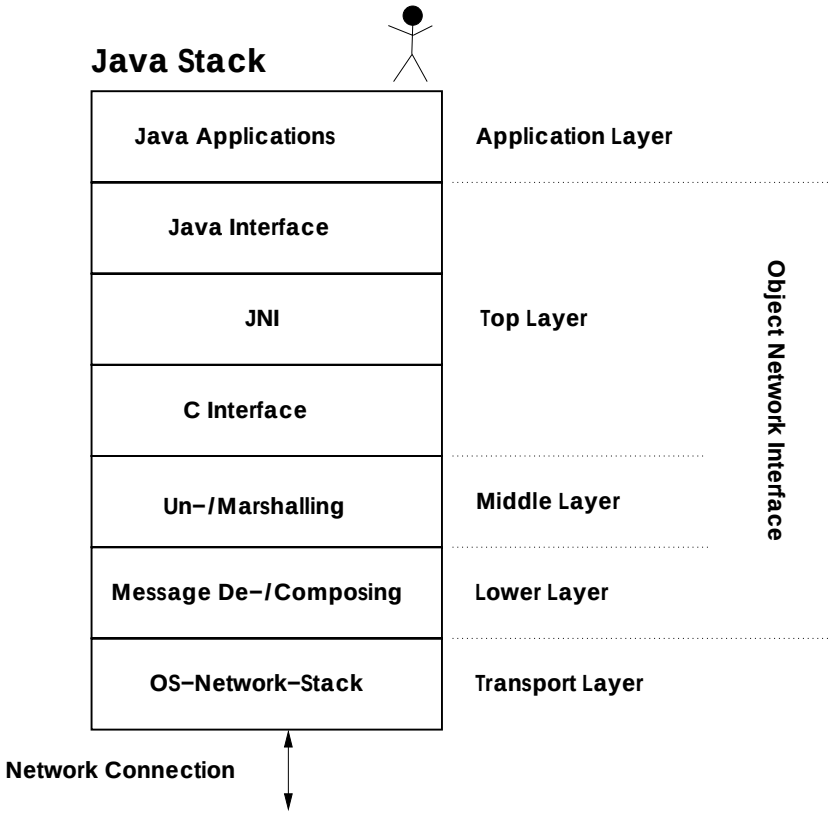

Figure 3. ONI protocol stack for Java modules

The module `ONIparser` provides some basic parsing functions as mentioned below. `debug` is a module that allows to generate scalable debug output without incrementing the volume of the final library. This is very useful, because with this module the debugging process can easily be focused on a single layer or module. The main module is `ONIinterface`, which implements the interface functions as well as the biggest part of the internal functionality.

Looking at Figure 4 it should be much easier to see which module implements which layer or part of ONI.

ONI Interface

This is the interface of ONI for C users. The interface is declared within the header file `ONIinterface.h`. It consists of function declarations for connection management and for the remote procedure calls, as well as of the definition of the data structures which are used for the communication with ONI.

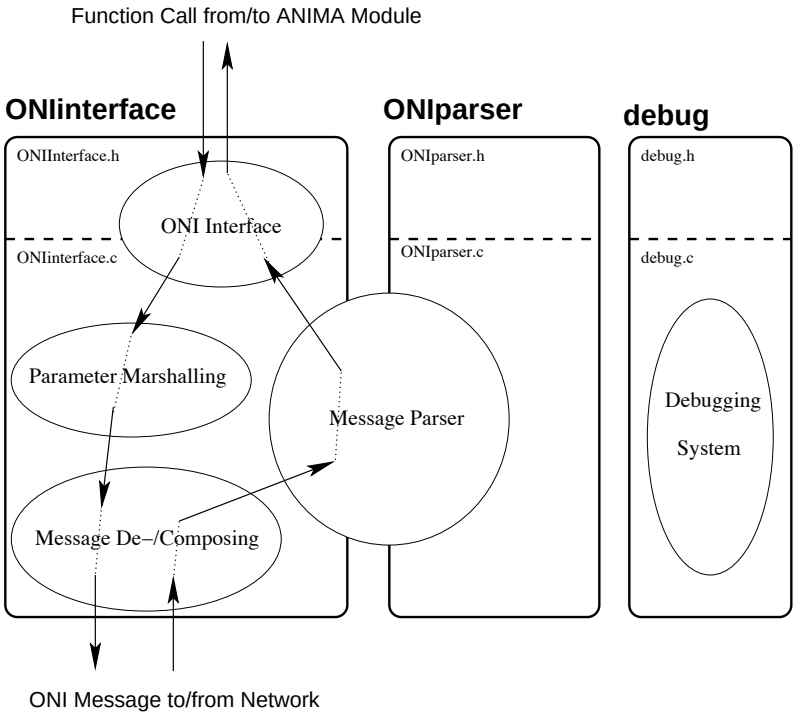

Figure 4. Parts of ONI and path of a function call

Parameter Marshalling
This is the part where the parameters of the function calls are converted into the ONI message format. Parameter marshalling is done within the main code file `ONIinterface.c`.

Message Parser
This is the reverse of the parameter marshalling. ONI messages are parsed into parameters for call-back functions. The message parser is divided into basic functions, which "know" how a single parameter is parsed, and a function called `oni_handle_message`, which controls how a distinct message has to be parsed. The basic functions are implemented within the separate module `ONIparser`, while the controlling function is found within the main code file `ONIinterface.c`.

Message De-/Composing
This means generatint or interpreting the ONI header, which consists of one length field. Therefore this part is implemented within the send and receive functions `oni_send` and `oni_recv` in the main code file.

Debugging System
This part is used for debugging only, and consists of a code and a header file named `debug.c` and `debug.h`, respectively. It is possible to control

the debug output through preprocessor macros at compile time. If the compilation is made with a set debug flag, the debugging system increases the size of the library.

It is not shown in Figure 4 how the different modules use each other, but this is swiftly told. `ONIinterface` uses all other modules, and `debug` is used by all other modules. As a result, `ONIparser` has to be used by `ONIinterface` and uses `debug`, too.

Java Implementation

As mentioned before, the ONI version in Java is not a complete implementation, but rather a Java interface to the ONI C library. This is only possible because we can use the Java Native Interface library to translate Java method invocations into C function calls and vice versa.

The path of a function call through the different parts can be seen in Figure 5.

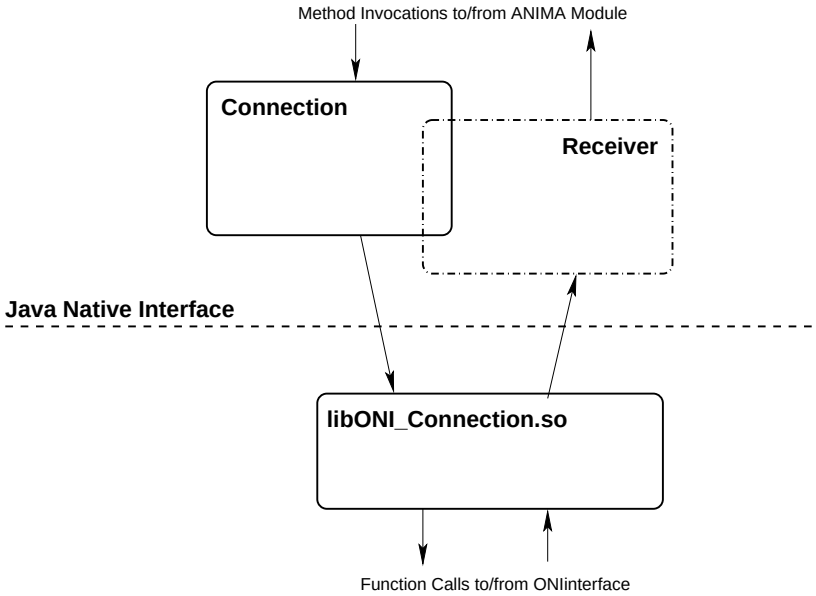

Figure 5. Parts and path of method/function calls

ONI Java Interface

This is mostly representated through the `Connection` class for outgoing function calls and connections and the `Receiver` interface for incoming

calls. An instance of `Connection` is used as a representation for a single connection. Each `Connection` has to know exactly one instance of a `Receiver` interface through which function calls will be made by ONI. Equally important for the whole interface are *data types*, which are represented either as basic data types or instances of the classes `Matrix` or `Point`.

As shown in Figure 5, a method invocation of a remote method of an instantiated Connection object is processed through JNI methods, the `libONI_Connection`, and results in a matching function call for the `ONIinterface`. In the other direction, a function call from the `ONIinterface` is translated into a method invocation for the appropriate `Receiver` instance, through the JNI, by `libONI_Connection`.

JNI Interface

This part realizes the translation between Java and C. It is implemented in the C program `ONI_Connection.c` and compiles to a shared object library, which is then used from within the `Connection` class by the Java Virtual Machine. The translation library itself uses the ONI library and the library of the JNI which provides access to Java classes and methods from within a C program.

Exceptions

This part is responsible for the handling of occurring errors. There are two major error classes: errors which occur during connection setup and errors which occur during communication (receiving and sending). This is reflected in three exceptions: `ConnectionFailedException`, `ConnectionBrokenException` and `SendFailedException`. The first is thrown if the instantiation of a `Connection`, and therefore the setup of a connection fails. The third exception is thrown if something unrecoverable goes wrong during an ONI function call. A `ConnectionBrokenException` is thrown if the connection is terminated by the other side or something happens during receiving that has a similar effect.

Message Format

The message format is influenced by the *Mbus* protocol[2], which is to be standardized by the IETF[3]. The *Mbus* defines a way to exchange messages on a flexible multicast address scheme.

A message consists of a header and a body. The header (four bytes) contains the length of the body. The body starts with the function name, followed by a space and a list of parameters encapsulated in parentheses. The *Mbus* Transport Draft [OPK01, Section 5.3] describes how the parameters are encoded. We do it similarly, with the exception that we allow integers encoded to any base up to 36, where the alphabet is used for digits above 9, just as in the usual hexadecimal notation.

[2] (http://www.mbus.org)
[3] Internet Engineering Task Force (http://www.ietf.org)

A list of commands and parameters can be found in Section 6. A little bit more about sending is said below. A detailed specification for each command would go beyond the goal of this paper; please consult the above mentioned literature and the source code of ONI.

Connection Setup

ONI has two methods to open a connection: one waits for an incoming connection on a port, and the other actively seeks to build up a connection to a named host and port.[4] In Java these two functions are mapped to two different constructors for `Connection`. One needs an address with port number, and the signature of the other constructor requires only the port number. In both languages one has to pass a structure which holds references of a call-back function. In Java this is the interface `Receiver` which has to be implemented by the user of ONI. In C this is a structure which holds function pointers.

The connection is then realized as a Berkeley Socket which either waits for a connection or tries to connect from or to another socket. ONI has its own connection descriptor whose memory address is used as host-unique identifier.

Sending

To send a command with ONI, the user only has to call the member function of the right connection instance or the C function with the identifier of the right connection. ONI defines a set of commands which, when called, will be executed on the host at the other end of a connection. Every function will be sent as a different command message and has its very own set of parameters.

The call of a remote function is translated into a call of an internal send function. The send function only sends one string of symbols, which has to be generated by the parameter marshalling. This means that, depending on which function has been called, each parameter is interpreted and written as part of the resulting string. The message format defines how each parameter type is translated and how the different parameters of a function are ordered within the string[5]. The name of the function is the first parameter within this string.

The send function then takes this string and encapsulates it into a message. This simply means that the four byte header is generated and concatenated with the message string. To reduce operating expenses these four bytes are allocated previously within the parameter marshalling. The last step of a sending process is to send the data over the socket of the connection until either an error occurs, which would be really bad, or the whole message is translated. Normally the socket buffer should be large enough to keep a single message, but if a lot of messages are sent in a short period of time, this

[4] Hopefully this is the one on which someone accepts connections.
[5] To simplify matters the order within the signature is kept.

buffer might overflow, which leads to a blocking of the send function. Sending is always completed unless an error occurs.

When using the Java version of ONI, the only difference to the above description is that function calls have to be forwarded and parameters have to be translated from Java to C.

Receiving

Receiving is implemented through call-back functions. For each sending function, there is a matching call-back function declared within the structure `callbacks`. Similar to C there is a method within the interface `Receiver`, for each call-back function, which has to be implemented.[6]

When a message is received by the receive function `oni_recv`, the message header is read, and afterwards a number of bytes matching the number in the length field. This is then processed by the function `oni_handle_message` which, if an appropriate call-back function is defined within the `callbacks` structure of the connection, parses the string, allocates memory and translates the string into a set of variables. Then the call-back function is called. After the return of the call-back function, whether successful or not is not checked, the memory of the parameters is freed and the receive function returns.

If the Java implementation is used, every function of the `callbacks` structure is defined by the library `ONI_Connection`. If any of these functions is called by `oni_handle_message`, the function parameters are translated from C to Java variables, or classes and the right method of the `Receiver` interface is called. When the `Receiver` method has finished, the translation function simply returns like a normal C call-back function.

5.4 Future Goals

Although the state of ONI is quite pleasing, there are still some things left to wish. Beside some new commands, which can be implemented, there are some more conceptional challenges.

Maybe the most interesting one is the creation of a module discovery protocol. Currently an operator has to know what module is waiting on which host and port in order to set up a connection. This is not satisfying, as the operators are only interested in the modules and their function. Therefore a protocol which finds AnimaLab modules within the network could make the whole system less abstract for both users and operators.

Most of the annotations from users referred to the complicated way in which lists are dealt with. In a next version these pointer lists should be changed to some more comfortable growable arrays or other easier-to-use concepts.

[6] This includes an empty implementation.

6 Application Programming Interface (API)

In Section 5.3 the construction of ONI is described. In this section the functions and classes that are part of ONI are explained.

6.1 C API

Below the functionality of all provided functions is described, but as ONI is only passing the function call to the receiver, the receiver is actually responsible for the real functionality. ONI is only a transport medium for AnimaLab applications.

First we describe the functions used for connection management, receiving, and then the functions which are called at the remote host.

Connection Management

The functions in this section handle the establishing and closing of connections.

`connection` **oni_makeconnect**`(const char* addr, const int port, unsigned char flags)`

`addr`	An host name or IP address in dot notation of your communication partner.
`port`	The port on which the partner module is waiting for a call.
`flags`	This determines whether the connection is for example a blocking or nonblocking one.
`cbf`	The structure which holds all the funny call-back functions one likes to call.

Connect to the port on the specified host. The function returns a new connection structure.

`connection` **get_connect**`(const int port, unsigned char flags)`

`port`	The port on which is waited for an incoming call.
`flags`	This determines whether the connection is for example a blocking or nonblocking one.
`cbf`	The structure which holds all the funny call-back functions one likes to call.

Open socket for new connection and wait for incoming connections. Returns a new connection structure if successful.

`int` **oni_closeconnect**`(connection con)`

`con` Connection identifier (0 means close all connections).

Close a single or all connections.

Receiving

There is only one function which has to be called in order to receive commands. This function is called `oni_recv`. The process of receiving is explained in Section 5.3.

```
int oni_recv(connection con, void* data,
             int max_recv, int timeout)
```

`con` The connection on which one wishes to receive. If is this `NULL` then all connections are searched for incoming data.

`data` This pointer is handed over to your call-back functions.

`max_recv` Receive at most this many commands before returning, if they are already in the socket buffer. Any value $<=0$ is illegal and will be treated as 1.

`timeout` Sets the maximum waiting time, in milliseconds, for new commands. A negative value means waiting until `max_recv` messages are received.

This function receives and handles incoming commands if any are present. Any incoming command message is parsed and leads to the call of a call-back function.

Remote Functions

These functions are sent to the remote host, to which a connection has been established, to trigger a matching call-back function on this host.

`int` **oni_NTset**`(connection con, const char** nts)`

`con` Connection on which the command should be send.

`nts` A list of nonterminal strings; each one should be unique.

This command sets the set of nonterminals which are used by this object generator. It should be used before any other commands immediately after opening a connection.

```
int oni_combPrim(connection con, const char* name,
                 const char** prims,
                 transformation*** trafos)
```

`con`	Connection on which the command should be sent.
`name`	This parameter names the *new* entity; it has to be a new name.
`prims`	List of already existing (!) primitives.
`trafos`	A list of transformations for each primitive in `prims`

Combine some primitives to create a new one. Every primitive that is used can be modified by a transformation. This command should make it possible to create all kinds of fancy primitives.

```
int oni_addPrim(connection con, const char* name,
                point*** corners, const char** texture)
```

`con`	Connection on which the command should be sent.
`name`	A new designator for a new primitive.
`corners`	A list of polygons which define the primitive.
`texture`	List of textures, one for each polygon. If this list has only one element, then this element is used as texture for all polygons.

Create a new primitive out of polygons with textures.

```
int oni_createTex(connection con, char* name, int height,
                  int width, unsigned char* bytes, int type)
```

`con`	Connection on which the command should be sent.
`name`	A new name which from now on refers to this texture.
`height`	Height of the texture in pixel.
`width`	Width of the texture in pixel.
`bytes`	The bytes which define the texture (portion of 4 bytes defining a pixel color (rgbα) or (rgb)).
`type`	Either `TEX_TYPE_RGBA` or `TEX_TYPE_RGB`.

Create a new texture. In reality the data is just sent to the viewer and given a name so that it is referable.

```
int oni_loadTex(connection con, const char* name,
                const char* path)
```

`con`	Connection on which the command should be sent.
`name`	A module-unique name with which the texture will be referenced from now on.
`path`	The path where the texture is found. Must be of the form `[<hostname>:]<filepath>`.

Binds a texture file to a texture name and makes the viewer load the texture.

```
int oni_create(connection con, const char* name,
               const char* primitive,
               transformation** trafos,
               const char** nterms,
               transformation*** nt_trafos)
```

`con`	Connection on which the command should be sent.
`name`	Name of the entity to create.
`primitive`	Name of the primitive which should be used.
`trafos`	List of transformations which should be used on the entity.
`nterms`	List of nonterminals.
`nt_trafos`	One transformation list for each nonterminal.

Display an entity. This is the command that puts something on the screen. It creates an entity as picture of a primitive, transformed by a list of transformations and with nonterminals which will be resolved by the Arranger.

```
int oni_modify(connection con, const char* name,
               transformation** trafos)
```

`con`	Connection on which the command should be sent.
`name`	Name of the entity to be transformed.
`trafos`	List of transformations to be used for the entity.

Transform an entity. If the entity is too small, too large or has the wrong position, it can be modified by applying a list of transformations.

```
int oni_set(connection con, const char* name,
            const char* primitive, transformation** trafos,
            const char** nterms, transformation*** nt_trafos)
```

`con`	Connection on which the command should be sent.
`name`	Name of the entity to change.
`primitive`	Name of the primitive which should be used from now on.
`trafos`	List of transformations which should now be used on the entity.
`nterms`	New list of nonterminals for the entity.
`nt_trafos`	One transformation list for each nonterminal.

Redefine an entity. If the entity has the wrong texture, a shape from the wrong primitive, or any of the other things that were defined at the creation, putting the new definitions in here makes the according changes. This command of course has the same parameters as create. Any parameter except *name* and *con* can be a `NULL`-pointer to leave the corresponding property unchanged.

```
int oni_destroy(connection con, const char* name, int dtype)
```

`con`	Connection on which the command should be sent.
`name`	Name that identifies the entity.
`dtype`	Defines the type of deletion (`SIMPLE_DESTROY` is always implemented).

Delete an entity. Simply deletes the specified entity using the effect chosen to dissolve it. The available effects depend on the Viewer.(See Chapter 12)

```
int oni_createComp(connection con, const char* name,
                   const char** entList)
```

`con`	Connection on which the command should be sent.
`name`	The name for the compound.
`entList`	A list of entities, ending with `NULL`.

Create a compound of entities. Several entities can be grouped together and referred to as one entity, which facilitates modifying all original entities with the same transformations.

`int` **oni_start**`(connection con)`

`con` Connection on which the command should be sent.

Restart a stopped or paused stream.

`int` **oni_pause**`(connection con)`

`con` Connection on which the command should be sent.

Pause a stream (if the destination reacts this way; otherwise it has no effect).

`int` **oni_stop**`(connection con)`

`con` Connection on which the command should be sent.

Stop a stream. This command, like pause, signals the receiver to stop transmission. This lies solely in the responsibility of the receiver, with the one exception that ONI clears its buffers on both sides when it sends or receives this command, thereby deleting all the data that is still within the space of the interface.

`int` **oni_faster**`(connection con)`

`con` Connection on which the command should be sent.

Tells the receiver to speed up. The semantics of this command is totally up to the receiver.

`int` **oni_slower**`(connection con)`

`con` Connection on which the command should be sent.

Tells the receiver to slow down. The semantics of this command is totally up to the receiver.

`int` **oni_tick**`(connection con, unsigned int type)`

`con` Connection on which the command should be sent.
`type` The tick type. (Which must somehow be negotiated between the modules)

Tick you are caught. This command is used to send any kind of stimulation or indication that can be thought of, e.g. time ticks (that indicate the

start of a new time interval), a request for the next object, or whatever one may need. Please remember to tell your receiver what you mean by tick.

Miscellaneous

The two functions in this section have been created for comfort. Of course there are many other functions which could be useful.

`void` **oni_free_trafo_list**`()`

`trafos` Pointer to a list of transformations.

Frees a list of transformation lists.

`void` **oni_free_trafo_list_list**`()`

`trafos` Pointer to a list of lists of transformations.

Frees a list of lists of transformations.

6.2 Java API

The methods that are used in the Java implementation do not differ very much from the functions of the C implementation. The biggest difference is that there is now an object for each connection within which the methods are implemented, and an interface which resembles the old structure `callbacks`. Therefore the explanations might be shorter than the explanations in the C API.

Class
`public` **Connection**

This class implements a connection between two AnimaLab modules, using the ONI C library `libONIinterface.so` by means of the Java Native Interface. A connection is established by using a constructor of this class and having somebody using the other constructor waiting at the destination address.

Fields

`public static final int` **SIMPLE_DESTROY**

Constructors

`public` **Connection**`(String addr, int port, Receiver rec)`
throws *ConnectionFailedException*

`String addr`	Host name/IP-address of the destination server.
`int port`	Port address of the destination server.
`Receiver rec`	Reference to an instantiated `Receiver`-object.

Opens a new connection.

`public void` **Connection**`(int port, Receiver rec)`
throws *ConnectionFailedException*

`int port`	Port on which this server should wait for incoming connections.
`Receiver rec`	Reference to an instantiated `Receiver`-object.

Opens a new connection.

Destructors

`public void` **close**`()`
throws *Throwable*

Close this connection.

Methods

`public static public void` **broad_receive**`(int maxrecv, int timeout)`

`int maxrecv`	Maximum number of commands that are to be received at once.
`int timeout`	Number of milliseconds to wait for incoming commands.

Receive on all connections. This is not implemented yet.

`public void` **receive**`(int maxrecv, int timeout)`
throws *ConnectionBrokenException*

`int maxrecv`	Maximum number of commands that are to be received at once.
`int timeout`	Number of milliseconds to wait for incoming commands.

Receive on this connection.

`public void` **NTset**`(String[] nts)`
throws *SendFailedException*

`String[] nts`	Array of nonterminal strings; each one must be unique.

Define the set of nonterminals that are to be used by your module.

`public void` **combPrim**`(String name, String[] prims, Matrix[][] trafos)`
throws *SendFailedException*

`String name`	Names the new entity; this has to be a new name.
`String[] prims`	Array of existing (!) primitives.
`Matrix[][] trafos`	An array of transformations for each primitive in `prims`.

Concatenate (combine) primitives to get a new primitive. Each primitive can be transformed by a list of transformations.

`public void` **addPrim**`(String name, Point[][] corners, String[] texture)`

`String name`	Yes, it is a new name.
`Point[][] corners`	An array of polygons which define the primitive.
`String[] texture`	Array of textures, one for each polygon. If this list has only one element, then this element is used as texture for all polygons.

Create a new primitive out of polygons with textures. Be creative!

`public void` **createTex**`(String name, int height, int width, Byte[] bytes, int type)`

`String name`	The name for the texture.
`int height`	Height of the texture in pixel.
`int width`	Width of the texture in pixel.
`Byte[] bytes`	The bytes which define the texture (portion of 4 bytes defining a pixel color (rgbα) or (rgb)).
`int type`	Defines whether one pixel is defined by three or four bytes.

Create a new texture. In reality the data is just send to the viewer and given a name so that it is referable.

`public void` **loadTex**`(String name, String path)`

`String name`	Module-unique name with which the texture will be referenced from now on.
`String path`	The path where the texture is found. Must be of the form `[<hostname>:]<filepath>`.

Binds a texture file to a texture name and makes the viewer load the texture.

```
public void create(String name, String primitive,
                   Matrix[] trafos, String[] nterms,
                   Matrix[][] nt_trafos)
```

`String name`	Name of the entity to create.
`String primitive`	Name of the primitive which should be used.
`Matrix[] trafos`	List of transformations which should be used on the entity.
`String[] nterms`	List of Nonterminals.
`Matrix[] nt_trafos`	One transformation list for each nonterminal.

Display an entity. This is the command that puts something on the screen. It creates an entity as a picture of a primitive, transformed by a list of transformations and with nonterminals which will be resolved by the Arranger.

```
public void modify(String name, Matrix[] trafos)
```

`String name`	Name of the entity to be transformed.
`Matrix[] trafos`	List of transformations to be applied to the entity.

Transform an entity. If the entity is too small, too large or has the wrong position, it can be modified by applying a list of transformations.

```
public void set(String name, String primitive,
                Matrix[] trafos, String[] nterms,
                Matrix[][] nt_trafos)
```

`String name`	Name of the entity to change.
`String primitive`	Name of the primitive which should be used from now on.
`Matrix[] trafos`	List of transformations which should now be used on the entity.
`String[] nterms`	New list of nonterminals for the entity.
`Matrix[] nt_trafos`	One transformation list for each nonterminal.

Redefine an entity. If the entity has the wrong texture, a shape from the wrong primitive, or any of the other things that were defined at the creation, putting the new definitions in here makes the according changes. This command of course has the same parameters as create. Any parameter except *name* and *con* can be a `NULL`-pointer to leave the corresponding property unchanged.

`public void` **destroy**`(String name, int dtype)`

`String name`	Name identifies the entity.
`int dtype`	dtype defines the type of deletion.

Delete an entity. Simply deletes the specified entity using the effect chosen to dissolve it. The available effects depend on the viewer.

`public void` **createComp**`(String name, String[] entList)`

`String name`	The name for the compound.
`String[] entList`	A list of entities.

Create a compound of entities. Several entities can be grouped together and referred to as one entity, which facilitates modifying all original entities with the same transformations.

`public void` **start**`()`

Restart a stopped or paused stream.

`public void` **pause**`()`

Pause a stream (if the destination reacts this way; otherwise it has no effect).

`public void` **stop**`()`

Stop a stream. This command, like pause, signals the receiver to stop transmission. This lies solely in the responsibility of the receiver, with the one exception that ONI clears its buffers on both sides when it sends or receives this command, thereby deleting all the data that is still within the space of the interface.

`public void` **faster**`()`

Tells the receiver to speed up. The semantics of this command is totally up to the receiver.

`public void` **slower**`()`

Tells the receiver to slow down. The semantics of this command is totally up to the receiver.

`public void` **tick**`(int type)`

`int type`	The tick type (see definitions for globally used ticks).

Tic. This command is used to send any kind of stimulation or indication that can be thought of, e.g. time ticks (that indicate the start of a new time interval), a request for the next object or whatever is needed. Please remember to tell your receiver what you mean by tick.

Class

`public` **Matrix** `implements Serializable`

Implements a 4×4 transformation matrix.

Constructors

`public void` **Matrix**`()`

Creates a new `Matrix` instance. This is an identity matrix.

`public void` **Matrix**`()`

`Matrix4f matrix` A matrix.

Creates a new `Matrix` instance, just like the matrix specified with the parameters.

`public void` **Matrix**`(float[] fa)`

`float[] fa` A float array describing a transformation matrix. The parameters are interpreted as first columns from top to bottom, then lines from left to right.

Creates a new `Matrix` instance, just like the matrix specified with the parameters.

`public void` **Matrix**`(double[] fa)`

`double[] fa` A double array describing a transformation matrix. First columns from top to bottom, then lines from left to right.

Creates a new `Matrix` instance, just like the matrix specified with the parameters.

```
public void Matrix(float a00, float a10, float a20,
                   float a30, float a01, float a11,
                   float a21, float a31, float a02,
                   float a12, float a22, float a32,
                   float a03, float a13, float a23,
                   float a33)
```

`float aXX` Each parameter defines the value of one cell of the matrix.

Creates a new `Matrix` instance, just like the matrix specified with the parameters.

Methods

```
public void setValues(double[] a)
```

`double[] a` An array which defines the new value of the `Matrix`.

Sets the values of a `Matrix`.

```
public double _XX()
```

These methods return the value of a single element. This description explains a set of functions where the "XX" is a placeholder for the index of the corresponding array element.

```
public void mul(Matrix matrix)
```

`Matrix matrix` A `Matrix` value.

Multiplication of the current instance of `Matrix` with the matrix provided as parameter. The current instance is redefined as `this ∘ matrix`.

`public void` **rotation**`(double angle, double x,`
`double y, double z)`

`double angle`	The rotation angle.
`double x`	The x-value of the rotation vector.
`double y`	The y-value of the rotation vector.
`double z`	The z-value of the rotation vector.

Make this matrix a rotation matrix about the given axis (`x,y,z`) with the given angle (`angle`).

`public void` **rotationX**`(double angle)`

`double angle`	A `double` value.

Make this matrix a rotation matrix about the x-axis with the given angle.

`public void` **rotationY**`(double angle)`

`double angle`	A `double` value.

Make this matrix a rotation-matrix around the y-axis with the given angle.

`public void` **rotationZ**`(double angle)`

`double angle`	A `double` value.

Make this matrix a rotation matrix about the z-axis with the given angle.

`public void` **scaling**`(double x)`

`double x`	A `double` value; the factor by which the matrix is scaled.

Make this matrix a scaling matrix.

`public void` **scaling**`(double x, double y, double z)`

`double x`	Scaling factor in the direction of the x-axis.
`double y`	Scaling factor in direction of the y-axis.
`double z`	Scaling factor in direction of the z-axis.

Scale this `Matrix`.

`public void` **translation**`(double x, double y, double z)`

`double x`	Translation factor at the x-axis.
`double y`	Translation factor at the y-axis.
`double y`	Translation factor at the y-axis.

Make this matrix a translation matrix.

`public void` **rotate**`(double angle, double x, double y, double z)`

`double angle`	The rotation angle.
`double x`	x-value of the rotation vector.
`double y`	y-value of the rotation vector.
`double y`	z-value of the rotation vector.

Add to this matrix a rotation of `angle` degrees about (`x`,`y`,`z`).

`public void` **rotateX**`(double angle)`

`double angle` Rotation angle.

Add to this matrix a rotation of `angle` degrees about the x-axis.

`public void` **rotateY**`(double angle)`

`double angle` Rotation angle.

Add to this matrix a rotation of `angle` degrees about the y-axis.

`public void` **rotateZ**`(double angle)`

`double angle` Rotation angle.

Add to this matrix a rotation of `angle` degrees about the z-axis.

`public void` **scale**`(double x)`

`double x` Scaling factor.

Add to this matrix a scaling matrix.

`public void` **scale**`(double x, double y, double z)`

`double x` Scaling factor on axis x.
`double y` Scaling factor on axis y.
`double z` Scaling factor on axis z.

Add to this matrix a scaling matrix.

`public void` **translate**`(double x, double y, double z)`

`double x`	Translation on x-axis.
`double y`	Translation on y-axis.
`double z`	Translation on z-axis.

Add to this matrix a translation matrix.

14 Chaincode Object Generator

Lutz Albrecht

1 Introduction

Chaincode picture languages have been investigated already for some time. In the literature, the emphasis lies on two-dimensional objects (see, e.g., Dassow and Hinz [DH89] and Maurer, Rozenberg and Welzl [MRW82]). In order to study three-dimensional objects, a visualization tool is helpful. Such a tool is the Chaincode Object Generator within AnimaLab. It allows to implement a chaincode grammar, which generates chaincode strings, and translates such a string into object data to be visualized by the Viewer.

2 Basic Notions

In this section, the basic notions and notations of three-dimensional chaincode picture languages are recalled.

1. The Cartesian product $\mathbb{Z}^3$, the set of points in the 3D space with integer coordinates is called *universe set of points*.
2. Each point (x, y, z) has six direct neighbours,
   one neighbour to the *east* $e(x, y, z) = (x + 1, y, z)$,
   one to the *south* $s(x, y, z) = (x, y - 1, z)$,
   one to the *west* $w(x, y, z) = (x - 1, y, z)$,
   one to the *north* $n(x, y, z) = (x, y + 1, z)$,
   one *up* $u(x, y, z) = (x, y, z + 1)$,
   and one *down* $d(x, y, z) = (x, y, z - 1)$.
3. The set of straight lines between all points and their direct neighbours forms the *universe set of unit lines*, which is given by

$$UL =$$

$$\{\{(x, y, z), (x', y', z')\} \mid x, y, z, x', y', z' \in \mathbb{Z},\ |x' - x| + |y' - y| + |z' - z| = 1\}$$

   where a unit line is represented by the pairs of its endpoints.
4. A *picture* p is a finite subset of UL.
5. The *set of points of a picture* p is defined as $point(p) = \bigcup_{l \in p} l$.

6. A picture p is called *connected* if a sequence of points (x_i, y_i, z_i) for $i = 1, .., n \ \ (n \geq 0)$ exists with $(x, y, z) = (x_0, y_0, z_0), (x', y', z') = (x_n, y_n, z_n)$ and $\{(x_i, y_i, z_i), (x_{i+1}, y_{i+1}, z_{i+1})\} \in p$ for $i = 0, .., n-1$ for every two points $(x, y, z), (x', y', z') \in point(p)$.
7. A *drawn picture* is a pair $d = (p, end)$ in which p is a connected picture and *end* is a point with the following property: $end = (0, 0, 0)$ if p=$\emptyset$, and $end \in point(p)$ otherwise. The component is called *underlying picture* with *end* as the *endpoint*.
8. A string $u \in \{e, s, w, n, u, d\}^*$is called *picture description*.
9. The *drawn picture* $drawing(u) = (p(u), end(u))$ of a picture description u is defined recursively:
   - $(p(\lambda), end(\lambda)) = (\emptyset, (0, 0, 0))$,
   - $(p(vx), end(vx)) = (p(v) \cup \{\{end(v), x(end(v))\}\}, x(end(v)))$ for $v \in \{e, s, w, n, u, d\}^*$ and $x \in \{e, s, w, n, u, d\}$.
10. Let $L \subseteq \{e, s, w, n, u, d\}^*$. The *chaincode picture language* of L is given by $picture(L)=\{p(u) | u \in L\}$.
11. To generate languages of picture descriptions, L systems are used in the object generator. Clearly, Chomsky grammars or other language-generating devices would also do.
12. An OL system $G = (V, \omega, P)$ consists of an alphabet V, an axiom $\omega \in V^*$ and a finite set of productions $P \subseteq V \times V^*$. The term $(A, u) \in P$ is also notated by $A \longrightarrow u$.
13. Let $a_1...a_n \in V^*$ with $a_i \in V$ be a word over V. Let $a_i \longrightarrow u_i \in P$ for $i = 1, .., n$. Then $a_1...a_n$ derives $u_1...u_n$ directly through P. This *direct derivation* is denoted by $a_1...a_n \longrightarrow u_1...u_n$.
14. A *derivation* $v_1 \longrightarrow ... \longrightarrow v_k$ consists of direct derivations $v_j \longrightarrow v_{j+1}$ for $j = 1, ..., k-1$. This can be abbreviated by $v_1 \longrightarrow^* v_k$.
15. The *language* generated by G contains all strings that can be derived from the axiom: $L(G) = \{v \in V^* \mid \omega \longrightarrow^* v\}$
16. A *deterministic OL system (DOL system)* is an OL system $G = (V, \omega, P)$ subject to the condition that, for each $a \in V$ there is exactly one $u \in V^*$ with $a \longrightarrow u \in P$.
17. An *EOL system* $G = (V, T, \omega, P)$ consists of an OL system (V, ω, P) and a terminal alphabet $T \subseteq V$. For $p \in P$ let $t \longrightarrow u$ be a valid derivation with $t \in T$.
18. Its generated language consists of all terminal strings that can be derived from the axiom, i.e.$L(G) = \{v \in T^* \mid \omega \longrightarrow^* v\}$
19. An *ETOL system* is defined as $G = (V, T, \omega, \mathbf{P})$ where V is the alphabet, T a terminal subalphabet, ω the axiom, and $\mathbf{P}$ is a finite set of tables, each of which being a finite set of productions.
20. A *derivation* of such system consists of direct derivations where one of the tables is applied in each step.
21. The *generated language* $L(G)$ contains all terminal words derivable from the axiom.

3 Examples

1. The Koch curve can be specified by a DOL system over the alphabet $\{e, n, w, s\}$ with the axiom $nesw$ and the rules

$$\begin{aligned} n &\longrightarrow nenwwnen \\ e &\longrightarrow esennese \\ s &\longrightarrow swseesws \\ w &\longrightarrow wnwsswnw. \end{aligned}$$

The drawn picture of the picture description which is derived from the axiom in four steps, is depicted in Figure 1. For better visualization the terminals were doubled before generating the picture.

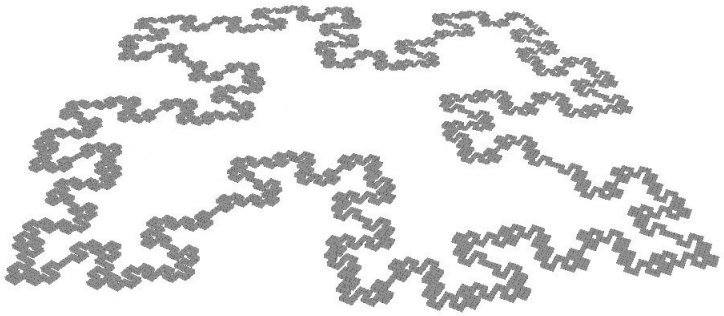

Figure 1. 2D Koch curve after four derivation steps

2. The 3D Hilbert curve can be specified by an ETOL system over the alphabet $\{e, n, w, s, u, d, A, B, C, D, E, F, G, H, I, J, K, L, M\}$ the first six letters of which are terminal. The axiom is A, and there are two tables.

Table 1

A	$\longrightarrow$	$FeBnBwMuMeIsIwH$
B	$\longrightarrow$	$AuFeFdJnJuKwKdL$
C	$\longrightarrow$	$IsLdLnFeFsMuMnJ$
D	$\longrightarrow$	$MuKwKdIsIuFeFdG$
E	$\longrightarrow$	$HwJnJeLdLwDsDeF$
F	$\longrightarrow$	$BnAuAsCeCnEdEsD$
G	$\longrightarrow$	$HwJnJeLdLwDsDeF$
H	$\longrightarrow$	$JnEdEsKwKnAuAsI$
I	$\longrightarrow$	$LdCeCuDsDdHwHuA$
J	$\longrightarrow$	$EdHwHuBnBdCeCuM$
K	$\longrightarrow$	$DsMuMnHwHsLdLnB$
L	$\longrightarrow$	$CoIsIwGdGeBnBwK$
M	$\longrightarrow$	$KwDsDeAuAwJnJeC$

Table 2

A	$\longrightarrow$	λ
B	$\longrightarrow$	λ
C	$\longrightarrow$	λ
D	$\longrightarrow$	λ
E	$\longrightarrow$	λ
F	$\longrightarrow$	λ
G	$\longrightarrow$	λ
H	$\longrightarrow$	λ
I	$\longrightarrow$	λ
J	$\longrightarrow$	λ
K	$\longrightarrow$	λ
L	$\longrightarrow$	λ
M	$\longrightarrow$	λ

In both tables, rules for terminals are omitted because each of them is replaced by itself. The drawn picture of the picture description which is derived from the axiom by two steps through Table 1 and one step through Table 2 is depicted in Figure 2.

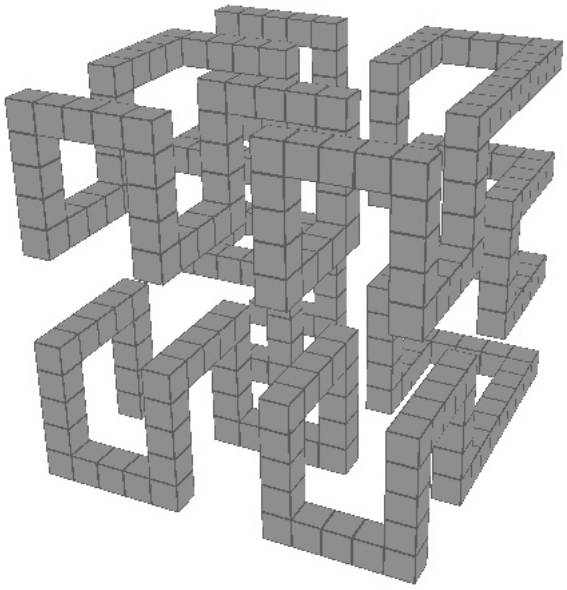

Figure 2. A Hilbert curve after three derivation steps

4 Syntax

The Chaincode Object Generator is written in Java. It consists of two components (see Figure 3). The first component is for generating chaincodes, i.e., strings over the alphabet $\{e, s, w, n, u, d\}$; this is typically done by providing a grammar and the number of derivation steps. The second component is for generating objects and sending them to the Viewer. The construction of the components is explained with the Koch curve as running example.

4.1 Component 1: Grammar and Derivation

The user provides a grammar file (e.g. `koch2d.java`) and a derivation file (e.g. `koch2d_to_cc.java`). Consider, for example the DOL system of the Koch curve in Section 3 as input grammar. This grammar is implemented in the grammar file `koch2d.java` as follows:

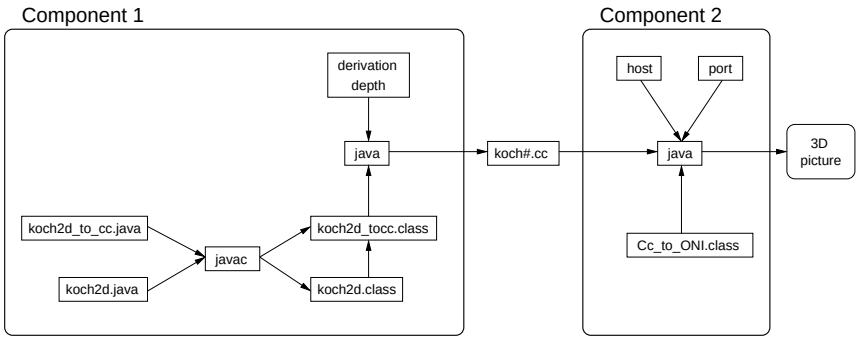

Figure 3. The chaincode file koch.cc is generated by Component 1 and is used as input for Component 2

```
...
//axiom
public String lhs_axiom=("nesw");
//Table1
public String lhs_rule11=("n");
public String lhs_rule12=("e");
public String lhs_rule13=("s");
public String lhs_rule14=("w");
...
if(lhs.equals(lhs_rule11)) {
  String rhs="nenwwnen";
  return rhs;
}
if(lhs.equals(lhs_rule12)) {
  String rhs="esennese";
  return rhs;
}
if(lhs.equals(lhs_rule13)) {
  String rhs="swseesws";
  return rhs;
}
if(lhs.equals(lhs_rule14)) {
  String rhs="wnwsswnw";
  return rhs;
}
```

The derivation process is encoded in the corresponding derivation file `koch2d.cc`, which looks as follows:

```
...
public class koch2d_to_cc {
   public static void main(String[] args) {
```

```
   //variables
   StringBuffer aktuell=new StringBuffer();
   String axiom="A";
   koch2d koch2d_instance=new koch2d();
   String result="";
   String inFile=null;
   String input=null;
   String sub;
   //read parameters
      try {
          depth = Integer.parseInt(args[0]); inFile = args[1];
          }
      catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("java koch2d_to_cc
        derivation depth sourcefile");
          }
      System.out.println("generating "
          +inFile+".....................");
   //start with derivation
   //use Table1
   for(int i=0;i<depth;i++) {
     for(int ii=0;ii<axiom.length();ii++) {
        result=result.concat(koch2d_
          instance.derive(String.valueOf(axiom.charAt(ii))));
     }
   axiom=result; result="";
   }
   //use Table2
   for(int iii=0;iii<axiom.length();iii++) {
     result=result.concat(koch2d_
       instance.end(String.valueOf(axiom.charAt(iii))));
   }
   axiom=result;
...
```

Calling the compiler with the command

```
javac DerivationFile GrammarFile
```

generates the corresponding class files, here `koch2d.class` (grammar part) and `koch2d_to_cc.class` (derivation part). Now a derivation is started with the following command line:

```
java ChaincodeGenerator DerivationDepth OutputFile
```

In this case the chaincode generator is named `koch2d_to_cc` and it is using the grammar `koch2d.java`. The corresponding files are called `koch2d_\ to_cc.java` and `koch2d.java`. For example type

```
java koch2d_to_cc 5 chaincodefiles/koch5.cc
```

to generate a file named `koch5.cc` in the directory chaincodefiles: this file contains the chaincode of a Koch curve at its fifth derivation. It is recommended not to use derivation depths beyond ten because the transfer of the object data would take more time than is helpful in capturing the recursive structure.

4.2 Generalization to Arbitrary Grammars

In order to implement new grammars, some modifications are necessary. The axiom and the rules of a new grammar are best developed on a piece of paper first. This grammar must be implemented in a grammar file analogous to `koch2d.java`. Moreover, in the derivation file `koch2d_to_cc.java` the text printed in bold must be adjusted correspondingly to the name of the new example. After compiling the two java files, two class files should be created and ready to use as decribed above. Feel free to write a chaincode generator on your own. It is not very difficult and you can use your favourite programming language.

4.3 Component 2: Generation and Transfer of Object Data

The java program that generates 3D objects from chaincode and transfers it to a running viewer instance is the matrix generator `CctoONI`. Use the following command line to start the transfer. Please keep in mind that *Cc_ | to_ ONI* requires a running viewer instance!

```
java Cc_to_ONI host port chaincode file
```

host is the name of the host where the viewer awaits the data at a specific portnumber *port*, and the chaincode file can in particular be the one generated by Component 1. The viewer visualizes the structure that is described by the chaincode file as follows. The primitive object is a cube, and there is one transformation for each prefix of the input chaincode, interpreted in the underlying coordinate system shown in Figure 4. Consider e.g. the string $w = nnneesss$. The first four letters are $nnnn$; the correspondingly transformed four cubes are shown in Figure 5a. In Figure 5b the interpretation of the string $nnnee$ is illustrated, and in Figure 5c the whole string w. In these pictures, the cubes have different shades of grey for better visualisation purposes only.

5 Error Messages

5.1 Chaincode Generators

When starting a derivation, the chaincode generator may return one of two error messages.

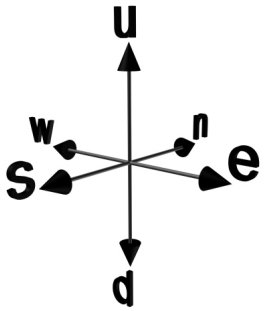

Figure 4. The coordinate system

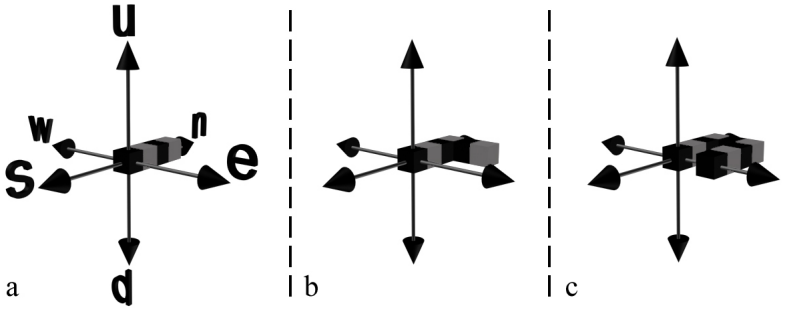

Figure 5. The initial cube and three transformations in n-direction, two in e-direction and three in s-direction

1. *Exception in thread 'main' java.lang.NumberFormatException* Cause: Parameters are missing or not complete.
2. *Sorry! filename could not be generated!* Cause: Chosen filename is too long or the disk is full.

5.2 Matrix Generator

When transferring objects to the viewer, the matrix generator may return the following error message.

1. *Sorry! filename could not be found!* Cause: The file is not present or typed wrong.

6 Conclusion

The Chaincode Object Generator produces chaincode over the alphabet u, d, n, s, w, e and interprets those strings in three-dimensional space. In addition to the Koch curve, implementations of the Dragon curve and the

three-dimensional Hilbert curve are available with Animalab. While small chaincode pictures can be drawn easily by hand, for large examples and especially in three-dimensional space a visualization tool like the Chaincode Object Generator is mandatory. Most examples presented in the literature are two-dimensional, and three-dimensional extensions can be developed, implemented and visualized as well.

15 NIFS Object Generator

Sarah Behrens, Maxim Bortin, and Jianfeng Chen

1 Introduction

Networked iterated function systems (NIFS) are a device for the generation of images. For detailed information we refer the reader to the chapter about the theoretical basics of NIFS.

The NIFS generator is based on those concepts and is one of the object generators integrated into AnimaLab. It was developed in *Java* under *SDK1.3* with the technology of *Java3D*. As a result of the platform independency of *Java* the generator can be run on any platform where *JRE* with *Java3D* is installed.

To be started, this program needs a definition file where the user can define one or more NIFS. The syntax and the semantics of this file are described in Section 4, and Section 5 contains a sample definition file.

2 Command Line Options

Name
nifsgen - object generator based on the concept of networked iterated function systems

This object generator is a part of the AnimaLab, but it can also be run alone. The command line syntax differs accordingly.

Synopsis

- first case (inside AnimaLab):
  nifsgen <file>.def [-h] [-p]
  Options
  -h <host> The host name of the receiver of the output data.
  -p <port> The port number of the receiver of the output data.

- second case (outside AnimaLab):
  nifsgen2 <file>.def [-f]

Options
-f pov The output data is written in povray format into a file called [file].pov.
Starting just with a definition file, the internal viewer will be used.

3 Error Messages

Errors can only occur during the parsing of the definition file, if the syntax rules are not followed or undefined symbols are used. Typical error messages are:

1. `Exception in thread "main"`
   `nifsmodule.ParseException: Undefined symbol "pyramide"`
   The term `pyramide` is unknown. Probably `pyramid` was meant.
2. `Exception in thread "main" nifsmodule.ParseException:`
   `Encountered "," at line 37, column 9. Was expecting: "(" ...`
   In this case, parentheses were omitted.

4 Configuration Files

In the following we explain the syntax of the definition file. (...)? means that the symbols inside the parentheses are optional; | is used for denoting alternatives. With <string>, an alpha-numerical string is denoted.

4.1 Grammar

```
(1) <nifsdef>    ::= (<clib>)? (<tlib>)? <nifslist>
                     "MAIN" "{"  <main> "}"
(2) <clib>       ::= "CONSTLIB" "{" <clist> "}"
(3) <clist>      ::= <string> "=" <expr> ";" (<clist>)?
(4) <tlib>       ::= "TRANSLIB" "{" <tlist> "}"
(5) <tlist>      ::= <string> "=" <trans> ";" (<tlist>)?
(6) <main>       ::= <call> ";" (<main>)?
(7) <call>       ::= <string> "(" <expr> ")"
(8) <nifslist>   ::= <nifs> (<nifslist>)?
(9) <nifs>       ::= "NIFS" "("  <string> ")"
                     "{" (<opt>)?<nodelist> "}"
(10) <opt>       ::= "OPTSIZE" "="  <expr> ";"
(11) <nodelist>  ::= <node> (<nodelist>)?
(12) <node>      ::= "NODE" "(" <string> (<out>)? ")"
                     "{" <parts> <edges> "}"
(13) <out>       ::=  "," "out"
(14) <edges>     ::= "EDGES" "{" (<elist>)? "}"
(15) <elist>     ::= <edge> (<elist>)?
(16) <edge>      ::= "(" <trans> "," <string> ")" ";"
```

```
(17) <trans>     ::=  <transform> ("*" <trans>)?
(18) <transform> ::= ("scale" | "shift" | "rotate")
                     "(" <expr> ","  <expr> ","
                     <expr> ")" | "id" "(" ")" |
                     <matrix> | <string>
(19) <matrix>    ::= "(" <expr> ","  <expr> ","
                     <expr> "," <expr> ","  <expr> ","
                     <expr> "," <expr> ","  <expr> ","
                     <expr> "," <expr> ","  <expr> ","
                     <expr> ")"
(20) <parts>     ::= "PARTS" "{" (<plist>)? "}"
(21) <plist>     ::= <obj> ";" ("{" <preprocess> "}")?
                     (<plist>)?
(22) <preprocess>::= <trans>
(23) <obj>       ::= ("cuboid" | "pyramid") "("
                     <expr> ","  <expr> ","  <expr> ","
                     <expr> "," <expr> "," <expr> ","
                     <expr> ")" | "sphere" "(" <expr> ","
                     <expr> "," <expr> "," <expr> ","
                     <expr> ")" | "cone" "(" <expr> ","
                     <expr> ","  <expr> "," <expr> ","
                     <expr> "," <expr> "," <expr> ")"
(24) <expr>      ::= <term> <expr'>
(25) <expr'>     ::= ( ("+" | "-") <term> <expr'>)?
(26) <term>      ::= <factor> <term'>
(27) <term'>     ::= ( ("*" | "/") <factor> <term'>)?
(28) <factor>    ::= <string> | "sin" "(" <expr> ")"
                     | "cos" "(" <expr> ")"
                     | "tan" "(" <expr> ")"
                     | "exp" "(" <expr> ")"
                     | "log" "(" <expr> ")"
                     | "(" <expr> ")"
                     | "-" <factor>
```

4.2 Semantics

Line (1) shows the essential parts of a NIFS definition file. The lines (2) and (3) explain the construction of the constant library (CONSTLIB). The CONSTLIB is intended to create a list (<clist>) of assignments. The value of a numerical expression (<expr>) is assigned to an identifier (<string>) which is used like a global constant in the definition file and therefore must be unique.

The same is valid for the transformation library (TRANSLIB) except for the assignment; here we need an affine transformation as value (<trans>).

<nifslist> in line (8) is a construct that consists of several NIFS declarations (<nifs>). A NIFS declaration (line (9)) starts always with the keyword "NIFS" followed by the unique name of the NIFS. The parameter <opt> is

optional and specifies the minimal size a part must have to be displayed; parts that are smaller than OPTSIZE are left out. The default value of OPTSIZE is 0. This concept allows us to accelerate the derivation process.

A <nodelist> (11) is a list of node definitions (<node> (12)). A node definition has to start with the keyword "NODE" followed by the node name that again has to be unique. The optional parameter <out> indicates whether the current node is going to be the output.

The second part of the definition of a node is <edges> (14) consisting of the keyword "EDGES" followed by a list of edges <elist> (15). An edge (<edge> (16)) consists of a transformation <trans> and a <string> representing the target node of this edge.

The nonterminal <trans> (17) represents a single transformation (<transform>), or a composition of transformations with the semantics: $(f * g)(x) = f(g(x))$. <transform> stands for an affine transformation in three-dimensional space. There are various possibilities for such a transformation (18):

1. It can be a scaling ("scale"), a translation ("shift") or a rotation ("rotate"), each with three parameters. In case of a scaling the parameters indicate the scaling factors for the different axes, in case of a translation they are the components of the translation vector, and in case of a rotation they denote the rotation angles about the different axes.
2. Another possibility is the identity, denoted by "id()".
3. You can also define your own transformation as a 3×3 matrix and a translation vector. This yields twelve values which have to be entered into the matrix definition (19). The first nine values correspond to the matrix in row-majored form, and the other three to the components of the translation vector.
4. Supposed you have already declared your own transformation in the TRANSLIB (4) you can just use its identifier from this on.

   The first part of the node definition consists of <parts> (20). It has to start with the keyword "PARTS" followed by a list of parts (<plist> (21)). A part consists of an object definition (<obj> (23)) and <preprocess> (22). An <obj> represents a three-dimensional object that can be one of the following kinds:
   (a) A "cuboid" or a "pyramid", followed by seven values that correspond to the width x, the height y, the depth z of the object and the rgbalpha values.
   (b) A "sphere" followed by five parameters corresponding to the radius of the sphere and the rgbalpha values.
   (c) A "cone" described by seven parameters: the height y, the upper radius, the lower radius and the rgbα-values.

The objects cuboid and pyramid are placed with the left front corners to the origin. The sphere has its center at the origin and the cone has the center of its lower base at the origin.

The non-terminal <preprocess> in line (21) allows you to define an affine transformation that will be applied once so that the object defined before can be modified to get the right form and position.

In lines (24)-(28) we describe the non-terminal <expr> which represents an arithmetic expression. A <factor> can be a <string> that is either a number or an identifier of a constant declared in CONSTLIB (2). Furthermore it is possible to use the mathematical functions sine, cosine, tangent, exponential and logarithm.

Finally we have to explain the "MAIN"-section (1) which is defined in line (6). <main> consists of a list of <call>s (7). A call is composed of an identifier (<string>) of a previously defined NIFS, followed by an <expr> representing the desired number of iteration steps this NIFS will have to compute.

5 Example

The following is an example that illustrates how to define an NIFS using the introduced grammar. This example defines one side of the well-known Koch Snowflake. It contains most of the introduced concepts.

```
CONSTLIB
{
  width = 12;
  height  = 0.3;
  depth  = 1;
}
TRANSLIB
{
  sc = scale(1/3, 1/3, 1);
  turn1 = rotate(0, 0, pi/3);
  turn2 = rotate(0, 0, -pi/3);
  s1 = shift((1/3)*width, 0, 0);
  s2 = shift(width/2, (width/3)*sin(pi/3), 0);
  s3 = shift((2/3)*width, 0, 0);

  f1 = sc;
  f2 = s1*turn1*sc;
  f3 = s2*turn2*sc;
  f4 = s3*sc;
}

NIFS(koch)
{
```

```
 NODE(a)
 {
   PARTS
   {
    cuboid(width, height, depth, 0, 0, 0, 1);
   }
   EDGES
   {
     (f1, a);
     (f2, a);
     (f3, a);
     (f4, a);
     (id(), b);
   }
 }

 NODE(b, out)
 {
   PARTS {}
   EDGES
   {
     (shift(0, 15*height, 0), b);
   }
 }

}

MAIN
{
 koch(4);
}
```

The output image for this NIFS is shown in Figure 1.

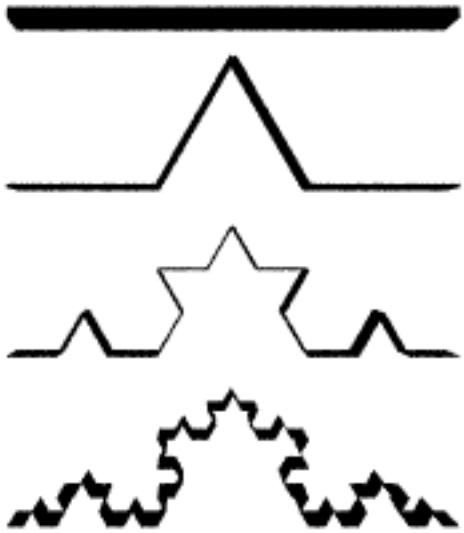

Figure 1. The Koch Snowflakes

16 Collage3D Object Generator

Carolina von Totth

1 Introduction

Collage3D is a tool for creating and displaying three-dimensional collages. It has been implemented within the framework of TREEBAG (see Drewes [Dre01]) , which is a tree-based generation and visualization system. The basic idea behind TREEBAG is quite simple: various types of tree grammars (called *generators*) yield trees, which are comprised of abstract symbols with no meaning whatsoever. Such a tree can then be interpreted in various ways by different algebras.

Tree grammars are purely syntactical devices for the generation of tree languages, and thus method-independent. The generated trees only gain meaning when interpreted by an appropriate algebra.

Although tree grammars as used in TREEBAG are independent of any picture generation method such as collage grammars, it is known that for every tree grammar and corresponding collage algebra there exists a collage grammar yielding the same language (see Chapter 5).

Therefore in order to generate three-dimensional collages, we need at least one tree grammar and the corresponding algebra, which then provides an interpretation for the trees the grammar has produced. Moreover we need a display component, so that we can actually *view* what we have built.

The paper consists of two parts. Section 2 contains a concise description of the graphical user interface of the *Collage3D* object generator as well as tree generation by tree grammars, 3D collage algebras and the display. Moreover, the impatient reader may find an intuitive introduction to the basic concepts of *Collage3D* in the mini-tutorial of Section 3.

2 Collage3D Module

From a modelling point of view, collage grammars are syntactical devices that can be used to create highly complex objects with a minimum of work for the user. Since TREEBAG already provides a multitude of tree grammar types for generation purposes, we have only had to add an algebra that interprets the output trees yielded by these grammars over a given signature as collages, and a display in order to visualize these collages as three-dimensional objects.

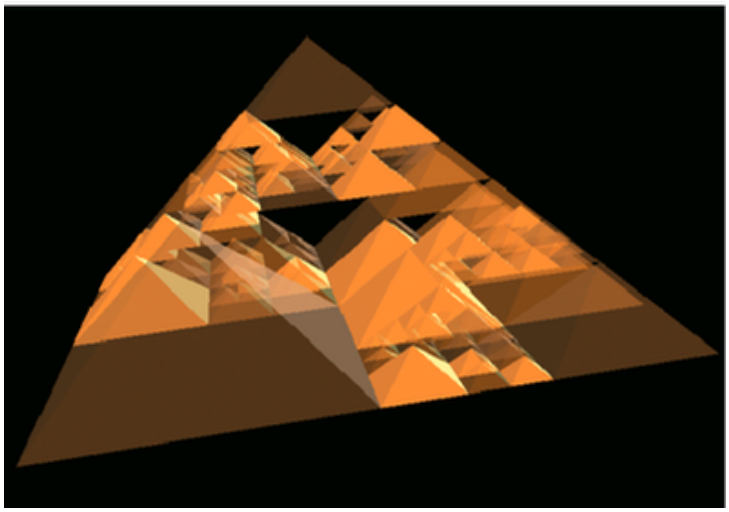

Figure 1. A small (and exceedingly well-known) example

2.1 Worksheet

The worksheet is the main window of the system. Here, the user can arrange instances of TREEBAG components as nodes of an acyclic graph and establish input/output relations by drawing edges between these nodes.

The following menu items are available within the worksheet:

- SAVE: saves the current worksheet configuration.
- ADD: opens a new file dialog from which a worksheet file can be selected. The selected worksheet configuration is then added to the current worksheet configuration, if such a configuration exists.
- ADD COMPONENT: opens a new file dialog from which a single node can be selected.
- LOAD: loads the selected worksheet configuration after removing the current configuration.
- CLEAR: removes all nodes from the worksheet.

Furthermore, single nodes can also be added by double-clicking in a free area of the worksheet. Nodes can be dragged around in the usual way; double-clicking on a node yields either a display or a menu window for the corresponding component. The commands offered by a component can also be reached by pressing the right mouse button on the corresponding node.

A worksheet can also be described by an ASCII configuration file; for the syntax of worksheet configuration files please refer to the TREEBAG Manual (Drewes [Dre01]).

2.2 Grammar

[1]

There are two special commands which every TREEBAG component accepts (and which, therefore, are not mentioned below): the command *delete node*, which deletes an instance from the worksheet, and the command *reload file*, which updates a node when the file from which it was initially loaded has been changed. The latter is possible only if the class of the object described in the respective file stayed the same. Otherwise, an error message appears and reloading is cancelled.

Regular Tree Grammars

Mathematically, a regular tree grammar is a 4-tuple (N, Σ, R, S), where N is a set of nonterminals, Σ is a signature, R is a set of productions $A \to t$ consisting of a nonterminal $A \in N$ and a tree t over $\Sigma \cup N$ (where the nonterminals are considered as symbols of rank 0), and $S \in N$ is the start symbol. It is required that Σ does not contain any symbol $A{:}\,0$ for which $A \in N$. The tree language generated by such a grammar is the set of all trees over Σ that can be derived from S by repeatedly applying productions in the obvious way, i.e., replacing a nonterminal with the right-hand side of an appropriate production. In addition, in TREEBAG one can assign a weight to every production.

Syntactically, in TREEBAG the set of nonterminals is a set of names. The syntax for productions is

```
< name > -> < tree > [ weight < rat+ > ] .
```

For example, the following is a syntactically correct regular tree grammar:

```
( { left, somewhere },
  { left:2, .:2, 0:0 },
  { left -> left[left[left, somewhere], somewhere]
            weight 4,
    left -> 0,
    somewhere -> (somewhere . somewhere),
    somewhere -> 0 },
    left )
```

The grammar generates all trees with zeroes as leaves, such that the leftmost branch consists of an even number of left symbols and all the other symbols (except leaves) are dots.

The implementation of regular tree grammars basically provides two modes, called *enumeration* and *random generation* mode. The enumeration mode yields an enumeration of the tree language generated by the grammar. In this mode the main commands are *advance* (compute the next tree

[1] This section in its entirety represents an excerpt from the TREEBAG Manual [Dre01], and is reproduced here with the permission of the author.

in the enumeration), *reset* (return to the first tree), and *random generation* (turn to random generation mode). In addition, there is the command *derive stepwise*, which yields a stepwise derivation of the current tree of the enumeration. An application of this command makes the grammar turn to the start of the derivation and brings up the additional commands *single step* (perform a single step of the derivation), *parallel step* (perform several steps of the derivation by applying productions to all nonterminals in parallel), *back* (undo the last single or parallel step), and *results only* (return to the enumeration mode).

The random generation mode iteratively applies productions to all nonterminals in parallel, using a random choice depending on the weight of productions. Here, the main commands are *refine* (replace all nonterminals in the current tree by the right-hand sides of randomly chosen productions), *back* (undo one refinement step), *enumeration* (apply terminating productions and return to enumeration mode), and *reset* (return to the initial nonterminal). As mentioned above, the random choice taken in the refinement step depends on the weight of productions (the default weight being 1). If w is the sum of the weights of all productions with left-hand side A and one of these productions has weight w_0, then this production will be chosen with probability w_0/w (where $0/0 = 0$, by convention). In case of the sample grammar above, for instance, this would mean that the first production is applied to an occurrence of the nonterminal `left` with probability $4/5$.

Parallel Deterministic Total Tree Grammars

Parallel deterministic total tree grammars are a very special, yet frequently useful class of tree grammars. They are similar to regular tree grammars, except that, for every nonterminal, there must be exactly two productions the second of which is terminating (i.e., contains no nonterminals). Derivations are maximum parallel, applying n times the first production to all nonterminals in parallel, and then, finally, the second production (which corresponds to the derivation mode of table-driven L systems, with a fixed number of two tables). Thus, for every $n > 1$ there is exactly one tree t_n that can be derived by a derivation of length n. The generated language is the set $\{t_1, t_2, ...\}$ of all these trees.

The syntax is like the one for regular tree grammars, the difference being that the two productions for each nonterminal are denoted as

```
< name > -> < tree > | < tree >
```

(where the left-hand side is the nonterminal and the two trees are the first and second right-hand side, separated by |). Because of the deterministic nature of these grammars, productions cannot be assigned a weight. An example is

```
( { start, left, right },
  { left:2, right:2, .:2, 0:0 },
```

```
{ start - > (left . right) | (0 . 0),
   left - > left[left, left] | left[0, 0],
   right - > right[right, right] | right[0, 0] },
 start )
```

which generates all trees $(t\ .\ t')$ such that t and t' are fully balanced trees of equal height, where t consists of the symbols `left` and 0, and t' consists of the symbols `right` and 0.

The implementation provides enumerations of the generated terminal and nonterminal trees, the nth tree in the enumeration being the one that can be derived by a parallel derivation of length n. The basic commands are *advance* (turn to the next tree in the enumeration), *back* (return to the previous tree in the enumeration), and *reset* (return to the first tree). In addition, the commands *terminal results* and *nonterminal results* allow to switch between terminal and nonterminal output terms.

ETOL Tree Grammars

ETOL tree grammars are a tree-grammar version of the well-known ETOL systems (extended context-free Lindenmayer systems with tables). They are parallel grammars which generalize the previously described class `generators.pdtGrammar`.

Syntactically, an ETOL tree grammar is a regular tree grammar except for three differences. First, the set of rules is now a set of sets of rules $\{R_1, ..., R_n\}$. Each set R_i is called a table. Second, the output signature is allowed to intersect with the set of nonterminals. An example is

```
(  { c },
  { a:2, b:2, c:0 },
  { { c -> a[c, c] }, { c -> b[c, c] } }, c ).
```

A derivation step is done by choosing a table and then replacing all nonterminals in parallel, using rules of the chosen table. If, for a given nonterminal A, the table does not contain an appropriate rule, then the implicit rule $A \to A$ is applied (i.e., A stays as it is). The example above is deterministic in the sense that every table contains only one rule for each nonterminal. In this case, the result of a derivation is uniquely determined by the chosen sequence of tables. (The grammar generates all fully balanced binary trees, where the internal nodes are labelled with a respectively b, the leaves are labelled with c, and nodes at the same distance from the root have identical labels.)

The third syntactical deviation from regular tree grammars is that one may add as a fifth component (i.e., after the specification of the initial nonterminal and separated by a comma), a regular expression which specifies the admissable table sequences. In such a regular expression, the numbers 1,...,n denote the tables, juxtaposition denotes concatenation of table sequences, commas separate alternatives, $E*$ denotes arbitrarily many repetitions of E

(i.e., the Kleene star), $E+$ denotes at least one repetition of E (which is equivalent to $EE*$), square brackets enclose optional parts, and parantheses (...) are used to override the precedence rules. Without parentheses, * and + bind strongest, concatenation binds second strongest, and commas bind weakest. For instance, in a grammar with three tables, the expression (1,2 3)*2+ requires admissable table sequences to start with a sequence of tables 1, 2, 3 where tables 2 and 3 occur only in direct succession (and in this order). After this initial part, the sequence must end with at least one application of table 2 (note the space between 2 and 3; without it, this would refer to table number 23).

The implementation of ETOL tree grammars translates such a grammar into a regular tree grammar whose output signature is monadic, and a top-down tree transducer. Roughly speaking, an output tree of the regular tree grammar determines the table sequence (every symbol corresponds to a table) and the top-down tree transducer implements the actual rules, using the input symbols in order to choose the tables consistently. The parsing routine of the class `generators.ETOLTreeGrammar` stores the two components in files named `file.1` and `file.2`, where `file` refers to the parsed file. (The two files are syntactically correct descriptions of TREEBAG components and can therefore be studied or loaded onto the worksheet if someone wants to do so.)

The available commands are rather similar to those of regular tree grammars. There are two basic modes: *table enumeration* (the initial one) and *random tables*. The first implements an enumeration of the table sequences. In this mode, the following commands are available:

- *advance* turns to the next table sequence in the enumeration. A derivation based on this sequence is chosen nondeterministically, and the resulting tree is the output tree. If the grammar is deterministic, this yields an enumeration of the generated tree language;
- *reset* returns to the first table sequence;
- *derive stepwise* is similar to the corresponding command of regular tree grammars and allows to view the current derivation step by step, using the commands *derivation step* (perform one parallel step of the current derivation), *back* (back up one step), and *results only* (switch back to the state in which only the terminal results of derivations are shown);
- *random generation* switches to the second main mode.

If the grammar contains nondeterministic tables, an additional command *new derivation* is available. Upon invocation, it randomly chooses a new derivation based on the current table sequence (which is not affected by the command).

In the *random tables* mode a derivation based on a random table sequence is performed using the commands *refine*, *back*, and *reset*, similar to the corresponding mode of regular tree grammars. Note, however, that only the table sequence is chosen at random. If the grammar is nondeterministic,

the command *new derivation* should be used in order to get further output trees.

Pragmatically, if you do not restrict the table sequences by a regular expression, it is often a good idea to have one or more tables which contain terminating rules for all nonterminals. In this case, the translation into a regular tree grammar and a top-down tree transducer guarantees that the enumeration mode yields only defined results (which may otherwise fail to be the case because the tree transducer does not always turn a table sequence into a terminal tree).

The translation of the optional regular expression into a regular tree grammar does not perform state minimization. Therefore, it is not as efficient as one would like it to be. Moreover, depending on the expression used, certain table sequences may occur repeatedly in the table enumeration mode.

2.3 Algebra

In this implementation of the 3D collage algebra, a collage is a list of three-dimensional objects called *parts*, which possess various appearance attributes like *colour* and *transparency*. Colour is given either in *rgb* values (three numbers between 0 and 1), or in *rgbalpha* values: four numbers between 0 and 1, where the last number denotes the transparency of the part in question.

Parts can also be textured, in which case the following texture attributes can be defined: *mirror, transparency* and *blending*.

Several predefined primitives are available: *cube, pyramid, cylinder, frustum, sphere* and *polyhedron*.

A transformation operation consists of one or more concatenated affine transformations. In the trees yielded by the grammar it is represented by a ranked symbol with a rank greater than 0. The transformation is applied to each of its argument collages.

The syntactical appearance of a collage algebra for two-dimensional collages is described in detail within the corresponding section of the TREEBAG Manual [Dre01]. The generalisation of collage algebras for three-dimensional collages as implemented in *Collage3D* follows the syntactical specifications presented there, except for the few places where the nature of three-dimensional collages requires specific handling.

The syntax for a three-dimensional collage algebra in *Collage3D* is described by the following rules:

```
<definition> ::= <parts> | <trans>
<trans>      ::= transformations { <transformation>
                 (, <transformation>)* }
<transformation> ::= <name> = <primitive transformation>
                     (. <primitive transformation>)*
<primitive transformation> ::= scale (<rat>
                     [, <rat>, <rat>])
                     //a uniform/non-uniform scaling
```

```
                    | translate <point>
                    | rotate (<rat>, <rat>, <rat>, <rat>)
                    //rotate about vector and angle
                    | matrix (<rat>, <rat>, <rat>, <rat>,
                    <rat>, <rat>, <rat>, <rat>,
                    <rat>, <rat>, <rat>, <rat>,
                    <rat>, <rat>, <rat>, <rat> )
<parts>      ::=  parts { <part> (, <part>)* }
<part>       ::= <name> = <geometry>, <appearance>
<geometry>   ::= cube    { <rat>, <rat>, <rat>
                 pyramid { <rat>, <rat>, <rat>
                 cylinder{ <rat>, <rat>, <rat>
                 frustum { <rat>, <rat>, <rat>, <rat>
                 sphere  { <rat>, <rat>
                 polyhedron{ points {<named point> (,
                 <named point>)*},
                 polygons{<polygon> (, <polygon>)*}
                 [, texcoords{<point> (, <point>)*}]
<point>      ::= (<rat>, <rat>, <rat>)
<named point>::= <name> = <point>
<polygon>    ::= (<name> (, <name>)*)
<appearance> ::=    colour <rgb-colour>
                  [, <texture>] }
<rgb-colour> ::= [<rat>, <rat>, <rat> [, <rat>]]
                 // (r,g,b, optional alpha value)
<texture>    ::= texture [<name> [, mirror] [, <rat>]
                 [, blend]]
                 // mirror, transparency and blending
```

The algebra menu only offers the two standard commands *reload file* and *delete node*.

2.4 Display

The 3D-Display is implemented in such a way that it can be used with every generator/algebra combination that yields three-dimensional objects as their output.

The display menu contains the following items:

- JPEG OUTPUT: writes a screenshot of the current content of the display to file.
- ENABLE ANTIALIASING: enables line and colour antialiasing, possibly resulting in a slowed navigation.
- DISABLE ANTIALIASING: disables antialiasing.
- ENABLE PICKING: switches the navigation mode to picking. In picking mode, the individual parts contained in the displayed collage can be rotated by clicking on them with the right mouse button and dragging.

Such a part can also be translated by pressing the middle mouse button while on it, and dragging.

As soon as the mouse button is released, an additional window opens, containing the transformation matrix used to transform the manipulated part into its current shape and position.

- DISABLE PICKING: switches from picking to navigation mode.
- PARALLEL VIEW: changes the view to parallel view.
- PROJECTION VIEW: changes the view to projection view.
- EXPORT TO VRML: exports the displayed three-dimensional object into the *VRML* V1.0c format.
  (For more information see Chapter 17.)

These three-dimensional objects are displayed in a separate window and can be viewed from any direction.

Furthermore, it is possible to export either screenshots into the *JPEG* image compression format, or the entire geometry and appearance of the current scene into the *VRML* file format for further processing.

3 Mini-Tutorial

We present the simplest conceivable combination of grammar, algebra and display, yielding a simple cube. So, in order to get that cube displayed, we have to follow a few easy steps:

1. Build an algebra where we describe the 3D object we want to be displayed, in this case a simple cube.
2. Build the corresponding grammar, containing one single production with the start symbol on the left-hand side, and the symbol for our cube on the right-hand side.
3. Construct a simple display.
4. Build a worksheet containing the algebra, grammar and display.

3.1 Algebra

An algebra file always starts with a line indicating which kind of algebra is meant, and optionally a name that will be displayed on the worksheet alongside the algebra node. For a *Collage3D* algebra, the first line is

```
applications.pictures3D.collageAlgebra():
```

or, if we want to assign a name to the algebra in question,

```
applications.pictures3D.collageAlgebra("simple cube
                                        algebra"):
```

respectively.

The actual content of the algebra is enclosed by braces. Our algebra, containing the definition of a cube, looks like this:

```
applications.pictures3D.collageAlgebra("simple cube
                                          algebra"):
{
   parts {
            cube = cube{1,1,1, colour[1,1,1]}
         }
}
```

The `cube` primitive used to build a three-dimensional cube takes four parameters: width, height, depth, and rgb colour. The current colour of the cube is white.

3.2 Grammar

We are going to use the simplest kind of grammar TREEBAG offers: a regular tree grammar. As with the algebra, we first need to describe the type of grammar we are going to use — and here we assign a name to it right away, so that the first line of the file reads:

```
package generators.regularTreeGrammar("simple cube
                                       grammar"):
```

The content of the grammar is enclosed in parentheses, and is divided in three main parts — each enclosed by braces and separated by commas:

```
package generators.regularTreeGrammar("simple cube
                                       grammar"):
   (
     { S },
     { cube:0 },
     { S -> cube },
     S
   )
```

The first part is a set of the nonterminals used in the grammar.

The second part is a set of all the ranked symbols that can appear in a tree yielded by this grammar — a sort of terminal alphabet. The only symbol we use here is `cube`, and because it is a part and not an operation on some other symbols, it has the rank 0.

The third part is a set of productions — the left-hand side is always *one* nonterminal symbol, the right-hand side is a term (tree). At the moment, we

have only one production: one which replaces the start symbol with a tree that is composed of only one symbol.

Since a grammar can contain any number of nonterminals, we have to explicitly name the start symbol, and this is what we do in the fourth part — we assign the function of start symbol to the nonterminal S.

3.3 Display

Since no special display configuration options are implemented yet, the 3D display file looks like this:

```
applications.pictures3D.display3D:
```

3.4 Worksheet

The simplest way to build a worksheet is by making use of the TREEBAG-GUI. Therefore, we start TREEBAG and see an empty worksheet (Figure 2). By double-clicking on the worksheet surface we get a dialog (Figure 3) with the help of which we can load the files we have edited previously. If their syntax is correct, they appear as nodes on the worksheet.

Figure 2. Empty worksheet

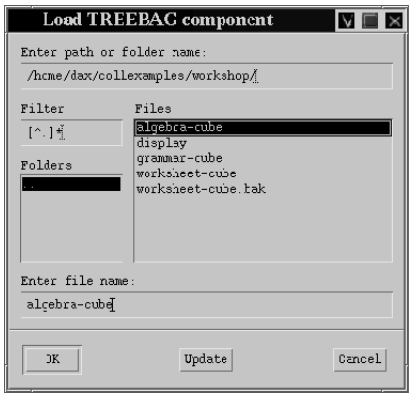

Figure 3. File dialog

We can load the algebra, grammar and display nodes that way. The display needs an input from both the algebra and grammar nodes — we can provide that input by clicking once on the respective source node (the algebra or grammar node) and then clicking once on the target node (the display node in this case). The result should be a correct worksheet configuration as shown in Figure 4.

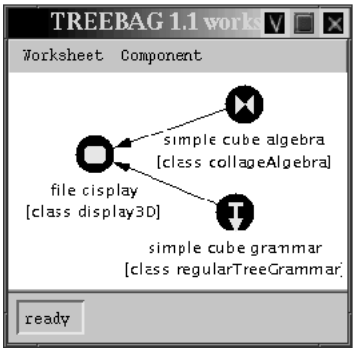

Figure 4. Worksheet

Double-clicking on each of the worksheet nodes yields the corresponding dialogs. In the case of the display node, the first double-click yields the display window, the second double-click the display dialog. An example how the screen can look now is shown in Figure 5.

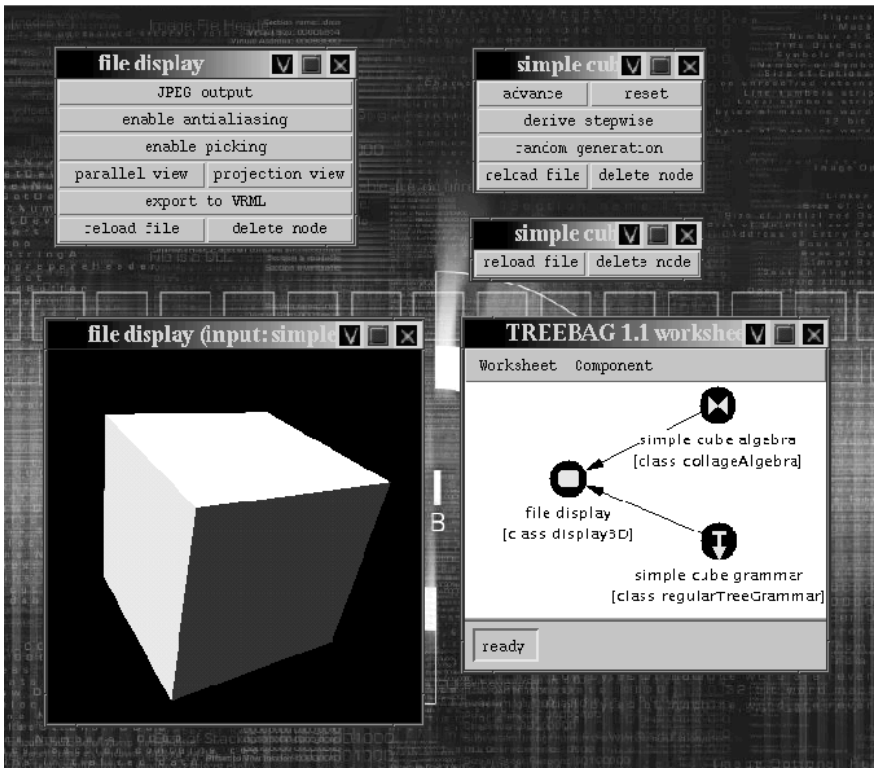

Figure 5. The whole screen

17 VRMLSaver – Exporting Data from Java3D to VRML

Björn Cordes

1 Introduction

What is the *VRMLSaver* good for? The *VRMLSaver* allows to transform a *Java3D* scene graph into a *VRML* (Virtual Reality Modeling Language) scene graph, which is then written to a file. It has been developed primarily as an export module for 3D collages in *Collage3D*. Such an export allows to edit and render the scenes created with *Collage3D* with other software and thus allows the integrated use of *Collage3D* within larger processes of 3D picture generating. Only a subset of the possibilities that *Java3D* offers is transformed into *VRML* due to a lack of corresponding concepts in *VRML*, noteworthy differences between certain concepts in *Java3D* and *VRML*, and the fact that *Collage3D* uses only a subset of *Java3D*.

Why is *VRML* used as the destination format? One major advantage of *VRML* is that one does not need expensive 3D modelling software to view this format. As there are freeware or shareware viewers and plugins available for the common web browsers. Nevertheless, the majority of modelling and rendering software offers possibilities for an import of *VRML* data.[1] Furthermore, *VRML* is a format which humans can comprehend and, provided some familiarity with the format, edit with any text editor.

Which version of *VRML* is used? There are two versions of *VRML* available. The first and older version is *VRML* version 1.0c. Despite the less comfortable way of representing the structure of a graph this version offers a matrix representation for transformations. Since *Collage3D* exclusively uses transformation matrices, *VRML* version 1.0c was the first choice. An export to *VRML97*, a derivate of *VRML* V2.0, requires a solution for the nontrivial problem of decomposing a transformation matrix into basic transformations, as these are the only transformations which are supported by *VRML97* or which can be emulated. When matrix decomposition became available, the second version of the *VRMLSaver*, now exporting to *VRML97*, could have been implemented. However, currently only an implementation exporting to *VRML* version 1.0c exists.

[1] This is true at least to a certain degree. This topic will be covered in the section about the concrete implementation of the *VRMLSaver*.

Why the odd name? *Java3D* introduced the concept of Loader classes that can create scene graphs from files for an implementation-specific file format. A loader is, in other terms, an import module that converts the 3D data structure described by a 3D file format into the 3D data structure of *Java3D*. Doing the opposite, i.e. converting a *Java3D* scene graph into a 3D file format, one has an export module that we call Saver in the style of *Java3D*'s Loader.

The structure of this paper. Since the subject of the *VRMLSaver* is to transform *Java3D* data into *VRML* data, the first step is to reflect the input data. Therefore, Section 2 describes how the input scene graph is structured, what *Java3D* classes are used, and how they relate to each other. This section does not treat *Java3D* in general but concentrates on its use in *Collage3D*. The following two sections describe how the *Java3D* scene graph is transformed into a *VRML* V1.0c, respectively *VRML97*, scene graph. The specification uses graph transformation. The final section covers some topics about the implementation of the *VRMLSaver*. Some basic knowledge about 3D graphics, *Java3D*, *VRML*, and graph transformation can prove helpful for understanding.

2 *Java3D* Scene Graphs and Their Use in *Collage3D*

Java3D scene graphs consist of nodes and node components. In principle all nodes form a tree. There are two different kinds of nodes: leaves, which can never have child nodes, and groups, which may have. Scene graphs in *Collage3D* use three kinds of groups: `BranchGroups`, `TransformGroups`, and `SharedGroups`. (The structure of *Java3D* scene graphs in *Collage3D* is also graphically represented by means of an UML class diagram in Figure 1.) `BranchGroups` are used for no other purpose than grouping nodes. `TransformGroups` refer furthermore to a transformation which is applied to all elements of the subtree. `SharedGroups` are special groups that cannot be child nodes of other group nodes. They are the root nodes of subgraphs and can be referred to by one or more so-called `Link` nodes. The main purpose of a `SharedGroup` is efficiency since it allows the re-use of data at several locations in the scene graph (but this also turns the tree into a graph). Of leaf node types occurring in *Java3D*, only three can be exported by the *VRMLSaver*: `Light` nodes, the already noted `Link` nodes, and most important the `Shape3D` node. A `Light` node allows the placement of light sources in the scene. `PointLight` and `DirectionalLight` are the only subclasses that are supported for export. Directional light sources illuminate a scene along rays parallel to a 3-dimensional vector while point light sources illuminate a scene equally in all directions from a given position, though the brightness decreases with increasing distance from the source. The only function of `Link` nodes is to refer to `SharedGroups` as noted above. `Shape3D` nodes are the most important nodes because they contain the data of a 3D object. This data can

be separated into two categories: the geometry, which contains coordinates, normals, and texture coordinates for all vertices of the 3D object and the appearance, which describes the transparency value, the material[2], and the textures. In *Collage3D* each `Shape3D` refers to exactly one `Geometry` node component (though more than one would be possible) and one `Appearance` node component. More precisely, the `Geometry` is an `GeometryArray`, and to be exact, it is either a `TriangleArray` or a `TriangleStripArray`. In *Java3D*, `Appearance` can refer to much more information than only material, textures, and transparency, but these are the only ones with corresponding concepts in *VRML*, and thus the only ones which will be translated. Concerning the texture, it is worth knowing that *Collage3D* only uses textures with one image map. All other image maps will be ignored for export purposes. (For more detailed information on *Java3D* see, e.g., Sowrizal et al. [SRD00] and the Java 3D Engineering Team [Tea00].)

3 Translation of *Java3D* Scene Graphs into *VRML* V1.0c Scene Graphs

3.1 *VRML* V1.0c Basics

Pesce, Bell and Parisi, the developers/designers of the Virtual Reality Modeling Language (*VRML*), describe the basic concept of *VRML* in [BPP95] as follows:

> *VRML* defines a set of objects useful for doing 3D graphics. These objects are called Nodes. Nodes are arranged in hierarchical structures called scene graphs. Scene graphs are more than just a collection of nodes; the scene graph defines an ordering for the nodes. The scene graph has a notion of state — nodes earlier in the world can affect nodes that appear later in the world. For example, a Rotation or Material node will affect the nodes after it in the world. A mechanism is defined to limit the effects of properties (Separator nodes), allowing parts of the scene graph to be functionally isolated from other parts.

VRML V1.0c allows the emulation of trees through the use of separator nodes. However, one has to keep in mind that the order of the child nodes of each node is important. Different orders can result in different scenes. A child node can affect all nodes of its subtree, as well as its following sibling nodes and their subtrees, but not its precedent siblings or their subtrees. Thus the first child node can affect all nodes of its subtree, as well as all its sibling nodes and their subtrees, while the last child node can only affect the nodes of its own subtree. Furthermore a node type does not affect all other node types, e.g. a transformation node does not affect a material node. Apart from

[2] This is the way how light is reflected off the surface of a 3D object to create colour.

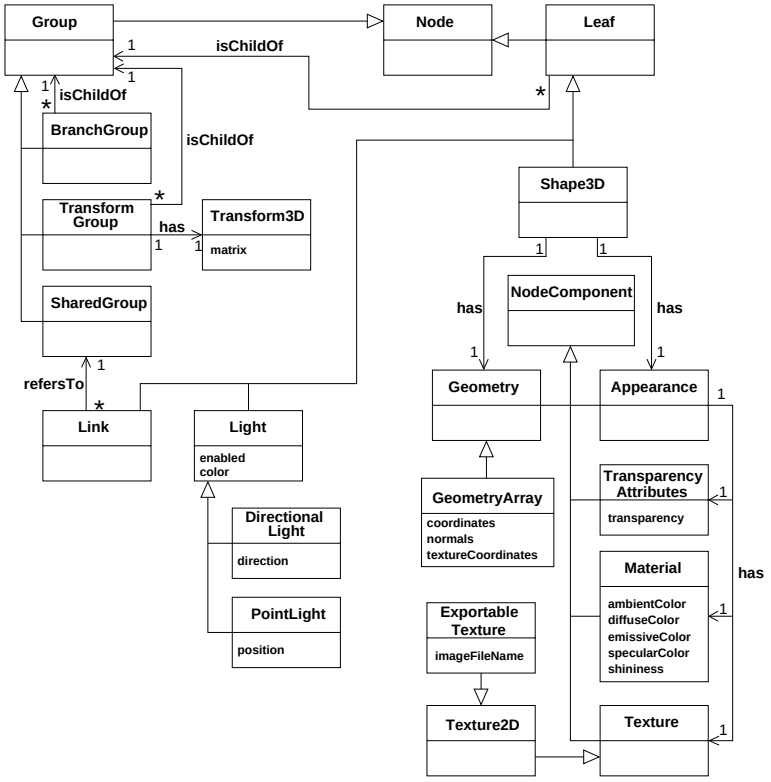

Figure 1. UML class diagram: *Java 3D* classes used by *Collage3D* and the *VRML* Export (The multiplicities of the associations do not reflect the possibilities offered by *Java3D* but their use in *Collage3D*.)

transformation nodes whose effects stack, all other node types affect nodes of the same type in an overriding manner (the local one overriding the more global one).

3.2 Transformation

The translation of a *Java3D* scene graph into a *VRML* V1.0c scene graph is specified by means of graph transformation rules (see Corradini et al. [CMR+97]) on UML object diagrams. Since they are not of interest for our purpose we omit the data values of the attributes, and the operations in the object diagrams.

In order to ease the understanding of the following explanation of the graph transformation rules an explicit knowledge of what a graph is can prove very helpful. For this reason a short, general definition of a marked,

targeted graph is given here: A graph consists of a set of vertices, and a set of edges. Every edge connects two vertices. Since the graph is targeted one can distinguish between the source vertice and the target vertice of an edge. Both vertices and edges are marked (even if this is only an invisible mark). In the case of vertices these marks are referred to as labellings and in the case of edges as markings.

The graph transformation rules consist of a left hand side L and a right hand side R which both are graphs. Furthermore, a rule has a set I that specifies vertices that are, ignoring labellings of vertices for this purpose, vertices of L as well as of R. In the following these vertices are referred to as interface vertices. The application of a rule to a graph G comprises several intermediate steps. The first step is to find an image of the left hand side L in the graph G. As a second step it is necessary to determine whether all edges connecting the image of L in question with the rest of G are edges connecting interface vertices of the image with the rest of G. The rule can be applied only if this so-called contact condition is met. After the positive test of the contact condition all elements of the image of L except for interface vertices of the image are removed from G. Thus an intermediate graph T is created. Next to last R, excluding its interface vertices, is added to T whereas the inserted elements are connected with the interface vertices of the image in exactly the same manner as they were connected with the interface vertices in R. The application of the rule is completed by substituting the labellings of the interface vertices of the image with the labellings of the interface vertices of R.

In addition to the graph transformation rules an initial and a terminal state are specified. The initial state defines a condition that the graph subject to the transformation must fulfill before the application of the rules. The terminal state describes a state of the scene graph where the translation from *Java3D* to *VRML* has been successfully completed. Rules are applied until the scene graph has reached a terminal state or no further application of rules are possible. In the latter case the translation has failed.

The transformation rules below are based on the following assumptions:

- Vertices are instances of either *Java3D* classes or *VRML* node types.
- Interface vertices are numbered.
- Edges are associations.
- Labellings of vertices are *Java3D* class names, *VRML* node type names, or attribute names (*Java3D*, *VRML*).
- Markings of edges are association names.
- The labellings and markings of *VRML* constructs are written in italic letters while those of *Java3D* constructs are not.
- Edges with an invisible mark have the following semantics: the source of the edge has the role of a parent node to the target of the edge in the scene graph.

- Child nodes are ordered. In the graphical representation the order descends from left to right, i.e. the left outermost child node is the highest-ordered and the right outermost the lowest-ordered.

Graph Transformation Rules: *Java3D* to *VRML* V1.0c

Initial state:

All graphs are entirely composed of vertices representing instances of the following classes:
`BranchGroup`, `TransformGroup`, `Transform3D`, `Link`, `SharedGroup`, `DirectionalLight`, `PointLight`, `Shape3D`, `Appearance`, `Material`, `TransparencyAttributes`, `ExportableTexture`, `GeometryArray`.
All graphs are trees except for edges connecting vertices labelled with 'Link' and 'SharedGroup' and the root is labelled 'BranchGroup'.

Rules:

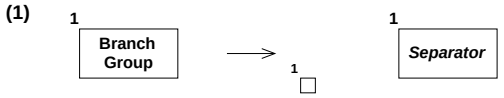

Figure 2. *Java3D* to *VRML* V1.0c — rule (1)

Since the `Separator` is the only *VRML* group node we use, we must apply it to model all three *Java3D* group nodes used by *Collage3D*. *Collage3D* uses `BranchGroups` purely for grouping purposes. Therefore, they can be mapped straightforwardly to the corresponding node type, the `Separator` (see Figure 2).

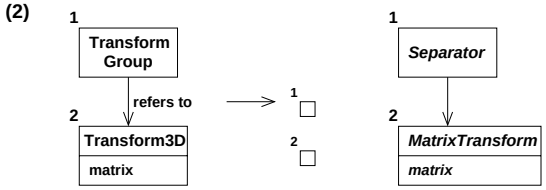

Figure 3. *Java3D* to *VRML* V1.0c — rule (2)

The `TransformGroup` has no counterpart in *VRML* V1.0c but it can be emulated with a `Separator` (grouping) and a `MatrixTransform` node (transforming). It is most important, though not evident from Figure 3, that the `MatrixTransform` node is the highest-ordered child of the `Separator` in order to apply the transformation to the complete group. In

order to be semantically equivalent the matrix of the `MatrixTransform` must equal the transposed matrix of `Transform3D`. The reason for this is that *Java3D* uses column-major ordered matrices while *VRML* uses row-major ordered matrices.

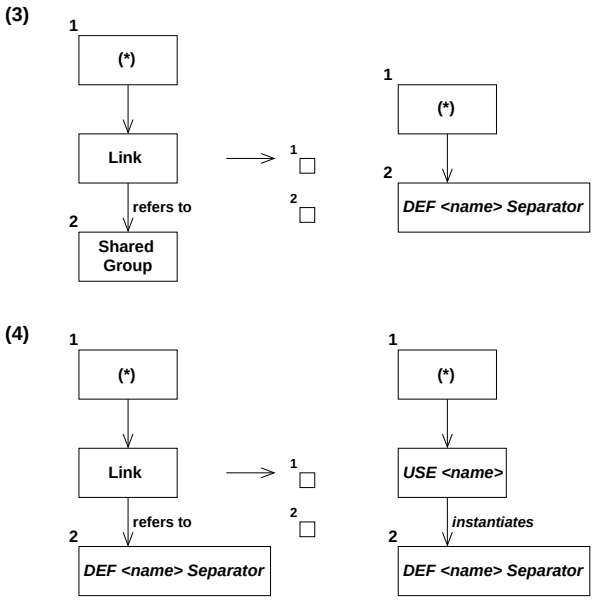

Figure 4. *Java3D* to *VRML* V1.0c — rule (3) and rule (4)

The rules (3) and (4) (Figure 4) are used for transforming `Links` and `SharedGroups`. While rule (3) applies whenever a `SharedGroup` is encountered for the first time during the transformation of the diagram, rule (4) applies when a `Link` is encountered that refers to an already transformed, and thus already used, `SharedGroup`. Each rule of this kind represents a set of (concrete) rules. The vertices labelled with the wildcard '(*)' can either be labelled as `BranchGroup`, `TransformGroup`, `SharedGroup` or as `Separator`.

Links refer to `SharedGroups` and are used for re-usability. In *VRML* this concept is called 'Instantiation' and allows the re-use of nodes. Instantiating `Separators` even makes it possible to re-use complete groups of nodes. The instantiation of a node requires an identifier for this node. In *VRML* a node is usually defined by the name of the node type followed by the contents of the node (attributes or children) enclosed in braces. In the case of a named node, the node type name is preceeded by the key-

word `DEF` and a unique name for the node. With a `USE`-statement, whose only parameter is the name of the node, the node can be instantiated.
The lighting concept of *Java3D* is more powerful than the one used in *VRML* V1.0c. In both languages, the default is that only elements of the subgraph of a light node are illuminated. In *Java3D* the scope of a light node can be extended beyond its subgraph by explicitly specifying these nodes. This feature is not used in *Collage3D* and a translation into *VRML* would require a complete rebuild of the structure of the scene graph. Thus this feature is discarded in the *VRMLSaver*.

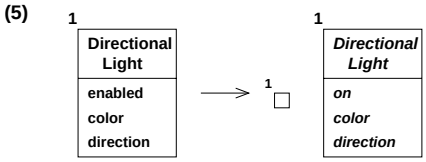

Figure 5. *Java3D* to *VRML* V1.0c — rule (5)

Concerning the `DirectionalLight`, the translation from *Java3D* to *VRML* V1.0c is reduced to the attributes specifying whether the light source is activated or not, the color of the light, and the direction vector (Figure 5).

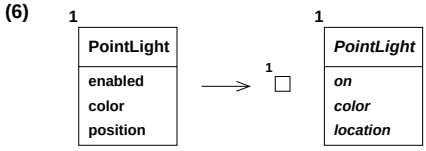

Figure 6. *Java3D* to *VRML* V1.0c — rule (6)

Like the translation of a `DirectionalLight`, the translation of a `PointLight` node leads generally to a loss of information because the scope of a *Java3D* `PointLight` can also be extended beyond its subgraph. Furthermore the information about the attenuation, i.e. the decrease of brightness connected with the increase of distance from the light source, is inevitably lost due to the lack of a corresponding feature in *VRML* V1.0c. Thus the translation shown in Figure 6 is reduced to the attributes specifying the activation status (on/off), the colour, and the 3D location of the light.
The rule shown in Figure 7 translates `Shape3D` nodes, i.e. the data of the 3D objects themselves. Since we have no construct in *VRML* V1.0c which

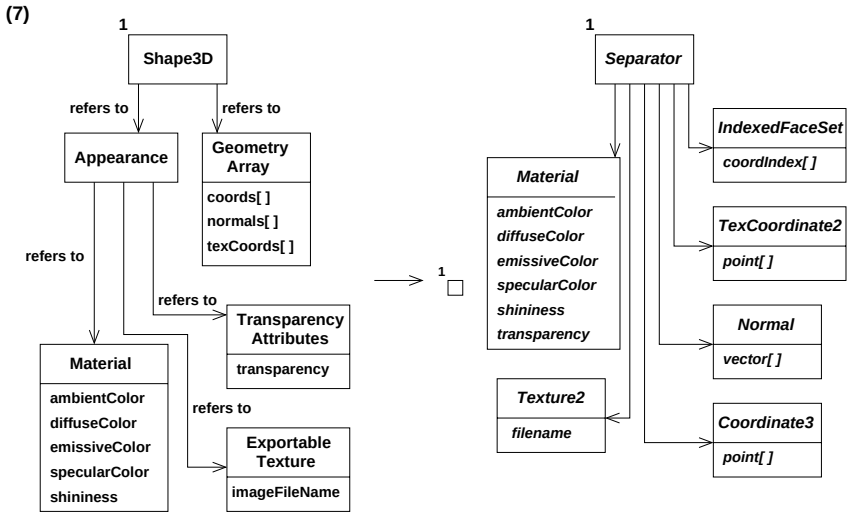

Figure 7. *Java3D* to *VRML* V1.0c — rule (7)

is equivalent to a `Shape3D`, we use a `Separator` node as a container in order to bundle the data. *VRML* V1.0c also does not provide constructs to bundle either appearance or geometry data, thus all the nodes storing appearance and geometry data directly descend from the `Separator` node. We translate the appearance data before the geometry data. This ensures that the material and texture is applied to the current 3D object and not to the following one(s).

The transformation of a *Java3D* `Material` into a *VRML* `Material` is straightforward, since the four material colours are identical. The only difficulty is the translation of the shininess. Since the range of this attribute runs from 1 to 128 (integer values) in *Java3D* while it ranges from 0.0 to 1.0 in *VRML*, it must be mapped accordingly. Finally the `transparency` attribute of a *VRML* `Material` is assigned the transparency value of the corresponding `TransparencyAttribute` node.

The `Texture2` nodes and the `ExportableTexture` objects correspond in a straightforward way. Both nodes contain only a reference to an image file.

Concerning the geometry data, we already mentioned above that *Collage3D* uses `TriangleArrays` and `TriangleStripArrays`. Both of them store the data of the vertices of a 3D object. A vertex consists of a coordinate, a normal, and a texture coordinate [3]. These are stored in three separate arrays; one array for the coordinates of all vertices, one array for

[3] In the terminology of *Java3D* a coordinate as well as a texture coordinate is a tuple whose components represent the coordinates of a point. Thus the term

the normals of all vertices and so on. The coordinate, normal and texture coordinate of one specific vertex can be found at the same index in the respective array.

The way how the vertices define the polygons that finally make up the 3D object is given by the type of `GeometryArray`. The `TriangleArray` draws the array of vertices as individual triangles. Each group of three vertices defines a triangle. The `TriangleStripArray` draws triangles that share an edge, i.e. each triangle re-uses the two last vertices of the previous triangle. *VRML* V1.0c does not offer the possibility to store coordinates, normals, and texture coordinates within one node. It has a node type for each of them: `Coordinate3`, `Normal`, and `TexCoordinate2`. The way how polygonal 3D objects in *VRML* are constructed from the data of the vertices is — unlike in *Java3D* — not inherently given by the type of a node. It must be explicitly defined with an `IndexedFaceSet` node. The `coordIndex` array of an `IndexedFaceSet` allows to define a sequence of polygons by using the indices of the coordinates of the currently used `Coordinate3` node. The polygons are delimited by a -1 (which is obviously an invalid value for an index and thus cannot lead to ambiguous interpretations). Thus `IndexedFaceSet` allows the description of heterogeneous 3D objects as well as homogeneous 3D objects which are composed entirely of one type of polygon, e.g. triangles. It is quite obvious that, concerning the coordinates, every `TriangleArray` can be easily mapped to a `Coordinate3` node and an `IndexedFaceSet` node. `IndexedFaceSets` allow only individual polygons and thus do not allow a direct mapping of `TriangleStripArrays`. Since a `TriangleStripArray` only offers a more compact representation than a `TriangleArray` and no additional modelling power, it is possible to transform every `TriangleStripArray` into a `TriangleArray` and thus to provide a mapping to *VRML* V1.0c. So far we have considered the mapping of the coordinates of the vertices. *VRML* V1.0c provides analogous mechanisms for the normals and texture coordinates. Since *Java3D* ensures that coordinates, normals, and texture coordinates of the same vertex have the same index in the respective array and since these arrays are identically mapped to the according *VRML* node type (`Coordinate3`, `Normal`, and `TexCoordinate2`), the resulting `normalIndex` and `textureCoordIndex` arrays will equal the `coordIndex`. *VRML* allows to omit the `normalIndex` and `texCoordIndex` attributes of an `IndexedFaceSet` and in this case uses the data of the `coordIndex` attribute for this purpose. Summarizing the translation of the geometry data from *Java3D* to *VRML* V1.0c is done by a direct mapping of the vertex data, and the reconstruction of the polygons from the vertices by means of an `IndexedFaceSet` node.

Terminal state:

'coordinate' describes a point rather than a coordinate in the original sense of the word.

The graph contains only *VRML* constructs, i.e. all labels and marks (unless invisible) are written in italic letters. This is equivalent to a state where none of the above rules can be applied.

3.3 Sample Graph Transformation

In this section we will illustrate the functionality of the transformation unit described above by means of an example.

Our 3D scene consists of two transformed instances of one and the same 3D object. Figure 8 shows the scene graph in its initial state.

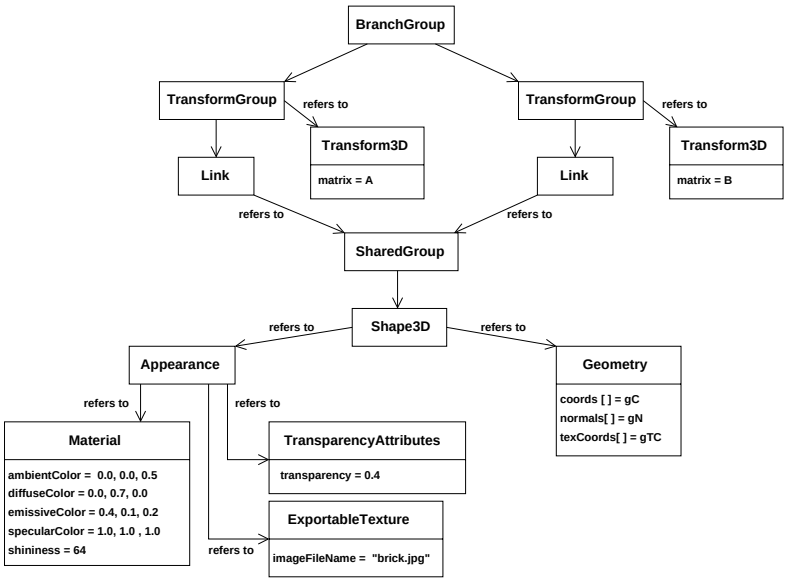

Figure 8. Initial scene graph before the application of the rules

When starting to apply the rules one does not necessarily need to start with the root node of the scene graph and then work downward. To demonstrate this the example starts with the application of rule (2) translating the "left" `TransformGroup` and the appendant transformation. Note that the `MatrixTransform` resulting from the `Transform3D` node is the left outermost child in the graphical representation (Figure 9) and thus the highest-ordered child of the `Separator` node.

In the next step we apply rule (3), replacing the `SharedGroup` with a named `Separator`. There are two possibilities to find an image of the left side of the rule in the graph. We can either use the subgraph containing the `Link` node which is a child of the `Separator`, its parent and the `SharedGroup`

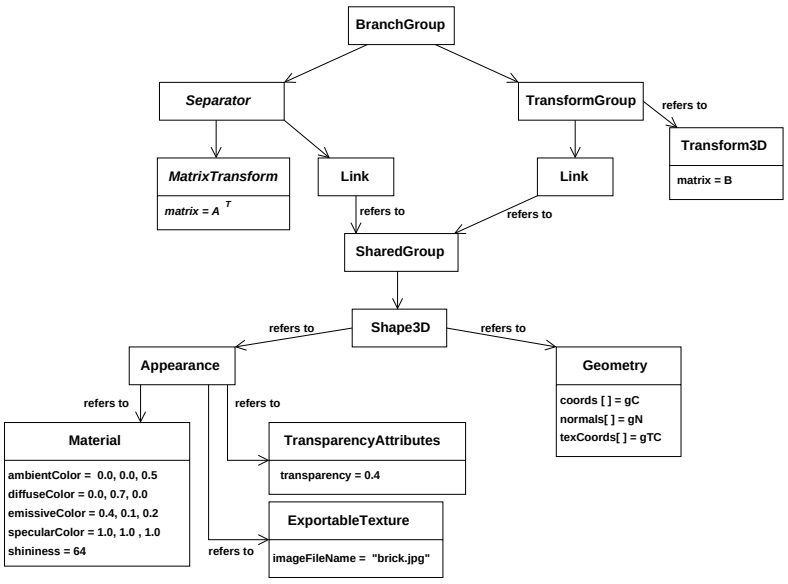

Figure 9. After translation step 1: application of the rule (2)

it refers to, or we can use the subgraph containing the other `Link` which is a child of the `TransformGroup`, its parent, and the `SharedGroup`. In this case we choose the latter of the two options (though neither offers any advantage). First we remove the `Link` node and its edges, then we insert a new edge connecting the `TransformGroup` vertice with the `SharedGroup` vertice and finally replace the `SharedGroup` with a named `Separator` (speaking in graph transformation terms we actually do not replace the vertice but change its labelling). We choose "group1" as the identifier of this `Separator`. Figure 10 shows the scene graph resulting from the application of rule (3).

In step number three (Figure 11 shows the result) we will deal with the remaining `Link` node. Since the `Link` refers to an already translated node, we must apply rule (4) in order to use the instantiation mechanism of *VRML*.

Translation step number four, whose result is shown in Figure 12, demonstrates the application of rule (7) — the translation of a 3D object. The bundling `Shape3D` node is translated into a `Separator`. The `Appearance` node and the three nodes it refers to are replaced by a `Material` and a `Texture2` node who directly descend from the `Separator`. Note that the new `Material` node has a real shininess value between 0.0 and 1.0 rather than an integer value between 1 and 128, since it is a *VRML* node rather than a *Java3D* node. It furthermore contains the transparency value which was formerly stored in a separate node.

The `Geometry` node is replaced by a set of nodes: a `Coordinate3` node, a `Normal` node, and a `TexCoordinate2` node which store the values of the

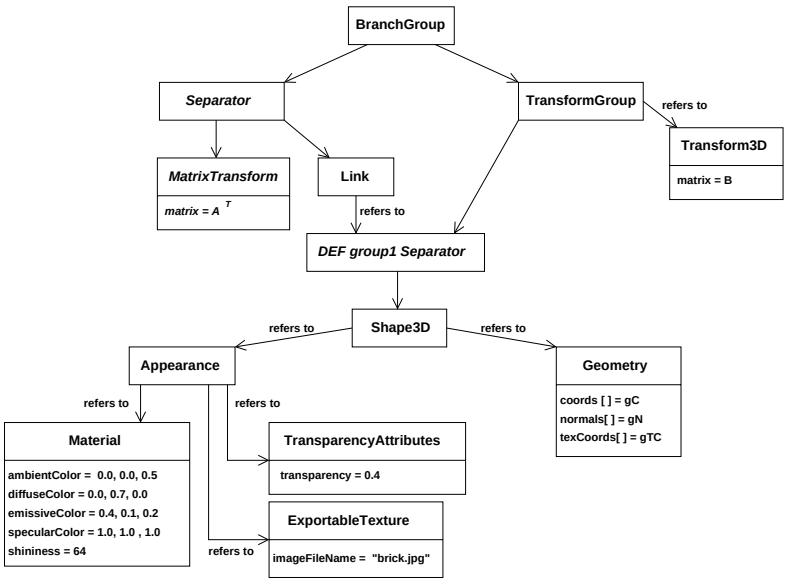

Figure 10. After translation step 2: application of the rule (3)

attributes of the `Geometry` node. Finally we add an `IndexedFaceSet` node that holds the information on how the coordinates make up polygons. These informations are inherently given by the type of the `Geometry` node. Supposed the `Geometry` node in our example is a `TriangleArray` with n vertices, the `coordIndex` array of the `IndexedFaceSet` will be an array containing the ascending numbers from 0 to n with a -1 delimiter inserted after every three numbers: *[0, 1, 2, -1, 3, 4, 5, -1, ..., n]*.

The two final translation steps necessary to reach a terminal state comprise the translation of the `BranchGroup` and the remaining `TransformGroup` and its appendant transformation. The first one is achieved by the application of rule (1) and the latter by the application of rule (2) as shown above. After these two rule applications the graph (Figure 13) contains only *VRML* nodes and no further rules can be applied.

4 Translation of *Java3D* Scene Graphs into *VRML97* Transformation Hierarchies

4.1 The Differences Between *VRML* V1.0c and *VRML97*

VRML V1.0c provides the functionality to describe static scenes. *VRML97* extends this functionality by interaction and animation. A *VRML97* scene graph keeps static data and interaction data conceptually separated. The Route Graph of the scene graph stores the interaction data. It is a set of

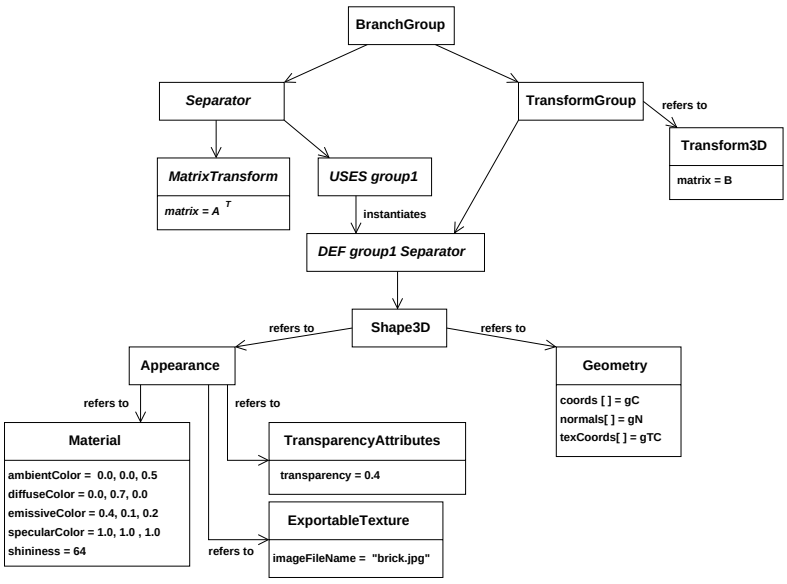

Figure 11. After translation step 3: application of the rule (4)

routes. A route is defined as a "...connection between a node generating an event and a node receiving the event." (Carey, Bell, and Marrin [CBM97] and Carey, Bell [CB97]). The Transformation Hierarchy is a graph that stores the static data of the scene. It is the counterpart of a whole *VRML* V1.0c scene graph. Since *Collage3D* can produce only static scenes, we will focus solely on the translation from *Java3D* into a *VRML97* scene graph which consists only of a transformation hierarchy but no route graph.

4.2 Transformation

The translation of a *Java3D* scene graph into a *VRML97* transformation hierarchy will be, again, specified with graph transformation rules based on the same assumptions made in Section 3.2. The fact that *VRML97* is similar to *VRML* V1.0c induces that the rules translating into either of these languages will be quite similar, too. For this reason, only the differences between the rules of this transformation rule set and the previously described will be explained in the following.

Graph Transformation Rules: *Java3D* to *VRML97*

Initial state:

All graphs are entirely composed of vertices representing instances of the following classes:

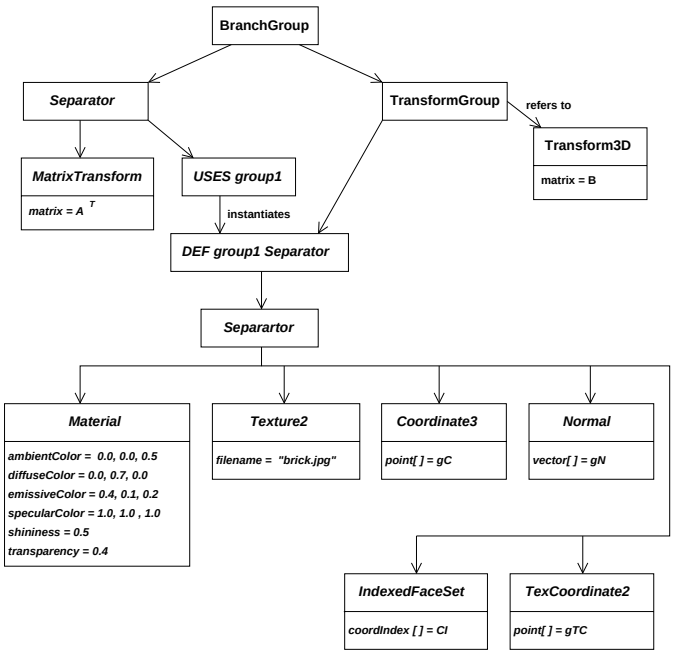

Figure 12. After translation step 4: application of the rule (7)

`BranchGroup`, `TransformGroup`, `Transform3D`, `Link`, `SharedGroup`, `DirectionalLight`, `PointLight`, `Shape3D`, `Appearance`, `Material`, `TransparencyAttributes`, `ExportableTexture`, `GeometryArray`.
All graphs are trees except for edges connecting vertices labelled with 'Link' and 'SharedGroup' and the root is labelled 'BranchGroup'. (This is identical to the initial conditions described in Section 3.2.)

Rules:

Unlike *VRML* V1.0c, which only provides the `Separator` node that allows us to emulate groups, *VRML97* provides "true" grouping nodes that do not require to take care of the order of the child nodes. The most basic form of this grouping nodes is the `Group` node, the counterpart of the `BranchGroup` of *Java3D*. Therefore a `BranchGroup` is mapped to a `Group` (Figure 14).

VRML97 does not support transformation matrices, instead basic transformations must be used to manipulate 3D objects. Thus the translation of transformations from *Java3D* to *VRML97* is done in two steps. First, the matrix of the `Transform3D` is decomposed into basic transformations as introduced in Chapter 3 which are then used to create a *VRML97* construct equivalent to the `TransformGroup` and its appendent `Transform3D`.

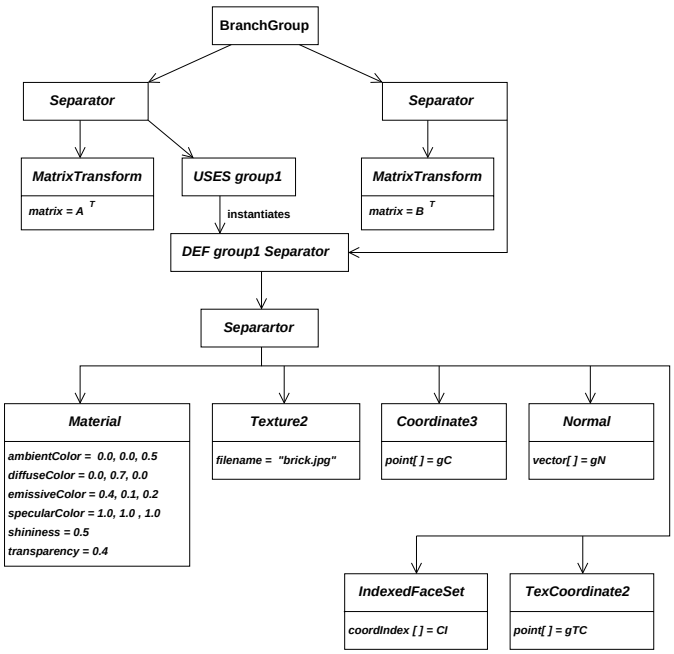

Figure 13. After translation steps 5 and 6: application of the rules (1) and (2)

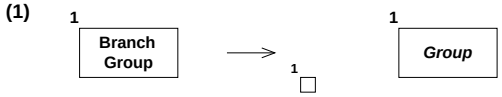

Figure 14. *Java3D* to *VRML97* — rule (1)

The decomposition of the matrix M yields a scale, a rotation, two different shear, and a translation transformation. Each shear transformation is actually resolved into an equivalent sequence of intermediate transformations: a rotation, a scale, and a rotation reverting the previous one. If S (scale), R (rotation), $H1$ (shearing 1), $H2$ (shearing 2), and T (translation) are matrix representations of these basic transformations yielded by the decomposition, then the original matrix is obtained by the composition

$$M = S \circ R \circ H1 \circ H2 \circ T.$$

The *VRML97* node type correspondent to the `TransformGroup` of *Java3D* is the `Transform` node, which has five fields that define a transformation: `translation`, `rotation`, `scale`, `scaleOrientation` and `center`. While the first three are rather self-descriptive the latter two need some

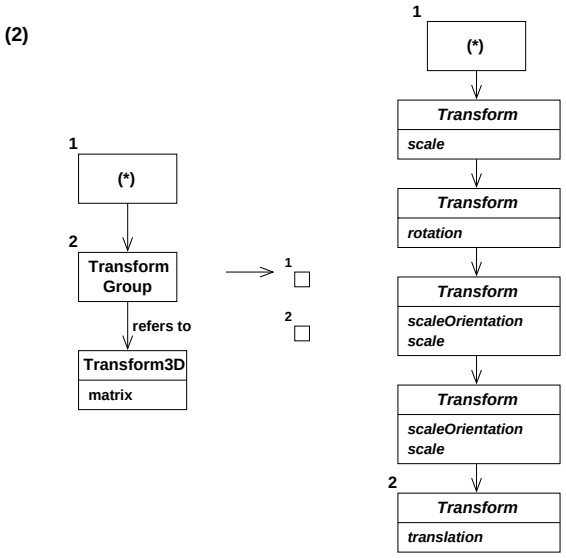

Figure 15. *Java3D* to *VRML97* — rule (2)

explanation. The `scaleOrientation` defines a rotation of the coordinate system just before applying the scalation. This rotation is reverted just after the scalation. Thus `scaleOrientation` allows to specify scaling in arbitrary orientations and this way it is possible to shear 3D objects. The `center` field allows to define a temporary translation offset from origin of the local coordinate system which influences the application of the `rotation`, `scale`, and the `scaleOrientation`. Given a `Transform` node and a 3-dimensional point P, the point P is transformed into P' by a sequence of intermediate transformations. If C (`center`), SR (`scaleOrientation`), T (`translation`), R (`rotation`), and S (`scale`) are transformation matrices equivalent to the corresponding fields of the `Transform` node, the transformation is defined by

$$P' = T \circ C \circ R \circ SR \circ S \circ -SR \circ -C \circ P.$$

Thus a fixed order of the basic transformation is defined which is clearly different from the order of transformations yielded by the matrix decomposition. Since the `Transform` node is a group node another order of the basic transformations can be achieved by using a nested sequence of `Transform` nodes where each one utilizes only a subset of its transformation fields as shown in Figure 15. The rule shown in this figure is actually a set of rules whereas the vertice labelled with the wildcard '(*)' represents any possible group node of *Java3D* (`BranchGroup`, `TransformGroup`, `SharedGroup`) or one of the two group nodes of *VRML97* (`Group`, `Transform`) used for the translation.

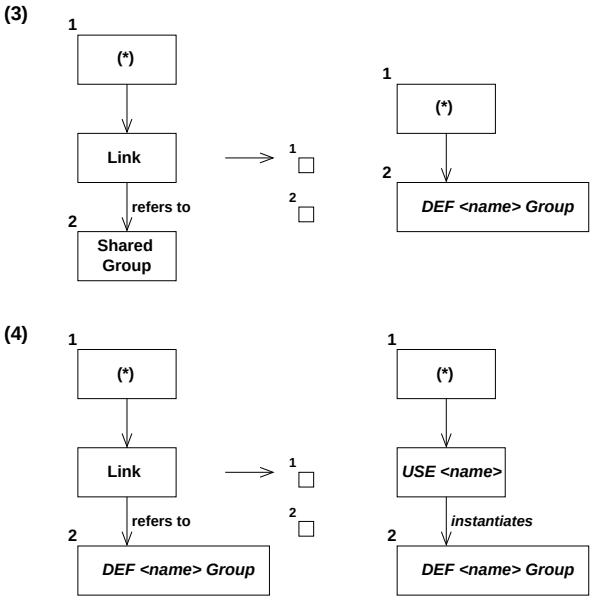

Figure 16. *Java3D* to *VRML97* — rule (3) and rule (4)

The re-use or so-called 'instantiation' of nodes works just the same way as in version 1.0c of *VRML*. Thus the rules (3) and (4) shown in Figure 16 are identical to those of the transformation unit translating into *VRML* V1.0c with the exception that the named node is a `Group` node and not a `Separator` node.

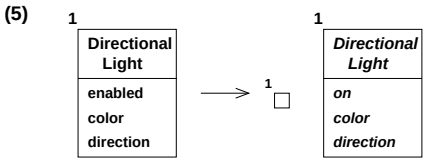

Figure 17. *Java3D* to *VRML97* — rule (5)

The translation of directional light nodes (Figure 17) from *Java3D* to *VRML* is identical for both versions of *VRML*.

With the enrichment of the lighting model of *VRML* by attenuation of light in the second version it became possible to translate this already offered feature of *Java3D* as shown in Figure 18.

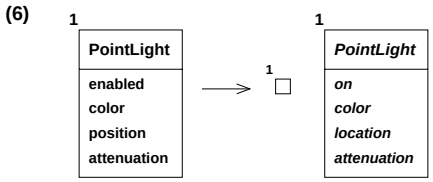

Figure 18. *Java3D* to *VRML97* — rule (6)

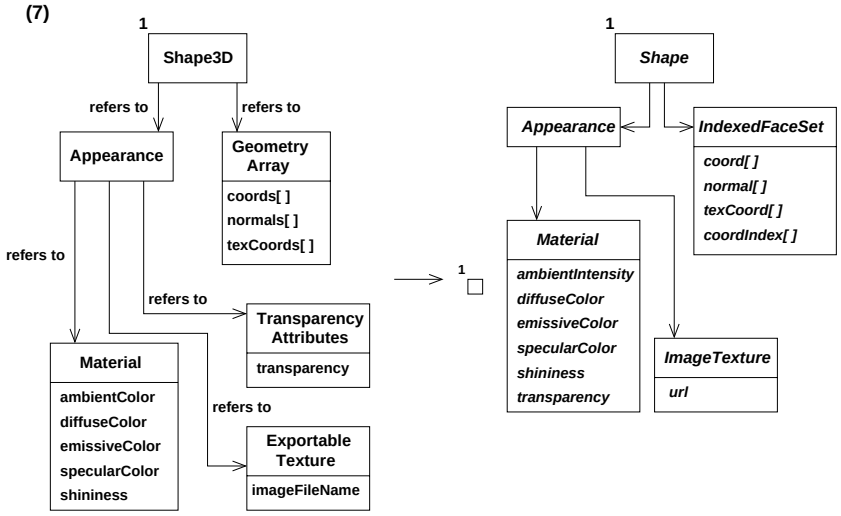

Figure 19. *Java3D* to *VRML97* — rule (7)

VRML97 provides several node types that allow a more bundled storage of data of 3D objects. Thus the rule translating 3D objects from *Java3D* to *VRML97* (Figure 19) differs from the corresponding rule described in Section 3.2. The new node types include the `Shape` node which is a more appropriate counterpart for the `Shape3D` node of *Java3D* than the `Separator` we used in *VRML* V1.0c. Furthermore, the `Appearance` node is used to bundle the appearance data. The `Texture2` of version 1.0c is replaced by the `ImageTexture` node which is used in an identical way. The `Material` has remained identical since version 1.0c with the exception that the `ambientColor` attribute has been replaced by the `ambientIntensity` attribute, which has only a single real value in contradiction to the rgb value of its predecessor. In *VRML97* the ambient light colour is calculated as *ambientIntensity* × *diffuseColor*. The ambient intensity can be approximated by dividing the ambientColor of *Java3D* by the diffuse colour component-wise and calculate the average of the

components of resulting quotient. This is of course not very precisely, but *VRML97* understands the ambient colour of a 3D object as a fraction of its diffuse colour, while *Java3D* and *VRML* V1.0c regard ambient colour as a colour that is more or less independant of the diffuse colour. The several components of the geometry data — coordinates, normals, texture coordinates and the coordinate indices — are no longer separately stored but within an `IndexedFaceSet` node which thus became more similiar to the `Geometry` node of *Java3D* than its previous version. Apart from the structural changes described above, this rule is identical to the corresponding one of the transformation unit described before.

Terminal state:

The graph contains only *VRML* constructs. This is equivalent to a state where none of the above rules can be applied.

5 Implementation Notes

The transformation presented in Section 3.2 can be easily implemented if one abandons the option to translate the nodes of the scene graph randomly but in depth first order starting with the root node.
The resulting *VRML* files work fine with many *VRML* viewers and 3D design tools. The only known exception so far is *Cinema4D* from Maxon. This 3D design tool has problems with named `Separators`. As far as observable the scalation informations of the transformation matrices are not correctly interpreted when the attempt is made to name `Separators` for further use. Unfortunately *Cinema4D* was the tool mainly used for further processing of 3D data produced with *Collage3D*. Thus a workaround for this problem became necessary.

To understand the workaround we have take a closer look at the *Java3D* scene graphs in *Collage3D*. Generally every 3D object is intended to be reusable. This means that we have a `SharedGroup` for each `Shape3D` which contains nothing else than the `Shape3D`. This is also the only way that `SharedGroups` are used by *Collage3D*. Furthermore, the initial location of each 3D object is the origin of the scene. This means that a transformation for every 3D object is required. This results in the following structure for each 3D object: a `TransformGroup` referring to a transformation, and parent of a `SharedGroup` which contains nothing but a `Shape3D`. These restrictions or properties of the scene graph were not mentioned in Section 2 because they are irrelevant for a translation under normal circumstances. However, for the workaround these restrictions are essential.

When translating the scene graph to *VRML* V1.0c we do not name the `Separators` that represent the `SharedGroups` but instead name the `IndexedFaceSet` nodes within these `Separators`. Whenever we intend to instantiate a SharedGroup-`Separator` we instantiate the corresponding `IndexedFaceSet` instead. This means that only the data of the named

`IndexedFaceSet` node is re-used and while the remaining data of the SharedGroup-`Separator` would be replaced by the most recently defined nodes of the according node types within the scene graph, i.e. concerning a 3D object the most recently defined `Material`, `Texture2`, `Coordinate3`, `Normal`, and `TexCoordinate2` nodes within the current `Separator`. Actually *Cinema4D* does re-use *all* the nodes of SharedGroup-`Separator` as intended, although only the `IndexedFaceSet` node is instantiated. Besides the fact that this workaround allows the correct re-use of subgraphs it also ensures the correct application of transformations. However, this works only if the initial scene graph only contains `SharedGroups` that consist of a single leaf node.

Of course the implementation allows to export scene graphs to *VRML* V1.0c as described above as well as an export customized for *Cinema4D*.

Another problem that frequently occurred during tests is one of a computer-numeric character. The `double` data type of *Java*, which is used for representing the normals of a 3D object, provides 64 bits for the representation of real numbers. This limitation means that not all real numbers can be represented with this data type. Sometimes, however, normals can be such numbers that are not representable. *Java* then uses a `NaN` (Not-a-Number). Actually *Java* can even make calculations with this `NaN`[4] but *VRML* cannot. *VRML* does not even have a corresponding concept. For this reason the implementation, that merely writes string representations of the `double` data type into a file would produce an incorrect *VRML* file whenever a `NaN` occurs. In order to solve this problem the export of normals is optional.

5.1 Application Programming Interface (API)

Class

`public` **VRML10cSaver**

The class `VRML10cSaver` provides the possibility to generate a *VRML* V1.0c file from a given *Java3D* `BranchGroup` object.

Constructors

`public` **VRML10cSaver**`(String fileName, BranchGroup branchgroup)`

`String fileName`	Name of the *VRML* file to be written.
`BranchGroup branchgroup`	The branchgroup to be translated to *VRML*.

[4] Whether the resulting values of such a calculation make sense or not is another question that is not discussed here.

Creates a new `VRML10cSaver` instance. It sets the source (branchgroup) and the destination (fileName) of the export.

`public` **VRML10cSaver**`(String fileName, BranchGroup branchgroup, double scaleFactor, boolean exportForC4D, boolean exportNormals)`

`String fileName`	Name of the *VRML* file to be written.
`BranchGroup branchgroup`	The branchgroup to be translated to *VRML*.
`double scaleFactor`	Allows a resizing of the exported scene.
`boolean exportForC4D`	Sets a flag to indicate whether the export is made for *Cinema4D* or not.
`boolean exportNormals`	Sets a flag that indicates whether normals should be exported or not.

Creates a new `VRML10cSaver` instance. It sets the source (branchgroup) and the destination (fileName) of the export. Furthermore, the scale of the exported scene is set as well as flags that indicate whether normals should be exported or not, and whether the export should be made for *Cinema4D*.

Methods

`public void` **openNewFile**`(String Filename)`

`String fileName`	Name of the new file

`openNewFile` Opens a new file with the specified name which will be the destination of the next export.

`public void` **setBranchGroup**`(BranchGroup branchgroup)`

`BranchGroup branchgroup`	Reference to the `BranchGroup`.

`setBranchGroup` Sets the `BranchGroup` object which will be the source of the next export.

`public void` **setScaleFactor**`(double scaleFactor)`

`double scaleFactor`

`setScaleFactor` Sets the factor which will be used for resizing the exported scene.

`public void` **setExportForC4D**`(boolean flag)`

`boolean flag`

`setExportForC4D` Sets a flag to the specified value which states whether the *VRML* export is for *Cinema4D* or not.

`public void` **setExportNormals**`(boolean flag)`

`boolean flag`

`setExportNormals` Sets a flag to the specified value which determines whether normals will be exported or not.

`public void` **export**`()`

`export` Generates a *VRML* V1.0c file from the given `BranchGroup`.

18 Turmite Object Generator

Antonio Krampe

1 Introduction

This is the handbook of the Turmite Object Generator. Based on the concept of the Turing machine, the object generator is used to simulate the behaviour of so-called turmites. A Turing machine consists of a tape of cells each of which contains a symbol of an alphabet, a pointer to the read/write position of the Turing machine which points to one of these cells and can be moved to one of its neighbour cells, and a control unit which is in one of a finite number of states and decides whether the content of the actual cell is to be changed and the pointer moved.

A Turing machine is a one-dimensional model; for turmites it is generalized to a three-dimensional version where the tape of the Turing machine is replaced by a cuboid world which is divided into cells. These cells can be modified by finitely many turmites, which are pointers to a cell of the defined world. Each of these turmites has its own tables where its behaviour is defined. Thus the tables are the replacement of the control unit from the Turing machine.

One can define a world with walls and turmites. A world has a width, a height and a depth, giving rise to an array of cells where the turmites can exist. Each of these cells has the property *colour* which represents one of the items from the alphabet of the world. The colour can be changed by a turmite of a certain type, implying a change in the behaviour of the other turmites. Each turmite has tables in which its behaviour is defined. The path which a turmite follows depends on the entry in their tables and on the colour of the cells. There are four different types of turmites and four different types of walls, each with different functionality. The walls are used to hinder a turmite from following its way; one can define a wall which blocks or even kills a turmite on contact.

In a world one can define arbitrarily many turmites and walls. After starting the object generator and connecting it to the Viewer, one sees a chaotic but well defined scene.

2 Overview

The document describes the functionality of the object generator and what one can do with this tool. It discusses which functionality has been implemented, and which extended concepts are planned but not implemented yet.

2.1 Implemented Functionality

At first there is the world in which the turmites are living. This world is a cuboid with a width, height and depth. At the border of a world one can say whether a turmite should be killed, blocked, reflected or transferred to the other side of the world.

Four types of turmites are implemented in this object generator (queen, worker, killer, drone). Each of these four different types has its own function in the life of the turmites. A queen generates new turmites dynamically, a worker changes the colour of the cells, a killer kills existing turmites, and a drone helps a queen with the birth of new turmites.

Four types of walls are implemented to hinder the turmites walking through the world. One blocks a turmite, one reflects a turmite, one transfers a turmite to the other side, and one kills a turmite. Each turmite has a set of tables. Two turmites can have the same tables. In these tables the behaviour of the turmite is exactly described. There one can define a lot of properties which change the behaviour of the turmite which uses this table.

2.2 Projected Extensions

The appearance of the world need not look like a cuboid, but may resemble every object one can imagine. If you want a world that looks like an elephant, then you can define such a world. To implement this feature the time of the project was too short.

Another feature which is not implemented yet is the possibility to say that the turmites are living on rather than in the world. In particular, the idea is to generate a world with another object generator and use this one for the turmites which can live on this object. An extending idea is that then the turmites can make holes in this object like real termites.

3 Turmite Module

3.1 Requirements

The Turmite Object Generator runs on a Linux system. The ONI libraries are needed from AnimaLab in order to connect the object generator to AnimaLab. If one only wants to generate an output file, then these libraries are not needed. The Viewer can show the output as an animation, and the Arranger allows to arrange a scene with different object generators.

3.2 Installation

The Turmite Object Generator is installed by copying the executable of the object generator in a directory of your choice.

3.3 Starting

To start the Turmite Object Generator write:

```
programname [options] <scenefile>
```

These options are used for a connection with ONI:

- **-p** [*portnumber*] – This is the number of the port which is used for the connection to the other components of AnimaLab. This port has to be the same for each of the components.
- **-h** [*hostname*] – The hostname states where the object generator has started. Use $LOCALHOST for the local system.
- **-e** – Starts the editor of the Turmite Object Generator. This option is not implemented yet.

These options are used to generate an output file with the object generator:

- **-f** [*outputfile*] – This is the option to generate only an output file. It must be set to generate an output file. The parameter *outputfile* includes the path of the output file.
- **-o** [*every derivation step onefile*] – This option describes whether the object generator should write an output file after each derivation step (parameter = 0) or only after the last one (parameter = 1).
- **-a** [*quantity of derivation steps*] – This option informs the program how many derivation steps it has to do before the output file will be written.
- *scenefile* The scene file contains a description of the world with all its turmites, tables and walls. This file can be generated manually or with the help of the editor.

3.4 Examples

Here are some examples how to start the Turmite Object Generator.

- Start the object generator with a connection to the other components of AnimaLab:

```
programname -p 5000 -h \$LOCALHOST ./scenes/test.tur

programname -h MyComputer -p 3200 ./scenes/myscene.tur
```

- Start the object generator and generate an output file after the last derivation:

```
programname -f ./output/myoutputfile.txt -o 1  -a 2000
                  ./scenes/myscene.tur
```

- Start the object generator and generate an output file after each derivation:

```
programname -f  ./output/o.txt  -a 1200 -o 0
                  ./scenes/test.tur
```

- Start the editor:

```
programname -e
```

4 Scene File

In the following the structure of the scene file is described. This structure is well defined and you have to generate only scene files such a structure. The editor generates it automatically. It is important that each property stands alone in one line in the scene file. The keywords of the properties are in German.

4.1 Comments

A comment may be written anywhere in the scene file:

- # **This is a comment.**

4.2 World Properties

The world properties begin with the tag <**WELT**> and end with </**WELT**>.

<WELTGROESSE> width height depth
: This property defines the size of the world. **width**, **height** and **depth** have to be numbers greater than 0.

<RANDVERHALTEN> no
: This property defines the behaviour of a turmite reaching the border of the world. The parameter **no** may be 1, 2, 3 or 4, with the following meaning.
  1. The turmite is moved to the other side of the world. The direction of the turmite is not changed.
  2. The turmite is not moved to the next cell, and the direction is not changed. It remains in the current cell. The colour of the cell is changed so that the turmite can later move in a direction other than towards the wall.

3 The turmite is not moved to the next cell, but the direction is changed to the opposite direction. It remains in the current cell. The colour of the cell is changed so that the turmite can later move in a direction other than towards the wall.
4 The turmite is killed.

<AENDERUNGSART> no
This property is not implemented yet and the parameter should always be 3.

4.3 Turmite Properties

The turmite properties start with the tag **<TURMITE>** and end with **</TURMITE>**.

<ID> no
This identifies an individual turmite. Each turmite has to have an unambiguous identity given by a number.

<TYP> no
This is the type of the turmite. There are four different types with individual functionality. The parameter **no** may be 1, 2, 3 or 4, with the following meaning.
1 This defines the type *queen*. A queen has some extra properties (see below) and generates new turmites dynamically while the object generator is running.
2 This defines the type *worker*. A worker can change the colour of the cells of the world.
3 This defines the type *killer*. A killer has some extra properties (see below) and kills other turmites.
4 This defines the type drone. A drone has only the functionality to help the queen generate new turmites.

<POSITION> x y z
This property defines the start position of the turmite in the world. The parameters **x**, **y** and **z** should be within the border of the world.

<GROESSE> width height depth
This property defines the size of the turmite but is not implemented yet.

<RICHTUNG> no
This property defines the start direction of the turmite. Six possibilities exist for the parameter **no**:
1 Left
2 Right
3 Up

4 Down
5 Front
6 Back

<TEXTUR> file
This defines the texture of the turmite. The parameter **file** includes the path to the texture. The format of the texture file has to be one of the texture formats the Viewer supports.

<TABELLE> id
This property defines the start table. The parameter **id** is an identifier of a defined table.

<DARSTELLUNGSFARBE> r g b a
This property defines the colour of the turmite and is not implemented yet. The parameters **r**, **g**, **b**, and **a** define the red, green, blue, and alpha values of the colour, respectively. The parameters **r**, **g**, **b** are values between 0 and 255, and the parameter **a** is a value between 0 and 1 where 1 means full colour and 0 means totally transparent.

<LEBENSLAENGETYP> no
This property defines the lifespan of the turmite. The parameter **no** has one of seven values:
1 The lifespan depends on the number of derivation steps.
2 The lifespan depends on the time.
3 The lifespan depends on how many steps a turmite has done.
4 The lifespan depends on a colour. If the turmite moves to a cell with this colour, the turmite will be killed.
5 The lifespan depends on how many other turmites have been killed by this turmite. This one is only valid for turmites of the type killer.
6 The lifespan depends on how many new turmites have been generated by this turmite. This one is only valid for turmites of the type queen.
7 The lifespan is infinite.

<LEBENSLAENGE> value
This property depends on the property **<LEBENSLAENGETYP>** and defines how long the turmite lives, or the colour.

<ZEITINTERVALL> value (queen only)
This property defines after how many derivations the turmite generates new turmites. Setting **value** to -1 means that the turmite generates new turmites whenever it meets another turmite of the type drone.

<ANZAHL> from to (queen only)
This property defines how many new turmites are generated at the same time. The parameter **from** defines the minimal and the parameter **to** the maximal amount of newly generated turmites.

<NEUETURMITE> id (queen only)
This property defines the start properties of the newly generated turmites. The parameter **id** refers to an existing turmite that passes its properties on to the new turmite.

<ZUFALL> no (queen only)
This property specifies whether the properties of the new turmites will be defined depending on the property **<NEUETURMITE>**, or whether the new turmites get randomly the foundation of their properties. One has the following possibilities to instantiate **no**:

1 The queen chooses randomly an existing turmite whose properties are passed on to the new turmites.
2 The queen uses the turmite specified with **<NEUETURMITE>** to generate the new turmites.

<ENTFERNUNGSTYP> no (queen only)
This property defines how far the newly generated turmites are away from the queen. There are the following possibilities to actualize **no**:

1 The new turmites are randomly far away from the queen.
2 The new turmites are generated somewhere in a defined area around the queen.
3 The new turmites are generated at a fixed distance to the queen.

<ENTFERNUNG> width height depth (queen only)
If **<ENTFERNUNGSTYP>** is not set to 1, then this property defines the area around the queen or the relative position to the queen where the new turmites will be generated.

<KILLINGTYP> no (killer only)
This defines the type of turmites which the killer should kill. One has the following possibilities to instantiate **no**:

1 The killer kills turmites of the type queen.
2 The killer kills turmites of the type worker.
3 The killer kills other turmites of the type killer.
4 The killer kills turmites of the type drone.

<KILLINGBEREICH> left down front right up back (killer only)
This property defines the area in which the killer eliminates its victim.

4.4 Table Properties

The table properties begin with the tag **<TABELLE>** and end with **</TABELLE>**.

<ID> id
This identifies an individual table. Each table has to have an unambiguous identity given by **id**.

<TABELLENEINTRAG>
A finite number of table entry properties (see below).

4.5 Table Entry Properties

The table entry properties begin with the tag <**TABELLENEINTRAG**> and end with </**TABELLENEINTRAG**>.

<ALTEFARBE> r g b a
This property defines the colour of the cell when a turmite steps into it. The parameters **r**, **g**, **b**, and **a** define the red, green, blue, and alpha values of the colour, respectively. The parameters **r**, **g**, **b** are values between 0 and 255, and the parameter **a** is a value between 0 and 1 where 1 means full colour and 0 means totally transparent.

<VERAENDERUNGSTYP> no
This property defines the change when the turmite steps into a cell with the colour defined in <**ALTEFARBE**>. There are the following posibilities to instantiate **no**:

1 The colour of the cell is changed.
2 The colour of the turmite is changed.
3 This is reserved for metamorphosis (but not implemented yet).
4 The size of the turmite is changed.
5 The type of the turmite is changed (not implemented).
6 The turmite is killed.
7 The turmite is moved to an absolute position.
8 The turmite is moved relative to its position.

<VERAENDERUNG> value
This property depends on the property <**VERAENDERUNGSTYP**> and specifies the new colour (**value** = r g b a), or the new position (**value** = x y z), or the distance to the current position (**value** = dx dy dz), or the new size (**value** = width height depth).

<SCHRITTWEITE> dx dy dz
This property defines how many steps the turmite will be moved in one derivation. The parameters **dx**, **dy** and **dz** are relative values to the actual position of the turmite.

<NAECHSTETABELLEID> id
This property defines the table which the turmite uses for its next derivation. The parameter **id** has to be the valid identification of an existing table.

<RICHTUNG> no
This property defines the direction in which the turmite will be turned. One has the following possibilities to actualize **no**:

1 Left
2 Right
3 Up
4 Down
5 Front
6 Back

4.6 Wall Properties

The wall properties begin with the tag <**WALL**> and end with </**WALL**>.

<ID> id
This identifies an individual wall. Each wall must have an unambiguous identity given by **id**.

<WANDTYP> no
This defines the type of the wall. There are the following possibilities to instantiate **no**:

1 Cuboid
2 Object (not implemented)
3 Sphere (not implemented)

<OBJEKT> value
This specifies the size of the wall depending on the property <**WANDTYP**>. If it is a cuboid, then **value** = width height depth; if it is a sphere, then **value** = r (radius of the sphere). If it is an object, then **value** depends on the object.

<POSITION> x y z
This property defines the left upper front corner of the wall.

<FARBE> r g b a
This defines the colour of the wall. The parameters **r**, **g**, **b** and **a** define the red, green, blue and alpha value of the colour, respectively. The parameters **r**, **g**, **b** are values between 0 and 255 and the parameter **a** is a value between 0 and 1 where 1 means full colour and 0 totally transparent.

<TEXTUR> file
This defines the texture of the wall. The parameter **file** includes the path to the texture. The file format of the texture file has to be one of the texture formats supported by the Viewer.

<RANDVERHALTEN> no
This property defines the behaviour when a turmite connects with the border of the wall. One has the following possibilities to instantiate **no**:

1 The turmite is moved to the other side of the wall. The direction of the turmite is not changed.
2 The turmite is not moved to the next cell, and the direction is not changed. It remains in the current cell. The colour of the cell is changed so that the turmite can later move in a direction other than towards the wall.
3 The turmite is not moved to the next cell, but the direction is changed to the opposite direction. It remains in the current cell. The colour of the cell is changed so that the turmite can later move in a direction other than towards the wall.
4 The turmite is killed.

<SICHTBAR> no
This property defines whether one sees the wall in the Viewer or not. **no** is one of the following possibilities:
1 One can see the wall in the Viewer.
2 One does not see the wall in the Viewer.

5 Example

The following example specifies a world with a width, height and depth of 50. The border of the world reflects a turmite on contact. Moreover, there are four turmites in the world: two workers, a killer, and a queen. These turmites start on different positions in the world. A table with three entries guides the turmites through the world. Four walls, one of each type, hinder the turmites in their movements through the world.

```
#
# Welt
#
<WELT>
# Example
        <WELTGROESSE> 50 50 50
        <RANDVERHALTEN> 3
        <AENDERUNGSART> 3
</WELT>
#
# Turmiten
#
<TURMITE>
# JIM
        <ID> 0
        <TYP> 2
        <POSITION> 20 20 20
        <GROESSE> 1 1 1
        <RICHTUNG> 3
        <TABELLE> 0
        <DARSTELLUNGSFARBE> 100 100 1 1
        <LEBENSLAENGETYP> 7
        <LEBENSLAENGE> 0
</TURMITE>
<TURMITE>
# JOE
        <ID> 1
        <TYP> 2
        <POSITION> 80 12 45
        <GROESSE> 1 1 1
        <RICHTUNG> 6
        <TABELLE> 0
```

```
        <DARSTELLUNGSFARBE> 100 1 100 1
        <LEBENSLAENGETYP> 7
        <LEBENSLAENGE> 0
</TURMITE>
<TURMITE>
# BOB
        <ID> 2
        <TYP> 3
        <POSITION> 50 10 20
        <GROESSE> 1 1 1
        <RICHTUNG> 3
        <TABELLE> 0
        <DARSTELLUNGSFARBE> 1 255 255 1
        <LEBENSLAENGETYP> 7
        <LEBENSLAENGE> 0
        # Killer
        <KILLINGTYP> 1
        <KILLINGBEREICH> 1 1 1 1 1 1
</TURMITE>
<TURMITE>
# Elisabeth
        <ID> 3
        <TYP> 1
        <POSITION> 10 23 76
        <GROESSE> 1 1 1
        <RICHTUNG> 1
        <TABELLE> 0
        <DARSTELLUNGSFARBE> 100 100 100 1
        <LEBENSLAENGETYP> 7
        <LEBENSLAENGE> 0
        # Koenigin
        <ZEITINTERVALL> 2000
        <ANZAHL> 1 2
        <NEUETURMITE> 0
        <ZUFALL> 1
        <ENTFERNUNGSTYP> 1
        <ENTFERNUNG> 0 0 0
</TURMITE>
#
# Tabellen
#
<TABELLE>
# Tab_1
        <ID> 0
        <TABELLENEINTRAG>
                <ALTEFARBE> 255 1 1 1
                <VERAENDERUNGSTYP> 1
                <VERAENDERUNG> 1 255 1 1
```

```
                <RICHTUNG> 1
                <SCHRITTWEITE> 1 1 1
                <NAECHSTETABELLEID> 0
        </TABELLENEINTRAG>
        <TABELLENEINTRAG>
                <ALTEFARBE> 1 255 1 1
                <VERAENDERUNGSTYP> 1
                <VERAENDERUNG> 1 1 255 1
                <RICHTUNG> 4
                <SCHRITTWEITE> 1 1 1
                <NAECHSTETABELLEID> 0
        </TABELLENEINTRAG>
        <TABELLENEINTRAG>
                <ALTEFARBE> 1 1 255 1
                <VERAENDERUNGSTYP> 1
                <VERAENDERUNG> 255 1 1 1
                <RICHTUNG> 5
                <SCHRITTWEITE> 1 1 1
                <NAECHSTETABELLEID> 0
        </TABELLENEINTRAG>
</TABELLE>
#
# Wände
#
<WAND>
# OtherSide
        <WANDTYP> 1
        <POSITION> 30 30 30
        <OBJEKT> 5 5 5
        <RANDVERHALTEN> 1
        <FARBE> 100 100 100 1
        <SICHTBAR> 1
</WAND>
<WAND>
# Blocker
        <WANDTYP> 1
        <POSITION> 70 70 70
        <OBJEKT> 5 5 5
        <RANDVERHALTEN> 3
        <FARBE> 100 100 100 1
        <SICHTBAR> 1
</WAND>
<WAND>
# Reflector
        <WANDTYP> 1
        <POSITION> 90 90 90
        <OBJEKT> 5 5 5
        <RANDVERHALTEN> 2
```

```
        <FARBE> 100 100 100 1
        <SICHTBAR> 1
</WAND>
<WAND>
# KillWall
        <WANDTYP> 1
        <POSITION> 25 25 80
        <OBJEKT> 5 5 5
        <RANDVERHALTEN> 4
        <FARBE> 100 100 100 1
        <SICHTBAR> 1
</WAND>
# ENDE
```

6 Description

6.1 World

The world is a cuboid with a specified width, height and depth. It is divided into cells which are cuboids themselves. Each of these cells has the same width, height and depth as the other ones. A cell has a colour which represents its state. A turmite entering a cell can react to this state. The world has a border which can react with one of four possibilities to entering turmites. One can define that on contact with the border, a turmite will be blocked, reflected, killed, or moved to the other side of the world. Thus, a turmite can never leave the world.

In a world, one can define turmites and walls with different abilities and possibilities. So, the world, may be seen as the container of the turmites and walls.

6.2 Turmite Types

There are various types of turmites which can be defined in a world. A lot of properties are common to every turmite, but some are type-dependent. One can define four types of turmites:

- The *worker*, who changes the state of the cells in the world.
- The *killer*, who kills other turmites.
- The *drone*, who helps the queen to generate new turmites.
- The *queen*, who generates new turmites.

Each of these types has a particular function in the world. The worker is the most known and used type of the turmite because it can change the states of the cells. If it changes such a state, this influences the behaviour of

the other turmites because now they have to react differently when entering this cell.

The killer is very dangerous to the other turmites. It kills them. But not all types of turmites are involved in its killing process. Every killer can only kill one particular type. So a killer who can only kill queens never kills a worker. Moreover, one can also define a killer who kills other killers, like a 'police' turmite who saves the other turmites.

The queen is generating new turmites dynamically. One can define that it only generates a new turmite on entering a cell with a drone, or that it generates new turmites periodically. A queen can generate a predefined amount of new turmites, e.g. two to five new turmites, whenever it generates new turmites.

The only function of a drone is to help the queen generate new turmites. It is only useful if there is a queen who needs a drone to generate new turmites.

6.3 Turmite Tables — How the Turmites React to States of Cells

The tables of a turmite are very important. They allow to define how the turmite reacts to a state on entering a cell. There are different possibilities what might happen. Normally the turmite will change the colour (the state) of the cell. But there are some other possibilities. The turmite may grow, or change colour, move to an absolute or relative position in the world. The turmite might also change its own type, but this is not implemented yet. The cruelest thing one can define is that the turmite is killed on entering a cell with this state. But even this is possible.

6.4 Wall Types

Walls are obstacles with a fixed position in the world of the turmites. There are various types of walls which can be defined in a world. Each of these walls reacts differently to connecting turmites. There are four different types of walls:

- The blocking wall, which hinders the turmite to walk on (see the right illustration in Figure 1).
- The reflecting wall, which turns the turmite to the other direction.
- The other side wall, which moves the turmite to the other side of the wall (see the left illustration in Figure 1).
- The killing wall, which kills the turmite.

6.5 Interaction Between World, Turmites and Walls

The world, turmites and walls interact. The world is the container of the other objects. The walls have a fixed position and the turmites are guided

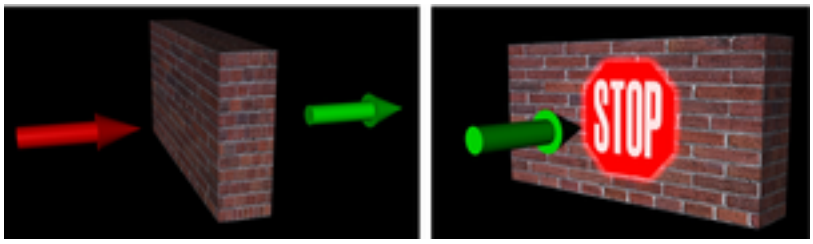

Figure 1. Wall types

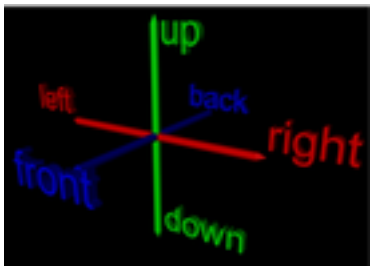

Figure 2. The six directions in which a turmite can move

through tables. A turmite can walk in up to six different directions, which are shown in Figure 2.

The direction in which a turmite moves depends on the state of the cell in which the turmite is and the entry in the current table of the turmite. Entering a new cell can mean changing the world. It might be that a worker enters a cell and changes the state of the cell, or its own colour. There are eight possibilities how the world may be changed when a turmite enters a cell:

- The state of a cell is changed.
- The colour of the entering turmite is changed.
- The entering turmite metamorphoses into another object (not implemented yet).
- The size of the entering turmite is changed (not implemented yet).
- The type of the entering turmite is changed (not implemented yet).
- The entering turmite is killed.
- The entering turmite is moved to an absolute position.
- The entering turmite is translated along a vector to a new position.

Clearly, there are a lot of possibilities to change the world. But how can one say that on a defined state of a cell a certain thing will be changed? The answer is the table in which the behaviour of a turmite is defined. A table has entries, each showing a state of a cell. If there is a state of a cell which is not defined in the table, then the first entry of the table is used by default. Normally one defines an entry for each state which is possible in the world.

In these entries one defines the behaviour of the turmite and the change to the world when it enters a cell.

7 Langton's Ant

Langton's ant is the best-known example of this part of chaos theory. It is set on a two-dimensional world where the ant can move into four directions (north, south, east and west). There is one ant in the world, which looks to the east. In a derivation step, it tests whether the colour of the cell it points to is black. In this case, the ant changes the colour to white and turns 90 degrees to the right. Otherwise, it changes the colour to black and turns 90 degrees to the left. Then the ant goes one step in the new direction to the neighbour cell. In Figure 3 the first twelve states of Langton's ant are shown, where the ant, black cells, and white cells are represented by the yellow sphere, blue cubes, and transparent (invisible) cubes, respectively.

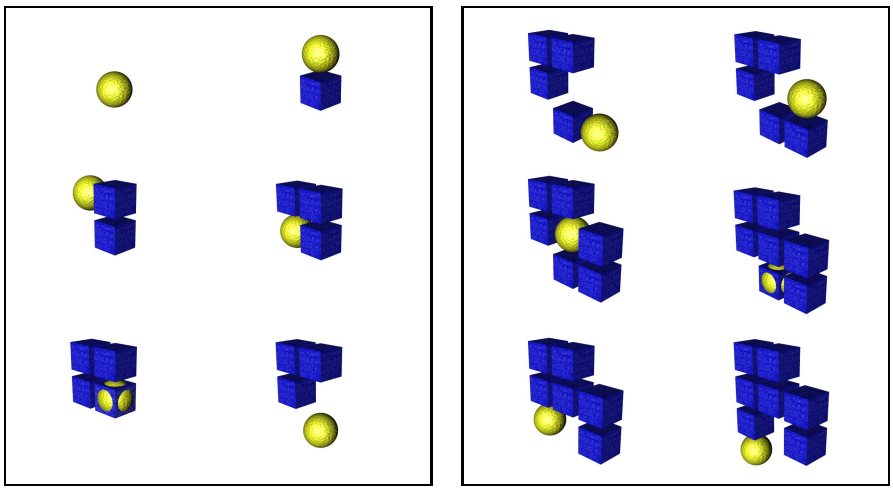

Figure 3. The first 12 states of Langton's ant

Part III

ANIMA-**Projekt**

19 ANIMA-Projekt

André René Gefken

1 Vorwort

Dieses projektbegleitende Dokument soll die meisten Fragen bezüglich des Projektes ANIMA der Universität Bremen abdecken. Es kann als Resümee über unser Projekt angesehen werden. Verfasst wurde es vor allem für diejenigen, die sich mit der Leitung eines Projektes beschäftigen wollen, diejenigen, die an einem Projekt teilnehmen wollen und als Hilfestellung für diejenigen Personen, die im Rahmen ihrer Projektarbeit mit Problemen zu tun haben und von anderen Projekten, in diesem Fall unserem, etwas lernen wollen.

Sie werden in diesem Dokument Informationen über die Mitglieder, den Projektinhalt und den Projektverlauf finden. Außerdem werden Sie viel über die Arbeitsweise innerhalb des Projektes erfahren und mit welchen Methoden vorgegangen wurde. Seien es allgemeine Erkenntnisse, die im Projekt erlangt wurden, oder aber persönliche Eindrücke der Mitglieder, Sie werden sicherlich äußerst interessante und hoffentlich lehrreiche Informationen vorfinden.

In diesem Dokument werden Ihnen allerdings keinerlei tiefer gehende Informationen zu dem Produkt AnimaLab gegeben werden, das im Rahmen des Projektes entstanden ist. Für diesbezügliche Informationen wollen wir, die Mitglieder des Projektes ANIMA, Ihnen die Programmdokumentation ans Herz legen. In dieser werden Sie genaue Informationen zu AnimaLab finden. Ebenso werden Sie keine Informationen zu den theoretischen Grundlagen unseres Projektes vorfinden, die wir ebenfalls in einem weiteren Abschnitt unseres Projektberichtes abhandeln.

Wir wünschen Ihnen auf jeden Fall viel Spaß beim Lesen unseres Projektdokumentes.

2 Dankeswort

An dieser Stelle soll von uns für die Hilfe und Unterstützung danke gesagt werden. Abgesehen von uns Teilnehmern, die wir natürlich den Löwenanteil zum Gelingen unseres Projektes beigetragen haben, gibt es eine Reihe von Personen, denen wir danken wollen.

Zunächst geht ein dickes Lob und Dankeschön an Hans-Jörg Kreowski (und seine Arbeitsgruppe), Frank Drewes und Renate Klempien-Hinrichs.

Diese drei, die uns betreuten, haben dem Projekt einen guten Start, einen erfolgreichen Ablauf und ein erfolgreiches Ende ermöglicht. Sei es mit den nötigen Vorlesungen, die begleitend stattfanden, sei es mit Rat und mit Tat, mit dem Bereitstellen von zusätzlichen Geld- und Arbeitsmitteln, oder aber mit konstruktiver Kritik. Hierfür ein herzliches Danke von uns allen!

Wir wollen aber auch nicht vergessen, wer uns dieses Projekt überhaupt ermöglicht hat und uns im Rahmen unseres Studiums die Chance gegeben hat, wichtige Erfahrungen mit Projektarbeit zu machen. Hierfür ein dickes Dankeschön an den Fachbereich 3 und natürlich an die Universität Bremen selbst.

Neben den Projektteilnehmern und Betreuern gab es zahlreiche Personen, und das sollte man auf keinen Fall vergessen zu erwähnen, die nichts mit dem Projekt ANIMA an sich zu tun hatten, aber dennoch mit Rat und Tat zur Seite standen oder uns ein angenehmes Arbeiten ermöglicht haben. Ein recht herzliches Dankeschön also auch an diese Personen und natürlich auch an all die Leute, die an dieser Stelle noch nicht erwähnt wurden, die sich aber dennoch unserer Probleme angenommen und uns mit Rat, Tat und einem offenen Ohr hilfreich zur Seite gestanden haben.

Danke von uns allen.

3 Die ANIMA-Projektmitglieder

Lutz Albrecht
Sarah Behrens
Maksym Bortin
Mathias Bremer
Philip Büschenfeld
Jianfeng Chen
Björn Cordes
Frank Drewes (B)
Lars Fischer
André René Gefken
Thomas Harlos
Karsten Hölscher
Renate Klempien-Hinrichs (B)
Martin König
Antonio Krampe
Hans-Jörg Kreowski (B)
Reiner Leins
Thomas Meyer
Thomas Oliver Moll
Michael Prahl
Rafael Trautmann
Carolina von Totth

(B) Betreuer/Betreuerin

4 Das Projekt ANIMA

4.1 Näheres über unser Projekt

Wenn man sich fragt, was denn ANIMA überhaupt für ein Projekt ist, so sollte man vielleicht mit dem Namen anfangen. ANIMA deutet es schon gut an. Das Projekt hat mit Bildern, aber auch mit ANIMAtion zu tun. Der Titel der Projektveranstaltung bzw. des Projektes ANIMA – *Regellabor für*

Visualisierung und Animation geht in dieser Hinsicht noch weiter. Es geht also um regelbasierte Animation.

Um das Geheimnis zu lüften, ANIMA ist ein Projekt des Fachbereiches 3 der Universität Bremen. Im Rahmen der Theoretischen Informatik beschäftigt man sich dort mit den syntaktischen Methoden der Bilderzeugung. Mit anderen Worten erstellt man Bilder mit Hilfe von Regeln. Bekannte Bilder, in diesem Zusammenhang, sind z.B. die Sierpinski-Pyramide oder die Drachenkurve.

Noch sehr viele andere Bilder lassen sich ebenso, wie die erwähnten, aufgrund von Regeln erzeugen. Diese alle hier aufzuzählen, wäre zum einen nicht gerade leicht, und würde zum anderen auch nicht unbedingt zum Verständnis bzw. zum Kennenlernen des Projektes beitragen.

Das Projekt ANIMA beschäftigt sich also mit der Erzeugung von Bildern mit Hilfe von syntaktischen Methoden und erstellt Animationen. So wäre es z.B. vorstellbar, sich durch eine dreidimensionale Welt bewegen zu können, die aus syntaktisch erzeugten Gebilden besteht. Man könnte sich z.B. die Erkundung eines von sogenannten Ameisen erzeugten Turmes vorstellen. Solche Dinge waren es, die wir vor allem im Sinn hatten, als wir uns für den Beitritt zu diesem Projekt entschieden hatten.

Zeitlich gesehen fand das Projekt vom Wintersemester 1999/2000 bis zum Ende des Sommersemesters 2001 statt. Geleitet wurde es durchgehend von Hans-Jörg Kreowski, wobei er zu Beginn von Frank Drewes, und am Ende von Renate Klempien-Hinrichs unterstützt wurde. Diese drei haben mit Fachwissen und Engagement einen nicht unerheblichen Teil zum Gelingen des Projektes beigetragen.

4.2 Unsere Motivation

In erster Linie wollten alle Teilnehmerinnen und Teilnehmer ihren Spass haben. Eine sehr trockene, ins kleinste Detail durchstrukturierte Projekthandhabung, oder gar haufenweise Vorschriften, Deadlines, und schriftliche Ausarbeitungen wollte eigentlich keiner. Unter der Voraussetzung, dass also eine gewisse lockere Handhabung vorherrscht, die einem die persönliche Entfaltung innerhalb des Projektes ermöglicht, und unter der Voraussetzung, dass die Teilnehmer untereinander halbwegs gut miteinander auskommen, waren alle bereit Teamgeist zu entwickeln, und zum Gelingen des Projektes beizutragen.

Die meisten von uns hatten außerdem gehofft mehr über Projektarbeit und die damit verbundenen Probleme lernen zu können. Zu lernen, wie man an Projekte herangehen kann, sei es von der entwicklerischen Phase her, von der organisatorischen, aber auch von der Durchführung her, war ein persönliches Ziel, das nahezu alle Teilnehmer teilten. Nur wenige sahen das Projekt als reine Pflichtsache an. Für diejenigen stand also in erster Linie der Erwerb des "Projektscheines" im Vordergrund. Dieser Leistungsnachweis, der

am Ende des Projektes an die Teilnehmer vergeben wird, soll die Mitarbeit innerhalb des Projektes über den Zeitraum von zwei Jahren widerspiegeln.

In manchen Projekten legen die Verantwortlichen einen hohen Wert darauf, dass nach jedem Semester eine bewertbare schriftliche Ausarbeitung im Rahmen des Projektes vorliegt. Dies war in unserem Projekt nicht der Fall, und trug vor allem dazu bei, unsere ungezwungene Arbeitsweise zu fördern. Jeder konnte, je nachdem wie er gerade Lust und Laune hatte, mal richtig ranklotzen, oder aber eine Schaffenspause einlegen.

Allgemein wurde diese Vorgehensweise als positiv empfunden und trug also im Sinne unserer Motivation dazu bei, auch weiterhin bei diesem Projekt zu bleiben, und nicht zu einem anderen über zu wechseln. Als eigentlich notwendige Bewertungsgrundlage, ist laut unserer Studienordnung ohnehin nur die Mitarbeit am Projektbericht wichtig. Von dementsprechenden schriftlichen Ausarbeitungen, wie sie manche Projektleiter eben als Voraussetzung für ihre Benotung sehen, ist da nicht die Rede. Unser Projekt hat sich also in einem völlig legitimen Rahmen bewegt.

4.3 Unsere Lernziele und Erwartungen

Betrachtet man den inhaltlichen Aspekt des Projektes, so waren die Lernziele, Vorstellungen, und Voraussetzungen der Teilnehmer doch etwas unterschiedlicher. Einige waren, was bei einem theorieorientierten Projekt wohl zu erwarten war, in erster Linie an den theoretischen Grundlagen und der Umsetzung dieser Erkenntnisse interessiert und weniger am Programmieren. Von diesen Leuten haben sich dann einige mit der Modellierung von Bildern beschäftigt und die syntaktischen Methoden der Bilderzeugung aufgearbeitet bzw. überarbeitet und erweitert. Andere waren wenig an der Theorie und wenig an der programmiertechnischen Umsetzung interessiert und wollten allgemein nur die Projektarbeit an sich kennen lernen. Nun war es aber schon so, dass diejenigen sich wenigstens teilweise für Bilderzeugung und Programmierung interessierten. Eine wirkliche Abneigung gegen die theoretische und praktische Arbeit während des Projektes war dementsprechend nicht gegeben. Als dritte Gruppe gab es also noch die Teilnehmer unseres Projektes, die wenig mit der Theorie zu tun haben wollten und vor allem auf das Programmieren und die mit dem Projekt verbundenen neuen Techniken ihren Schwerpunkt gelegt hatten. Tendenziell war dies auch der größte Teil unserer Projektmitglieder.

Auch Erwartungen gab es, wie bereits angedeutet. Allgemein wurde z.B. vorausgesetzt, dass alle die sich am Projekt beteiligen wollten, sich auch dementsprechend mit Engagement ins Zeug legen sollten. Alle sollten ein gewisses Grundinteresse mitbringen und zum Erfolg des Projektes beitragen. Diese Voraussetzung wurde auch allgemein erfüllt. Wenn sich auch die Interessen und Schwerpunkte gelegentlich verschoben, so hatten alle doch zumindest immer eine Aufgabe an der sie irgendwie arbeiteten.

Teamfähigkeit der Teilnehmer, gegenseitiger Respekt und die Bereitschaft Kritik zu äußern, anzuhören und zu verarbeiten waren ebenso Voraussetzungen, die an die Projektmitglieder gestellt wurden. Es sollten alle die Möglichkeit haben sich mitzuteilen und ein Vetorecht haben, wenn sie mit dem Projektverlauf oder inhaltlichen Dingen nicht einverstanden waren. Diese Voraussetzungen wurden an sich auch erfüllt. Zwar hatte man nicht immer den Eindruck, aber im Endeffekt war es doch so.

Wichtig war es für uns Teilnehmer auch, dass wir zumindest halbwegs brauchbare Materialien und eine gewisse Grundausstattung für das Projekt zur Verfügung gestellt bekommen würden. Es war eigentlich allen klar, dass wir nicht die hochmodernste Rechnerausstattung bekommen würden, aber ein halbwegs guter Rechner, der für Grafikbearbeitung nun mal notwendig ist, und relativ aktuelle Bücher wurden schon vorausgesetzt. Mit einem netten Arbeitsraum, der der Anzahl an Teilnehmern angemessen ist, wurden wir leider nicht beglückt, worauf viele unseres Projektes doch schon Wert gelegt hätten.

Ansonsten gab es eigentlich nur noch eine entscheidende Voraussetzung, die alle Projektteilnehmer an das Projekt gestellt hatten. Es war allen klar, dass sie das Projekt nur dann als wirklich erfolgreich, im Sinne der persönlichen Ziele ansehen würden, wenn auch ein fertiges Produkt dabei entsteht. Eine halbfertige Arbeit sollte auf keinen Fall nach den zwei Projektjahren liegen bleiben.

4.4 Unser benötigtes Vor- und Fachwissen und seine Erarbeitung

Vorwissen und Fachwissen sind für jede erfolgreiche wissenschaftliche Arbeit notwendig. Natürlich zählen auch Studien im Rahmen eines Projektes zur wissenschaftlichen Arbeit. Im Rahmen unseres Projektes brauchten wir also auch ein gewisses Fachwissen, um unser Ziel erreichen zu können.

Ein bestimmtes Vorwissen war allgemein gesehen nicht nötig gewesen. Nun ist es ja nicht so, dass wir kein Vorwissen mitbrachten. Da wir gerade unser Grundstudium beendet hatten, konnten wir zu Beginn des Projektes zumindest schon mal Programmierkenntnisse vorweisen, die wir mit Sicherheit für die grafische Umsetzung unserer Bilder auf einem Rechner gebraucht hätten. Ebenso waren auch noch die Verfahrensweisen der Softwareentwicklung gerade frisch vorhanden.

Um die entwicklungstechnische Seite, also das damit verbundene Fachwissen, brauchten wir uns keine grossen Sorgen zu machen. Was blieb dann aber übrig um ein Thema wie dieses als Teil dieses Dokumentes zu rechtfertigen. Natürlich die Theorie. Zu Beginn des Projektes hatten wir eigentlich noch keine richtige Ahnung, was uns erwarten würde. Allgemein hatten wir nur im Hinterkopf, was wir im Grundstudium über regelbasierte Sprachen im Bereich der theoretischen Informatik gelernt hatten. Doch jetzt wurden wir mit den syntaktischen Methoden der Bilderzeugung konfrontiert. Es ging für

uns also darum, wenn wir überhaupt erkennen wollten, was wir für Modellierungsmöglichkeiten haben würden und eventuell umsetzen möchten, uns erst mal diese anzueignen.

Zunächst waren da natürlich die sehr hilfreichen Vorlesungen über die syntaktischen Methoden der Bildbearbeitung, die unser Projektleiter selbst gehalten hatte. Mit diesem Fachwissen, konnten wir dann langsam unser Projektziel festlegen. Ausserdem gab es mehr als genug gute Literatur, die uns zum einen inspiriert aber vor allem auch mit ergänzendem Fachwissen versorgt hat. Die Möglichkeit diese Literatur zu nutzen war jederzeit gegeben und wurde auch gut genutzt. Fehlende Bücher wurden angeschafft und füllten eigentlich auch die letzten Lücken. Internetquellen zählen in diesem Sinne natürlich mit zu unseren Quellen für zusätzliches Fachwissen.

Von der programmiertechnischen Seite her war ergänzendes Fachwissen natürlich auch unumgänglich. Aber auch in diesem Fall gab es mehr als genug Möglichkeiten sich per Vorlesungen oder dementsprechender Literatur weiterzubilden. Abgesehen von diesen fast als Standard anzusehenden Möglichkeiten sich Fachwissen zu erarbeiten, konnten wir auch jederzeit unsere Projektbetreuer um Rat fragen. Dabei braucht eigentlich nicht extra erwähnt zu werden, dass diese Ratschläge nicht nur gut waren, sondern auch sehr hilfreich.

Um das nötige Fachwissen, das wir umgesetzt haben, und das unsere Projektarbeit auch wirklich als wissenschaftliche Arbeit rechtfertigt, brauchten wir uns also keine Sorgen zu machen. Hilfestellung beim Erarbeiten gab es mehr als genug.

4.5 Uns zur Verfügung stehende (Hilfs-)Mittel

Betrachtet man die Ausstattung des Projektes äußerst kritisch, dann gäbe es sicherlich einige Punkte, über die man streiten könnte. Vergleicht man z.B. den uns zur Verfügung gestellten Raum mit den Projekträumen, von anderen Projekten, dann fällt er doch ziemlich klein aus. Für ein stark besetztes Projekt wie unseres ist er nicht unbedingt angemessen. Wenn man auf der anderen Seite allerdings weniger kritisch auf diesen Missstand blickt, dann muss man sagen, dass nur selten alle Mitglieder gleichzeitig in den Raum hinein mussten und meistens auch gar nicht wollten. Viele unserer Projektmitglieder waren zu unterschiedlichen Zeiten im Projektraum und wie unsere Heimarbeiter, fast gar nicht im Projektraum anzutreffen. Von daher ist das Projekt also gut mit unserem Projektraum ausgekommen. Die Verteilung der Rechner, die einem Projekt für die Arbeit zustehen, würde bei kritischer Betrachtung auch eher schlecht bewertet werden. Ein so grafisch orientiertes Projekt, wie das unsere, sollte schon eine gute Rechnerausstattung, mit dementsprechenden schnellen Rechnern haben. Auch die Anzahl sollte dem Projekt angemessen sein, was nicht gerade den Anschein gemacht hat.

Fairerweise muss man sagen, da sind wir Projektteilnehmer uns eigentlich schon einig, dass wir mit den vorhandenen Rechnern ausgekommen sind.

Aufgrund der beschränkten Anzahl von Leuten, die zugleich im Projektraum arbeiten wollten, waren letztendlich gerade genug Rechner vorhanden. Und aufgrund der Prioritäten, wurden die Rechner den entsprechenden Leuten gelassen, die z.B. für die grafisch aufwendigeren Dinge die schnelleren Rechner brauchten. Nebenbei gab es dann ja auch noch die Möglichkeit in der 0. Ebene die Rechnerausstattung zu nutzen, die zwar ebenso begrenzt, aber dennoch allgemein für alle Informatikstudierenden zu Verfügung steht. Ob nicht auch deshalb weniger Leute im Projektraum gearbeitet haben, weil wenige Rechner zur Verfügung standen, sei in diesem Moment einfach mal dahingestellt.

Wie bereits zuvor erwähnt, brauchte unser Projekt nicht sehr viel. Wir hatten einen Raum, einige Rechner, die mit den nötigsten Programmen und Hardwareteilen ausgestattet waren, und diverse Literatur. Mit der Literatur sind wir, wie auch schon angedeutet, sehr gut ausgekommen. In dieser Beziehung hat uns eigentlich nichts gefehlt. Wenn uns dennoch mal geringe Mittel, besonders Geldmittel gefehlt haben, die uns nicht vom Fachbereich für unsere Zwecke gegeben wurden, oder gegeben werden konnten, dann sprang die Arbeitsgruppe unseres Projektleiters hilfreich ein.

5 AnimaLab

5.1 Unsere Suche nach einem Ziel für ANIMA

Zunächst sollte man bei diesem Thema eins nicht vergessen. Es war eine grosse Gruppe, die sich zu Beginn des Projektes zusammengesetzt hatte. Jeder hatte eine eigene Vorstellung von dem Projekt, für das er sich angemeldet hatte. Keiner wusste wirklich, was syntaktische Methoden der Bilderzeugung waren. Und was die eigentlichen Verantwortlichen mit dem Projekt bezwecken wollten, war eigentlich auch noch unklar. Das einzige, was man mit Sicherheit wusste, war doch bloß das, was man bei der Einführungsveranstaltung gesehen und gehört hatte.

Da waren schicke Bilder, die sich unterschieden, aber vom Aufbau her doch irgendwie ähnelten. Da waren Pflanzen, Landschaften, Kristalle, sogar Tiere, die auf irgendeine Weise mit regelhaften Erscheinungen und Formen aufwarteten. Ausserdem war die Rede von einem System gewesen, das es ermöglichen sollte, mit Hilfe von Regeln, die schönen Formen selbst zu erzeugen. Und dieses auch noch mit Hilfe von Bildern, die in den dreidimensionalen Raum übertragen werden sollten, sodass syntaktisch erzeugte Körper entstünden, durch die man vielleicht hindurchgehen oder die man im Sinne von Animationen betrachten kann.

Da entstand allerdings ein Problem. Es wurde nicht, wie es zuvor wirkte, ein Ziel vorbestimmt, das es zu verwirklichen galt, sondern auf einmal die Wahlmöglichkeit eröffnet, sich selber ein Ziel für das Projekt zu suchen. Das Problem dabei war folgendes. Im Groben waren sich alle einig über das Projektziel. Es sollte ein System entwickelt werden, das möglichst das macht, was man sich in seiner Phantasie schon vorgestellt hatte. Bilder erzeugen, die auf

Regeln basieren und vielleicht nachher so aussehen, wie die schicken Fotografien in den Büchern, die man gesehen hatte. Dann sollten sie im 3D-Raum sein und animiert ablaufen. Aber was für Bilder wollte man jetzt überhaupt erzeugen können? Nur eine Art oder vielleicht alle im Sinne eines ultimativen syntaktischen Systems, das es im Projekt zu entwickeln galt?

Zunächst hatten sich nahezu alle entweder ihre Gedanken über ein Projektziel gemacht oder aber schon angefangen sich die seltsamsten Dinge vorzunehmen und zu überlegen, was daran wohl mit Regeln syntaktisch erzeugbar ist, und ob man es nicht zu einem umsetzbaren Teilziel unseres Systems machen könnte. Dabei kamen die schönsten, aber auch merkwürdigsten Ideen heraus. Die entsprechenden Personen stellten ihre Vorschläge in kleineren oder grösseren Präsentationen vor und waren hochmotiviert diese auch umzusetzen. In Ansätzen gab es auch schon Regeln, mit deren Anwendung einige der vorgestellten Bilder bzw. Gebilde mehr oder weniger erfolgreich erzeugt werden konnten.

Durch die gerade erst begonnene Vorlesung zu den syntaktischen Methoden der Bilderzeugung, also quasi durch das fehlende Fachwissen, war allerdings auch noch vieles bei den Vorschlägen, das nicht unbedingt als realisierbar einzustufen war. So kam es dann, dass schon bald wieder einige der Ideen verworfen wurden. Andere Vorschläge, die darauf hinaus liefen, dass sich das Projekt mit fachbereichsübergreifenden Themen hätte beschäftigen müssen, wurden ebenfalls schnell verworfen. Da sich das erste Semester, das in der Regel für die Zielfindung verbraucht wird, langsam fortschreitend dem Ende näherte, entschloss man sich also zu einem, wie sich herausstellte, erfolgreichen ersten Projektseminar.

Dieses Seminar hatte nun den primären Zweck, ein Ziel für unser Projekt festzulegen. Im Rahmen eines fast dreitägigen Aufenthaltes in einer Jugendherberge, sollte nun noch mal intensiv mit allen Teilnehmern des Projektes überlegt werden, wie weiterhin mit dem Projekt umgegangen wird, in welche Richtung es inhaltlich gehen sollte und wie man organisatorisch verfahren wollte. Dass nebenbei der Spaß nicht zu kurz kam und wir Teilnehmer uns näher kennen lernten, braucht an sich nicht extra erwähnt zu werden, aber es zeigt deutlich, dass sich die Voraussetzungen an das Projekt auch in dieser Hinsicht erfüllten.

Auf dem Seminar wurden wieder Vorträge gehalten, was man denn machen könnte, geklärt, ob es realistisch ist, diese Vorschläge auch umzusetzen, und die bisher offenen Fragen bezüglich älterer Vorschläge beantwortet. Der Erfolg hierbei war, dass man endlich einen besseren Überblick über die möglichen zu realisierenden Teilziele bekam, weil die entsprechenden Projektteilnehmer sich nochmals in ihre Vorschläge vertieft hatten.

Ein wichtiger Bestandteil des ersten Projektseminars war unbestreitbar der durchgeführte Workshop zur Zielfindung. Im allgemeinen war es ein abgewandelter Workshop im Rahmen von allgemeiner Projektarbeit, mit dem Ziel, Schwächen innerhalb des Projektes zu finden und Ziele zu festigen. Das Wir-Gefühl sollte durch diesen gestärkt werden, negative Stimmungen abge-

baut, und eine gemeinsame Vorstellung der Projektarbeit und des Projektzieles aufgebaut werden. Das Ziel bei dem Workshop war uns eigentlich genauso wenig klar, wie unser Projektziel, aber auf ein brauchbares Resultat hoffend machte jeder bei diesem Workshop mit. Es stellte sich auch ein brauchbares Ergebnis ein. So wurde z.B. eine allgemein negative Stimmung im Projekt festgestellt.

Den meisten ging es nicht schnell genug. Für sie war es unbefriedigend nach fast einem Semester noch kein richtiges Projektziel zu haben. Andere gingen nach drei bis vier Monaten tatsächlich schon so weit, sich über die Arbeitsmoral anderer aufzuregen, wobei dies schon irgendwie verfrüht erschien. Einige hatten sich eben schon so sehr in Arbeit gestürzt, dass sie aus den Augen verloren, dass andere noch mehr als genug Zeit haben würden sich im Projekt einzuarbeiten. Nebenbei gab es ja auch noch nicht viel Arbeit, in die man sich sinnvollerweise hätte stürzen können, da die Zielfindung noch nicht einmal abgeschlossen war.

Festgestellt wurde mit Hilfe des Workshops allerdings auch, dass viele schon eine genaue Vorstellung von dem hatten, was sie mit dem Projekt erreichen wollten. Und so kam es dann fast schlagartig zu einer Festlegung unseres Zielsystems und einer allgemeinen, auf Interessen beruhenden Verteilung der Mitglieder auf einzelne Arbeitsgruppen. Alle hatten nun ein grundlegendes Wissen aus der ersten Vorlesung und schlossen sich einer Arbeitsgruppe an, die entweder mit der Modellierung von Bildern und der theoretischen Weiterentwicklung dieser zu tun hatte oder aber mit der Entwicklung unseres Systems zur Erstellung von Bildern und Animationen. So stand also nach dem ersten Semester für die meisten ein gemeinsames Projektziel, und es waren alle zur Beruhigung der Arbeitswütigen mit irgendeiner Arbeit versehen.

5.2 Unser Projektziel – AnimaLab

Die Meinungen sind doch geteilter gewesen, als ursprünglich angenommen. Nach dem ersten Projektseminar glaubten viele, eine einheitliche Meinung über unser Projektziel zu haben. Da hiess es, wir würden ein System, also ein Programm entwickeln, das auf verschiedenen Rechnern lauffähig ist. Es soll die anfallende Arbeit, sofern möglich, auf verschiedenen Rechnern erledigen und auf einem das Ergebnis ausgeben können. Bestehen sollte es aus zahlreichen sogenannten Modulen, einem damals sogenannten Editor, einer zentralen Schnittstelle, die für die Kommunikation benötigt wird, und einem sogenannten Viewer. Die Module sollten eigenständige Programme sein, die auf einem Rechner gestartet werden können, auf der Grundlage von Ableitungsregeln Bilder erzeugen und das Ergebnis zur Verfügung stellen können. Für jede syntaktische Methode, die wir zur Erzeugung von Bildern nutzen wollten, sollte dementsprechend ein Modul da sein. Der Editor als quasi vereinendes Element hatte nun die Aufgabe die Möglichkeit zu bieten, erzeugte Bilder bzw. Teilbilder mit sogenannten Nichtterminalen-Elementen zu vereinen, z.B. grössenmässig aufzubereiten und zur Anzeige an den Viewer weiter-

zugeben. Ausgegeben werden sollten entweder Standbilder oder aber im Sinne einer Animation entstehende dreidimensionale Welten, denen man sozusagen beim Wachsen zusehen kann.

Nachdem im zweiten Semester allerdings fast nur gruppenintern gearbeitet wurde und allgemein nicht mehr viel über das Gesamtsystem geredet wurde, stellte sich im dritten Semester so langsam eine Art Verwirrung ein. Mit fortschreitender Zeit bemerkte man, dass man doch kein gemeinsames Bild vom Projektziel hatte. Auch die spärlich dokumentierten bzw. selten aufgesetzten Spezifikationen waren keine Hilfe. Es musste ein zweites Mal ernsthaft überlegt werden, was denn nun das Ziel sei und ob es so überhaupt noch realisierbar ist.

Wenn es auch in gewisser Hinsicht so scheint, dass dies ein Nachteil gewesen ist, dass man wirklich alles sofort hätte aufschreiben und festlegen müssen, dann stimmt das irgendwie nur halb. Sicherlich hätte das so manche Diskussion verhindern können und man hätte durchgängig dasselbe Ziel verfolgt, aber fraglich bleibt doch, ob es unsere Arbeitsweise wirklich beeinflusst hätte und ob es nicht sowieso gut ist, gelegentlich das gemeinsame Projektziel und -verständnis zu überprüfen.

Gerade in unserem Projekt konnten wir gut merken, was passiert, wenn die Kommunikation nicht funktioniert. In unserem Fall verloren wir eben das gemeinsame Ziel aus den Augen. Ein Resultat, das sich für unser ursprüngliches Ziel ergab, war die Reduktion der Module. Von allen geplanten Modulen, also sozusagen den syntaktischen Methoden, wurden einige gestrichen. Zum einen, weil sich die Zuständigen nicht mehr weiterhin mit ihnen beschäftigen wollten und kein Ersatz gefunden wurde, und zum anderen, weil sie sich doch nicht als so erfolgversprechend herausstellten. Um die übrige Arbeit mit Erfolg beenden zu können, strichen wir also einige der Module.

Im Groben blieb die Struktur unseres ursprünglich geplanten Systems erhalten genauso wie der restliche Ablauf bzw. das Zusammenspiel innerhalb des Systems. Mit den notwendigen Änderungen, die sich aus der Reduktion ergaben, waren eigentlich auch alle Mitglieder des Projektes einverstanden. Von daher kann man schon sagen, dass in gewisser Weise zu diesem Zeitpunkt ein einheitliches Projektziel gegeben war.

Unser letztendliches Projektziel bestand zum einen also darin, ein wenn auch reduziertes, aber lauffähiges und verteiltes System zu entwickeln, das Bilder und Animationen mit Hilfe von syntaktischen Methoden erzeugt. Und zum anderen sollte dieses System auf dem bevorstehenden Projekttag vorgeführt werden können. Davon mal abgesehen sollte der vorliegende Projektbericht entstehen, der zum einen unsere Projektarbeit bewertet und zusammenfasst, und zum anderen als Hilfestellung und Anregung für andere Projekte zu gebrauchen sein sollte. Genauso sollte er eine Dokumentation und Weiterentwicklungshilfe beinhalten, aber auch einen Theoriebereich für die theorieinteressierten Leser. Zur Abrundung für diejenigen, die sich nur eine Auswahl an schicken Bildern angucken möchten, die unser System erzeugen kann, war neben den drei anderen Bänden unseres Projektberichtes

ein Bildband geplant, mit diversen Bildern, die mit unserem System modelliert und erzeugt wurden. Auf jeden Fall war aber ein zweites Hauptziel ein Leistungsnachweis bzw. der sogenannte Projektschein. Das Projekt sollte auf jeden Fall fertig und erfolgreich abgeschlossen sein, da waren sich alle einig.

Eine Frage, die sich jetzt stellen könnte, wäre die Frage nach der Realisierbarkeit unseres Vorhabens. Betrachtet man unser erstes Projektziel, das sich nicht wirklich von unserem späteren unterscheidet, so kann man durchaus sagen, dass es möglich gewesen wäre, es zu realisieren, auch wenn einige von uns Projektmitgliedern bis zuletzt glaubten, dass es utopisch war und die Ziele zu hoch gesteckt waren. Die zur Verfügung stehende Zeit war auf jeden Fall ausreichend, wenn man betrachtet, was in der Endphase noch für grosse Fortschritte gemacht wurden. Durch unsere allgemeine selbstbestimmende bzw. lockere Arbeitsweise, die auch bewusst nicht von unseren Projektbetreuern forciert wurde, ging allerdings einiges an Zeit verloren. Resultate standen für gewöhnlich erst nach mehreren geplatzten Terminen, sofern überhaupt Termine festgesetzt wurden. Die Ergebnisse waren für gewöhnlich allerdings durchaus gut, denn durch den Mangel an Druck haben sich die dementsprechenden Personen mehr Zeit genommen, einige Dinge noch mal überdacht und so verbesserte Versionen abgeliefert. Man muss dennoch gestehen, dass es nur durch unsere mangelnde Selbstdisziplin dazu kam, dass Abstriche an unserem ersten Entwurf gemacht werden mussten, um sicher zu stellen, dass überhaupt ein funktionierendes Komplettsystem fertig wird. Das erste System wäre nach dem Anbruch des dritten Semesters nicht mehr realisierbar gewesen.

5.3 Die Entwicklungsphase

Das Ziel stand also wenigstens grob nach dem ersten Projektseminar zur Findung des Projektvorhabens. Die Frage ist jetzt natürlich, wie man das Vorhaben realisieren wollte. Zunächst einmal gab es da zwei Möglichkeiten für uns Projektteilnehmer und mindestens zwei für unsere Projektleitung.

Wir hatten uns zu entscheiden, ob wir uns selber eine feste Strukturierung der nächsten eineinhalb Jahre ausdenken, uns selber feste Zielpunkte setzen und diese in einem angemessenen Rahmen erledigen wollten. Das hätte bedeutet, uns selbst Druck zu machen. Alternativ dazu hatten wir uns zu entscheiden, ob wir uns gefühlsmässig an die Sache ranwagen sollten und darauf hoffen würden, dass alle mit genügend Selbstdisziplin arbeiten und das Endprodukt bis zum Projekttag zusammen fertigstellen würden.

Unser Projektleiter allerdings hatte nun die Möglichkeit uns entweder sehr zu forcieren, sodass auf jeden Fall ein Endprodukt und zuvor natürlich fertige Teile rechtzeitig entstehen würden, oder aber die Möglichkeit den weiteren Projektverlauf völlig uns zu überlassen, sodass wir zwar einen Lerneffekt, aber nicht unbedingt ein fertiges Produkt am Ende unserer zwei Jahre haben würden.

Wie ich bereits angedeutet hatte, war die Tendenz innerhalb des Projektes auf jeden Fall die, dass niemand wirklich Druck wollte, und jeder gerne zu der Zeit und mit der Geschwindigkeit, die er bevorzugte, an seiner Arbeit herumbasteln wollte. Natürlich sollten hierbei die einzelnen Teile unseres Systems trotz freier Zeiteinteilung rechtzeitig fertiggestellt sein. In guter Hoffnung auf ein Endprodukt war diese Zeiteinteilung also quasi beschlossene Sache.

Unser Projektleiter hat sich auf unsere Anti-Druck-Haltung eingestellt, die wir erfolgreich mit Nichtbeachtung seiner gelegentlichen Hinweise, doch mal etwas Schriftliches bei ihm zur Ansicht einzureichen, unterstützt hatten. Ob er eventuell mit einer eher druckbehafteten Art unser Projekt zu führen, das Projekt besser geleitet hätte, sei dahingestellt, aber die Möglichkeit uns mehr unter Druck zu setzen hätte er sicher gehabt.

Betrachtet man unsere Haltung gegenüber der Arbeitsverteilung, so sollte erwähnt werden, dass eigentlich einheitlich die Meinung vorherrschte, die Arbeit im Projekt nicht zu sehr in Kleingruppen aufzuteilen. Jedenfalls war das zu Beginn des Projektes so. Jeder wollte im grossen Rahmen programmieren, sich nicht nur mit seinem Nachbarn einig werden, sondern wie bei "richtigen" Projekten, sich mit allen Projektmitgliedern arrangieren, kommunizieren und lernen, die vielen kleinen Teile mit allen zu entwickeln und zusammenzufügen.

Davon war nach dem ersten Projektseminar nichts mehr zu spüren. Was passiert war, könnte man sich jetzt fragen. Das ist leicht beantwortet. Froh über das endlich gefundene Projektziel, begannen auf einmal alle, sich irgendwelchen Teilen unseres Gesamtsystems zuzuteilen. Was eigentlich nur als Feststellung der Interessen der Mitglieder und als Sicherstellung begann, auch alle Teile mit Interessenten irgendwann entwickeln zu können, artete auf einmal so aus, dass jeder das Gefühl hatte, dass er in der "Arbeitsgruppe" eingeteilt sei, der er sich zugeordnet hatte. Das, was also eigentlich nicht gewollt war, nämlich die Kleingruppenbildung, war in schlimmster Form eingetreten. Es gab Gruppen zwischen 2 bis 5 Leuten, und sogar Gruppen, wenn man die überhaupt so nennen kann, in denen sich nur einzelne Personen befanden. Aber es schien keinen zu stören.

Durch diese Gruppenbildung waren alle Mitglieder also einer Aufgabe zugeteilt. Jeder konnte sich jetzt an die Arbeit machen und irgendetwas Produktives zum Projekt beisteuern. Wie bereits angedeutet geschah dies vor allem in Gruppen- oder Alleinarbeit. Selten wurde etwas Inhaltliches bei den Projekttreffen, die wöchentlich stattfanden, preisgegeben oder gar aufgeschrieben und noch seltener allen zur Verfügung gestellt. Auch die Dokumente, die an die Projektleitung gingen, und sozusagen auch als Bewertungsgrundlage dienen sollten, waren eher gering an der Zahl.

Es wurde kaum noch daran gedacht, dass eventuell noch Missverständnisse bezüglich des Gesamtsystems bestehen könnten oder, dass die Selbstdisziplin bei uns Mitgliedern vielleicht doch nicht die höchste ist. Also wurde in dem Semester kaum etwas produziert, geklärt oder besprochen. Ganz im Gegenteil zu der anfänglichen Euphorie, sich kommunikativ mit allen anderen Leuten zu arrangieren und abzusprechen.

Es schlich sich leider auch die Unlust ein, überhaupt etwas genau zu spezifizieren. Nicht nur dass man seine eigene Arbeit selten oder gar nicht dokumentierte, es wurden sogar gelegentliche Vorschläge abgelehnt, die eine gewissenhaftere Arbeitsweise gefördert hätten, vor allem in Hinsicht darauf, was wir über Softwareentwicklung und dementsprechende Spezifikationen gelernt hatten. Nahezu jedes Diagramm, das eventuell mal angezeichnet wurde, war so ungenau, dass man damit eigentlich keine gemeinsame Vorstellung z.B. des benötigten Funktionsumfanges hätte haben können. Der Wille diese Ungenauigkeiten zu entfernen oder das Diagramm sorgfältig für alle abzulegen war eben nicht da. Auch das übliche Pflichtenheft gab es eigentlich nicht. Sofern einige Gruppen überhaupt eine Vorstellung hatten, was ihre Module können sollten, und vielleicht sogar für sich selber so etwas wie ein Pflichtenheft hatten, so wurden auch diese Dokumente nicht zusammengefügt, sodass etwas Schriftliches als Anhalt für alle da gewesen wäre.

Dennoch hatte jeder etwas mit dem er sich eifrig beschäftigen wollte und konnte, und genoss die Freizeit, die sich durch die extrem kurzen, oder manchmal auch ausgefallenen Projekttreffen so nebenbei ergab.

Zu Beginn des dritten Semesters wurde dann erneut nach vorhandenen Ergebnissen gefragt. Wenige hatten überhaupt Ergebnisse, die sie vortragen wollten oder konnten, weil sie einfach noch nicht ausgereift genug dafür waren oder in manchen Fällen auch nicht vorhanden.

In den nächsten zwei Monaten bemerkte man so langsam, dass nicht alles aufgrund mangelnder positiver Arbeitshaltung so vorwärts ging, wie man es ursprünglich erwartet oder gehofft hatte. Von dem produktiven Verlauf des ersten Projektseminars beeindruckt, entschloss man sich, ein zweites noch innerhalb dieses Semesters zu planen und durchzuführen. Das Ziel war diesmal folgendes. Es sollte bis zum Seminar an dem weitergearbeitet werden, womit man gerade beschäftigt war. Auf dem Seminar sollte dann allgemein darüber diskutiert werden, was wir uns früher vorgenommen hatten, was davon überhaupt noch bis zum Projekttag realisierbar sein könnte, und was auf jeden Fall noch fertiggestellt werden sollte. Hierbei wurden dann, wie bereits zuvor angedeutet, einige Streichungen vorgenommen und das System neu überdacht.

Gerade dabei fiel zum ersten Mal richtig auf, wie sehr die Meinungen über unser System auseinandergingen und wie unterschiedlich sich einige die Umsetzung vorgestellt hatten. Das Klären dieser Ungereimtheiten erforderte einiges an Zeit und hatte zur Folge, dass man diesmal erneut festlegte, wie das Verhalten unseres Systems denn nun sein sollte bzw. wie also die einzelnen Komponenten zusammenspielen sollten. Nebenbei wurden dabei auch noch einige namentliche Änderungen vorgenommen. Da wurden z.B. aus den Modulen die *Object-Generators* und aus dem Editor der *Arranger*. Viele solcher Nebensächlichkeiten wurden allerdings in unserem Projekt sehr wichtig genommen und heiß diskutiert.

Zum einen hatte also die erneute Diskussion über unser System einen beträchtlichen Teil der Seminarszeit eingenommen und zumindest fürs Erste

eine Einigung bewirkt. Das allgemeine Verständnis schien jedenfalls gegeben zu sein. Zum anderen wurde das zweite Seminar inhaltlich von dem bevorstehenden Projekttag und dem noch anzufertigenden Projektbericht bestimmt. Hierzu wurde eine ausführliche inhaltliche Beschreibung des Projektberichtes vorgestellt und eine anschließende Diskussion zu diesem ersten ausgearbeiteten Projektberichtvorschlag und zum Projekttag geführt.

In produktiver Weise entstand ein relativ einheitliches Bild von dem, was wir Mitglieder gern in den Projektbericht haben wollten und wie er in etwa aufgebaut werden sollte. Das Problem, den Projektbericht mit Inhalt zu füllen, wollte man auf umfrageähnliche Weise lösen. Da ja bisher kaum schriftliche Ausarbeitungen vorlagen, obwohl des öfteren darum gebeten worden war, und da auch das extra für die schriftliche Arbeit zur Unterstützung eingerichtete elektronische Aktensystem nicht gerade mit vielen Dokumenten gefüllt war, schien dies ein möglicher Weg zu sein.

In Hinsicht auf den Projekttag kam man zumindest auf eine grobe Richtung, wie man das Projekt und seine Arbeit in angemessener Art den anderen Projekten und Besuchern präsentieren könnte. Die hierzu entfachte Diskussion verlief vielleicht etwas träge in Anbetracht der fortgeschrittenen Zeit und vergangenen Diskussionen, war aber dennoch produktiv. Dies kann man übrigens allgemein von den beiden Seminaren behaupten. Wenn sie auch etwas anstrengend waren, so haben sie sich auf die Produktivität, das gemeinsame Verständnis und natürlich auf den gemeinsamen Spaß äußerst positiv ausgewirkt. Solche Seminare kann man in dieser Hinsicht nur weiterempfehlen.

Mit den neu gesetzten bzw. überdachten Zielen machte man sich also frisch und voller Tatendrang an die Arbeit. Durch die schon fortgeschrittene Zeit war man sich einig die Projekttreffen mehr auszunutzen und sich auch in der vorlesungsfreien Zeit wenigstens alle zwei Wochen zu treffen und gemeinsame Probleme zu klären. Die Einhaltung der gesetzten zweiwöchentlichen Termine hatte allerdings nicht für jeden von uns den Stellenwert, den sie hätte haben sollen. Obwohl durchaus Probleme bei einzelnen Gruppen auftauchten, die sie gern mit allen besprochen hätten, waren nie alle Mitglieder da und selten mal eines aus jeder Gruppe.

Die Folge war natürlich, dass wieder keine Übersicht über die laufenden Arbeiten und deren Stände gegeben war und dass auch wichtige Absprachen und Problembeseitigungen nicht stattfanden. Bereits zu einem frühen Zeitpunkt wurde innerhalb unseres Projektes festgelegt, dass alle von uns regelmäßig ihre E-Mails lesen sollten. Dies wurde selten getan und demzufolge antworteten auch nicht alle auf die Fragen und beachteten wichtige Hinweise auch nicht. Die ursprüngliche Kommunikationsbereitschaft war immer noch nicht vorhanden bzw. umgesetzt worden.

Nach den Semesterferien kam dann allerdings eine entscheidende Wende. Von der nun vorhandenen Zeitknappheit beflügelt und von der unbefriedigenden Arbeitsweise und Arbeitshaltung einiger enttäuscht, entschied man sich endlich eine gewisse Organisation in das Projekt hineinzubringen. Man

erkannte eigentlich ganz klar, welche Probleme für einen erfolgreichen Abschluss unseres Projektes gelöst werden mussten.

Ein entscheidender Faktor war die Zeit. Das daraus resultierende Problem war nun mal die Knappheit. Wir mussten das restliche zur Verfügung stehende halbe Jahr unbedingt besser nutzen als die drei vorigen. AnimaLab sollte bis zum Projekttag in einer vorführbaren und natürlich funktionierenden Version fertiggestellt sein. Der Projekttag selber musste noch detailliert geplant und vorbereitet werden. Und der Projektbericht musste auch endlich ausgearbeitet und verfasst werden, sofern man das Projekt nicht verlängern wollte.

Ein weiteres wichtiges Problem war unsere Kommunikation. Um das System fertig stellen zu können, musste mehr miteinander gesprochen werden. Da wir bisher schon alle recht diskussionsbereit gewesen waren, entschloss man sich also, auch zu Gunsten der geringen Zeit, die Projekttreffen zu verlängern, sodass mehr Zeit zur Verfügung stand, um ausführlicher und in Ruhe über die kleineren und vor allem auch grösseren Probleme in Bezug auf Implementation, Organisation und Planung sprechen zu können.

Die Übersicht über den aktuellen Projektstand war bisher immer noch nicht gegeben. Zur Behebung dieses sehr hinderlichen Problems und um besser mit der verbliebenen Zeit auszukommen, entschloss man sich eine Projektkoordinationsgruppe (Proko-Gruppe) einzurichten. Diese sollte als Anlaufstelle für alle Gruppen dienen, um zum einen zur besseren Übersicht von allen die Arbeitsstände einfordern zu können und zum anderen bei dringenden Fragen vermitteln oder helfen und die einzelnen Arbeiten besser unterstützen und koordinieren zu können.

Ebenso wurde eine Projekttagsgruppe eingerichtet, die ihren Schwerpunkt auf den bevorstehenden Projekttag und die damit verbundenen Probleme setzen sollte, wie z.B. den organisatorischen Ablauf, den vorzustellenden Inhalt oder die Absprache mit den anderen laufenden Projekten. Schließlich war der Projekttag eines der wichtigen Ereignisse im Laufe unseres Projektes und somit ein weiteres Problem, mit dem man sich noch ausführlich beschäftigen musste.

Nebenbei ergab sich widererwartend auch noch das Problem, den Projektbericht nicht in Angriff nehmen zu können. Das geplante und getestete Umfragesystem trug nicht die erwarteten Früchte. Trotz mehrfacher Hinweise, sich daran zu beteiligen, und trotz diverser Termine, die geplatzt waren, stellte sich nicht der gewünschte umfangreiche Inhalt ein. Zunächst beteiligten sich nur vereinzelte Mitglieder, sodass sich nach einer langen Zeit gerade mal die Hälfte der Teilnehmer unseres Projektes überhaupt in geringem Umfang an der Materialsammlung für den Bericht beteiligt hatten. Einige hatten sich nicht mal die Mühe gemacht, sich diese überhaupt anzusehen. Dieses Problem musste unbedingt gelöst werden, um nicht am Ende des Projektes vor einem Berg von Berichtsarbeit zu stehen.

Vor allem durch die Proko-Gruppe, die allerdings weniger Rückmeldungen bekam als ursprünglich erwartet wurde, sind einige wichtige Neuerungen

eingeführt worden. Zur Beseitigung des Zeit- und Übersichtsproblems wurden feste Zeitpunkte gesetzt und auf Zeitlinien eingetragen. Jeder konnte sehen, wie seine Arbeit auf die der anderen abgestimmt war und welchen zeitlichen Rahmen er einhalten musste, um im Projekt wenigstens mit der verbleibenden Zeit weiterhin auszukommen. Leider sind von den von uns sogenannten Deadlines einige geplatzt, aber dennoch tat sich nun einiges mehr. Man konnte mittlerweile doch einige Fortschritte in Richtung eines gemeinsamen Systems sehen. Die erwarteten Rückmeldungen der Arbeitsgruppen blieben leider überwiegend aus. Dies hatte zwar zur Folge, dass die Proko-Gruppe nicht immer viel erzählen konnte, aber dennoch versuchte sie eifrig nicht die Übersicht zu verlieren, berichtete jede Woche über noch so kleine Fortschritte und Probleme und ließ nicht locker. Man kann sagen, dass diese Arbeit, die mit Sicherheit schon früher hätte stattfinden müssen, auf jeden Fall einen wichtigen Beitrag zur Produktivitätssteigerung und zur Lösung einiger wichtiger Probleme leistete.

In etwa genauso hilfreich waren die verlängerten Projekttreffen. Es wurde zwar gelegentlich immer noch über relativ eindeutige und manchmal verhältnismässig unwichtige Dinge wild diskutiert, aber jetzt hatte man sich wenigstens genügend Zeit reserviert, sodass keine wichtigen Dinge mehr aus Zeitdruck ungeklärt blieben. Gelegentlich fanden sogar lange nach Beendigung der Projekttreffen noch Workshops am Nachmittag statt, die ebenfalls zum allgemeinen Verständnis dienten. Alles in allem könnte man behaupten, dass wir das Zeitproblem zumindest im Ansatz im Griff hatten.

Das Kommunikationsproblem, das bei uns vorherrschte, war durch die verlängerten Treffen etwas entschärft worden, aber noch nicht gelöst. Wenn auch diverse Ansätze da waren, sich besser mit den anderen Gruppen abzusprechen, so fiel eigentlich nur die Kommunikation zwischen den einzelnen Gruppen und der zentralen Schnittstellengruppe ins Gewicht. Diese war schließlich auch für alle anderen entscheidend. Anderweitige Kommunikation kam wie immer etwas kurz. Dennoch war zumindest jetzt die Tendenz zur Besserung zu bemerken. Um die einzelnen implementierten Teile unseres Systems überhaupt zusammenfügen zu können, war dies auch dringend notwendig gewesen.

Neben der Projektkoordinierungsgruppe gab es wie erwähnt die Projekttagsgruppe. Auch diese hatte sich gut bewährt. Wo zuerst nur eine ungenaue Vorstellung war, festigte sich so langsam ein konkreter Entwurf dessen, was am Projekttag vorgestellt werden sollte. Es wurden Absprachen mit den anderen Projekten unseres Jahrgangs gehalten, organisatorische Dinge angesprochen und wenn möglich auch abgehakt, inhaltliche Vorstellungen diskutiert und in Arbeit gegeben, und natürlich auch schon erste Anfragen bezüglich des benötigten Materials und der Kosten getätigt. Man kann also mit Recht behaupten, dass der im Projekt entstandene Stress sich auch für das Problem mit dem bevorstehenden Projekttag so langsam abbaute. Natürlich war es zu dem Zeitpunkt noch zu früh um zu jubeln, aber man bekam doch langsam das Gefühl, innerhalb einer grösseren Gruppe an der selben Sache zu arbei-

ten. Man könnte fast sagen, dass sich erst jetzt das Projekt ANIMA so richtig entwickelte.

Was den Projektbericht anging, der nun an sich das letzte grosse Problem darstellte, diskutierte man erneut, in welcher Art man denn nun die vielen einzelnen Dokumente unter einen Hut bringen konnte, die in den Teilgruppen in unterschiedlichster Form zumindest in Ansätzen vorhanden waren. Auch die bisher gesammelten Antworten auf allerlei Fragen zum Projekt sollten nicht verloren gehen. So gründete man also eine Redaktionsgruppe, die sich um die Anfertigung der vier beschlossenen Teildokumente des Projektberichtes kümmern sollte. Die ersten Rohdokumente für den Projektbericht wurden vorgestellt, besprochen und weiterbearbeitet. Man könnte sagen, dass auch der Projektbericht trotz anfänglicher Schwierigkeiten langsam vorwärts kam.

Es war insgesamt Land in Sicht, das man auch erreichen konnte, sofern man sich anstrengen würde.

6 Der Projekttag

6.1 Unsere Vorhaben

In Anbetracht dessen, dass sich das Projekt erst dann entscheidend entwickelte, als doch etwas Druck gemacht wurde, könnte man denken, dass sich unter dem Druck des bevorstehenden Projekttages wieder eine Leistungssteigerung hätte ergeben müssen. Dem war eigentlich nicht so. Vielmehr war es so, dass wir zur Höchstform aufliefen. Die Leistungssteigerung war eher eine Leistungsexplosion. Eine so hohe Arbeitsmoral und Einsatzbereitschaft war vor den zwei entscheidenden Vorbereitungswochen noch nicht da gewesen.

In der heißen Phase vor dem Projekttag waren zunächst einmal alle Teilnehmer des Projektes den grössten Teil der Zeit über da. Auch wenn es schon allein wegen des begrenzten Raumes und der zahlreichen Arbeiten etwas chaotisch zuging, so war doch jeder bereit seinen Teil zum allgemeinen Chaos beizutragen. Besonders erfreulich war auch die Tatsache, dass die Arbeit nicht mehr durch die mangelnde Kommunikation behindert wurde. Schließlich hatte man während dieser zwei Wochen fast immer jeden zur Verfügung, mit dem man etwas zu bereden hatte.

Da die Zeit im gewissen Sinne schon sehr drängte, weil zum einen das AnimaLab noch nicht vollkommen vorführreif war und auch noch diverse andere Dinge geplant waren, zeigten alle eine hohe Einsatzbereitschaft. Diese äußerte sich vor allem darin, dass jeder an irgendetwas mithalf und nicht eher aufhörte, bis die Arbeit entweder erledigt war, oder zumindest einen gewissen Stand erreicht hatte.

Wenn hier die Rede von zahlreichen geplanten Dingen und der damit verbundenen Arbeit ist, so sind natürlich damit unsere Vorhaben bezüglich des Projekttages gemeint. Wir hatten uns als Hauptziel die Repräsentation des Projektes ANIMA vorgenommen. Ganz im Sinne des in der Informatik an der Universität Bremen so üblichen Projekttages, wollten wir vorstellen, was wir

in der vergangenen Projektzeit entwickelt und fertiggestellt hatten. Ebenso sollte ein gewisser Showanteil vorhanden sein und das Projekt im Sinne einer zusammengehörigen Gruppe vorgestellt werden. Mit anderen Worten wollten wir also die recht nüchterne Welt der Informatik etwas ansprechender gestalten und unseren Gemeinschaftsgeist demonstrieren. Was wir uns hierfür vorgenommen hatten waren die folgenden Ziele.

Unser Projektstand sollte zum einen mehrere Computer enthalten und somit eine Vorführung unseres Programms AnimaLab ermöglichen. Hierfür brauchten wir nicht nur ein vorführbares Programm, sondern auch eine Leinwand und einen Projektor, die es beide noch zu organisieren galt.

Da unser System zur Erzeugung von Bildern geschaffen wurde, sollten diese natürlich auch einen Hauptteil beim Projekttag einnehmen. Zum einen waren wir uns über die Werbewirksamkeit der Bilder bewusst und zum anderen sollten möglichst viele Besucher unseres Standes diese gleichzeitig bewundern können. Also nahmen wir uns vor, den ganzen Projekttag über, Vorführungen, Bilder und Animationen auf unserer Leinwand sichtbar zu haben, um zum einen den Stand interessant zu halten und zum anderen neue Besucher anzulocken. Abgesehen von der Hardware brauchten wir für dieses Vorhaben natürlich auch das Programm selbst und die entsprechenden Bilder und Animationen.

Allein mit einigen Rechnern und einer grossen Leinwand bestückt, wäre unser Stand zwar vielleicht schon halbwegs ansprechend gewesen, sofern die geplanten Vorführungen, Bilder und Animationen gewesen wären, aber das war uns doch etwas zu wenig. Da man Bilder auch wunderbar auf anderen Medien präsentieren kann, also auch auf Papier, entschieden wir uns dafür, nicht nur elektronische Mittel einzusetzen, sondern ganz im Werbesinne auch herkömmliche Poster zu benutzen. Zu jedem unserer Bilderzeugungsmodule sollte nach Möglichkeit eines vorhanden und auf dem Projektstand aufgehängt sichtbar sein.

Selbst das war uns noch zu wenig. Um Werbung für unseren Stand zu machen und um ein wenig plastisch zu verdeutlichen, wie die dreidimensionalen Modelle auf der Leinwand in Wirklichkeit aussehen, entschlossen wir uns plastische Modelle zur Dekoration unseres Standes aufzuhängen. Diese sollten, wie alles Andere ebenfalls, noch in der kurzen Zeit vor dem Projekttag fertiggestellt werden.

Zwar hätte man auch einfache Flugblätter verteilen können, um auf den Stand aufmerksam zu machen, aber wir hatten uns tatsächlich noch vorgenommen, Faltblätter mit von uns erzeugten Bildern und einer kurzen Vorstellung des Projektes anzufertigen. Auch diese mühevolle Kleinarbeit musste also bis zum Projekttag erledigt werden.

Eine Sache hatten wir noch, die man auch in die Sparte Werbung und Repräsentation stecken kann, allerdings war diese mehr dazu gedacht unseren Gemeinschaftsgeist zu verdeutlichen. Das “Wir-Gefühl”, wie man so schön sagt, sollte mit dem Tragen eines gemeinsamen T-Shirts auch den Besuchern unseres Standes demonstriert werden. Das geplante T-Shirt, das zum

einen ein für unser Projekt typisches Bild enthalten sollte, sollte ebenso den ANIMA-Schriftzug enthalten, also unser Logo in groß. Dieses T-Shirt, mit anderen Worten die Gestaltung und die Beauftragung einer Druckerei mit dem Drucken, musste ebenfalls bis zum Projekttag fertig werden. Dies galt allgemein gesehen für unsere Drucksachen. Auch die Poster und Flyer mussten rechtzeitig gestaltet und in Druck gegeben werden.

Von unserer ANIMA-Werbekampagne mal abgesehen, gab es natürlich noch weitere Arbeiten, wie z.B. die Planung und der Aufbau des Standes selbst, oder die Vorstellung des Projektes bei der allgemeinen Vorführung der Projekte. Gemeint ist damit, dass jedes Projekt ca. eine Stunde lang Zeit hat, sich und seine Arbeit den anderen Projekten und Besuchern vorzustellen. Es hat sich dabei schon fast zur Tradition entwickelt eine Art Game- bzw. Talkshow zu veranstalten oder aber eine Art Bühnenstück aufzuführen.

Innerhalb unseres Projektes hatten wir uns vorgenommen ein Theaterstück aufzuführen, das den Ablauf innerhalb unseres Programms AnimaLab verdeutlichen sollte, wenn ein Benutzer mit der ihm zur Verfügung stehenden Benutzungsoberfläche ein Bild erzeugt. Man kann mit Recht behaupten, dass die Schwierigkeiten, die hiermit in Zusammenhang standen, die ungewöhnlichsten während des gesamten Projektes waren. Zum einen galt es die Requisiten zu besorgen, zum anderen Darsteller für die Aufführung zu motivieren, aber vor allem musste man auch noch die üblichen Probleme des Schauspielens überwinden. Diese waren zum Beispiel Lampenfieber, die Angst sich lächerlich zu machen, Ausfälle von Schauspielern bei den Proben oder auch das Fehlen von Erfahrungen mit der Bühnengestaltung, der Ausleuchtung, der Akustik und ähnlichen Dingen. Und von diesen Problemen mal abgesehen, die schon übel genug waren, kam das grösste Problem der Schauspielerei immer noch erschwerend hinzu, die Zeit, denn wie bereits angedeutet musste auch unser Theaterstück innerhalb der kurzen zwei Wochen noch fertig geprobt und organisiert werden.

Die knappe Stunde, die jedes Projekt zur Vorstellung hat, war also zum größten Teil mit dem Theaterstück verplant. Für den Rest wollte man aber noch mal einen drauf setzen, wie man so schön sagt. In Anbetracht dessen, dass kaum jemand glaubte, in so kurzer Zeit ein erstklassiges Bühnenstück auf die Beine zu stellen, entschied man sich, um die Enttäuschung der Zuschauer in Grenzen zu halten, noch einen Film vorzuführen. Dieser Film sollte eine Animation zeigen, wie sie von unserem System im wesentlichen erzeugt wird. Zusätzlich sollte er noch mit diversen Effekten aufgewertet und von einer Hintergrundmusik spannungssteigernd untermalt werden. Es war allen klar, dass dieser Film noch sehr viel Arbeit machen würde und vor allem nur noch knapp in der Zeit zu schaffen sein würde. Die Vorführung, die wohl mit Abstand den größten Beitrag zum Aufbau des Images des Projektes ANIMA leisten sollte, wollte man nebenbei gern auf Video festhalten. So musste also kurzfristig das nötige Equipment besorgt werden und die Hürde genommen werden, dass sich keiner mit der Videofilmerei auskannte, der nicht auf der Bühne stehen sollte. Aber auch dieses Problem wollte man noch lösen.

6.2 Die heiße Phase

Wie bereits erwähnt hat sich das Meiste während der Zeit kurz vor dem Projekttag ereignet. In den zwei Monaten davor waren viele Gruppen mit ihren Programmierarbeiten erheblich vorwärts gekommen. Das Gesamtsystem war zwar schon vorhanden, aber es lag quasi noch in Einzelteilen vor uns. Leider fehlte bis dahin auch noch die Oberfläche, die zumindest optisch den Anschein erweckt, dass man ein zusammengehöriges System hat.

Mit der Angst im Nacken, am Projekttag kein vorführbares Gesamtsystem zu haben, machte man sich daran diese Schwierigkeiten aus dem Weg zu räumen. Es wurde kommuniziert, programmiert und resigniert. Es gab tatsächlich einige Dinge, die generell als leicht erschienen, aber partout nicht funktionieren wollten. Dennoch wurde einiges an Arbeitszeit und Nerven investiert, sodass besonders in den zwei Wochen nahezu alles fertiggestellt wurde, das für die Vorführung nötig war, um wenigstens den Anschein zu erwecken, als hätte man in den vergangenen eineinhalb Jahren durchgängig mit Fortschritten auf den Projekttag hingearbeitet. Wer schon mal beobachtet hat, wie innerhalb der Softwareprogrammierung kurz vor dem Fertigstellungsdatum noch am System gearbeitet wird, der kann sich eine Vorstellung davon machen, was Stress sein kann. Und man kann bestimmt mit Recht behaupten, dass sich die Teilnehmer unseres Projektes dem Stress mit Erfolg gestellt haben.

Neben der Fertigstellung von AnimaLab liefen wie bereits angedeutet noch zahlreiche andere Arbeiten. Ein besonderer Stressfaktor war, wie beschrieben, der Film für die Projektvorstellung bzw. -vorführung. In mühevoller Kleinarbeit entstand bis quasi zur letzten Minute ein kleines Animationsmeisterwerk. So belastend es auch war, dass entweder die nötigen Programme fehlten oder aber kurzfristig noch aufhörten zu funktionieren, weil gerade an dem Tag die Probezeit ablief, so befriedigend war es doch, zu sehen, dass sich der Stress und die Liebe in die vielen kleinen Details doch gelohnt hatten. Schlaflose Nächte und zahlreiche Geduldsproben waren nötig gewesen, aber sie würden bestimmt bald mit dem Applaus vergessen werden, da war man sich sicher.

Während die einen am System oder an dem Animationsfilm bastelten, widmeten sich andere den Beispielen, die am Stand zu bewundern sein sollten. Zahlreiche syntaktische Gebilde wurden extra für den Projekttag geschaffen und zur Vorführung auf der Leinwand vorbereitet. Auch die dreidimensionalen Deko-Sierpinski-Pyramiden wurden wie geplant in Angriff genommen und tatsächlich fertiggestellt. Jetzt fehlten also eigentlich nur noch die Poster, die Leinwand und der Projektor.

Die restliche Hardware zu besorgen, war kein grosses Problem, dessen sich einige annahmen, allerdings machten die Poster doch einige Schwierigkeiten. Zum einen sollten sie an der Uni gedruckt werden und zum anderen auch noch rechtzeitig. Ausserdem war man sich auch noch gar nicht so sicher, welche Motive man haben wollte und wie viele Poster überhaupt noch rechtzeitig ge-

staltet und fertig werden würden. Dieses Problems nahm man sich ebenfalls an. Bis kurz vor dem Projekttag entstanden zu den einzelnen bilderzeugenden Modulen Poster, die typisch für die Bilder waren, die mit den Modulen herstellbar sind. Natürlich wurden auch diese noch in mühevoller Kleinarbeit durchgestylt und aufgewertet.

Das Drucken machte dann doch noch einen üblen Strich durch unsere Rechnung. Aufgrund der begrenzten Druckmöglichkeiten entstanden Wartezeiten beim Drucken. Andere Projekte wollten natürlich auch zu dem Zeitpunkt schnell noch einige Poster haben. Wir entschlossen uns also, schon allein wegen der schönen Bilder und der mangelnden Druckqualität, unsere Poster bei einer richtigen Druckerei anfertigen zu lassen. Wie sich heraus stellte, waren die Poster nicht nur in einer hervorragenden Qualität, sondern sahen richtig gut aus in den Bilderrahmen, die wir extra noch dafür besorgt hatten.

Alles Nötige für den Stand war also in die Wege geleitet worden und mit Schwierigkeiten, aber immerhin rechtzeitig, fertiggestellt worden. Es blieben also nur noch die T-Shirts, die Flyer, die Theateraufführung und die Videoaufnahme, die es parallel zu den anderen Arbeiten noch zu organisieren gab.

Die T-Shirts wurden, da das Motiv für die T-Shirts relativ schnell gefunden war, von einigen in Angriff genommen. Kaum war die Gestaltung erledigt, wurden sie auch schon in Druck gegeben. Auch wenn dies ebenfalls erst in der kurzen Zeit vor dem Projekttag geschah, so wurden sie doch rechtzeitig fertig. Die Werbeflyer zum Verteilen sollten ebenfalls wie die T-Shirts und die Poster in Druck gegeben werden. Das Problem war allerdings, dass noch nicht feststand, was auf den Flyern stehen bzw. abgebildet sein sollte. Mit einigem Stress, der sozusagen schon Dauergast bei uns war, entstand gerade noch rechtzeitig der Inhalt, sodass der Druck, wenn auch spät, aber immer noch rechtzeitig, in Auftrag gegeben werden konnte. Für die Videoaufnahme unseres Theaterstücks fand sich mehr oder weniger freiwillig schnell jemand. Für die Aufführung selbst war das schon schwieriger. Viele fanden die Idee gut, aber nur wenige waren bereit sich auf die Bühne zu stellen, und sich eventuell lächerlich zu machen, was zum Glück nicht eintrat.

Unser Theaterstück sollte, wie bereits erwähnt, den Ablauf unseres Systems verdeutlichen. Allgemein gesehen war das schon in Ordnung, aber noch etwas wenig, um damit die Zeit zu füllen. Letztendlich wurde zu dem eigentlich eher trockenen Ablauf der Bilderzeugung durch einen Benutzer eine Art fiktiver Rückblick auf die Teilnehmer des Projektes hinzugefügt, der auf lustige Weise auf die Charakterzüge der entsprechenden Projektmitglieder hinweisen sollte. Die Planung, die Requisitenorganisation, die Proben und das Drehbuch wurden also auch noch innerhalb der heißen Phase durchgeführt. Die Einsatzbereitschaft und der Wille noch rechtzeitig ein halbwegs gutes Theaterstück auf die Beine zu stellen, war enorm groß.

Während mehrerer Proben, die relativ stressig waren, behielten alle Schauspieler unseres Projektes die Nerven und hatten gelegentlich sogar noch einen Scherz auf den Lippen. Es entstand ein annehmbares Bühnenstück, mit dem

man mit ruhigem Gewissen, zur Vorführung antreten konnte. Sogar kleinere Ausfälle bei den Proben, die zur Folge hatten, dass man nicht immer mit allen Schauspielern den Text durchgehen konnte, der sich ja eigentlich nebenbei noch ständig änderte, waren letztendlich doch kein Problem. Mit Improvisationstalent und Selbstvertrauen wurden auch die Schwierigkeiten der Schauspielerei überwunden.

Wenn man jetzt abschließend noch mal über die zwei Wochen vor dem Projekttag nachdenkt, dann fallen vor allem einige Dinge auf, die fast selbstverständlich da waren, aber dennoch hätten Schwierigkeiten machen können, wie z.B. die Arbeitseinteilung. Was oftmals das übliche "Wer sich zuerst rührt, hat verloren!"–Spielchen war, war in den meisten Fällen kaum noch ein Problem. Wenn eine Aufgabe entdeckt wurde, fand sich eigentlich schnell jemand, der sie erledigte. Der gemeinsame Wille, am Projekttag nicht zu versagen, war doch sehr groß.

Dann wäre da auf jeden Fall nochmals die finanzielle Unterstützung zu erwähnen, denn kleinere Beträge wurden eigentlich generell wohlwollend über unsere Projektleitung beglichen. Für andere Dinge, die über Arbeitsmaterial und notwendige Dinge hinausgingen, wurde nahezu selbstverständlich etwas Geld von uns Teilnehmern locker gemacht. Sogar dabei haben wir noch kleinere Zuschüsse über unsere Projektbetreuer bekommen. An dieser Stelle gleich nochmals danke dafür!

Geld- und Materialsorgen hatten wir also allgemein gesehen keine. Aber auch die üblichen Probleme, die wir bisher hatten, verschwanden fast von selbst. Die Kommunikationsbereitschaft war hoch wie nie. Anwesend war jeder der Zeit hatte. Und diejenigen, die notwendigerweise arbeiten mussten oder Dringendes zu erledigen hatten, waren zumindest die restliche Zeit über da. Wie gewohnt, waren auch unsere Projektbetreuer mit Rat und Tat da, und machten sich nicht aus dem Staub als es stressig wurde. Man kann also sagen, dass alles vorhanden war oder gut geklappt hat, das nicht unmittelbar mit zu unseren selbst verursachten Stressfaktoren gehörte.

6.3 Der Tag X

Früh am Morgen begann der übliche Stress. Man könnte sagen für fast alle Teilnehmer, denn für einige hatte er noch gar nicht geendet. Für die meisten Teilnehmer jedenfalls begann der Tag, wie verabredet etwas früher als sonst, um die restlichen organisatorischen Massnahmen in Angriff nehmen zu können. Der Projektstand musste aufgebaut und dekoriert werden. Das System, das aus mehreren Rechnern bestand, musste aufgebaut, vernetzt und nochmals kurz getestet werden. Man hatte schließlich vor, nicht mehr als nötig dem Zufall zu überlassen.

Die letzten Arbeiten an unserer Software und dem Animationsfilm wurden noch getätigt. Eigentlich dürfte mittlerweile jedem klar sein, der den Abschnitt über die heiße Phase unseres Projektes gelesen hat, welcher Stress vorgeherrscht hatte und welche Kleinarbeit letztendlich die Fertigstellung bis

kurz vor die Vorführung herausgezögert hatte. Dennoch möchte ich an dieser Stelle nochmals darauf zurück kommen. Der Anblick, der sich am Morgen des Projekttages bot, war zunächst einmal etwas entmutigend. Zum Teil wurde noch am System und am Video gearbeitet und nebenbei bereits die Hardware abgebaut, die für die Vorführung am Projektstand benötigt wurde. Es sah doch alles recht chaotisch aus. Zwischendurch liefen zahlreiche Teilnehmer im Projektraum umher und holten Dekorationsteile, Drucksachen und Tische und Stühle heraus. Der Anblick, der sich zeitweise bot, glich den von unserem Turmiten-Modul erzeugten Animationen, quasi einem Gewusel von Teilnehmern, die wie wild durcheinanderliefen, aber dennoch ein stricktes Ziel verfolgten.

Es gab während des Projekttages, wie bereits angedeutet, zwei Arbeitsbereiche für die Teilnehmer, in denen sich im Verlauf des Tages alles weitere abspielen würde. Die einen sollten sich nach einem vorher festgelegten Zeitplan um die Betreuung unseres Standes kümmern, und die anderen um die Theateraufführung.

Nachdem die letzten Arbeiten im Projektraum teilweise widerwillig beendet wurden, widmeten sich zunächst fast alle den abschliessenden Arbeiten am Projektstand. Bereits dabei fielen wir Teilnehmer durch unsere einheitlichen und farbenfrohen T-Shirts auf. Einige Besucher schauten uns sogar schon bei den Aufbauarbeiten und ersten Systemtests über die Schulter. Ebenso fruchteten auch unsere weiteren Werbemaßnahmen. Die angebrachten Poster und Dekorationsobjekte und natürlich die ersten Bilder auf der großen Leinwand, lockten bereits erste Besucher an unseren Stand. Da die erforderlichen technischen Geräte und das benötigte Zubehör vorhanden waren, gab es auch kaum etwas das beim Aufbau unseres Standes schief ging. Es war alles da was gebraucht wurde und genügend Leute, die mithalfen.

Ebenso war es an dem zweiten Hauptschauplatz dieses Tages. Die Bühnenvorbereitungen liefen auf Hochtouren. Letzte Änderungen an der Beleuchtung und dem Ton wurden vorgenommen und ein letztes Mal in Gedanken geprobt. Das Lampenfieber war zwar stark, aber wich mit der Zeit nahezu einer routinemäßigen Professionalität. Das mag zwar etwas übertrieben klingen, doch dafür, dass Informatiker ein kurzfristiges Theaterstück auf die Beine gestellt hatten und nun vorführen wollten, lief alles doch widererwartend reibungslos.

Nachdem sich das erste Projekt vorgestellt hatte, unsere gedruckten Werbeflyer auf den Sitzplätzen verteilt waren und die Garderobe ein letztes Mal zurecht gerückt worden war konnte dann endlich der Projekttag für das Projekt ANIMA beginnen.

Die Zuschauer, die sich zu recht früher Stunde zu uns verirrt hatten, waren zunächst etwas verblüfft. Das Theaterstück schien nichts mit dem Inhalt unseres Projektes zu tun zu haben. Stattdessen waren es eher nicht ganz ernst gemeinte Anspielungen und Witze über die Projektteilnehmer, die dem Zuschauer geboten wurden. Doch obwohl viele Anspielungen eher nur von uns Teilnehmern richtig gedeutet werden konnten, kamen sie überwiegend als

amüsante Witze rüber und fanden auch dementsprechenden Anklang beim Publikum.

In etwa zur Mitte unserer Aufführung hin wurde das Stück abgebrochen und das eigentliche AnimaLab vorgestellt. Die zwei Hauptdarsteller, die sozusagen den Rahmen mit ihrem Dialog bilden sollten, wurden um weitere Darsteller ergänzt und unser System in seinem Ablauf vorgestellt. Besonders positiv fiel auf, dass die Schwierigkeiten der Generalprobe des vorigen Tages nahezu alle beseitigt waren, und dass der Ablauf um einiges reibungsloser war. Die eingebauten komischen Elemente kamen ebenfalls beim Publikum an. Man kann also durchaus behaupten, dass sich keiner bei der Aufführung lächerlich gemacht hat. Am Ende wurde dann durch die beiden Hauptdarsteller der Dialog wieder aufgenommen und im bereits bekannten Stil weiter gespielt.

Der Animationsfilm, der so viele Nerven und Zeit gekostet hatte, verdeutlichte das Wachstum eines Baumes, so wie es quasi mit Hilfe unseres AnimaLab simuliert werden kann. Durch die dreidimensionale Darstellung, die speziellen Lichteffekte, die eingesetzte Rotation und die musikalische Untermalung wurde er zu einem beeindruckenden Schauspiel. Die investierte Mühe hatte sich mehr als gelohnt, denn man könnte sicherlich ohne zu lügen behaupten, dass er eines der Highlights des Tages war. Mit Sicherheit hatten nicht nur wir, sondern auch die zahlreichen Besucher unserer Vorführung noch den ganzen Tag über die Melodie des Zauberlehrlings als Ohrwurm im Gedächtnis.

Generell wurde von vielen bestätigt, dass die Theateraufführung und der Film unseres Projektes sehr gelungen waren. Etwas kritischer waren wir Teilnehmer da schon, doch auch wir hatten von vielen Dingen, die wir als problematisch eingestuft hatten, den Eindruck, dass sie super gelaufen waren. Hierzu zählte unter anderem auch der Text. Wo bei den Proben noch üble Lücken waren oder Patzer auftraten, da war auf einmal alles in Ordnung oder im Handumdrehen durch Improvisation gerettet.

Nach der Aufführung und dem zwischengeschobenen Film wurde durch unseren Moderator nochmals kurz Werbung für unseren Stand gemacht. Diese verfehlte ihre Wirkung nicht, denn noch mehr Besucher als vorher drängten sich um unseren Stand. Unsere Standbetreuer, die leider nicht den Film und die Aufführung sehen konnten, hatten plötlich doch einiges mehr zu tun. Zahlreiche Anfragen bezüglich der Bilder, des Films und unseres AnimaLab, mussten beantwortet werden. Wer jetzt allerdings den Eindruck gewonnen hat, dass wieder enormer Stress auf uns und insbesondere auf die Standbetreuer herein brach, der irrt sich. Auch hierbei verteilte sich die Arbeit recht schnell. Da viele von uns in der Nähe des Standes waren und wir durch die T-Shirts gut erkennbar waren, eröffneten sich unseren Besuchern quasi mehr als genug fachkundige Anlaufstellen, um den Wissensdurst zu stillen. Dies entlastete die Standbetreuer doch schon um einiges.

Betrachtet man den Projekttag insgesamt, dann lief er doch sehr gut für uns. Die vergleichenden Blicke zwischen den einzelnen Projekten, hatten für gewöhnlich immer dasselbe Ergebnis. Unser Stand war nahezu optimal einge-

richtet. Es war genügend Platz für zahlreiche Besucher da, um sich gleichzeitig umsehen und informieren zu können. Der Stand lag strategisch günstig, denn zum einen lag er in der Nähe des organisierten Buffets, und zum anderen leicht erhöht, was beides zusätzliche Werbewirkung ausmachte. Ausserdem war er leicht zugängig und lag in der Verlängerung eines Ausgangs des Theaterraumes. Jeder, der eine Projektvorstellung gesehen hatte, lief mit einer Wahrscheinlichkeit von 50 Prozent genau auf unseren Stand zu. Die auffälige Standdekoration lockte ebenfalls bereits von weitem und verfehlte ihre Wirkung nicht. Zugegebenermaßen gab es eigentlich nur einen Stand der noch etwas mehr auffiehl, da er sich sozusagen in Dunkelheit hüllte und durch das Geheimnisvolle lockte. Allerdings war dieser schnell überfüllt und nicht unbedingt leicht zugänglich. Dennoch war er in Punkto Werbewirksamkeit doch ein gewisser Konkurent.

Ein nicht unwichtiger Punkt, den ich sozusagen abschliessend erwähnen möchte, ist der, dass unser Stand nicht nur durch sein Marketingkonzept glänzte, sondern auch alles hielt, was unsere Werbung versprach. In Anbetracht dessen, was wir in den letzten zwei Jahren und besonders in den letzten Wochen getan hatten, war der Projekttag mehr als ein großer Erfolg, der uns von vielen Seiten bestätigt wurde, und er hätte auch kaum besser laufen können. Allerdings machte uns der Gedanke daran, was wir hätten erreichen können, wenn wir nur etwas früher mit unserer Projekttagsarbeit angefangen hätten, doch etwas nachdenklich und ließ uns teilweise sogar ins Schwärmen geraten.

Den Projekttag selbst haben wir nach den nötigen Abbauarbeiten ruhig in einer geselligen Runde ausklingen lassen, wobei uns unsere Projektleitung noch großzügig eingeladen hatte. Schon allein deshalb hatte sich der Projekttag und die damit verbundenen Strapazen gelohnt. Und ein schönes Andenken nahm jeder mit dem tollen T-Shirt auch noch mit nach Hause.

7 Der Projektbericht

7.1 Unsere Vorhaben

Zu allererst soll zum Projektbericht erwähnt werden, was wir Teilnehmer für Möglichkeiten hatten, unseren Projektbericht anzufertigen und welche Ausgangsituation für unsere Vorhaben gegeben war.

Zunächst war völlig unklar, wie ein Projektbericht auszusehen hatte. Einige von uns hatten zwar schon mal einen Projektbericht von weitem gesehen, aber eigentlich auch keine richtige Ahnung, wie man ihn formell anzufertigen hat. Das war auch eigentlich gar nicht möglich, denn wirklich fest vorgeschrieben scheint das gar nicht zu sein. Bedingung für den Projektbericht ist nur, und das scheint auch das Entscheidende zu sein, dass ein Dokument nach Beendigung der Projektlaufzeit abgegeben wird, das die Leistung der Projektteilnehmer und die zweijährige Arbeit widerspiegelt. Um den sogenannten Projektschein, also einen Leistungsnachweis zu bekommen, muss

ein Teilnehmer am Projektbericht mitgearbeitet haben. So seltsam es auch klingt, aber theoretisch könnte man zwei Jahre lang nichts für das Projekt tun und dennoch am Ende durch Abgabe eines guten Textes erfolgreich am Projekt teilgenommen haben und den Projektschein mit nach Hause nehmen. Dementsprechend könnte ein Projektbericht letztendlich die seltsamsten Formen haben, solange er nur die genannten Kriterien erfüllt, also eine Bewertung möglich macht.

Aufgrund unserer Projektleitung, die uns kein festes Aussehen des Berichtes vorschrieb, standen uns alle Möglichkeiten offen. Wir erkundigten uns nach Berichten anderer Projekte und stießen dabei auf diverse Dokumente in Papierform, aber auch auf ein interessantes Programm.

Dieses Programm dient zur automatischen Erstellung von Dokumenten in druckbaren aber auch rein elektronischen Formaten, die nicht zum Druck gedacht sind. Dem System liegt ein SGML-Dokument in reiner Textform zugrunde, das mit Hilfe von Auszeichnungselementen gekennzeichnet wird und anhand dieser automatisch nach bestimmten Regeln in diverse Formate übertragen wird. Mit anderen Worten hatten wir ein sogenanntes Tool gefunden, bei dem sich nur wenige über das Layout Gedanken machen mussten, nämlich die, die das SGML basierte System verwalteten. Alle Anderen konnten fast einfach drauflos schreiben.

Wenn eben das Wort "fast" verwendet wurde, dann eigentlich nur deshalb, weil man sich zwar nicht mehr über das Layout Gedanken machen musste, aber dennoch mit einigen Schwierigkeiten zu kämpfen hatte. Das Tool war nämlich noch nicht ganz ausgereift, schwierig zu administrieren, oftmals nur unzureichend dokumentiert und war sehr intolerant, was bestimmte Auszeichnungselemente oder "unsichtbare" Fehler anging, wie z.B. ein Zeilenwechsel an der falschen Stelle. Besonders diejenigen von uns, die nicht so oft an der Uni ihre Texte schrieben oder andere Betriebsysteme und Programme zur Bearbeitung bevorzugten, hatten es nicht gerade leicht, sich an das System zu gewöhnen.

Trotz der Skeptiker, der Schwierigkeiten und der geteilten Meinungen entschied man sich dennoch, das Aktensystem zu benutzen. Einige von uns glaubten zwar weiterhin bis zum Ende des Projektes, dass man den Bericht auch gut auf unkompliziertere Weise hätte erstellen können und sich die vielen Probleme, die sich durch das verwendete Aktensystem ergaben hätte sparen können, aber letztendlich hatten sich wieder mal die wenigen Befürworter gegen die wenigen Skeptiker durchgesetzt, weil es dem Rest mal wieder völlig egal war.

Man brauchte allerdings noch mehr als nur ein System, das gemeinsames Arbeiten ermöglicht, sodass ein einheitliches Dokument entsteht. Man brauchte auch noch den entsprechenden Inhalt. Und der Inhalt war natürlich das Hauptproblem, denn hier stellte sich wieder die Frage, wie unser Bericht denn nun sein sollte.

Wir hatten, wie bereits angedeutet, einige Berichte von anderen Projekten gesehen. Diese waren allerdings sehr unterschiedlich. Einige waren ähnlich wie

eine Programmdokumentation gehalten. Das soll heissen, dass weniger auf theoretische Dinge Wert gelegt wurde, sondern mehr auf die Beschreibung des entstandenen Programms. Andere Berichte legten mehr Wert darauf die, Theorie in den Vordergrund zu stellen. Und wieder andere Berichte waren mehr von den Erkenntnissen beeinflusst, die während der Projektarbeit bezüglich einiger Themen gewonnen wurden.

Da wir uns noch nicht schlüssig darüber waren, wie wir unseren Bericht inhaltlich füllen wollten, und da ja auch noch viel Zeit war, entschieden wir uns zunächst, nur die wöchentlichen Protokolle und gelegentlich nebenbei entstehenden Dokumente im Aktensystem zu erfassen und zu sammeln. Das Aktensystem sollte schließlich ganz automatisch einen Projektbericht aus den erfassten Dokumenten erstellen können.

Ungefähr nach dem ersten Projektjahr beschäftigten sich dann einige erneut damit sich Gedanken über den Projektbericht und seinen Inhalt zu machen. Obwohl der Großteil der Teilnehmer hierfür immer noch keine rechte Notwendigkeit sah, entstand dennoch ein Ansatz, der sich von seinem Konzept her nur wenig bis zum Ende hin geändert hatte.

Das Ziel sollte es sein, einen Projektbericht zu erstellen, der zunächst einmal seinem Namen gerecht werden sollte und ausserdem die Theorie und unser Endprodukt beschreiben sollte. Als kleinen Zusatz sollte der Bericht auch noch eine Bildersammlung umfassen, um einige unserer erzeugten Bilder und somit die Möglichkeiten unseres AnimaLab vorzustellen. Der Bericht sollte also vor allem die Projektarbeit mit all ihren Höhen und Tiefen dokumentieren und unsere Erfahrungen widerspiegeln bzw. weitergeben und ebenso die theoretischen Erkenntnisse beinhalten und unser Programm beschreiben und seine Funktionsweise erklären.

Entscheidende Fortschritte in Richtung Projektbericht wurden nach dem zweiten Projektseminar gemacht, denn ein ausgearbeitetes Konzept bildete von da an eine Grundlage für die Erstellung des Gesamtberichtes.

7.2 Die Endphase des Projektes

Das erste ausgearbeitete Konzept für unseren Projektbericht beinhaltete sowohl eine Grobaufteilung, die die inhaltliche Aufteilung des Berichtes in seine Teile widerspiegelte, als auch detailliertere Aufteilungen innerhalb der Hauptteile, die sowohl als inhaltliches Ziel des Berichtes angesehen werden konnten oder aber als Grundlage, um Schwerpunkte setzen und Erweiterungsmöglichkeiten erkennen zu können.

Um den Projektbericht mit Inhalt zu füllen, musste ein Weg gefunden werden, um Material zu sammeln. Da in dem verwendeten Aktensystem nur wenige brauchbare Dokumente vorlagen, die zusätzlich auch nicht unbedingt in ihrer Form für den Endbericht geeignet waren, musste quasi von Grund auf neu angefangen werden. Aufgrund der Tatsache, dass die meiste Arbeit noch bei der Implementierung des AnimaLab anfiel, waren nur wenige überhaupt

bereit sich nähere Gedanken zum Projektbericht zu machen, der der meistverbreitetsten Ansicht nach noch mehr als genug Zeit hatte. Dennoch machten sich einige vorsorglich Gedanken und entwickelten eine Möglichkeit, von allen Teilnehmern wenigstens schon kleine auswertbare und für den Projektbericht zusammenstellbare Beiträge zu sammeln. Wenn auch noch nicht viel in Richtung theoriebezogene Dokumente vorlag oder in Richtung AnimaLab-Dokumentation, so sollte auf diesem Weg wenigstens schon der Inhalt für den eigentlichen Projektbericht zusammenkommen. Diese allgemeine Projektauswertung war nicht unbedingt an den Projektzeitplan gebunden und konnte bereits frühzeitig begonnen werden, sodass der Inhalt nicht erst am Ende der Projektlaufzeit zusammengesucht werden musste. Dies bedeutete natürlich eine Entlastung für die für gewöhnlich ziemlich stressige letzte Phase des Projektes.

Zur Füllung des Gesamtberichtes mit Inhalt, und um die verschiedenen Themen und nötigen Beiträge auf geordnete Weise zusammenzufassen und zu verwalten, wurde eine WWW-basierte Umfrage ausgearbeitet und realisiert. Über diese Internetseiten war es möglich alle Projektmitglieder zu spezifischen Themen zu befragen. Durch die Benutzung der Onlineumfrage konnte man einige Vorteile ausnutzen. Da sie jederzeit abrufbar war, war es zum einen jedem selbst überlassen, wann er die Fragen innerhalb eines gesetzten Zeitraumes beantwortet. Es musste also kein Extratreffen organisiert werden, um so etwas wie eine Fragestunde abzuhalten. Zum anderen konnte man neu hinzugefügte Fragen daran erkennen, dass sie als "nicht beantwortet" gekennzeichnet erschienen. Dies ermöglichte es quasi, Themenerweiterungen und zusätzliche Aspekte zur aktuellen Berichtstruktur hinzuzufügen, die dann anhand des Fragenkatalogs in gewisser Weise automatisch mit Inhalt gefüllt wurden. Als Voraussetzung hierfür und für das parallel zur sonstigen Arbeit laufende Füllen des Gesamtberichtes, war natürlich ein gewisses Engagement aller Teilnehmer und auf freiwilliger Basis auch der Betreuer nötig. Nur unter der Voraussetzung, dass die Fragen möglichst von allen beantwortet werden und dass gelegentlich auf Aktualisierungen geachtet wird, ist eine solche Art der Materialbeschaffung möglich.

Wie bereits erwähnt hatten sich in erster Linie drei Teile unseres Projektberichtes ergeben, die natürlich auch bei dem Online-Fragenkatalog die Grundlage bildeten. Als Versuch, von dem man noch nicht sicher war, ob er überhaupt etwas bringt, war die Befragung zum Bereich Programmdokumentation gedacht. Im Ansatz war eine Befragung zu den Wünschen und Vorhaben der einzelnen Implementierungsgruppen in Richtung Dokumentation durchaus sinnig, da man sich vor allem davon versprach, dass eine Übersicht entsteht, die es ermöglicht die verschiedenen Dokumentationsinhalte so aufeinander abzustimmen, dass eine einheitliche Gesamtdokumentation entsteht. Die Option, dass der Inhalt geringfügig anders sein kann, schon allein vom Schreibstil der einzelnen Autoren her, wurde so offen gelassen. Uns reichte es, wenn das Gesamtdokument vom äusserlichen Aufbau her einheitlich war und eine gewisse inhaltliche Gleichstrukturierung der Einzelbeiträge vorlag.

Durchaus vielversprechender war es, den Theorieteil (ANIMA-Methods) unseres Projektberichtes mit Hilfe der Onlinebefragung mit Inhalt zu füllen bzw. für ihn eine Strukturierung zu finden. Da die Phase der Aneignung von theoretischen Grundlagen im Grunde genommen abgeschlossen war, sollte eigentlich jede Gruppe, die etwas implementierte, genau wissen, was sie an theoretischem Inhalt dem Gesamtbericht hinzufügen würde. Genau wie bei dem praktischen Teil sollte durch die Befragung Inhalt zusammenkommen, der den allgemeinen Rahmen des Theorieteils bilden könnte und schon frühzeitig eine Vereinheitlichung ermöglichen sollte. Das Ziel der Befragung war schließlich, nicht wie so viele andere Projekte den gesamten Stress mit dem Projektbericht am Ende zu haben und eventuell sogar noch über das offizielle Projektende hinaus, sondern einen frühzeitigen Beginn des Endberichtes zu ermöglichen.

Wenn auch eine Onlinebefragung bei den zwei anderen Hauptteilen nicht hundertprozentig erfolgversprechend war, so war sie es bei dem vorliegenden Projektteil doch schon eher. Da es sich bei diesem Teil hauptsächlich um allgemeine Dinge des Projektes drehen sollte, um Arbeitsweisen, Meinungen, Grundlagen, und vieles mehr, konnte man viel auswertbares Material erwarten. Die Vermutung, die sich auch bestätigte, war vor allem, dass man diesen Teil als ersten mit Inhalt füllen können würde, und dass dieser vermutlich inhaltlich umstrittenste Teil des Berichtes genügend Zeit haben würde um noch heiß diskutiert zu werden.

7.3 Der fertige Bericht

Betrachtet man die Intention, die hinter dem Online-Fragenkatalog stand, dann könnte man meinen, dass ein Grossteil des Berichtes sich fast von selbst ergeben hatte und dass für die einzelnen Teile schon mehr oder weniger Inhalt vorhanden gewesen sein muss nachdem die Teilnehmer die Fragen beantwortet hatten.

Dem war nur bedingt so. Leider hatte die Arbeitsmoral, die unserem Projekt schon so oft zum Verhängnis wurde, auch in diesem Fall wieder ein Hindernis dargestellt. Nur weniger als die Hälfte der Teilnehmer hatten beim zweiten Versuch nach Ablauf des ersten Beantwortungszeitraums einen Teil der Fragen beantwortet. Wie erwartet waren die Antworten zu den Dokumentationsfragen eher wenige an der Zahl, da die meisten Teile unseres AnimaLab noch implementiert wurden und sich eigentlich noch keiner über die Dokumentation Gedanken machte und auch nicht machen wollte. Diese Befragung war wirklich noch etwas verfrüht, aber immerhin einen Versuch wert. Und letztendlich war sie ja auch so ausgelegt, dass die Fragen, bei denen keine frühzeitige Beantwortung möglich war, auch später noch beantwortet werden konnten.

Auch die Befragung zu den späteren Inhalten des Theorieteils war wenig erfolgreich. Doch war es wenigstens aufgrund der Strukturierung der Fragen und der dahinter stehenden Themenbereiche und Gedanken möglich, sowohl

für den Theorieteil, als auch für den Dokumentationsteil ein gewisses Aussehen des späteren Inhalts abzuschätzen. Und indirekt wurde ja auch immerhin der Effekt erzielt, dass sich diejenigen, die sich mit den Fragen überhaupt beschäftigten, auch einige Gedanken zum späteren Bericht machten.

Wie erwartet stellte sich der meiste Erfolg bei dem Projektteil ein. Nahezu alle Antworten, die zu den Fragen kamen, waren überaus aufschlussreich und auswertbar und wurden auch fast vollständig von denjenigen beatwortet, die sich die Fragen angesehen hatten. Anhand der Antworten, die besonders bei den meinungsbezogenen Fragen in viele Richtungen gingen, konnten viele Erkenntnisse gewonnen werden, die letztendlich dazu führten, dass der vorliegende Bericht nicht nur beschönigend, sondern vor allem auch kritisch und somit hilfreich für andere verfasst werden konnte. Ob das letztendlich geklappt hat ist natürlich Ansichtssache des Lesers.

Aufbauend auf dem Stand des Aktensystems zur Erstellung des Gesamtberichtes und auf den durch die Umfrage gewonnen Inhalten sollte nach dem Projekttag von der bisherigen Einzelarbeit bezüglich des Projektberichtes zur allgemeinen Arbeit aller Projektteilnehmer am Bericht übergegangen werden. Der als erster Teil festgelegte Projektteil entstand mehr oder weniger ohnehin schon selbständig und bildete eigentlich kein größeres Problem. Für die anderen zwei Teile, die bisher nur grob geplant waren musste jedoch noch einiges getan werden, schon allein wegen der geringen Ergebnisse, die sich für diese Teile durch die Befragung angesammelt hatten. Zu diesem Zweck wurde wie bereits früher eine Redaktionsgruppe gebildet, die die Koordination und weitere Planung des Berichtes und des Drucks in Zusammenarbeit mit der Projektleitung übernahm. Das Aktensystem musste angepasst werden und weitere Strukturierung und Inhalt geplant und gesammelt werden.

Das Aktensystem funktionierte zwar allgemein gesehen, allerdings waren die Teile für die Berichterstellung noch nicht an die Bedürfnisse unseres Projektberichtes angepasst. Letztendlich machte diese Arbeit noch die meisten Schwierigkeiten, denn nötige Vereinbarungen und Absprachen waren bisher eigentlich noch nicht vorhanden, und im Endeffekt trudelten Wünsche oder Probleme auch nur völlig unkoordiniert ein. Das einfache Erstellen von Berichten, wie es quasi von den Befürwortern des Aktensystems versprochen wurde, stellte sich als doch nicht so einfach heraus. Gerade bei ausgefallenen Wünschen waren die Anpassungen nur schwer durchzuführen. Und für das Zusammenfügen des Gesamtberichtes musste dann doch noch einiges von Hand getan werden.

Auch wenn die letzten Bemerkungen nicht unbedingt positiv waren, so soll an dieser Stelle nicht der Eindruck erweckt werden, dass das Aktensystem völlig unbrauchbar war. Für ein reibungsloses Funktionieren war es nötig, im Gegensatz dazu, wie wir es gehandhabt haben, das System ständig zu warten, und den Anforderungen gerecht regelmäßig auf dem neuesten Stand zu halten. Da bei uns allerdings die bereits beschriebenen Schwierigkeiten vorherrschten und diejenigen, die sich ursprünglich für die Wartung zur Verfügung gestellt hatten, sich nicht mehr verantwortlich fühlten, war

das System zu einem gewissen Teil verwahrlost und es musste umso mehr getan werden. Und wäre das SGML-basierte Aktensystem besser dokumentiert gewesen, vor allem auch nach unseren Änderungen, dann wären viele Anpassungen und die Benutzung durch alle Projektmitglieder auch viel leichter gewesen. Oftmals scheiterte ein erfolgreiches Arbeiten nur daran, dass kleine undokumentierte Fehlermöglichkeiten eintraten und unerklärliche Fehler erzeugten, die letztendlich gar nicht so schlimm waren, wie es den Anschein hatte.

Abgesehen von den Arbeiten am Aktensystem liefen, wie bereits angedeutet parallel die Arbeiten am weiteren Inhalt des Berichtes. In Voraussicht auf das spätere Zusammenfügen wurden zur einheitlichen Gestaltung der Texte von der Redaktion einige Richtlinien herausgegeben, an die sich die einzelnen Autoren beim Erstellen ihrer Texte halten sollten. Unter Berücksichtigung der allgemeinen Richtilinien war es dennoch jedem möglich, seinen Beitrag zum Bericht inhaltlich frei zu gestalten und zu verfassen und trotzdem nicht gegen das Gesamtkonzept und gegen den einheitlichen Aufbau zu verstossen.

Da zu diesem Zeitpunkt jeder an dem Bericht arbeitete, entstanden so langsam die erforderlichen Dokumente, die gegenseitig gelesen und korrigiert wurden. Hierbei wurde allerdings in erster Linie Wert darauf gelegt, dass vor allem unsere Projektleitung, schon allein wegen ihrer Erfahrung und besseren Englischkenntnisse, die Dokumente quer gelesen hat. Letztendlich waren das ja auch die Texte, die als Bewertungsgrundlage des Projektscheines galten und in dem zu veröffentlichenden Gesamtbericht auftauchen sollten. Ehrlicherweise soll an dieser Stelle erwähnt werden, dass wir es nicht ganz geschafft haben, unser Ziel, den Bericht mit Ende des Projektes fertig zu haben, zu erreichen, allerdings ist er noch innerhalb des Folgesemesters fertig geworden und in Druck gegangen.

8 Das Projekt-Resümee

8.1 Unsere positiven Eindrücke

Grundsätzlich kann man zu unserem Projekt wohl auf jeden Fall sagen, dass es trotz der Probleme und Schwierigkeiten oder der gelegentlichen Unstimmigkeiten und Diskussionen auch durchaus Gutes gab, das uns auffiel.

Die positive Grundeinstellung, die wohl alle mitbrachten, fiel besonders auf. Jeder war bereit, dem Projekt zum Erfolg zu verhelfen. Zu der positiven Einstellung einer guten Projektgemeinschaft gehört mit Sicherheit die gegenseitige Achtung. Allgemein gesehen kann man schon behaupten, dass sich die Mitglieder unseres Projektes schon irgendwie respektierten. Wenn man sich auch nicht unbedingt gut mit jemandem verstand, so hörte man sich doch wenigstens an, was derjenige zu sagen hatte. Verschiedene Meinungen existierten natürlich. Diese wurden im allgemeinen angehört, besprochen, und bei Bedarf zur Abstimmung vorgeschlagen.

Die hohe Bereitschaft zur Diskussion könnte man innerhalb unseres Projektes durchaus als positiv werten. Wenn sich auch nicht jeder mit einem Problem beschäftigt hatte, so versuchte er durchaus innerhalb der entstehenden Diskussion noch einen produktiven Beitrag zur Lösung beizusteuern.

Wir Projektmitglieder waren natürlich nicht nur bereit zu diskutieren. Die Bereitschaft miteinander auszukommen war bei uns auch durchaus gegeben. In den kleineren Gruppen, die sich teilweise durch frühere Bekanntschaft oder aber auf Anhieb gut miteinander verstanden, war es eigentlich kein Problem miteinander auszukommen. In der grossen Gruppe war es nicht ganz so leicht, einen guten Draht zueinander zu bekommen. Zu diesem Zweck waren vor allem das erste, aber auch das zweite Seminar abgehalten worden. Man sollte miteinander ins Gespräch kommen, sich kennen lernen und versuchen mit den anderen Mitgliedern auszukommen, die schließlich alle unterschiedlicher Natur waren. Zwar hatten die Seminare in dieser Hinsicht nicht unbedingt den hundertprozentigen Erfolg gebracht, aber sie verliefen durchaus positiv und förderten das Arbeitsklima. Zu demselben Zweck wurde auch gelegentlich gemeinsam gefrühstückt oder auch mal gegrillt.

Als äußerst positiv wurde auch die lockere Projektführung empfunden. Durch die ungezwungene Arbeitsweise, die wir selber bestimmen durften, wurde die Kreativität gefördert. Ebenso kam man nicht unbedingt in Zeitdruck, sodass man sich nach eigenem Ermessen selber Druck machen konnte, wenn man ihn denn wollte. Dass dieses natürlich auch seine Nachteile hatte, soll an dieser Stelle nur noch mal kurz erwähnt werden, Näheres zur Erklärung stand schon bei der Beschreibung unserer Entwicklungsphase. Da nur ein geringer Druck gemacht wurde, könnte man natürlich vermuten, dass es mit der Arbeit innerhalb der Kleingruppen nicht geklappt hat. Das stimmt nur bedingt. Widererwartend gab es eine hohe Gruppendynamik und -kommunikation. Es wurde eifrig in den Gruppen gearbeitet, wenn es auch nach außen vielleicht nicht immer den Anschein hatte.

Betrachtet man den Umstand, dass wir es in der letzten Phase unseres Projektes für nötig hielten uns doch mehr Druck zu machen, um das Ziel bis zum gesetzten Zeitpunkt zu erreichen, dann sollte man auf jeden Fall die Organisationsgruppen erwähnen. Die Einführung hatte sich als sehr produktiv erwiesen. Zahlreiche Probleme, die während unserer Arbeit aufkamen, wurden durch die Organisationsgruppen behoben oder gemildert. Zwar hatten sie nicht den vollen erhofften Erfolg gebracht, aber mit Sicherheit haben sie noch mal alle dazu angeregt, gewissenhafter im Projekt zu arbeiten, und somit die Produktivität gefördert.

Was durchaus gut geklappt hat, auch ohne entsprechende Organisationsgruppen, war die Verteilung unserer zur Verfügung stehenden Mittel. Trotz knapper Räumlichkeiten und den wenigen guten Rechnern, hatte man es ohne Probleme geschafft mit den vorhandenen Ressourcen auszukommen. Durch die indirekte Verteilung und dementsprechende auf Prioritäten beruhende Rücksichtnahme gab es keine diesbezüglichen Probleme.

Auch kann man sagen, dass es sehr positiv war, dass von Anfang an Protokolle und Tagesordnungen der einzelnen Projekttreffen geschrieben wurden. Zwar waren diese nicht immer sehr detailliert und oft unausgereift, aber zumindest waren sie ein grober Hinweis auf die bisherige Arbeit und die gelegentlichen Ergebnisse. Das Führen der Protokolle und das eine Woche später geführte Plenum des Protokollanten, der mit den Ergebnissen des letzten Treffens und den anstehenden Tagesordnungspunkten vertraut war, wurde konsequent durchgehalten. In dem von uns benutzten elektronischen Aktensystem sind nahezu alle Protokolle vorhanden gewesen. Ausnahmen kamen eigentlich nur dann zustande, wenn das Aktensystem mal nicht hundertprozentig funktionierte oder wie am Anfang nicht vorhanden war. Für diesen Fall wurden allerdings, was man auch durchaus als hilfreich und positiv einstufen kann, in einem Extraordner, die handschriftlichen oder ausgedruckten Kopien abgelegt.

8.2 Unsere negativen Eindrücke

Zugegebenermaßen muss man wohl sagen, dass mit Sicherheit vieles nicht so gut in unserem Projekt gelaufen ist, wie man es sich hätte wünschen können. Dennoch gab es einiges, das gut lief. Aber auch die negativen Eindrücke, die wir gesammelt haben, sehen wir nicht als schlecht an. Allgemein herrscht in unserem Projekt die Meinung vor, dass man ebensoviel aus den schlecht gelaufenen Dingen lernen kann, wie aus denen, die gut liefen. Wir sehen das Projekt von daher immer noch als Erfolg, da sich ein guter Lerneffekt bei uns eingestellt hat. Es haben sich eigentlich alle ihre Meinung darüber gebildet, was sie beim nächsten Projekt besser machen würden.

Was die negativen Seiten bzw. Eindrücke unseres Projektes angeht, so fiel uns besonders die mangelnde Kommunikation unter uns auf. Nicht das wir nicht kommunikationsbereit gewesen wären, nur praktisch hat es dann nicht so geklappt, wie wir uns das gedacht hatten. Letztendlich dürfte dies auch an der geringen Selbstdisziplin gelegen haben, die wohl bei allen vorherrschte. Da fielen eben besonders solche Verhaltensweisen auf, wie die Nichterreichbarkeit der Mitglieder, mangelndes Verantwortungsbewusstsein, geringe Beschwerdebereitschaft oder das Ignorieren anderer Meinungen und die Nichtbeachtung von Beschlüssen.

Trotz dementsprechender Vereinbarungen, waren selten alle Mitglieder bei den Treffen anwesend und nicht mal regelmäßig per E-Mail erreichbar. Auch war die Bereitschaft nicht sehr hoch, regelmäßig zu übernommenen Arbeiten, Berichte oder dementsprechende Papiere abzuliefern. Davon mal abgesehen war selten jemand bereit zu seiner bisherigen Aufgabe weitere Verantwortung zu übernehmen. Wenn sich jemand dazu bereit erklärte, dann eigentlich nur weil sich sonst keiner meldete und irgendjemand gefunden werden musste. Es fiel dabei auf, dass dies auch immer dieselben Leute waren, die ohnehin schon vieles übernommen hatten, während andere sich mit kaum etwas beschäftigten.

Was die Beschwerdebereitschaft anging, so war diese eigentlich kaum oder nur selten vorhanden. Eigentlich nur dann, wenn mal einer von uns irgendeinen früheren Beschluss ansprach, oder etwas, das ihm irgendwie nicht passte, dann gab es für gewöhnlich eine Welle an Meinungen, die sich ihm anschlossen oder noch andere Ungereimtheiten zu Tage förderten. Die Bereitschaft überhaupt etwas anzusprechen, das nicht so toll lief oder das keinen Sinn ergab, war allerdings kaum vorhanden und wurde zusätzlich von einigen gehemmt, die oft ihre Meinung über die der anderen, und gleichzeitig durch ihre persönliche Art in den Vordergrund stellten.

Generell könnte man zu den vorherrschenden Meinungen sagen, dass diese für gewöhnlich einseitig und unumstößlich waren. Sofern jemand etwas ausgearbeitet hatte oder eine feste Meinung von etwas hatte, dann hörte derjenige sich zwar aus Anstand die anderen Meinungen an, aber ignorierte diese für gewöhnlich. Es wurde dann solange diskutiert, bis entweder der Überblick über die Ergebnisse der Diskussion verloren ging, also auf den Entwurf zurückgegriffen wurde oder aber keiner mehr weiterdiskutieren wollte und derjenige so doch seine Meinung durchsetzte.

Die Diskussionen waren für gewöhnlich auch ein negativer Aspekt unserer Projektarbeit. Entweder waren sie völlig überflüssig und wurden heiß diskutiert oder aber deren Ergebnisse wurden nicht beachtet. Selten wurden die Ergebnisse aufgeschrieben, für gewöhnlich auch nicht in angemessener Form, und noch seltener von allen beachtet. Da oftmals Mitglieder nicht erreichbar waren und sich auch nicht genügend darum kümmerten, was während ihrer Fehlzeit an Wichtigem passiert oder beschlossen worden war, sind viele Ergebnisse nicht bei allen Mitgliedern bekannt geworden und dementsprechend nicht beachtet worden. Hier kam auch noch hinzu, dass man oftmals die Lust verlor, sich die Protokolle der letzten bzw. verpassten Treffen anzusehen, da oft nur wenig Wichtiges drin stand oder man mit dem üblichen Nichts an Wichtigem rechnete. Die erwähnten überflüssigen Diskussionen, die einer Ja-Nein-Frage für gewöhnlich auch noch ein "Vielleicht" oder "Mal sehen was rauskommt, wenn wir dies auch noch dabei abstimmen" hinzufügten, verschärften diesen Eindruck sicherlich auch noch.

Von diesen negativen Eindrücken mal abgesehen, die sicherlich mit mehr Selbstdisziplin hätten vermieden werden können, kamen einige weitere negative Eindrücke zustande. Nahezu alle von uns hatten den Eindruck, dass die Anfangsphase unseres Projektes zu lange gedauert hatte. Die Zielfindungsphase hätte kürzer sein sollen, glauben viele von uns. Im Gegensatz dazu versicherte uns unser Projektleiter, dass dies allgemein so wäre. Wie auch immer, für uns war dies ein negativer Eindruck. Das lag vor allem daran, dass wir nicht von Anfang an eine strikte Zeitplanung in Erwägung gezogen hatten. Diese fehlte letztendlich doch sehr und hatte, neben der Tatsache, dass wir viel Zeit deswegen vertrödelt hatten, mit Sicherheit auch diesen Eindruck erweckt.

Die Zeitplanung, die wir lieber hätten vornehmen sollen, hat bei uns einen doch sehr negativen Eindruck hinterlassen. Ähnlich war es eigentlich mit unseren anfangs sehr hoch gesteckten Zielen. Wie bereits erwähnt, wären diese sehr wahrscheinlich mit der entsprechenden Disziplin auch zu schaffen gewesen, aber letztendlich waren sie eindeutig zu hoch für unseren ungezwungenen Arbeitsstil. Diese sehr negative Erfahrung, während der Arbeit eine Reduzierung vornehmen zu müssen, wurde nur ungern innerhalb unseres Projektes gemacht, dennoch war es eine lehrreiche Erfahrung, die sicherlich an keinem von uns Mitgliedern vorbeiging.

Ein weiterer Punkt wäre der, der schlechten Gruppenbildung. Es entstand bei vielen Mitgliedern der Eindruck, dass wir unsere Gruppenbildung nicht ordentlich in den Griff bekommen haben, dass zu ungleiche Gruppen entstanden waren und diese auch nur ungenügend zusammengearbeitet haben. Schließlich hatten wir uns anfangs vorgenommen, keine Kleingruppen zu bilden und im großen Team zu arbeiten.

Wie auch immer, zu der Gruppenbildung kann man auf jeden Fall sagen, dass sie ungeordnet bzw. ohne System ablief. Zufällig wurden Mitglieder zusammengewürfelt, die sich für dasselbe Interessengebiet entschieden hatten, ungeachtet der Tatsache, dass sich dabei auch Gruppen mit nur einer Person gebildet hatten.

Als zusammengehöriges Projekt hätte man dem mit Sicherheit vorbeugen sollen. Dabei wurde nicht darauf geachtet, ob nicht eventuell Interessengebiete von vornherein hätten gestrichen werden können, und zum Wohle der gerechten Arbeits- und Kräfteverteilung eine bessere Verteilung der einzelnen Mitglieder auf Schwerpunkte hätte vorgenommen werden müssen. Diese sicherlich sinnvolle Vorgehensweise wurde allerdings nicht berücksichtigt und hinterliess natürlich so den negativen Eindruck, dass die Gruppenbildung schlecht verlief.

Zu der Gruppenbildung kann man noch etwas hinzufügen, dass nicht wirklich negativ zu deuten ist, aber in einem gewissen Sinne hierzu gehört. Die Bildung, der in der Beschreibung der Entwicklungsphase bereits erwähnten Organisationsgruppen, war durchaus ein positives Highlight unseres Projektes. Doch sind sich eigentlich alle einig, dass dieser Schritt in die richtige Richtung erst viel zu spät getan wurde.

Allgemein hatten wir den Eindruck, dass z.B. eine rechtzeitige Einführung der Projektkoordinationsgruppe einen besseren Verlauf des Projektes bewirkt hätte. Der geringe Druck, den wir uns nahezu alle gewünscht hätten, wäre so in einem sinnvollen Maß von uns selber bestimmt worden, aber zumindest auch vorhanden gewesen. Denn bei vielen entstand der Eindruck, dass zu wenig Druck von Seiten unserer Projektleitung kam. Natürlich hat sich darüber keiner beschwert, denn so war es schließlich möglich sich so viel Zeit mit der eigenen Arbeit zu lassen, dass man in keiner Weise Stress bekommen würde. Leider hatten wir da wohl zu wenig Selbstdisziplin, um rechtzeitig die Notbremse zu ziehen und uns selber Druck zu machen.

9 Unsere Erkenntnisse und Tipps

Abgesehen davon, dass man sicherlich zahlreiche Informationen und somit auch positive und negative Erkenntnisse aus den vorherigen Abschnitten dieses Dokuments gewinnen konnte, möchten wir uns an dieser Stelle doch noch einmal die Mühe machen, die wichtigsten unserer Weisheiten zusammenzufassen.

Zunächst ist bei einem Projekt mit Ausgangsbedingungen wie den unseren eines zu sagen. Zuviel Freiheit schadet, zu wenig Freiheit schränkt zwar die Kreativität ein, aber hält auch den Schaden in Grenzen. Vieles, dass wir in Eigenregie begonnen haben, ist aufgrund der lockeren Handhabung nicht beendet worden oder aber mit vielen Schwierigkeiten. Allgemein sollte darauf geachtet werden, dass jemand da ist, der zu jeder Zeit einen Überblick über das Projekt hat. Wir empfehlen daher schon frühzeitig so etwas wie eine Koordinationsgruppe zu bilden, die darauf achtet, dass kleine Ziele, die man sich innerhalb eines Projektes unbedingt setzen sollte, auch mit Sicherheit erreicht werden. Diese Gruppe sollte durch ihre Arbeit gewährleisten, dass in festen Abständen ein erkennbarer Projektstand vorhanden ist. Den Überblick über die gesetzten Ziele zu verlieren, hatte unseres Erachtens die schwersten Folgen für unser Projekt.

Unsere zeitweise sehr unkoordinierte Arbeitsweise und der Verlust des Überblicks und der gemeinsamen Ansicht über das Projektziel hat uns mehr als deutlich spüren lassen, dass wir eine Zeit lang aneinander vorbeigedacht und vorbeigearbeitet haben. Hätten wir dies nicht erkannt und durch die späte aber dann wenigstens schnelle Einführung von Koordinationsgruppen abgefangen, dann hätte es möglicherweise einige Probleme gegeben, unser Projekt noch mit den meisten gesetzten Zielen erfolgreich zu beenden.

Wie auch an vielen Stellen des Berichtes erwähnt und angedeutet, geben wir unserer Projektleitung keinesfalls die Schuld dafür, dass wir unsere Probleme hatten, koordiniert zu arbeiten. Im Gegenteil dazu möchten wir ihr eher für die lockere Projektführung danken. Diese hat uns nämlich zum einen viel Spielraum für Kreativität gelassen und zusätzlich dazu geführt, dass wir wichtige Erfahrungen in Bezug auf Projektarbeit sammeln konnten. Und wenn man es genau nimmt, dann war es eigentlich unsere allgemeine Haltung und Arbeitsmoral im Projekt, die verhindert hat, dass mehr Druck von der Projektleitung aus gemacht wurde. Oft kam der Wunsch nach schriftlichen Zwischenergebnissen von Seiten unserer Projektleiter zur Sprache, allerdings wurde dieser Vorschlag, mehr Schriftliches abzuliefern, eher abgelehnt.

Auch an dieser Stelle können wir wieder nur anraten, dass Projektleiter ruhig etwas Druck in Richtung Zwischenergebnisse machen und vor allem, dass Projektteilnehmer diese Möglichkeit ihre Leistung nachzuweisen auch nutzen. Denn zum einen macht es der Projektleitung eine Bewertung der einzelnen Teilnehmer leichter und zum anderen geht wie bereits erwähnt der Überblick über das Projekt nicht so leicht verloren.

Neben dem Tipp, unbedingt kleinere Zwischenziele zu setzen und zu Gunsten aller auch schriftliche Zwischenergebnisse abzuliefern, haben wir noch einen wichtigen Tipp, der im gewissen Sinne mit demselben Problem zu tun hat. Man sollte unbedingt alles schriftlich festhalten, das man beschließt, wie z.B. Haupt- und Zwischenziele oder Ergebnisse von Diskussionen und Vorträgen.

Gerade in unserem Projekt hat die Tatsache, dass sehr wenig oder gar nichts von den Vereinbarungen und Zielen niedergeschrieben wurde, dazu geführt, dass unsere unterschiedlichen Sichtweisen das gemeinsame Projektziel in Frage gestellt haben und ein gemeinsames Arbeiten behindert haben. Wir hatten zwar Protokolle zu unseren wöchentlichen Plena geschrieben, aber trotzdem den Überblick verloren. Dies lag vor allem daran, dass wir nicht alle Beschlüsse festgehalten hatten bzw. teilweise nur grob. Unsere Art Protokolle zu schreiben war doch sehr unterschiedlich und unausgereift.

Wir möchten also an dieser Stelle allen nahelegen, die mit Projekten zu tun haben, im Projekt zu klären, was als wichtiges Ergebnis angesehen wird und festgehalten werden soll, wie die schriftliche Form der Ergebnisse auszusehen hat, wer die Ergebnisse festhält, und wo und wann diese abgelegt werden sollen und durchgesehen werden können. Wir haben es zwar in unserem Projekt nicht so gemacht, aber wir könnten uns vorstellen, dass entweder eine kleine feste Gruppe hierfür zugeteilt wird, die in gleicher Art und Weise arbeitet bzw. die Ergebnisse festhält oder aber dass eine Art Vortrag darüber gehalten wird, wie ein gutes Protokoll auszusehen hat, sodass alle die gleiche Art von Protokoll schreiben und sich zum Beispiel wöchentlich abwechseln können. So würden jedenfalls qualitativ gleiche Protokolle mit den von allen als solche angesehenen wichtigen Ergebnissen vorhanden sein.

Bei uns gab es in Bezug auf dieses Problem eine Diskussion. Jemand hatte behauptet, dass die bisherigen Protokolle sehr verschieden und überwiegend unbrauchbar gewesen seien, da selten wichtige Dinge drin standen. Ein anderer behauptete das Gegenteil. Desweiteren kam zur Sprache, dass es nicht immer Sinn macht zwangsweise ein Protokoll zu schreiben, z.B. wenn es keine wichtigen Ergebnisse zu notieren gibt, da unwichtige Dinge nicht festgehalten zu werden bräuchten. Der oben Erwähnte hatte aus diesem Grund z.B. auch ein Protokoll nicht schriftlich niedergeschrieben, da aus den Notizen nur unwichtige Dinge hervorgingen. Als Antwort kam, dass es nicht seine Entscheidung wäre, was wichtig und was unwichtig ist. Die Frage, wer dies zu entscheiden hat, wenn nicht der, der alles aus seiner Sicht Wichtige mitschreiben soll, sei einfach mal dahingestellt. Doch das Beispiel zeigt gut, dass solche Diskussionen überflüssig sind, wenn man die kurz zuvor genannten Tipps beherzigt. Überflüssige Diskussionen kann man gut vermeiden, wenn man mehr festlegt bzw. vereinbart.

Im Zusammenhang mit Abstimmungen, bei denen es genauso wie bei Diskussionen bei uns lief, muss gesagt werden, dass wir über einen langen Zeitraum hinweg unkoordiniert diskutiert und letztendlich auch abgestimmt haben. Diskussionsleitungen haben sich bei uns im nachhinein nicht nur be-

währt, sondern auch als absolut notwendig herausgestellt. Gerade bei unserer großen Mitgliederzahl waren die Diskussionen überwiegend chaotisch abgelaufen. Letztendlich hatten diejenigen, die sich auch sonst lautstark hervorgetan hatten, in den Diskussionen dominiert, was zur Folge hatte, dass sich die anderen von vornherein schon geistig von der Diskussion verabschiedeten oder aber ihre Meinung kurz ansprachen und sofort übertönt wurden. Eine geordnete und von einer Diskussionsleitung beaufsichtigte Diskussion bringt da vor allem den Vorteil mit sich, dass alle zu Wort kommen und hat oft den positiven Effekt dass sich mehr Leute an der Diskussion beteiligen, die sich sonst im Durcheinander zurücklehnen können.

Wie bereits angesprochen kommt eine diskussionsleitungsähnliche Instanz auch Abstimmungen zu Gute. Oftmals wurde bei uns über völlig unwichtige Dinge abgestimmt oder aber eine Auswertung einer Abstimmung unmöglich gemacht. Dies vor allem durch falsche Fragestellungen beim Abstimmen, bei denen die Abstimmungsergebnisse letztendlich dasselbe aussagten. Ein Beispiel wären die Fragen danach, ob man sich nach Acht Uhr oder ob man sich nicht vor Acht Uhr trifft. Letztendlich trifft man sich so oder so nach Acht Uhr, egal ob ein Teil nach Acht will oder der andere nicht vor Acht. Solche überflüssigen Abstimmungen, die uns manchmal wirklich stundenlang beschäftigt haben, kann man also dadurch vermeiden, dass man sie geleitet durchführt und vor allem mit sinnvollen Fragestellungen. Ganz nebenbei gesagt, sollte man auch nicht vergessen die Ergebnisse der Abstimmungen festzuhalten.

Ein weiterer wichtiger Punkt, den wir hier nochmal hervorheben wollen, ist der folgende Punkt. Die klare Zuordnung von Aufgaben und das dementsprechende Notieren hätte bei uns einige Missverständnisse beseitigt. Wie bereits beschrieben war das Verantwortlichkeitsgefühl nicht sehr hoch bei uns. Wäre grundsätzlich notiert worden, wer wann welche Aufgaben übernommen hat, dann wäre zu jeder Zeit nachzuvollziehen gewesen wer, was nicht gemacht hat. Nur allzu oft ging man in unserem Projekt davon aus, dass jemand etwas schon machen würde, aber derjenige wusste davon schon nichts mehr oder fühlte sich nicht mehr zuständig. Dieser Zustand sollte unbedingt vermieden werden, ansonsten macht es letztendlich gar keiner und es wird eventuell erst zu spät gemerkt.

Ansonsten gäbe es eigentlich nur noch einen wichtigen Punkt den wir unbedingt noch ansprechen wollen, die Zeitplanung. Man hat am Anfang das Gefühl, dass man unheimlich viel und vor allem genug Zeit hat die Projektarbeit rechtzeitig zu erledigen. Das Gefühl mag zwar nicht völlig täuschen, doch vergeht die Zeit letztendlich sehr schnell, wenn man nicht einen groben zeitlichen Ablauf mit grob gesteckten Zwischenzielen im Auge behält.

Eine simple Hilfe, die uns in unserer letzten Projektphase sehr geholfen hatte, war eine Zeitleiste, auf der die verbleibende Projektzeit optisch übersichtlich und mit kleinen gut erfassbaren Meilensteinen gekennzeichnet war. Jeder konnte darauf sehen, bis wann eine Aufgabe erledigt werden musste, um noch im Zeitrahmen zu bleiben. Die sogenannten Deadlines, die sich durch

die Meilensteine ergaben, waren zwar desöfteren geplatzt, aber dennoch versuchte man sich dem indirekten Druck der Zeitleiste zu beugen und seine Arbeit rechtzeitig zu erledigen. Allgemein gesehen empfanden wir diese Hilfe als nützlich.

Dieser kleine Überblick war wie gesagt nur noch mal eine Zusammenfassung unserer Erkenntnisse und Tipps. Mehr nützliche Hinweise kann man wie bereits angedeutet aus diesem Gesamtdokument entnehmen, wenn man auch zwischen den Zeilen liest und sich eine eigene Meinung zu unserem Projekt bildet.

10 Schlusswort

Die Projektarbeit hat uns zum einen gefordert und Spaß gemacht, und zum anderen viel gebracht. Die Erfahrungen, die wir sammeln konnten, seien es die guten genauso wie die schlechten, werden uns sicherlich bei anderen Projekten weiterhelfen. Jedem, der die Möglichkeit hat, können wir nur raten, selber mal an einem Projekt teilzunehmen und vielleicht dabei sogar einige unserer Erfahrungen zu berücksichtigen. Es wird bestimmt genauso lehrreich wie interessant.

Abschließend möchten wir Teilnehmer uns bei Ihnen, der Leserin, für Ihr Interesse bedanken. Es freut uns doch sehr, dass sich jemand für unser Projekt und unsere Arbeit interessiert.

Nachwort

Hans-Jörg Kreowski

Das Studienprojekt ANIMA – *Regellabor für Visualisierung und Animation*, das vom Wintersemester 1999/2000 bis zum Sommersemester 2001 lief, wurde von mir geleitet, unterstützt von Frank Drewes im ersten Semester und von Renate Klempien-Hinrichs im vierten. Die Fertigstellung des Abschlussberichts hat noch ein zusätzliches halbes Jahr beansprucht. In dieser Notiz sind einige Beobachtungen zum Projekt und seinem Verlauf festgehalten, die mir als Projektleiter wichtig und interessant erscheinen.

Im Sommersemester 1999 haben Frank Drewes und ich im Rahmen der PROBE-Veranstaltung den damaligen Informatik-Studierenden im vierten Semester unsere Ideen zum ANIMA-Projekt erstmals vorgestellt. Als Gegenstand der Betrachtung haben wir die Modellierung visueller und visualisierter Phänomene und Prozesse aus Natur und Technik, Mathematik und Phantasie vorgeschlagen, wie sie bei Pflanzen und ihrem Wachstum, bei Kristallen, bei der Zeichnung von Schneckenhäusern und Muschelschalen sowie vielen fraktalen Bildern anzutreffen sind. Zur Illustration haben wir neben einigen echten Kristallen, Muschelschalen und kleinen Kunstwerken insbesondere Bilder aus den Büchern von Prusinkiewicz und Lindenmayer [PL90], Meinhardt [Mei95], Medenbach und Wilk [MW86] sowie Peitgen, Jürgens und Saupe [PJS92] gezeigt. Wir haben angeregt, dass im Projekt ein System entwickelt wird, das derartige Modellierungen unterstützt sowie insbesondere visualisiert und animiert. Während nach unserem Projektplan die Systemgestaltung und die Auswahl der Modellierungsbeispiele von den Teilnehmerinnen und Teilnehmern entschieden werden sollten, war von uns fest vorgegeben, dass ANIMA auf syntaktischen, regelbasierten Methoden der Bilderzeugung aufbauen sollte. Die Auswahl der im Projekt eingesetzten Methoden aus einem gegebenen Spektrum war allerdings frei. ANIMA hatte somit in seinen praktischen und angewandten Teilen eine offene Planung, und nur der theoretisch-methodische Teil hatte von vornherein einen fixierten Rahmen. Mit 19 Studierenden, von denen drei aus gesundheitlichen, persönlichen bzw. privaten Gründen im Laufe des Projekts ausschieden und einer sich nicht am gemeinsamen Abschlussbericht beteiligte, hat ANIMA ein reges Interesse gefunden.

Die organisatorische Struktur von ANIMA hatte keine auffälligen Besonderheiten. über alle vier Semester hinweg gab es ein Plenum, das wöchentlich immer Freitagvormittag stattfand – in der lehrveranstaltungsfreien Zeit jedoch nur unregelmäßig. Am Ende des ersten und dritten Semesters wur-

den mehrtägige Seminare in zwei Jugendherbergen der Umgebung organisiert. Während der ersten beiden Semester fand eine Begleitlehrveranstaltung zum Thema *Syntaktische Methoden der Bilderzeugung* statt. Sie wurde im WS 1999/2000 von Frank Drewes und mir und im SS 2000 von mir allein durchgeführt. Sie diente dazu, den methodischen Rahmen des Projekts einzuführen und die vorgestellten Methoden theoretisch zu fundieren. Ansonsten handelte es sich um eine normale Lehrveranstaltung im Rahmen des Informatik-Hauptstudiums im Fach Theoretische Informatik, die offen für alle Studierenden war und auch nicht nur von Projektmitgliedern besucht wurde. Alles Weitere, wie z.B. Teilnehmerinnen und Teilnehmer einzeln oder in Arbeitsgruppen dem Projekt zuarbeiten oder sich Kenntnisse über Graphische Datenverarbeitung verschaffen, war nicht näher geregelt. Da allerdings ein großer Teil der Plena in seminaristischer Form ablief, war es möglich, über die methodischen Grundlagen hinaus einen gemeinsamen Informationsstand zu realisieren. In den ersten drei Semestern wurden die Plena ebenfalls dazu genutzt, ANIMA fortlaufend gemeinsam organisatorisch und inhaltlich zu planen. Im vierten Semester hat diese Aufgabe insbesondere hinsichtlich des zeitlichen Ablaufs eine Projektkoordinationsgruppe aus dem Kreis der Studierenden übernommen.

In der zweisemestrigen projektbegleitenden Lehrveranstaltung *Syntaktische Methoden der Bilderzeugung* wurde eine Palette solcher Methoden vorgestellt und jeweils ein Stück weit theoretisch vertieft. Im ersten Semester wurden so Kettencode-Bildsprachen nach Dassow und Hinz [DH89] sowie Maurer, Rozenberg und Welzl [MRW82], Turtle-Bildsprachen nach Prusinkiewicz und Lindenmayer [PL90], zelluläre Automaten nach Toffoli und Margolus [TM87], iterierte Funktionensysteme nach Barnsley [Bar88] und Collagen-Grammatiken nach Drewes und Kreowski [DK99] behandelt. Im zweiten Semester wurden zusätzlich zweidimensionale Grammatiken nach Giammarresi und Restivo [GR97], Random-Context-Picture-Grammars nach Ewert und van der Walt [EvdW98, EvdW99], Tilings nach Grünbaum und Shephard [GS89] und vernetzte iterierte Funktionensysteme nach Peitgen, Jürgens und Saupe [PJS92] eingeführt. Auf die theoretischen Untersuchungen gehe ich hier nicht ein, weil sie im Projekt keine Rolle gespielt haben. Die Bilderzeugungskonzepte aber wurden wie geplant in ANIMA aufgegriffen und zum Teil intensiv weiter diskutiert und bearbeitet. Das ist im Projektbericht mit Kettencode-Bildsprachen, Collagen-Grammatiken, vernetzten iterierten Funktionensystemen und Tilings dokumentiert. Letzteres wurde sogar von der üblichen zweidimensionalen Darstellung auf 3D-Tilings verallgemeinert. Außerdem wurde im Projekt mit Ameisen und Turmiten noch eine sequentielle Variante der zellulären Automaten verfolgt.

Neben der engen Verknüpfung von Begleitlehrveranstaltung und Projekt auf der Ebene der Bilderzeugungskonzepte wurden auch die Beispiele für die Modellierung bildlicher Phänomene und Abläufe in ANIMA aufgegriffen. An den Pflanzen, Tilings und Fraktalen im Abschlussbericht ist das abzulesen. Es wurden darüber hinaus Modellierungen betrachtet, die in der Lehrveran-

staltung keine Vorbilder hatten wie Kristalle, Brownsche Bewegung, keltische Knoten. Das meiste wurde nur angerissen und dann nicht weiter verfolgt. Die Stadt- und Gebirgslandschaften sind allerdings bis zu einem berichtenswerten Zustand verfolgt worden. Insgesamt ist aber der Modellierungsaspekt in ANIMA nicht so intensiv und systematisch untersucht worden, wie ich mir das anfangs gewünscht hatte. Die meiste Energie der Projektteilnehmerinnen und -teilnehmer ist stattdessen in die Konzeption, Spezifikation und Systementwicklung von AnimaLab geflossen. Ein Grund dafür scheint zu sein, dass Informatikstudierende hier ihr eigentliches Metier sehen und sich, bestärkt durch ihre Studienschwerpunkte, sicher fühlen. Dagegen bedeutet die Auseinandersetzung mit Anwendungsproblemen fast immer auch ein Einlassen auf interdisziplinäre Bezüge und damit ein Betreten unvertrauten Terrains.

Eine Schlüsselrolle hatten die beiden mehrtägigen Projektseminare. Sie hatten jeweils ein interessantes inhaltliches Programm und trugen viel dazu bei, dass sich die Projektmitglieder untereinander besser kennen lernen konnten. Außerdem wurden auf beiden Seminaren wegweisende Entscheidungen getroffen. Am Ende des ersten Semesters wurden die Projektziele endgültig erarbeitet und formuliert. Auch wurden Arbeitsgruppen gebildet, in denen die jeweiligen Teilaufgaben bearbeitet und erledigt werden sollten. Diese Arbeitsgruppen haben allerdings, wie das zweite und dritte Projektsemester zeigten, nicht optimal funktioniert. Sie haben selten zeitliche Vorgaben eingehalten, nur zögerlich über den Stand der Arbeit berichtet, teils gar nicht in der personellen Zusammensetzung gearbeitet, die ursprünglich vorgesehen war, sich untereinander zu wenig ausgetauscht. In den wöchtlichen Projektplena ist es in dieser Phase auch nur bedingt gelungen, diese Probleme zu lösen oder auszugleichen. Daher war das zweite Projektseminar am Ende des dritten Semesters so wichtig. Denn dort wurden die Projektziele auf das noch realistisch Erreichbare zurückgenommen, die Arbeitsgruppen konsolidiert und aktiviert und die Planungen für den Projekttag und für die Erstellung des Abschlussberichts mit Nachdruck aufgenommen. Äußeres Zeichen für den "Schlusspurt" war die Einsetzung einer Projektkoordinierungsgruppe. Das Ergebnis kann sich sehen lassen. Die Präsentation von ANIMA auf dem Projekttag war voll gelungen und der Abschlussbericht ist lesens- und sehenswert.

Die Teilnehmerinnen und Teilnehmer des Projekts ANIMA wirkten auf mich hoch motiviert, interessiert, engagiert und talentiert, theoretisch fundierte Methoden praktisch umzusetzen. Während des Projektes herrschte in meinem Beisein immer ein freundlicher Umgangston und eine angenehme Atmosphäre, wobei allerdings zwischenzeitliche Frustrationen und interne Reibereien mir gegenüber meist versteckt wurden. Das liegt meiner Vermutung nach daran, dass Studierende gegenüber Hochschullehrenden eine deutliche Distanz empfinden. Der Nachteil daran ist, dass ich ein etwas geschöntes Bild vom Projektverlauf hatte und auf entstandene Probleme nicht rechtzeitig eingehen konnte. Meine Rolle als Projektleiter war von Anfang an so abgesprochen, dass ich für das systematische Einbringen des Grundlagenwissens zuständig war und als Berater und Kommentator des Projektgeschehens fun-

gierte. Auf Machtworte hatte ich freiwillig verzichtet. Es war von Anfang an klar, dass zum Schluss eine Note vergeben werden muss. Damit die Zensuren aber nicht wie ein Damoklesschwert über der Arbeit des Projekts hängen, war vereinbart, dass letzlich die Beiträge zum Abschlussbericht Grundlage der Bewertung sind und alle anderen Leistungen während des Projekts allenfalls zur Verbesserung der Note genutzt werden. So wurde die laufende Arbeit nicht durch drohende Zensuren bestimmt. Nachteilig jedoch hat sich ausgewirkt, dass Zwischenergebnisse nicht verbindlich eingefordert werden konnten. Als Projektleiter war ich gezwungen, mit meinen Ideen zu überzeugen. Ich stehe dazu, auch wenn es mir bei ANIMA ziemlich häufig nicht gelungen ist.

Aus meiner Sicht war ANIMA ein erfolgreiches Projekt. Es mag bedauerlich sein, dass das Kernsystem AnimaLab nicht voll funktionsfähig ist, weil die fertiggestellten Komponenten nur provisorisch integriert sind und weil die Übertragungsraten, die in der verteilten Architektur erzielt werden können, nicht groß genug sind, um hoch strukturierte und detaillierte dreidimensionale Szenen in ihrer Entwicklung live betrachten zu können. Aber bei einem studentischen Projekt kommt es gar nicht wirklich darauf an, dass ein lauffähiges System produziert wird, sondern wichtig ist, was gelernt wird. Im Mittelpunkt stehen dabei die Inhalte. ANIMA hatte da mit den syntaktischen Methoden der Bilderzeugung, ihrer Verwendung bei der Visualisierung und Animation von Phänomenen und Prozessen aus vielfältigen Anwendungsbereichen und mit den computergraphischen und informationstechnischen Mitteln zur Unterstützung solcher Modellierungen einiges zu bieten. Aber die vermeintlichen oder tatsächlichen Fehler bei der Projektplanung und -durchführung, die mangelnde Disziplin beim Einhalten von gesetzten Terminen, das zögerliche Verhalten beim schriftlichen Fixieren von Zwischenergebnissen, die Mängel in der Kommunikation, die Missverständnisse und Ungereimtheiten, die bei einer zweijährigen Teamarbeit auftreten, sind vielleicht genauso lehrreich wie ein konsequentes Projektmanagement. Ein Projekt soll seinen Mitgliedern die Möglichkeit bieten, sich fachlich und persönlich zu entwickeln. Dabei gibt es viele Schwierigkeiten, die vielleicht sogar unvermeidlich sind. Einige nutzen die Möglichkeiten voll, andere halten sich zurück. Wenn die einen loslegen wollen, müssen andere in ihrem sonstigen Studium oder zum Lebensunterhalt gerade viel Zeit aufwenden, so dass eine starke Asynchronizität entsteht. Was für die einen besonders interessant ist, reizt andere eher weniger. Während sich die einen mutige Ziele stecken, sind die anderen darüber erschrocken. Ein Beispiel ist der Beschluss von ANIMA, die inhaltlichen Teile des Abschlussberichts in Englisch zu verfassen. Das war eine große Herausforderung, die viele zu besonderen Leistungen angespornt hat. Aber in Einzelfällen wurden die inhaltlichen Hürden noch durch sprachliche erhöht. Insgesamt bieten studentische Projekte die Chance zum forschenden Lernen, die nach meiner Beobachtung in ANIMA in vielerlei Weise und mit vorzeigbarem Erfolg genutzt wurde.

References

[Bar88] M. Barnsley. *Fractals Everywhere.* Academic Press, 1988.

[Blo85] J. Bloomenthal. Modeling the mighty maple. *Communications of the ACM*, 19(3), 1985.

[BPP95] G. Bell, A. Parisi, and M. Pesce. *The Virtual Reality Modeling Language Version 1.0 Specification*, November 1995. (http://www.vrml.org/VRML1.0/vrml10c.html)

[CB97] R. Carey and G. Bell. *The Annotated VRML97 Reference Manual*, 1997. (http://parlevink.cs.utwente.nl/Students/seminars/211032/vrmlboek)

[CBM97] R. Carey, G. Bell, and C. Marrin. *ISO/IEC 14772-1:1997, Virtual Reality Modeling Language (VRML97)*, 1997. (http://www.vrml.org/technicalinfo/specifications/vrml97/index.htm).

[CD93] K. Culik II and S. Dube. Affine automata and related techniques for generation of complex images. *Theoretical Computer Science*, 116:373–398, 1993.

[CE95] B. Courcelle and J. Engelfriet. A logical characterization of the sets of hypergraphs defined by hyperedge replacement grammars. *Mathematical Systems Theory*, 28:515–552, 1995.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations, chapter 3, pages 163–246. World Scientific, 1997.

[DE98] F. Drewes and J. Engelfriet. Decidability of the finiteness of ranges of tree transductions. *Information and Computation*, 145:1–50, 1998.

[DH89] J. Dassow and F. Hinz. Kettenkode-Bildsprachen. Theorie und Anwendungen. *Wiss. Zeitschrift der Techn. Universität "Otto von Guericke" Magdeburg*, 33(6), 1989.

[DHKT95] F. Drewes, A. Habel, H.-J. Kreowski, and S. Taubenberger. Generating self-affine fractals by collage grammars. *Theoretical Computer Science*, 145:159–187, 1995.

[DK96] F. Drewes and H.-J. Kreowski. (Un-)decidability of geometric properties of pictures generated by collage grammars. *Fundamenta Informaticae*, 25:295–325, 1996.

[DK99] F. Drewes and H.-J. Kreowski. Picture generation by collage grammars. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 11, pages 397–454. World Scientific, 1999.

[DKL02] F. Drewes, H.-J. Kreowski, and D. Lapoire. Criteria to disprove context-freeness of collage languages. *Theoretical Computer Science*, 2002. To appear.

[Dre96a] F. Drewes. Computation by tree transductions. Doctoral dissertation, University of Bremen, Germany, 1996.

[Dre96b] F. Drewes. Language theoretic and algorithmic properties of d-dimensional collages and patterns in a grid. *Journal of Computer and System Sciences*, 53:33–60, 1996.

[Dre98a] F. Drewes. A characterization of the sets of hypertrees generated by hyperedge-replacement graph grammars. *Theory of Computing Systems*, 32:159–208, 1998.

[Dre98b] F. Drewes. TREEBAG—a tree-based generator for objects of various types. Report 1/98, Univ. Bremen, 1998.

[Dre00] F. Drewes. Tree-based picture generation. *Theoretical Computer Science*, 246:1–51, 2000.

[Dre01] F. Drewes. *The TREEBAG Manual Version 1.2*, 2001. (http://www.informatik.uni-bremen.de/theorie/treebag/manual)

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 241–286. Academic Press, New York, 1980.

[Eng94] J. Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Proc. CAAP 94*, volume 787 of *Lecture Notes in Computer Science*, pages 15–37. Springer, 1994.

[ES77] J. Engelfriet and E.M. Schmidt. IO and OI. I. *Journal of Computer and System Sciences*, 15:328–353, 1977.

[ES78] J. Engelfriet and E.M. Schmidt. IO and OI. II. *Journal of Computer and System Sciences*, 16:67–99, 1978.

[EvdW98] S. Ewert and A. van der Walt. Generating pictures using random forbidding context. *International Journal of Pattern Recognition and Artificial Intelligence*, 12(7):939–950, 1998.

[EvdW99] S. Ewert and A. van der Walt. Generating pictures using random permitting context. *International Journal of Pattern Recognition and Artificial Intelligence*, 13(3):339–355, 1999.

[FFC82] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):164–172, 1982.

[Fis95] Y. Fisher. *Fractal Image Compression: Theory and Application.* Springer, 1995.

[FvDFH00] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practise, Second Edition in C.* Addison-Wesley, 2000.

[Gol91] R. Goldman. More matrices and transformations: Shear and pseudo-perspective. In J. Arvo, editor, *Graphic Gems II*, pages 339–340. Academic Press, 1991.

[GR97] D. Giammarresi and A. Restivo. Two-dimensional languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3: Beyond Words, chapter 4, pages 215–267. Springer, 1997.

[GS84] F. Gécseg and M. Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[GS89] B. Grünbaum and G.C. Shephard. *Tilings and Patterns.* Freeman, New York, 1989.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 1, pages 1–68. Springer, 1997.

[Hil91] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[HK91] A. Habel and H.-J. Kreowski. Collage grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 532 of *Lecture Notes in Computer Science*, pages 411–429. Springer, 1991.

[HKT93] A. Habel, H.-J. Kreowski, and S. Taubenberger. Collages and patterns generated by hyperedge replacement. *Languages of Design*, 1:125–145, 1993.

[Hut81] J. Hutchinson. Fractals and self-similarity. *Indiana University Journal of Mathematics*, 30:713–744, 1981.

[Lan86] C.G. Langton. Studying artificial life with cellular automata. In D. Farmer, A. Lapedes, N. Packard, and B. Wendroff, editors, *Evolution, Games and Learning: Models for Adaptation in Machines and Nature*, pages 120–149. North-Holland, 1986.

[Mei95] H. Meinhardt. *The Algorithmic Beauty of Sea Shells*. Springer, 1995.

[MRW82] H.A. Maurer, G. Rozenberg, and E. Welzl. Using string languages to describe picture languages. *Information and Control*, 54:155–185, 1982.

[MV90] K. Meyberg and K. Vachenauer. *Höhere Mathematik*. Springer, 1990.

[MW67] J. Mezei and J.B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[MW86] O. Medenbach and H. Wilk. *The Magic of Minerals*. Springer, 1986.

[OPK01] J. Ott, C. Perkins, and D. Kutscher. A Message Bus for Local Coordination. Internet draft, IETF Networking Group, May 2001. (http://search.ietf.org/internet-drafts/draft-ietf-mmusic-mbus-transport-06.txt)

[PH93] P. Prusinkiewicz and M. Hammel. A fractal model of mountains with rivers. *Proc. Graphics Interface*, pages 174–180, May 1993.

[PJS92] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals. New Frontiers of Science*. Springer, 1992.

[PL90] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[RB85] W.T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Communications of the ACM*, 19(3), 1985.

[Ree83] W.T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *Computer Graphics*, 17(3), 1983.

[Rou69] W.C. Rounds. Context-free grammars on trees. In *Proc. 1st Annual ACM Symposium on Theory of Computing (STOC)*, pages 143–148, 1969.

[Rou70a] W.C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4:257–287, 1970.

[Rou70b] W.C. Rounds. Tree-oriented proofs of some theorems on context-free and indexed languages. In *Proc. 2nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 109–116, 1970.

[SA99] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specifiation (Version 1.2.1)*, 1999. (http://www.opengl.org/developers/documentation/specs.html)

[SD92] K. Shoemake and T. Duff. Matrix animation and polar decomposition. *Proc. Graphics Interface '92*, pages 258–264, 1992.

[She96] J. Shemitz. Fractal landscapes version 3.0. WWW, January 1996. (http://www.midnightbeach.com/jon/pubs/3D_Fractal_Landscapes.html)

[SRD00] H. Sowrizal, K. Rushforth, and M. Deering. *The Java 3D API Specification*. Java Series. Sun Microsystems Press, second edition, 2000.

[Tan96] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[Tea00] Sun Microsystems Java 3D Engineering Team. *Java 3D API Tutorial*, 2000. (http://developer.java.sun.com/developer/onlineTraining/java3d/)

[Tha70] J.W. Thatcher. Generalized^2 sequential machine maps. *Journal of Computer and System Sciences*, 4:339–367, 1970.

[Tho91] S.W. Thomas. Decomposing a matrix into simple transformations. In J. Arvo, editor, *Graphics Gems II*, pages 320–323. Academic Press, 1991.

[TM87] T. Toffoli and N. Margolus. *Cellular Automata Machines, A New Environment for Modeling*. MIT Press, Cambridge, MA, 1987.

[WP99] A. Watt and F. Policarpo. *The Computer Image*. Addison-Wesley, 1999.