A Behavioral Animation System Based on L-systems and Synthetic Sensors for Actors

THÈSE No 1609 (1997)

PRÉSENTÉE AU DEPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Hansrudi **NOSER**
ing. info. dipl. EPFL
Dipl. Phys. ETHZ
de nationalité Suisse

acceptée sur proposition du jury :

Prof. Daniel Thalmann, rapporteur
Prof. Werner Purgathofer, corapporteur
Prof. Pascal Lienhardt, corapporteur
Prof. Jacques Zahnd, corapporteur

Lausanne, EPFL
1997

Acknowledgment

Many thanks to my thesis supervisor Prof. Daniel Thalmann for his precious guidance, help, support and encouragement.

My appreciation goes to Ing. Dr. Habilitation Ronan Boulic for his collaboration and the fruitful discussion all over the years of this research. His and Luc Emering's reviews of the manuscript is gratefully acknowledged.

I also extend my thanks to all members of the LIG group at the department of Computer Science of EPFL in Lausanne and to all the members of the MIRAlab Group at the University of Geneva, directed by Prof. Nadia Magnenat-Thalmann, for their support and cooperation during my research work.

Finally, I would like to thank all members of the jury for their participation and their interest.

Abstract

The creation, or at least the understanding of living, autonomous and intelligent creatures has always been a challenge for scientists. Today, we have the possibility to experiment with primitive autonomous creatures in virtual environments simulated on powerful computers. By programming the actors with some behavioral rules we can study possibly meaningful emergent behaviors of individuals or societies. Through special interfaces a user can even interactively participate in a virtual scene. Autonomous synthetic actors can only exist in an environment. The way they perceive their environment influences drastically their behavior. In a behavioral animation system the three components consisting of the environment, the sensors of the creatures and the creatures themselves play an important role.

The aim of this thesis is to implement such an animation system in order to study behaviors and to create autonomous actors useful in film productions and interactive games. For this purpose, a formal theory of a behavioral L-system was developed. It is based on a timed, parametric, conditional, stochastic, context dependent and environmentally sensitive production system. The production system associates to its symbols some basic geometric primitives as cubes, spheres, trunks, cylinders terminated at their ends by half spheres, line segments, pyramids and imported triangulated surfaces. We define non generic environment elements as the ground plane, the sky, walls or other static objects directly in the axiom of the production system. The generic parts as growing plants are defined by production rules having only their germ in the axiom. The actors are also represented by special symbols. Their geometric representation can vary from some simple primitives like some cubes and spheres, over a more complicated skeleton structure to a fully deformed triangulated body surface usable in a ray tracer. It depends on the purpose of the applications that range from tests over interactive real time applications to ray traced video production.

An autonomous agent can capture this world by synthetic vision, sense of touch and audition. The synthetic vision is the simulated vision for a synthetic actor where the computer renders from the actor's point of view the virtual environment in a window corresponding to the actor's vision image. This Z-buffered and colored image permits an actor to recalculate the 3D position of pixels and to extract semantic data from the image. We implemented also vision sensor functions returning useful information from the vision system. These functions can be called in the condition of production rules defining, for example, collision avoidance behaviors.

To simulate sense of touch we defined functions used also in conditions of production rules evaluating the global force field at a given position. The force field function returns the

amount of the global force field at the sensor position, and by comparing this value with a threshold value, a kind of collision detection with force field modeled environments can be simulated.

All sound events produced by an animation are managed by a special sound event handler. To model an acoustic sensor for actors we defined a special function returning information about on-going sound events. This function can be used in the conditions of production rules to model sound event dependent behaviors of actors with production rules.

In our animation system we have several possibilities to define behavior. We describe the group animation of fishes and butterflies with force fields defined in the L-system. Additionally, we present an automata approach for defining behaviors based on synthetic vision, audition, force field sensing and visual memory integrated in the L-system animation system. With the extensions of the behavioral L-system we can also define behaviors directly with production rules. With the above described sensor functions in the conditions of the production rules and the concept of the query symbol of environmentally sensitive L-systems simple rules can lead to interesting emergent behaviors.

The animation system is a universal, real time structured multi-L-system interpreter. Its real time structure allows interactive games with autonomous actors as partners. Additionally, it is conceived for automatic ray traced image by image film sequence production with synchronized sound track generation. Through a shared memory interface articulated humanoid actors with body deformation, controlled by a concurrent process can be imported into a simulation.

Another shared memory interface allows the sensor based autonomous actors to participate in the "Virtual Life Network". We demonstrate this feature with a networked interactive tennis game simulation where autonomous actors play the role of the referee judging the game and the role of a virtual game partner.

The proposed animation system is a versatile research tool. It combines in a systemic approach several different components important and necessary for sensor based behavioral animation.

Résumé

La création, ou au moins la compréhension de créatures vivantes, autonomes et intelligentes a toujours été un défi des scientifiques. Aujourd'hui nous pouvons expérimenter avec de simples acteurs autonomes dans des environnements virtuels simulés par des ordinateurs puissants. En programmant les acteurs par quelques règles comportementales nous pouvons étudier des comportements émergeants et éventuellement intéressants d'acteurs individuels ou de sociétés d'acteurs. Par des interfaces spéciales un utilisateur peut même participer interactivement dans une scène virtuelle. Des acteurs autonomes peuvent seulement exister dans un environnement. La façon dont ils perçoivent leur environnement influence beaucoup leur comportement. Les trois composantes d'un système d'animation, l'environnement, les capteurs des acteurs et les acteurs jouent un rôle très important.

Le but de ce travail est de réaliser un tel système d'animation afin d'étudier des comportements et de créer des acteurs autonomes utiles pour des productions de films et dans des jeux interactifs. C'est pourquoi nous avons développé une théorie formelle d'un L-système comportemental. Cette théorie est basée sur des L-systèmes dépendants du temps, paramétriques, conditionnels, aléatoires, dépendants du contexte et sensitifs à l'environnement. Le système de production associe quelques formes géométriques de base à ses symboles. Ces formes géométrique sont des cubes, des sphères, des troncs de cône, des cylindres terminés par des demi sphères, des segments de lignes, des pyramides et des surfaces triangulées et importées. Nous définissons des environnements non génériques comme par exemple le sol, le ciel, des murs ou d'autres objets statiques directement dans l'axiome d'un L-système. Les parties génériques comme des plantes croissantes, sont définies par des règles de productions ayant seulement un symbole germe dans l'axiome. Les acteurs sont aussi représentés par des symboles spéciaux. Leur représentation géométrique peut s'étendre de quelques formes simples comme des cubes, via des structures articulées plus compliquées à des corps triangulés et sophistiqués utilisable dans des lanceurs de rayons. La forme géométrique dépend en général d'une application qui peut être un test simple, un jeu interactif ou une production de film utilisant un lanceur de rayons.

Un agent autonome peut percevoir son monde par sa vision synthétique, par ses capteurs tactiles et par l'audition synthétique. La vision synthétique est la vision simulée d'un acteur synthétique. L'ordinateur rend l'environnement virtuel du point de vue de l'acteur dans une fenêtre qui correspond à l'image de la vision de l'acteur. Cette image colorée avec Z-buffer permet à l'acteur de recalculer la position 3D d'un pixel et d'extraire de l'information sémantique de l'image. Nous avons défini des fonctions de type “senseur”. Elles peuvent être

appelées dans les conditions de règles de productions définissant des comportements comme par exemple l'évitement de collisions. Pour simuler les senseurs tactiles nous définissons des fonctions utilisables dans des conditions de règles de production qui évaluent le champ de force globale à la position du senseur. Cette fonction retourne la valeur de cette force et en comparant cette valeur avec un seuil donné nous pouvons simuler une détection de collision avec un environnement modélisé par des champs de force.

Tous les événements acoustiques produit par le L-système sont gérés par un contrôleur d'événement de son particulier. Une fonction utilisable dans les conditions des règles de productions et indiquant si un son référencé par un paramètre du symbole à remplacer est actif, sert comme capteur acoustique.

Dans le système d'animation il existe plusieurs possibilités de définir les comportements. Des champs de force définis dans un L-système peuvent décrire, par exemple, des animations de bancs de poissons et des volées de papillons. Nous présentons également une approche par automate pour définir des comportements basés sur la vision synthétique et d'une mémoire visuelle intégrée dans l'interpréteur de L-systèmes. De plus, les extensions du L-système comportemental permettent de modéliser directement des comportements par des règles de production. De simples règles de production peuvent mener à d'intéressants comportements émergeants en utilisant le concept du symbole d'interrogation "?" de L-systèmes sensitifs à l'environnement et les fonctions des capteurs utilisables dans les conditions des productions.

Le système d'animation est un interpréteur universel de multiples L-systèmes qui est structuré pour des applications en temps réel permettant son utilisation pour des jeux interactifs avec des acteurs autonomes comme partenaires. De plus, le système est conçu pour la production de vidéos en mode image par image rendu par un lanceur de rayons et la génération automatique d'une bande de son synchronisée. Des acteurs humanoïdes qui sont contrôlés par un processus concurrent produisant des corps triangulés avec déformations, peuvent être importés par une interface à mémoire partagée.

Une autre interface à mémoire partagée permet aux acteurs autonomes de participer dans VLNET (Virtual Life Network). Nous montrerons cette extension du système avec une simulation d'un jeu de tennis interactif à travers le réseau. Les acteurs autonomes jouent les rôles de l'arbitre et d'un partenaire virtuel du jeu.

Le système d'animation est un outil de recherche versatile. Par une approche systémique il combine plusieurs composants différents qui sont importants et nécessaires pour l'animation comportementale

Contents

Acknowledgment **1**
Abstract **3**
Résumé **5**
Contents **7**
1. Introduction **11**
1.1 Motivation and Objectives 11
1.2 Organization 12
1.3 Literature Review 13
1.3.1 L-systems 13
1.3.2 Vision 16
1.3.3 Acoustics 18
1.3.4 Behavioral Animation 19
2. The Virtual Environment **23**
2.1 Animation System 24
2.1.1 System Architecture and Animation Loop 24
2.1.2 Recursive L-system 26
Growing Tree Structure 29
2.1.3 The Iterative L-system 31
2.1.4 Syntax and Semantic 34
2.1.5 The Turtle Interpretation 36
2.1.6 Camera, Light and Material Control 38
2.2 Geometric Modeling 38
2.2.1 Symbols for Basic Shapes 38
2.2.2 Special Symbols 39
2.2.3 Imported and Exported Data 39
2.3 Physical Modeling 40
2.3.1 Particle System 40
2.3.2 Tropism 42
2.3.3 Some Force Field Examples 43
General Forces 44
2.4 The Virtual Acoustic Environment 45
2.4.1. General Considerations 46
2.4.2 Real Time Sound Modeling 47
2.4.3 A Sound Renderer for Film Productions 48
2.4.5 Speech Recognition 51
2.4.6 The Sound Symbol of the L-system 52
3. The Actor - Environment Interface **53**
3.1 Vision System 54
3.1.1 Synthetic Vision 55

3.1.2 Object Recognition and Color Coding....55
Color coding....55
Position and size of an object....56
3.1.3 The Octree as Visual Memory Representation....58
3.1.4 The Update of the Octree....59
3.1.5 The Vision in the L-system....60
3.3 Tactile Sensors....61
3.4 The Audition Sensors....61
4. Actor Representation....63
4.1 L-system Modeled Actors....63
4.2 Imported Humanoids....65
4.3 VLNET Interface....66
4.3.1 The VLNET System....67
4.3.2 The VLNET - L-system Interpreter Interface....68
5. Behaviors....69
5.1 The Behavior Control....69
5.2 Global Navigation....71
5.2.1 Path Searching....71
5.2.2 Path Exploring....73
5.2.3 Heuristics....73
5.2.4 The Navigation Automata....74
5.3 Observing....76
5.4 Talking....76
5.5 Listening....77
5.6 Tennis Players....78
5.6.1 Game Automata of an Actor....80
5.6.2 Impact Point Estimation and Learning....81
5.6.3 Impact Point Determination....83
5.6.4 Game Strategy and Stroke Planning....84
5.6.5 Hit Determination....85
5.6.6 Path Planning....86
5.7 Tennis Referee....88
5.8 Interactive Tennis Player....89
5.9. Walking on Sparse Foothold Locations....90
5.9.1 Introduction....90
5.9.2 Map Extraction....92
5.9.3 Planning....93
5.10 Behavioral Control by Production Rules....95
5.10.1 Escaping from a maze based on vision....96
5.10.2 Random motion with collision detection....97
6. Implementation....99
6.1 System Architecture....100
6.2 The L-system Definition File....101
6.3 L-system Interpreter....104

6.3.1 Visual Memory Implementation 105
6.3.2 Shared Memory Interface 106
6.3.3 Import and Export of Data 108
Surface Manager SM 108
Data Channels 109
Actor Trajectories 109
6.3.4 Behavior Control 110
6.4 POST Speech Recognition 111
6.5 Image by Image Film Production 112
6.6 Hardware and Software 114
7. Case Studies 115
7.1 Force Field Animation 115
7.1.1 Tree - particle interaction 115
7.1.2 Dynamics of a Tennis ball 116
7.1.3 Net of a tennis court 117
7.1.4 Ground and gravitation 117
7.1.5 Tennis racket 118
7.1.6 Wind 118
7.1.7 School of Fish and Butterflies 118
7.2 Plants, Fractal, Crystals 121
7.2.1 Surreal L-structures 121
7.2.2 Plants, Crystals 121
7.3 Navigation 121
7.3.1 Navigation by automata 121
7.3.2 Vision Based Human Free Walking on Sparse Foothold Locations 122
7.3.3 Imported Actor Trajectories 123
7.3.4 Navigation by Production Rules 123
7.4 Tennis 124
7.4.1 Tennis with Autonomous Actors 124
7.4.2 Interactive Tennis through VLNET 126
8. Conclusion 129
8.1 Future Work 129
8.1.1 Olfaction 130
8.1.2 L-system User Interface 130
8.1.3 Evolution of L-systems 130
8.1.4 Particle System 131
8.2 Epilogue 131
Appendix A: L-system BNF 133
Appendix B: Semantic of BNF 137
Appendix C: Semantic of Symbols 147
C.1 Turtle Manipulation Symbols 147
C.2 Camera, Material and Light Control Symbols 148
C.3 Symbols for Geometric Shapes 149
C.4 Force Field Symbols 152

C.5 Actor Symbols 153
C.6 Special Symbols 154
Appendix D: Sounds 157
Appendix E: Scripts 159
Appendix F: VLNET Interface 163
Appendix G: Heuristics for Path Searching 165
Appendix H: The Fourth-order Runge Kutta method 169
Appendix I: Color Figures 170
Literature 171
Curriculum Vitae 177

1. Introduction

1.1 Motivation and Objectives

Nowadays computers enable us to create virtual environments offering a vast field of applications in domains as interactive computer games, interactive education, simulations, synthetic film productions, architecture and engineering. The role of such virtual environments becomes more and more important for our society in the next years. With virtual environments we try to reconstruct reality or to create fantastic imaginary worlds limited only by our imagination. A real environment consists of inanimate matter and living creatures as plants animals and humans obeying physical laws and behaving according to their inherent nature. Living creatures have the possibility to reproduce themselves, to grow and to react to their perceived environment. Even plants sense their environment and react to certain stimuli as light, gravitation, temperature and humidity. Animals and humans, in addition, can move around, and they show instinctive behaviors. Finally, humans are also equipped with some intelligence. Of course, with the present technical possibilities we can't exactly reproduce reality in a virtual environment. All we can achieve is a simplified model of a real environment. We can try to imitate certain principles and rules of reality in such a way that it can be recognized by an immersed user.

The objective of this work is to create a versatile, real time structured behavioral animation system allowing to experiment with virtual environments and autonomous agents. In reality, all autonomous agents are equipped with sensors through which they get information from the environment. Everything they know from their environment they learn through their sensors. This information severely influences their behavior. Thus, the first step in modeling "natural" behavior of autonomous agents is the introduction of synthetic sensors through which they will get all their environmental information. We will present sensor models for vision, audition and touch, which seem to us to be the most important senses for humanoid actors. During a real time structured animation an autonomous agent can only know the past, and at each time step it has to decide what to do next, based on its sensorial current input, its internal state and its knowledge. Thus, autonomous actors can be partners in interactive applications.

Such a complex animation system is composed of several interacting sub-systems as geometric modeling, acoustic modeling, physical modeling, synthetic sensors or actor behaviors, for example. For its realization we adopted a systemic approach [JO94, GABO92] which views the animation system as an organized entity composed of interdependent elements which are understood in terms of their relationships inside the global entity. According to [GABO92] computer science, artificial intelligence, ecology and psychology are privileged domains for systemic approaches as all of them are composed of interacting elements forming a complex entity.

In a behavioral animation system we first have to model the environment, the actors and the behavior of the actors before starting an animation. The classic description of 3D environments by polygons can be a very tedious work. To exploit fully the advantages of a computer and to facilitate the designing work of complex worlds we decided to develop a rewriting system having an important data amplification factor. In a rewriting system or a production system at each time step the environment is described by a symbol string generated by the computer according to a set of production rules. With such a production system we can symbolically describe not only the topology, but also the growth and behavior of complex articulated or branched objects. A text file of one or several pages containing these rules can produce several billions of bytes of images of a video film sequence. It can be considered as a compression of the film with a factor typically bigger than one million. In general, a production system needs not to build up a 3D database of the complete environment. It can directly draw objects during the interpretation of a symbolic environment

string. Thus, the synthetic vision sensor approach for environmental information passing to the agents is an advantageous principle when using a production system.

In Computer Graphics image and film production is a necessity. Therefore, an animation system has to support it. Especially, film production by synthetic images is a tedious work for users of modeling and animation systems. Very often parts of the 3D environment are created by different programs with different scaling, orientation and data formats that have to be put together before the final rendering. Camera settings, key frame animation and sound tracks have to be coordinated. Very often the final product can only be seen after days of calculation revealing perhaps some errors or aesthetic problems that provoke a repetition of the whole procedure. That's why we paid special attention to an integration of an automatic film sequence production facility, allowing after a fast preview of the animation an automatic ray traced image by image video film production with synchronized sound track. Another objective is to enable a user of the software familiar only with L-systems to model and to produce easily film sequences without being a specialist for a dozen of other related software packages.

An important issue of this work is to be open to already existing [TNH96] or to future humanoid software at LIG. As these software libraries are already very complex, we decided to provide shared memory interfaces for inter process communication. This provides several advantages. First, the development can clearly be separated in smaller, easier compilable and maintainable units. Second, on multi - processor machines, as the Onyx from SGI, for instance, an application will turn much faster as parallelism is an inherent feature of this approach. Third, several different languages, such as C, C++ or perhaps Lisp in future development, can be used in the same project, and extensions to networked applications are easily implemented.

In all these different domains of behavioral animation a lot of specialized work has already been done. A mayor contribution of this work is that it combines much systems in a systemic approach to a new entity, a behavioral animation system suitable for studying and modeling synthetic sensor based humanoids in a convenient virtual environment. We think that the originality of the present work is mainly given by its following features:

- Development of a systemic, real-time structured animation system for sensor based humanoids.
- Control and modeling of the virtual environment (acoustic, geometric and force field model) by L-systems
- L-system extension with interacting particles, synthetic sensors (vision, touch, and audition) and behavior control through production rules and automata.
- Synthetic vision with visual memory and associated behaviors.
- Real time structured sound rendering for film production (and in future for VR participants)
- It supports interactive and networked partners mixed with autonomous sensor based actors. For example, a tennis game facility with an autonomous referee, with autonomous players and with interactive users as players has been developed.

1.2 Organization

The thesis is organized in 8 chapters. Chapter 1 summarizes the motivation and objectives of this work, and presents a literature review on L-systems, on computer vision and acoustics, on behavioral animation and on physical modeling.

In chapter 2 we present how we model the virtual environment where the behavioral animation takes place. We present in detail the animation loop of the L-system interpreter and the actual L-system we developed. We show how the geometric, physical and acoustic modeling of the environment is realized inside the L-system interpreter.

The actor - environment interface is presented in chapter 3. This chapter describes the synthetic sensor models for vision, touch and audition we use for the behavioral animation.

In chapter 4 we discuss the different actor representations. We start with simple L-system modeled actors. Then, we pass to humanoid actors imported through a shared memory interface. Finally, we present the immersion of the autonomous actors of the L-system interpreter into the Virtual Life Network.

Chapter 5 addresses behavior modeling. After the description of the behavior control, the currently implemented automata based behaviors are presented. The chapter finishes with a general description of behavior modeling by production rules of an L-system.

In chapter 6 we present the system architecture and some particular implementation details. In particular we present the L-system definition file format, some modules of the L-system interpreter, the speech recognition system and the image by image film production support offered by the animation system.

Chapter 7 provides some application examples realized with the animation system. In particular, we present force field animations, plant, fractal and crystal modeling, navigation results and tennis game simulations.

The conclusion and some future research directions are given in chapter 8. Finally, several appendices contain useful supplementary information.

1.3 Literature Review

The creation of the proposed versatile behavioral animation system covers a vast field of different domains as computer graphics, physical modeling, acoustics, artificial intelligence, formal languages, games and film making. In the next sections we give a non exhaustive overview of some literature that influenced our work. We mention also the points we adopted for our animation system, and in which way other work differs from our approach.

1.3.1 L-systems

Smith [S84], already in 1984, stated that procedural models of plants and trees handling plant growth, sport an efficient data representation and have a high "database amplification" factor. These models were based on an extension of the well-known formal languages of symbol strings to the lesser-known formal languages of labeled graphs, Keywords of his article giving an overview of procedural models at that time are: plant, tree, graftal, fractal, particle system, parallel graph grammar, L-system, database amplification, Computer Imagery.

Lindenmayer introduced L-systems already 1968 [L68] as a mathematical theory of parallel production systems. In [PRUS90] Prusinkiewicz and Lindenmayer present the Lindenmayer systems - or L-systems for short - as a mathematical theory of plant development with a geometrical interpretation based on turtle geometry. The authors explain mathematical models of developmental processes and structures of plants and illustrate them using beautiful computer-generated images. This book is highly recommended for everybody who wants to get familiar with basic notions related to L-systems and their application in computer graphics. Besides a lot of biological models some of the topics covered are rewriting systems, D0L-systems, turtle interpretation of strings, bracketed L-systems, stochastic, parameterized, conditional, context sensitive and timed L-systems, growth functions, and surface models.

Production systems and L-grammars are very powerful tools for creating images. From a user-defined axiom and a set of production rules, the computer creates images with a complexity dependent only on the number of times the productions are applied. The theory of L-systems has been mainly used for the visualization of the development and growth of living organisms like plants, trees and cells. In [NTTU92] we presented the software package LMobject that realizes a timed and parameterized L-system with conditional and pseudo stochastic productions for animation purposes. With this software package a user may create realistic or abstract shapes, play with various tree structures and generate a variety of concepts

of growth and life development in the resulting animation. To extend the possibilities for more realism in the pictures, we added external forces, which interact with the L-structures and allow a certain physical modeling. External forces can also have an important impact in the evolution of objects. Prusinkiewicz and Lindenmayer have proposed two simple cases of external forces. In the first method, the 3D turtle that interprets the symbolic grammar may be aligned horizontal to a vector representing the gravity. Thus, an object (a tree, for example) is able to "feel" the gravity and to react. The second case is specific to plant and tree modeling and allows the simulation of tropism, which is responsible for the bending of branches towards light sources.

In our extended L-system interpreter, we generalized the tropism concept to a general force field interaction between the force fields of particles and branching structures. Tree structures can be elastically bent and animated by time and place dependent vector force fields. The elasticity of each articulation can be set individually by productions. Therefore, the bending of branches can be made dependent on the branches' thickness, making animation more realistic. The force fields too, can be set and modified with productions.

In [PRUS93] Prusinkiewicz and al. introduce a discrete / continuous model of plant development that integrates L-system-style productions and piece-wise continuous differential equations. The model is suitable for animating developmental processes in a manner resembling time-lapse photography. The proposed technique is illustrated using several flowering plants. The key concept is the integration of discrete and continuos aspects of model behavior into a single formalism, called differential L-systems (dL-systems), where the productions express changes to the model, and differential equations capture continuous processes, such as the gradual growth of internodes. However, dL-systems are not easy to understand and to program as the user must be skilled in differential equation theory. We think, that the use of explicit growth functions is user friendlier and results in faster execution times, important for interactive real time applications.

Tunbridge and Jones present in [TJ93] a typical application of L-systems. To model the growth and structure of the fungus Aspergillus nidulans they developed a parametric L-system using biological understanding of the flow of nutrients through the fungus to control the simulation of growth, with L-system productions being dependent on the levels of nutrients, stored as parameters, in various parts of the model.

C. Streit and H. Bieri present in [SB93] a stochastic, parametric and context independent table L-system (spT0L-system) with extensions for particle systems, tropism forces, fractal mountain generation and boundary object definition for synthetic topiary. Moreover, they show that parametric L-systems are equivalent to Turing machines, which gives an idea of the theoretic power of parametric L-systems.

In "Synthetic Topiary" [PJM94] Prusinkiewicz and al. extend Lindenmayer systems to environmentally-sensitive L-systems by using parameterized special query modules. Their parameters are set to the position or orientation of the turtle during the interpretation of the string. At the following derivation step this position information can be used in the conditions of the production rules. They implemented these query modules in a parametric, conditional context sensitive and stochastic L-system handler and illustrated the formalism by modeling the response of sculptured plants found in topiary gardens. The query modules, one of them represented by the symbol "?", for instance, are necessary to import environment data into the formal parameter space of the L-system. We implemented this feature also in our L-system interpreter. But, as it is real time structured, it wouldn't be really necessary as we can directly access environment data or sensorial environment data through specialized functions, usable in the conditions of production rules. We used this feature to develop general behavioral L-systems published in [NT96].

In [MP96] Mech and Prusinkiewicz extend the formalism of environmentally sensitive L-systems to open L-system with constructs needed to model bi-directional information exchange between plants and their environment. They illustrate the proposed frame work with models and simulations that capture the development of tree branches limited by collisions, the colonizing growth of clonal plants competing for space in favorable area, the interaction

between roots competing for water in the soil, and the competition within and between trees for access to light. This recent work of Mech and Prusinkiewicz shows interesting parallels to our work. First the plant is considered as a living sensor based organism interacting with an environment. The concept of environmentally sensitive L-systems [PJM94] is extended with communication modules (symbols) for bi-directional communication with the environment. The environment itself is a concurrent process especially modeled for a given problem. In our work the sensor based actors are not limited to plants, they can be “animals” or “humanoids”. We introduced three types of environments, i.e. the geometric world, the force field world and the acoustic world with the corresponding sensors for the actors. To each of these worlds we have a set of symbols which allow the modeling of the corresponding world through the L-system. These symbols correspond to the communication modules of open L-systems which export parts of the plant to the concurrent environment. Whereas open L-system use special communication modules to import sensorial information into the parameter space of L-systems, we use special sensor functions usable directly in the conditions of production rules. With both approaches plant or actor behavior can be modeled by using production rules in a permanent feed back loop with the virtual environment(s). Whereas open L-systems are based on parametric L-systems, we developed timed parametric L-systems with growth functions which are especially useful for piece-wise continuos interactive simulations and film productions. Our behavioral L-system and open L-system opt conceptually for the same paradigms in the simulation of living organisms, i. e:

- a systemic approach (living organisms and the environment are complex and should be modeled by different communicating systems)
- sensor based behaviors (the “behavior” of plants, animals and humanoids is influenced by a constant feed back loop with the environment)
- rule based derivation and control system for data amplification and emergent behaviors

In "Turtle Geometry" [ABE84] Abelson and diSessa describe an innovative program of mathematical discovery that demonstrates how the effective use of personal computers can profoundly change the nature of a students contact with mathematics. The book proceeds from a novel "procedural" view of the elements of plane geometry to such central ideas in modern mathematics as symmetry groups and topological invariance. Geometric figures are regarded not as static entities but as tracings of an imaginary "turtle". Some of the subjects covered are random motion, branching processes, space-filling designs, vector operations in two and three dimensions, topology of curves, maze-solving algorithms, intrinsic curvature of surfaces, spherical and "cubical" geometry, piece-wise flat surfaces and General Relativity. The turtle is introduced as a mathematical "animal" with smell, touch and sight sensors and with "predator-prey" and "following-chasing" behaviors. By using the "Pledge Algorithm" the turtle can even escape from a maze by using only local information. This turtle concept is successfully reused in the interpretation of L-systems, and the integration of a turtle as mathematical autonomous animal or actor equipped with sensors leads to a versatile behavioral L-system, combining behavioral turtle geometry and L-grammars, where the behavior of actors and the dynamic environment are modeled by production rules.

L-systems are mainly used for modeling plants in computer graphics. Other theories have been developed by Aono [AK84] for the generation of trees, and by de Reffye et al. [REF88] who presented plant models faithful to botanical structure and development. Like de Reffye, Lecoustre et al. [LR92] are interested in botanical precision and content in their work. In [FL94] Lienhardt describes basic principles of topology-based methods for simulating metamorphoses of zero to two dimensional natural objects like particle systems, trees and plant organs (leaves, flowers). Although our behavioral L-system offers features for modeling botanical plants, we made no special effort, apart form some students work, to simulate precise botanical plants. Our interest is mainly in the data amplification factor of L-systems for defining virtual dynamic environments and in the behavioral modeling potential of L-systems extended with sensors.

1.3.2 Vision

Synthetic vision, in combination with a visual memory and path searching procedures, is the most important sensor of our behavioral L-system interpreter. Our synthetic vision is based on Z-buffer techniques and simple color coding for semantic information extraction from vision. The next lines describe some background research in synthetic vision related to our work.

E. Catmull [C75] introduced the Z-buffer algorithm which is a simple method for hidden surface removal. The Z-buffer consists of an array containing the depth value for each pixel of the image to be displayed. The algorithm uses these Z-buffer values for efficient rendering of 3D scenes. Nowadays, Z-buffer algorithms are generally implemented on graphics terminals which offer the corresponding technology.

Reynolds [R88] proposed the use of Z-buffer images providing a map of the distances to the nearest obstacles to detect obstacles along a flight path of an object.

> "This sort of image can be generated from the real world using various range-finding devices (radar, sonar, laser interferometer) or can be made synthetically from a geometric database. In fact a certain class of rendering algorithm produces a Z-buffer image as a side effect of creating the visual image."

He proposed very low resolution images of about 4x4 pixels to detect clear paths to travel before an obstacle will be encountered. Later, Renault et al. [RTT90] used the Z-buffering hardware graphics of workstations to render a bitmap projection of the actors point of view. The color of an object is unique and serves to identify the semantics of an object in the image. This synthetic vision was used to create an animation involving synthetic actors moving automatically in a corridor avoiding objects and other synthetic actors. The behaviors of the actors are modeled with displacement local automata (DLA). A DLA is an algorithm that can deal with a specific environment. Two DLA's called follow_the_corridor and avoid_the_obstacle were described in detail. The actors got their visual input exclusively from their environment through a 30x30 pixels image (color + Z-buffer map).

Jean-Michel Jolion [JO94] gives a general overview of methodologies for computer vision. First he discusses the Marr paradigm. Marr [MA92] proposes an information processing system as a framework for the basis of vision. He defines vision as follows:

> "Vision is a process that produces from images of the external world a description that is useful to the viewer and not cluttered with irrelevant information".

This approach lead to the recovery school. To overcome the ill-posed nature of most computer vision problems the regularization theory (scenes are made of regular surfaces) with a well-defined mathematical framework was developed but it failed to take into account the actual complexity of natural scenes. Then, Jean-Michel Jolion [JO94] presents the goal-directed approach.

> "This approach can be understood as a generalization of the active vision paradigm in that sense that new constraints, extracted from the goal, have been added in order to make the vision problem more tractable".

Thus, vision is studied in the context of behaviors. In the last part of his paper Jolion [JO94] proposes a systemic approach to vision. In this framework he tries

> " to exploit of all of the sources of constraints, and, there by, to reconcile some of the previous approaches like recovery school and purposive vision". .. "The systemic paradigm addresses the problem of designing a general methodology for system understanding which is not reductionist. .. "The systemic paradigm argues that the power of a vision system is not a particular architecture but its complexity."

The systemic approach of constraint analysis is based on the ten following rules.

1. Keep the variety. Do not try to oversimplify by looking to a unique framework.

2. Do not open the control loop as local modifications in one component can have global consequences.

3. Look for location amplifications. Look first for the components that have a large influence on the system.

4. Restore the equilibrium of the system by the way of local actions. A vision sensor, for example, maintains itself the focus.

5. Maintain constraints by avoiding oversimplifications and by accepting and treating noise data.

6. Vary to better unify. Union, in the sense of combination, creates complexity and results in systems of higher level of organization.

7. Be adaptive. A system should be able to learn.

8. Look for a goal rather than a detailed behavior.

9. Study the actual complexity of the communication network. Among the issues here are transfer mechanisms, their limits, and more generally all component behaviors which can lead to conflicts over access.

10. Respect the time delays in the interaction of the system components.

A vision system makes part of behavioral animation system which is a more complex system as it includes several sensors, behaviors and environment modeling. Therefore, a systemic approach to a behavioral animation system is recommended. We adopted such an approach and we refer to it in the following chapters.

Other examples of not Z-buffer based vision systems are given by the following authors. Reynolds [REY93] described an evolved, vision-based behavioral model of coordinated group motion. Tu and Terzopoulos [TTA94], [TTP94] introduced a vision-based perception for fishes. In [TR95] Terzopoulos and Rabie propose a new paradigm for active vision research. Their software prescribes artificial animals, situated in physics-based virtual worlds as autonomous virtual robots possessing active perception systems. The animals autonomously control their eyes and muscle-actuated body, applying computer vision algorithms to continuously analyze the retinal image streams acquired by its eyes in order to locomote purposefully through their world. By emulating the appearance, motion and behavior of real fishes these animat are capable of spatially non uniform retinal imaging, foveation, retinal image stabilization, color object recognition, and perceptually-guided navigation. The authors make an effort to re-model biological vision systems, based on 2D image processing, in order to approach natural animats and to make their results useful for robotics. Our approach by using the Z-buffer of the vision image is more pragmatic and profits from the hardware Z-buffer of the IRIS graphics system for fast calculation. It simulates vision more on the functional level than on the biological level.

An interesting special application of the principle of synthetic vision is presented by Benes [B96] who uses Z-buffer techniques and color coding in the estimation of light in simulation of plant development. He renders the tree from the point of view of the light source. As each leaf is color coded and identified a color histogram of the rendered image can be used to calculate the amount of light “seen” by each leaf. Then, this information can be employed for the further development of each leaf.

In designing autonomous robots people encounter similar problems as in designing autonomous virtual actors. In robotics, however, perceptual problems are much more difficult as sensorial data are not very precise, and as computational power is more limited. Nevertheless, a lot of principles and results from the domain of robotics can be assimilated for virtual autonomous actors. As literature on robotics is huge, we review only some typical papers that are related to our work.

Boulic [B86] describes modeling and simulation of active perception sensors in a CAD system for robotics. Roth-Tabak et Jain [ROT89] introduced a new algorithm using dense range data to generate, refine and update a 3D voxelized environment model of a static environment. These data can also be used by an autonomous system for navigation and path planning as well as object recognition and manipulation. Elfes [ELF90] described a

> "framework for robot perception, real-world modeling, and navigation that uses a stochastic tessellated representation of spatial information called the Occupancy Grid. It is a multi-dimensional random field model that maintains probabilistic estimates of the occupancy state of each cell in a spatial lattice. Bayesian estimation mechanisms employing stochastic sensor models allow incremental updating of the Occupancy Grid using multi-view, multi-sensor data, composition of multiple maps, decision-making, and incorporation of robot and sensor position uncertainty."

These Occupancy Grids (2D) are used in mobile robot navigation and obstacle avoidance.

In our system we adopted the 3D voxelized environment model for the representation of an actor's visual memory of the perceived dynamic environment. As the synthetic vision, however, is more precise than artificial vision used in robotics, our 3D octree occupancy grid needs only simple statistics for its construction and update. The reconstruction of the seen environment by a "visual memory" of an actor and its use in global navigation is published in "Vision-Based Navigation for Synthetic Actors" [NRTT93] and in "Navigation for Digital Actors Based on Synthetic Vision, Memory and Learning" [NRT95].

1.3.3 Acoustics

Real life is full of different sounds and the addition of music or other sounds in video productions can considerably increase the quality of the final product. In real life, the behavior of persons or animals is very often influenced by sounds. According to E. M. Wenzel [W92] "the function of the ears is to point the eyes". Audition is a temporal sense and we are very sensitive to changes in an acoustic signal change. We can locate objects in space, especially when we are moving. Acoustic signals transport a lot of semantic and emotional information and tell us a lot of the position of a sound source relative to us and the propagation paths in the environment.

Durand R. Begault [DB94] published the first introductory book on 3D sound for virtual reality and multimedia. It includes descriptions of reverberation modeling and auralization for acoustical design applications. It overviews many different applications for spatialized sound including: auditory feedback, communication systems, aeronautics, computer music, sonification, television and computer interfaces. Other key features are the introduction to physics and perception of sound and digital signal processing related to spatial hearing and the outline of the components of spatial auditory displays. It also includes a chapter on resources for hardware, software, and publications related to virtual acoustics. Begault wrote:

> "Video game designers, computer musicians, virtual reality hackers, multimedia sound card designers, audiophiles, human factors engineers, broadcaster, recording engineers, or anyone interested in synthesizing auditory space will find this book helpful in realizing a successful design for their particular application."

The following citations underline the importance of sound in virtual environments:

> "Our perceptual system is often explained as being dominated by vision. Indeed, although a loudspeaker may be displaced from the actual location of a visual image on a television or movie screen, we can easily imagine the sound as coming from an actor's mouth or from a passing car. This is an example of what is termed visual capture. .. The point is that a sound in itself has considerable "virtual potential". For instance, it is currently expensive to provide tactile feedback in a virtual environment, but it is possible to use sound to suggest tactile experiences".

E. Wenzel [WF90] presents an overview of current techniques of real-time virtual displays using special hardware to satisfy real-time constraints. A typical example is the Convolvotron real-time digital signal processor designed by Scott Foster [FO88]. This processor places each input signal in the perceptual 3D space of the listener by convoluting the input data with filter

coefficients determined by the positions and orientations of target and listeners. The resulting data streams are presented over headphones. Foster and Wenzel [FOWEV91], [FOWER91] describe typical applications where the processing speed is sufficient for simulating a single source plus six early reflections in small reverberate environments. However, as the processing speed of today's computer generation already allows simple real time sound rendering, we opted for a software sound rendering for our system, which is more flexible than a hardware solution.

Important contributions to the field of auditory virtual environments have been made by Blauert and Lehnert. In [B83] Blauert describes the psychophysics of human sound localization. In [L94] Lehnert reviews the fundamentals of auditory virtual environments and discusses the needs of real time implementations. Further references to their work can be found in [L94]. Moreover, in a state of the art report about acoustic simulation for visualization and virtual reality, P. Astheimer [A95] explains and discusses the general approaches for acoustic simulations, basic mapping and rendering techniques, device utilization, distributed and parallel systems, synchronization of graphical and acoustical presentations and real time aspects.

A general methodology to produce synchronized sound tracks for animation is described by Tapio Takala and James Hahn [TAKHA92] in "Sound Rendering". A sound world is modeled by associating a characteristic sound for each object in a scene. It is described with a sound event file, and is rendered in two passes. First, the propagation paths from 3D objects to each microphone are analyzed and used to calculate sound transformations according to the acoustic environment. In the second pass the sounds associated with objects are instantiated and summed up to the final soundtrack. Real time software sound rendering is only possible for simple sound environments. Nevertheless, we developed a real time structured, simple sound renderer described in [NT95b], which we use now for sound track generation for film sequences, and which we can hopefully use soon for interactive applications. On the multiprocessor ONYX workstation with a sound card it should already be possible to do stereo real time sound rendering for simple auditory environments.

1.3.4 Behavioral Animation

In [Z82], one of the first papers on task-level animation, Zeltzer describes the use of multiple layers of control, arranged in a hierarchy that successively refines the actor's task until low-level motor control units move the limbs. At the top level of the control structure is the task manager accepting task descriptions from the user and decomposing each of them into a list of component skills. This middle level of the control hierarchy consists of all the implemented skills and associated motor programs. This multiple level control hierarchy can be found in similar form in other task-level animation systems. In the L-system interpreter, however, it is difficult to localize these levels. Motor control units, for example, can be realized by production rules, by automata, by growth functions of symbols or by external processes. The humanoid motor controls such as walking or grasping are integrated in a special external process. The definition of a "tennis player" role was established by an automata control in combination with some production rules. Higher level roles could be defined with production rules, or a new automata could include the "tennis player" role.

In [CRO85] the authors describe a system for a mobile robot, useful for 2D navigation in a finite, pre-learned domain such as a house, office or factory. The robot is equipped with a rotating ultra sonic range sensor. Navigation is based on a dynamically maintained model of the local environment, called the composite local model. It integrates information from the rotating range sensor, the robot's touch sensor, and a pre-learned global model. The most recent sensor scan is used for constructing a line segment description of the local environment. The estimated position of the robot is corrected by considering the difference in position between observed sensor signals and the corresponding symbol in the composite local model. The authors also describe a learning technique in which the robot develops a global model and a network of places, which is used in global path planning. The segments are recalled from the global model to assist in local path execution. Conceptually, we do

something similar in the L-system interpreter navigation, but with a 3D generalization. The synthetic vision provides more precise sensory information, and we need only one environment model, a voxelized occupancy octree grid that is dynamically maintained. Local and global path planning can be done directly in this voxel space.

Behavioral animation was studied in detail by Reynolds [REY87]. In the introduction he treats different methods and problems of behavioral animation. He describes the aggregate motion of a simulated flock of birds by a distributed behavioral model, where the birds choose their own path. A bird is implemented as an independent actor that navigates according to its local perception of the dynamic environment, the laws of physics that rule its motion, and a set of behaviors. Then, the aggregate motion of the flock results from the dense interaction of the behaviors of the individual birds. Reynolds compares his birds to interacting particles of a generalized particle system where the particles have shape, orientation and more complex behaviors as the particles of normal particle systems [R83]. In an other comparison he relates the birds and their flight paths to the "3d turtles" of [ABE84] that unite position and heading. The behaviors that lead to flocking are collision avoidance, velocity matching with nearby birds and flock centering through an attempt to stay close to nearby flock mates. These independent behaviors are arbitrated by a navigation module, and then the results are passed to a pilot and flight module of the actors. Simulated perception plays an important role in his behavioral model. Reynolds states, that giving each simulated bird perfect and complete information about the world leads to obvious failures of the behavioral model. A result of Reynolds paper is just the fact that the aggregate motion that we intuitively recognize as "flocking" **depends** upon a limited, localized view of the world. The birds' perception system is quite ad hoc and avoids simulating artificial vision as it is an extremely complex problem. But Reynolds concludes that if the actors could "see" their environment, they would be better at path planning than the current model. The perception model tries to make available to the behavior model approximately the same information that is available to a real animal as the end result of its perceptual and cognitive processes. In Reynolds's implementation the perceived neighborhood is defined as a spherical zone of sensitivity centered at the actors local origin. The magnitude of the sensitivity is defined as an inverse exponential of distance. The results of his work are illustrated by the film "Stanley and Stella in Breaking the Ice" by Whitney / Demos, where a school of fish and flock of birds are shown in a short love story.

In his paper, Reynolds addresses also a typical problem of behavioral animation. As the animator is not any more responsible for all motions, an autonomous actor can behave in a unexpected and uncooperative manner. Reynolds wrote:

> "Thus, the animator's job becomes somewhat like that of a theatrical director: the character's performance is the indirect result of the director's instructions to the actor. One of the charming aspects of the work reported here is not knowing how a simulation is going to proceed from the specified behaviors and initial conditions, there are many unexpected, pleasant surprises. On the other hand, this charm start to wear thin as deadlines approach and the unexpected annoyances pop up. This author has spent a lot of time recently trying to get uncooperative flocks to move as intended ("these darn boids seem to have a mind of their own!")."

We made similar experiences with our autonomous actors. When we tested, for example, the "maze escape" behavior based on synthetic vision, visual memory and heuristic path searching, an actor found an object close to a wall that allowed it to step on the top of the wall. There, it continued its search by balancing on the top of the walls of the maze. It was an amazing result, but not intended by us as the object close to the wall was a flower.

Susan Amkraut and Michael Girard showed in the film "Eurhythmy" [GA90] a flock of birds flying around and avoiding collisions between themselves and obstacles in their environment using a force field animation system to realize the simulation. Repulsion forces around each bird and around static objects are responsible for collision avoiding. At the beginning of the animation, the space field and the initial positions, orientations, and velocities of objects are defined and the rest of the simulation is evolved from these initial conditions. We realized a similar approach to group animation with the integration of a force field based particle system into the L-system interpreter.

In [RC90] the authors present a method for producing computer-generated micro worlds populated by synthetic actors. They use a LISP expert system to interpret a script from known and inferred relational constraints. The user specifies a set of boundary events in time and space for each figure, and the expert system finds a set of behaviors consistent with those events. Their application did not run in real time and used boxy robot-like forms. There was no dynamic simulation. Lisp is a commonly used programming language of the AI community, and "higher level intelligence" for autonomous actors could be easier modeled with LISP. The computer graphics community, however, prefers C or C++ for fast and efficient execution of programs. Our behavior control mechanism, based on text string automata stacks allows easy future extensions with special LISP processes for more actor intelligence. Such LISP processes could create appropriate text strings with the automata syntax. Then, they could communicate them through socket interfaces, for example, to the L-system interpreter where they were simply popped on the actor's behavior stack for execution.

Ridsdale [RID90] believes that task-level animation using a connectionist model of skill memory, implemented as collections of trained neural networks may be able to breathe an interesting degree of life into the behavior of simulated actors. His long-range goal is to merge an existing expert system, responsible for laying out the general plan of action for the agents, with a series of connectionist networks each trained to perform a skilled task. He states, that explicit rules are most useful when constraints can be formulated from established principles, and that connectionist models are better applied when the desired action is ill defined. He advocates a hybrid approach using rules for scene-level planning and a connectionist model for skill-level planning. To test the idea of acquiring a set of skills in the form of a connectionist model, he learned an actor several skills in playing handball, skills as left-bounce serve, straight-serve or underhand serve. Until now, we didn't implement any connectionist model for skill - level planning as the currently implemented skills were well enough defined. Nevertheless, the use of neural networks for ill defined actions is a challenge for future work.

Wavish and Graham describe in [WG94] a three-layer model for structuring systems of interacting agents based on situated action. Agents performing roles form the top layer of the model. The middle layer consists of the skills that agents need to perform their roles, and the lowest level provides the behaviors that are needed to realize the skills. The authors developed a production rule language called RTA to implement interacting agents. Sets of production rules relating the agent's action to its situation define skills. Roles are defined as sets of rules governing the appropriate deployment of behaviors and skills in relation to the perceived behaviors of other role-playing agents. To operate multi-agent simulations in real time, RTA is compiled to an efficient asynchronous digital logic circuit representation. This language represents a very specialized system for role, skill and behavior modeling with production rules. The behavioral production rule part of the L-system is conceptually similar. The behavioral part of the L-system interpreter concerning the production rules is less sophisticated. But on the other hand, it is more versatile and combines traditional L-system modeling, some physical modeling and behavioral modeling by force fields, automata and production rules.

Badler et al.'s book [BPW93] is intended for human factors engineers requiring current knowledge of how a computer graphics surrogate human can augment their analysis of designed environments. It will also help to inform design engineers of the state-of-the-art in human figure modeling, and hence of the human-centered design central to the emergent notion of Concurrent Engineering. Finally, it documents for the computer graphics community a major research effort in the interactive control and motion specification of articulated human figures. After the introduction and a historical background description body modeling, spatial interaction, behavioral control, simulation with societies of behaviors and task level specifications are the topics of the book, which is finished by an epilogue drafting a "road map toward the future". As Badler's work is real time oriented, he ignored auditory information processing, environmental factors and super-accurate skin models. They only touched perceptual, reactive and cognitive issues and they left issues of learning for later study. For our system, however, perceptual issues are main topics. We think that today's

computers are now fast enough for a limited use of sensor based autonomous actors in real time applications. The accurate skin model of [EAgent8125] that we use in our application, however, is still left to image by image rendering for film production.

In [BG95] Blumberg et al. discuss the problem of building autonomous animated creatures for interactive virtual environments which are also capable of being directed at motivational level, at task level, and at motor level. They detail a layered architecture and a general behavioral model for perception and action selection which incorporates explicit support for multi-level direction. The autonomous creatures use three types of sensing:

- "Real world sensing using real world "noisy" sensors.
- "Direct" sensing via direct interrogation of other virtual creatures and objects.
- "Synthetic Vision" in which the creatures utilizes vision techniques to extract useful information from an image rendered from their viewpoint.

The resulting image of synthetic vision is used to generate a potential field from the creature's perspective. Subsequently, a gradient field is calculated, and this is used to derive a bearing away from areas of high potential. The authors state that synthetic vision

> "was simple to implement, works well in practice, and is general enough to allow our virtual dog to wander around in new environments without modification".

Their behavior system is a distributed system composed of a loosely-hierarchical network of "self-interested, goal-directed entities", called behaviors. This system must weigh the potentially competing goals of the creature, asses the state of its environment, and choose the set of actions which make the "most sense" at that instant in time. It enables high level motivational or behavioral control. Based on their system the authors developed an autonomous dog which interacts with a user in a 3D virtual world. The dog has a number of internal motivations which he is constantly trying to satisfy. He responds also to some gestures and postures of the user. Moreover, the dog obeys a computational director which provides "directions" so as to meet the requirements of the story.

Our behavior system is not a distributed one. It is more task oriented. Each behavior must plan the cases where it can switch to other behaviors. Our stack based behavior treatment, however, allows a versatile subgoal treatment when necessary, and to switch back to the original behavior. It reflects the fact that people in general execute consciously only one task at a time. It is difficult to concentrate simultaneously on two things. At the motivational level, however, which is very often unconscious, a distributed model is advantageous and should be considered for future extension with motivational behaviors.

2. The Virtual Environment

According to a systemic approach we see the virtual environment as an entity illustrated in figure 2. It is composed of several interacting sub-systems. There are systems which do the acoustic, the geometric and the physical modeling. Other systems model the senses of touch, sight and audition. The actors and their behaviors are systems which constantly interact and exchange information with the other components of the virtual environment.

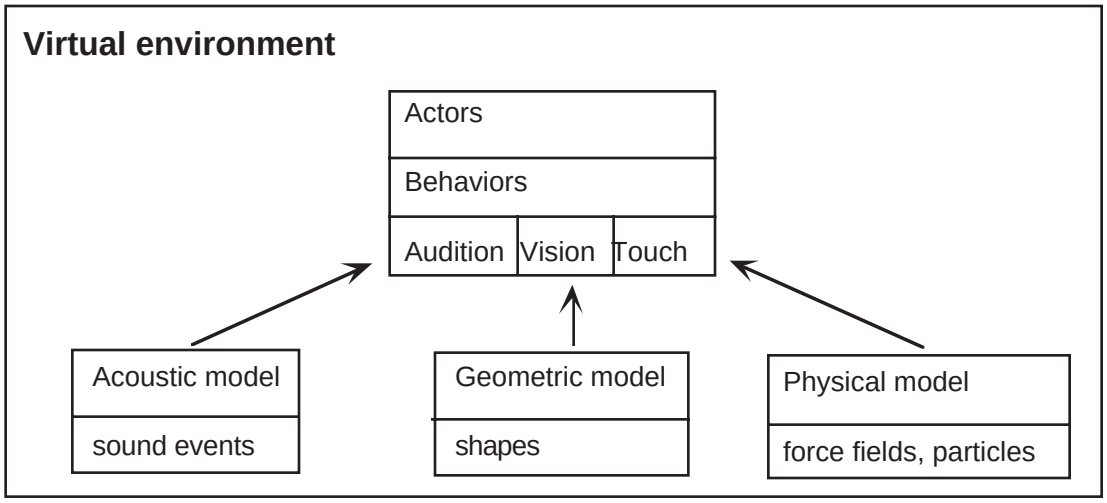


Figure 2: The components of the virtual environment for behavioral animation.

This virtual environment is controlled and synchronized by a behavioral L-system interpreter. Behavioral L-systems are timed production systems designed to model the development and behavior of static objects, plant like objects and autonomous creatures. Our L-system interpreter supports timed, parametric, stochastic and conditional production systems, force fields, synthetic vision and audition. We published parts of this system in [NTTU92], [NT93], [NT94c] and [NT95b].

A L-system is given by an axiom consisting of a string of parametric and timed symbols, and some production rules specifying how to replace corresponding symbols in the axiom during the evolution of time. The production system associates to its symbols some basic geometric primitives as cubes, spheres, trunks, cylinders terminated at their ends by half spheres, line segments, pyramids and imported triangulated surfaces. We define the non generic environment as a ground plane, imported surfaces or other static objects directly in the axiom of the production system. Generic parts like growing plants are defined by production rules having their germ symbol in the axiom. The actors are represented by a special symbol. The actors' geometric representation can vary from some simple primitives like cubes and spheres, over a more complicated skeleton structure to a fully deformed triangulated body surface. The actor representation depends on the purpose of an application. For tests a cube representation of an actor is sufficient. In interactive real time applications a simplified skeleton structure can be used. For ray traced video productions, however, only a fully deformed triangulated surface for an actor is convenient.

Such an L-system is given by an ASCII text file that can be created and edited by any text editor. The syntax and semantic of a formal L-system are discussed in the next sections and in section 6.2 (code 6.2.1) we can see a complete example of an L-system defining a tree. Our L-system animation system is able to read several L-system text files and to interpret them in parallel during an animation. One L-system text file typically models, for example, the

growth and topology of a plant or a tree or contains the rules of some actor behavior. Thus, an animation containing several types of plants and actors can easily be composed by using simply a library of already modeled L-systems. The modeling of an L-system is typically done by starting from a given L-system, and by alternatively editing and visualizing it. No compilation is necessary. Such an incremental development is especially useful for an artistic way to work, corresponding to a chaining of tests and creative decisions.

2.1 Animation System

In the following sections we present an overview of the system architecture with the animation loop of the L-system interpreter that drives and controls the whole virtual environment. Then, we establish two formal definitions of L-systems used during the development of the software package.

2.1.1 System Architecture and Animation Loop

Figure 2.1.1 shows the overall system architecture. In this section we only overview the functionality of these systems. In the following chapters they are explained in more detail. As we mentioned above, the L-system interpreter drives and controls the virtual environment where simulations take place. Moreover, it controls and synchronizes the external systems shown in figure 2.1.1.

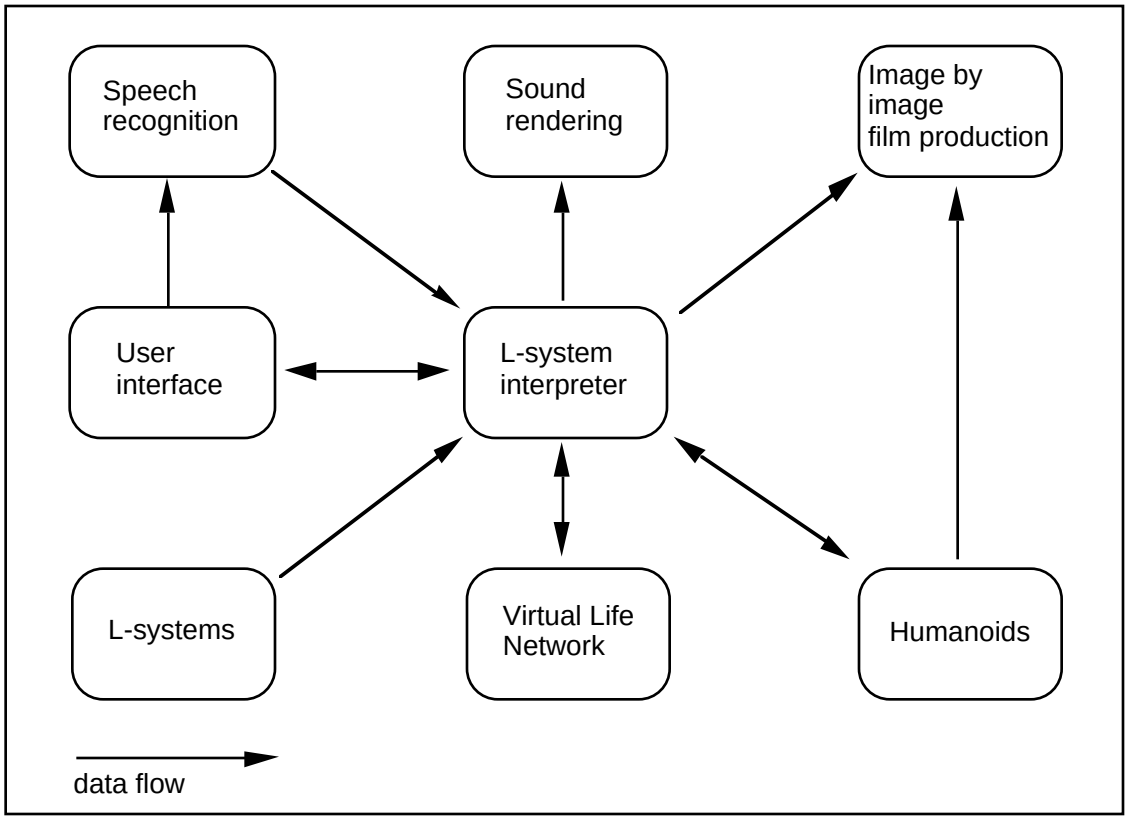


Figure 2.1.1: The overall system architecture

If a user wants to participate in an interactive real time animation, which is displayed on the screen, he can use interface devices as space ball, mouse or keyboard. He can also use a speech recognition system which transmits a limited set of isolated words to the acoustic virtual environment. Another possibility for a user to participate in the virtual environment is

offered by VLNET [PCTT95], that provides a natural interface for collaborative working and games. An animation is modeled by one or several user defined L-systems, which are read at the beginning of each animation. An animation can import virtual humanoids with sophisticated motor programs and skin surfaces. For ray traced image by image productions, the "Humanoids" system also produces a body surface file of each actor in the ray tracer's format. When sound rendering is specified, a log file of the sound events is created, which can be rendered to a sound track after the animation. This sound track can be added to the corresponding film sequence.

```
0.  - reading of N L-system definition files
    - initialization of modules and data structures
    - T = 0.0     /* the global time */

do { /* time loop */

    1.  - execution of a script file "Command0"
        - test for available disc space
        - write time to sound event track file
        - update file names
        - read data from shared memories and synchronize with concurrent processes

    for all windows of users and synthetic vision of actors {
    /* treat user and synthetic vision of actor s */

        2.  -set default camera
            - initialize window

            for all N L-systems {
                for some repetitions {
                /* repeated treatment of each L-structure
                defined by a L-system definition file */

                    3.  - determine position of the current object
                        - if (current object visible) treat  current object
                }
            }

            4.  - if (synthetic actor == current window) treat vision
                - if (rgb recording) record image
                - if (rgb file creation) save rgb file

    }

    5.  - execute a script file "Command1"
        - sound event update
        - T = T + ΔT

} while (T < end of animation)

6.  -close files
    -close video
```

Code 2.1.1: The animation loop of the L-system interpreter.

The animation system is real time structured, concurrent and essentially synchronized. The consequences of this architecture are that all sub-systems maintain an internal state and

proceed from one time step to the next by constantly exchanging information and updating their internal state. Generally, in interactive simulations the sub-systems can't know the future, as user reactions are not calculable. We opted for synchronous concurrent processes as the system development and maintenance is simplified by decomposition, and as the animation execution on multiprocessor machines is fast which is important for interactive applications.

The animation loop of the L-system interpreter is the heart of the system driving and controlling an animation and its associated concurrent processes. It is composed of the four nested animation loops shown in the following pseudo code. The numbers 0 to 6 mark parts of the code for further reference in the text.

In part 0 the program parses N L-system definition files and initializes the corresponding data structures, some modules and the global time T. The first loop is the time loop and drives the whole animation system. Each iteration corresponds to a frame and the global time T is incremented by the time step ΔT at the end of this loop (part 5). In code part 1 a script file "Command0" can be executed. The content of this file can be freely adapted to the needs of the actual animation. The parameter passing is explained in chapter 2.5. When image files are written to a file system, we can periodically test whether there is still disc space available and we can also update file names. When a sound track event file for later sound rendering is included, a new entry with the current time T is initialized. When articulated humanoid actors are used, data is read from the shared memory, which synchronizes the process managing the humanoids (see chapter 4.2.1).

The third loop treats each L-system. As L-systems very often represent plants, it is useful, when they can be repeatedly placed in a certain area of a virtual environment. Therefore we added a fourth loop that draws objects automatically by randomly changing their position, size and orientation. Thus, it is possible to create forests and flower fields. In part 3 of the code the position, size and orientation of each L-system defined object is determined, and when it is visible from the camera's point of view, it is drawn. The iterative step function S (see section 2.1.3) that iterates formal object is only applied once per time step and object. In all the other cases the symbolic object is only interpreted and displayed. Thus, for multiple actors and often repeated L-systems the iterative approach is much better than the recursive one. This fact will be further discussed in the corresponding section.

In code part 4 the synthetic vision of each synthetic actor is treated. When image by image recording is enabled, a selected window is recorded on the branched video device. When specified, the image can also be saved in rgb format on the disc.

At the end of the time loop in code part 5 another script file "Command1" can be executed. In general it is used to start a ray tracer rendering of the current image in film productions. In this code part the sound events have also to be updated. When the animation ends, eventually open files and the video device are closed in code part 6.

2.1.2 Recursive L-system

At the beginning of the development of the animation system we implemented the recursive definition of a timed L-system as described in [PRUS90]. We added some features useful for an animation system as parametric symbols with growth functions, conditional and stochastic productions and we made it sensible to external tropism forces. With this recursive definition a final symbolic string at a given age is always derived from the axiom. Thus, at high ages, the derivation becomes computationally expensive for complex objects. An advantage, however, is the fact that no entire symbolic string has to be memorized. The symbols of the right side of the production rules can be interpreted directly during the derivation. Additionally, an object at a given age can be directly created without all the intermediate time steps. Thus, very complex single images can be generated with low memory resources. Because of these inherent advantage we think it is worth to give a formal description of the recursive L-system. In the following work, however, we will refer only to the iterative version described in the next section as it is more efficient.

In definition 2.1.1 we can see some declarations used for the definitions of production rules and the derivation function D, which derives a symbolic string at a given age from an axiom by applying a set of production rules according to the axioms 1 to 4.

V:	an alphabet
C:	a set of constants
R^+:	the set of the real positive numbers
N :	the set of the integer positive numbers
Σ :	the set of parameters
$E(\Sigma)$:	the arithmetic expressions in Polish notation
$C(\Sigma)$:	the logical expressions in Polish notation
ω :	an axiom
P :	$P = \{p_{a,i} \mid a \in V, i \in N\}$ the set of productions
Π:	$p_{a,i} \rightarrow [0,1]$ a pseudo - stochastic distribution with $\sum_i \pi(p_{a,i}) = 1$
$(a, x_0, ..., x_6) \in V \times R^7$	a timed (x_0), parametric symbol

Definition 2.1.1: Formal definitions used for the L-system specification.

To simplify the implementation we associated to each symbol exactly three parameters and three growth functions. This simplification enables us to write an efficient array based L-system interpreter for fast interactive simulations. A production of the L-system is defined according to definition 2.1.2.

$P \in V \times R^7 \times C(\Sigma) \times \Pi \times (V \times E(\Sigma)^7)^*$

$p_{a,i} \in P$

$p_{a,i}: (a, \alpha_a, x_1, ..., x_6) \xrightarrow{Cond(\alpha_a, x_1 x_2, x_3, T) \text{ and } \Pi(p_{a,i})} (a_1, f_{10}, ..., f_{16}).....(a_n, f_{n0}, ..., f_{n6})$

α_a:	maximal age of symbol a
f_{j0}	initial age of symbol a_j
x_1, x_2, x_3	3 parameters
x_4, x_5, x_6	3 growth function values
$f_{j0} + t$	local age of the symbol
T	global time
$f_{j1}, ..., f_{j3}$	parameter expressions
$f_{j4}, ..., f_{j6}$	growth functions
f_{jk}	$f_{jk}(f_{j0}, x_{j1}, x_{j2}, x_{j3}, t, T)$
$p_{a,i}$	a conditional and pseudo - stochastic production

Definition 2.1.2: The definition of a production of the L-system.

Definition 2.1.3 contains the formal definition of the recursive derivation function, which generates a formal object string, based on the productions and an axiom string.

Derivation function *D:* $\left(V \times R^7\right)^* \times R \rightarrow \left(V \times R^7\right)^*$

Axiom 1: The development of each symbol is independent of each other in time

$$D\left(\left(a_1, x_{10}, \ldots, x_{16}\right) \ldots \left(a_n, x_{n0}, \ldots, x_{n6}\right), t\right) = D\left(\left(a_1, x_{10}, \ldots, x_{16}\right), t\right) \ldots D\left(\left(a_n, x_{n0,}, \ldots, x_{n6}\right), t\right)$$

Axiom 2: The growth of a symbol before terminal age

if $x_0 + t < \alpha_a$ then

$$D\left(\left(a, x_0, \ldots, x_6\right), t\right) = \left(a, x_0 + t, x_1, x_2, x_3, f_4, f_5, f_6\right)$$

Axiom 3: Application of stochastic production at terminal age

if $\left(x_0 + t \geq \alpha_a\right) \wedge \left(Cond\left(\alpha_a, x_1, x_2, x_3, T\right) = TRUE\right)$ then with a probability $\pi\left(\mathrm{p}_{\mathrm{a,i}}\right)$

$$D\left(\left(a, x_0, \ldots, x_6\right), t\right) =$$

$$D\left(\left(a_1, f_{10}, \ldots., f_{13}, x_{14}, x_{15}, x_{16}\right) \ldots \left(a_n, f_{n0}, \ldots., f_{n3}, x_{n4}, x_{n5}, x_{n6}\right), t - \left(\alpha_a - x_0\right)\right)$$

Axiom 4: Selection of random numbers for the productions

$$random\_number = rand\_table\left[\left(x + y[x]\right) \bmod z\right]$$

rand_table: table of size z with uniform random values between 0 and 1

z : size of rand_table

x : recursion depth of function D

y[x] : position of the production in the derivation tree of D at the x depth.

Definition 2.1.3: The definition of the recursive derivation function D.

Axiom 1 expresses the independence of each symbol in time. Axiom 2 controls the growth of a symbol before terminal age, and axiom 3 defines the application of stochastic productions at terminal age. Note that the parameter expressions of the symbols of the right side of the production rules are only evaluated once at the application of a production. The growth functions are without importance for the formal derivation, they are only evaluated during the interpretation of the symbols.

The nature of the derivation function is deterministic. To enable stochastic productions we added axiom 4 that shows how to associate random numbers to the productions. During the recursive derivation with the function D we can trace in a global variable the recursion depth x and the position of the symbol to be replaced in the derivation tree of D at depth x. This location of a production in the derivation tree identifies it and can be used to reference a random number in a table of random numbers. To maintain a temporal continuity of a symbolic object, it is necessary that at its derivation each stochastic production identified by its location in the derivation tree has the same random value associated for each age. Figure 2.1.2 shows an example.

The symbol a is non terminal and has no geometric signification. The symbols r and s represent rectangles or circles respectively. In this example all symbols have only the parameter t that corresponds to their local age.

axiom: **a**(0) **a**(0) **a**(0)

p1: **a**(1) --> **c**(0) **a**(0): probability = 0.7

p2: **a**(1) --> **r**(0) **a**(0): probability = 0.3
(c = circle, r = rectangle)

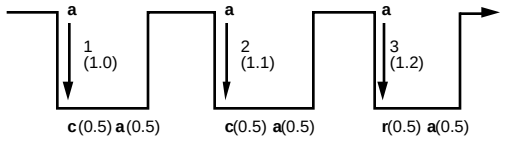
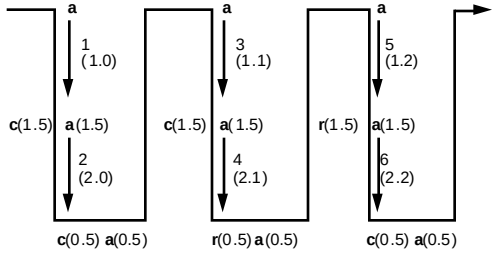
age t	recursion level of D	symbolic string	interpreted image
0.5	0	**a**(0.5) **a**(0.5) **a**(0.5)	
1.5	0 1		❍ ❍ ❑
2.5	0 1 2		❍ ❍ ❍ ❑ ❑ ❍

Figure 2.1.2: A recursive derivation example of a L-system

The example shows the development at three given ages. At time 0.5 still no production is applied and nothing is drawn. Only the local age of the symbols **a** has increased to 0.5. At time 1.5 production p1 has been applied for the 2 first symbols **a** and production p2 for the third symbol of the axiom. The arrows indicate the application of productions and they are numbered (x) according to their sequential execution through the derivation function D. The numbers (y.z) show the horizontal order z of the productions at a given recursion depth y. All the symbols beneath the curved arrow tracing the derivation function make part of the final symbolic object and are interpreted resulting in an image with two circles and one rectangle.

At time t=2.5 six productions have been applied in the order designed by the numbered arrows. From this last example it is evident that a sequential application of a random number generator for the production probabilities would not produce the same numbers for production 5, for example at time t=2.5, which correspond to production 3 at time t=1.5. Only the position (x.y) in the derivation tree identifies for each time a production and can be used to reference consistently random numbers for stochastic productions.

Growing Tree Structure

We illustrate the derivation function **D** by deriving the formal object of a continuously growing simple tree structure at several ages of the object. We consider the alphabet V = {**p**, **q**, **+**, **-**, **[**, **]** } and print all characters in bold. **p** and **q** are timed and parametric symbols with growth functions. We will use the following notation:

(**p**, a, x, f(x, t))

To establish the link with the formal description of section 2 we have the following correspondences:

p	the parametric symbol
a	the parameter x_0 that is the actual or initial age of the symbol.
x	the parameter x_1
f(x, t)	the growth function f1 with the parameter x and the time t as arguments.

The symbols **p** and **q** represent line segments with a length of the actual value of the growth function. The line segment is drawn from the actual position of the turtle (see section 2.1.5) in direction of its heading vector H. After having drawn the line segment, the turtle is placed at its end without changing its direction. The symbols **+** and **-** of the alphabet V represent rotations of the turtle around its "up" vector U by an angle of +-90 degrees. The brackets **[** and **]**, which are introduced to delimit branches, correspond to push and pop operations of the turtle state (position and orientation). The brackets enable the construction of tree structures.

The tree structure is defined by an axiom and one production for the symbol **p**.

Axiom: (**p**, a=0, x=1, f = 6xt)

Production: (**p**, α = 1) ----------> (Condition = TRUE, probability = 1)
(**q**, a=0, x = x, f=6x+t) **[+** (**p**, a=0, x = x/2, f=6xt) **] [-** (**p**, a=0, x = x/2, f=6xt) **]**

The axiom corresponds to a line segment (symbol p) with initial age a=0, a parameter x=1 and a linear growth function f = 6xt = 6t (as x =1). Thus, the segment will linearly grow until it reaches its maximal age alpha = 1 given in the left side of the production. The production is applied with a probability of 1.0 without any precondition. Figure 2.1.3 illustrates some applications of the production.

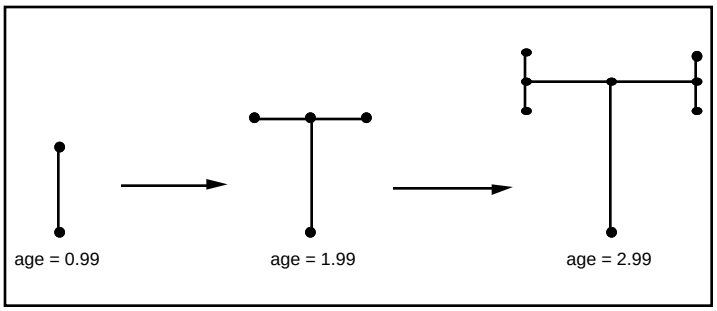


Figure 2.1.3: Continuous growth of a simple tree.

The line segment **p** at maximal age of 1 has to be replaced by a segment **q** of the same length at its initial age of a=0. The parameter x of the symbol **p** (which is replaced) is passed without any change and the growth function f_q=6x+t satisfies this continuity condition as $f_p(t=1, x=1) = f_q(t=0, x=1) = 6$. The term t is responsible for a further linear but smaller growth rate of the segment q. Thus, each branch of the segment continues to grow. The two new branches p should be a factor 2 shorter then the preceding one. Therefore, the parameter x is passed to the new symbols by dividing it by 2 (x := x/2). As the branches start their growth from zero, the same growth function has to be used as for the symbol **p** in the axiom. Thus, a continuous and consistent further replacement is guaranteed during the following iterations.

In the next paragraph we detail some derivation steps of the formal object. We show how the derivation function **D** works and when the corresponding axioms of the derivation function D are applied. To avoid confusion the reader is asked to distinguish between the enumerated mathematical axioms of the derivation function **D** (see definition 2.1.3) and the symbolic axiom of an L-system defining the tree.

Object age t = 0.5:
D(Axiom, 0.5)= D((**p**, 0, 1, 6xt), 0.5) =
/* Axiom 2 as a + t = 0.5 < α=1, the growth function is evaluated */
(**p**, a + t, x, 6xt) =
(**p**, 0+0.5, 1, 6*1*0.5) =
(**p**, 0.5, 1, 3)

Object age t = 0.9
D(Axiom, 0.9) = ... = (**p**, 0.9, 1, 5.4)

Object age t = 1.5
D(Axiom, 1.5) = D((**p**, 0, 1, 6xt), 1.5) =
/* Axiom 3 as a+1.5 = 0+1.5 = 1.5 < α = 1,
the parameter x of the preceding symbol p is passed to the symbols of the
production by evaluating the corresponding expressions */
D((**q**, 0, x, 6x+t) **[+** (**p**, 0, x/2, 6xt) **] [-** (**p**, 0, x/2, 6xt) **]**, 1.5 - (α-a)) =
D((**q**, 0, 1, 6x+t) **[+** (**p**, 0, 0.5, 6xt) **] [-** (**p**, 0, 0.5, 6xt) **]**, 0.5) =
/* Axiom 1*/
D((**q**, 0, 1, 6x+t), 0.5) **[+** D((**p**, 0, 0.5, 6xt), 0.5) **] [-** D((**p**, 0, 0.5, 6xt), 0.5)**]** =
/*Axiom 2 as a + t = 0+0.5 < α=1,
the growth functions are evaluated by using the evaluated parameter x*/
(**q**, 0.5, 1, 6.5) **[+** (**p**, 0.5, 0.5, 1.5) **] [-** (**p**, 0.5, 0.5, 1.5) **]**

2.1.3 The Iterative L-system

We have seen that the most important advantage of the recursive definition of the L-system is the fact that it is not necessary to memorize explicitly the actual formal or already interpreted object. In this case, however, at each frame the whole object has to be derived from the axiom, which can be very time consuming for complex objects. Additionally, the symbol identification needed for stochastic productions is not trivial and the symbols can not be efficiently deleted. The extension with context dependent productions is nearly impossible or very complicated.

Therefore, we developed an iterative step function, which takes as input the time step Δt and the actual symbolic object at the global time T. It derives the symbolic object at time T + Δt. In this version we need memory space for two symbolic objects, the current one and the new one. Note that the symbolic object is only a table of evaluated symbols shown in figure 2.1.4, which can procedurally represent very complex geometrical objects. Nevertheless, the memory space needed to represent the whole environment is still much smaller than for conventional databases where each geometrical object is represented by points, polygons, lines and materials.

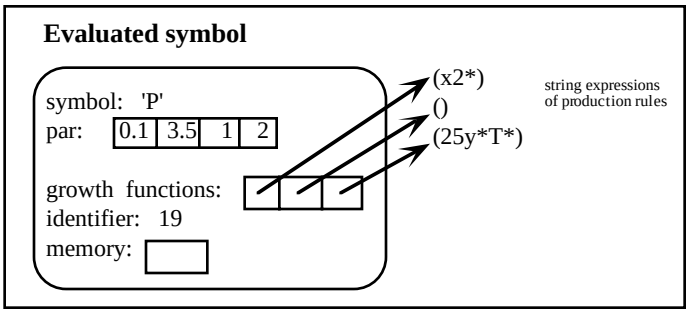


Figure 2.1.4: Evaluated parametric symbol.

With this iterative approach the evolution of a formal object by one time step is now faster. The symbol identification and debugging are also much more transparent, and the elimination of symbols and context dependency can be efficiently implemented. In the iterative L-system the formal derivation of an object and its interpretation are now separated. This is a considerable advantage, if the same object has to be drawn several times as the formal derivation has to be done only once. We profit from this situation, when the whole scene is drawn for several actors from their point of view, or when the same L-structure (a flower, for example) is drawn several times in an animation sequence. Referring to the definitions given

for the recursive derivation function D, the step function S can be characterized by the following axioms:

Step function *S*: $\left(V \times R^7\right)^* \times R \rightarrow \left(V \times R^7\right)^*$

Axiom 1: The development of each symbol is independent of each other in time

$$S\left(\left(a_1, x_{10}, \ldots, x_{16}\right) \ldots \left(a_n, x_{n0}, \ldots, x_{n6}\right), \Delta T\right) = S\left(\left(a_1, x_{10}, \ldots, x_{16}\right), \Delta T\right) \ldots S\left(\left(a_n, x_{n0,}, \ldots, x_{n6}\right), \Delta T\right)$$

Axiom 2: The growth of a symbol before terminal age

if $(x_0 + \Delta T < \alpha_a) \vee \left(Cond\left(\alpha_a, x_1, x_2, x_3, T, t\right) = FALSE\right) \vee$ (no context matching) then

$$S\left(\left(a, x_0, \ldots, x_6\right), \Delta T\right) = \left(a, x_0 + \Delta T, x_1, x_2, x_3, f_4, f_5, f_6\right)$$

Axiom 3: Application of stochastic production at terminal age

if $(x_0 + \Delta t \geq) \alpha_a \wedge \left(Cond\left(\alpha_a, x_1, x_2, x_3, T, t\right) = TRUE\right) \wedge$ (context matching)

then with a probability $\pi\left(\mathrm{p}_{a,i}\right)$

$$S\left(\left(a, x_0, \ldots, x_6\right), \Delta T\right) = \left(a_1, f_{10}, \ldots., f_{13}, x_{14}, x_{15}, x_{16}\right) \ldots \left(a_n, f_{n0}, \ldots., f_{n3}, x_{n4}, x_{n5}, x_{n6}\right)$$

Definition 2.1.4: The definition of the iterative step function of L-systems.

In the axiom 2 the age of the symbol **a** is increased by the time step Δt. The parameters x_1, .., x_3 are not changed and the growth functions f4, .. ,f6 are evaluated only during the interpretation of the symbol.

Note that in Axiom 3 there is no recursive call to the step function S, and the local age $t=x_0$ of the symbol **a** can be bigger than its maximum age because of the presence of a condition that must be true to replace the symbol **a**. To avoid inconsistent development of the object the time step Δt must be smaller than the smallest maximum age of the symbols to be replaced. The evaluation in axiom 2 is done from left to right. When a new symbol coming from a production rule is added to the symbolic object, its unique identifier is determined by incrementing a global counter. This symbol identifier remains unchanged during the whole life time of the symbol.

A (1,1) context dependency of the production rules is supported. A production is context dependent if its application depends on the left or right neighbor symbols of the symbol to be replaced. This effect is useful in simulating interactions between plant parts, due for example to the flow of nutrients or hormones. In a (k, l) - system, the left context is a word of length k and the right context is a word of length l. Note that according to [PRUS90] the left context is looked for only on the path to the root of the branching structure whereas the right context can be a sub-tree built by the symbols **[** and **]**. In our L-system interpreter the following symbols are ignored in the context matching routine, as they don't represent geometric shapes:

+, -, /, $, ^, &, m, n, []

In code 2.1.2 we illustrate the left context matching algorithm realized by a simple loop that places the index i, which points at the beginning to the symbol to be replaced, to the left neighbor symbol by skipping ignored symbols and sub-trees if present. Then, the algorithm returns the result of the comparison of the left context with the left neighbor symbol.

Algorithm 1: Left context matching

```
i:              start index of the symbol table pointing on the symbol to be replaced.
symbol[]:       the formal object or the symbol table
level = 0:      used for skipping branches

do {
    i = i - 1;
    if (symbol[i] == ']')
        level = level + 1;
    else if (symbol[i] == '[')
        level = level - 1;
    else if ((not ignored(symbol[i]) and (level ==0))
        exit do
}

return (left_context == symbol[i])        : returns TRUE if the right context matches
```

Code 2.1.2: Left context matching algorithm.

In code 2.1.3 the recursive right context matching algorithm reflects the sub-tree nature of the right context. This algorithm tests the current branch for the right context. If there are sub-trees in its neighborhood, the function is recursively called for this sub-tree. The 6 possible cases are illustrated with a symbol string where **a** is the symbol to be replaced and **b** is the right context, and (+, -, $) are the symbols that are ignored. The bold symbol is referenced by the index i before the operation and the cursive symbol is referenced by the index i after the operation.

Algorithm 2: Right context matching

```
symbol[]:   global symbol table or formal object
i:          start index of the symbol table pointing on the symbol to be replaced.

int right_context_ok(i) {

    i: points to the first symbol after the symbol to be replaced
    skip symbols to be ignored => i                 // a + $ b
    if (right_context == symbol[i])
        return TRUE;
    else if (symbol[i] == '[' {                     // there is a sub-tree a + $ [ ... ]
        if (right_context_ok(i+1))
            return TRUE;                            // a + $ [ + - b... ]
        else {
            skip to end of branch by ignoring all sub-trees in the branch =>
                i points now to the end of the branch       // a + $ [ + - c..]+ -
            return right_context_ok(i+1)            // a + $ [ + - c..]+ -
        }
    } else if (symbol[i] == ']') {                  // a + $]
            return right_context_ok(i+1)
    }
    else
        return FALSE                                // a + $ c
}
```

Code 2.1.3: Right context matching algorithm

2.1.4 Syntax and Semantic

According to the formal definition of the L-system described in the previous section a timed parametric symbol of the axiom or of the right side of a production rule is given formally by the following BNF:

ParametricSymbol ::= Symbol (**Expr**) (**Expr**) (**Expr**) (**Expr**) (**Expr**) (**Expr**) (**Expr**)
Symbol ::=[|] | { | } | / | ^ | & | $ | + | - | a .. z | A .. Z | 0..9 |
Expr ::= **Expr Expr Bin_op** | **Expr Un_op** | **Const** | **Function** | **Parameters** | empty

Parameters ::= t | T | x | y | z | X | Y | Z | u | v | w | r
Const ::= **Number** | **Number**.**Number**
Number ::= **Number Digit**
Digit ::= 0..9
Bin_op ::= + | - | * | - | ^ | % | ~
Un_op ::= | | _ | $ | & | : | ”

Definition 2.1.5: BNF of a parametric symbol.

In all subsequent BNF definitions all the keywords and the following types are terminal:

int: integer type
float: floating point type
Word: a string without blank, carriage return, or tab separators.

Non terminal productions are printed in bold. The symbol ‘ |’ expresses alternatives and the Kleen star * a list of items which can be empty. The upper script $^+$ indicates an non empty item list.

A complete definition of a L-system and its semantic is given in the appendices A, B and C. The numerical expressions, **Expr,** are in Polish notation with post fixed operators allowing a fast stack based evaluation. Moreover, to increase interpretation speed we defined an easily extensible set of special precompiled functions usable in the numeric expressions of growth functions of symbols and conditions of production rules. A complete description of these functions is given in appendix B, table B.5.

A symbolic L-system object at a given age T is a word (table) of evaluated parametric symbols. Observe the difference of a parametric symbol of a production rule and the symbol of a L-structure shown in the next table.

	symbol	max. age t local age t	parameter x	parameter y	parameter z	growth function 1	growth function 2	growth function 3
axiom or right side of a production rule	**s**	(expr_t)	(expr_x)	expr_y)	(expr_z)	(expr_f1)	(expr_f2)	(expr_f3)
evaluated symbol of a L-structure	**s**	par0	par1	par2	par3	par4	par5	par6

Table 2.1.1: Parametric and evaluated symbol

The values par¡ of an evaluated symbol are the results of the corresponding expressions that are evaluated according to the following rules and parameter semantic.

Age expression:
par0 = expr_t : local age of the symbol

Parameter expressions:
par1 = x = expr_x: the parameter x of the symbol s

par2 = y = expr_y: the parameter y of the symbol s
par3 = z = expr_z: the parameter z of the symbol s

Growth function expressions:
par4 = expr_f1: the first growth function of the symbol s
par5 = expr_f2: the second growth function of the symbol s
par6 = expr_f3: the third growth function of the symbol s

Note that the parameters t, x, y and z have different meanings in the above expressions. In the age and parameter expressions the parameters t, x, y and z correspond to the current values of the **symbol to be replaced** (!! not to the symbol **s**). These expressions are only evaluated once at the moment of the application of the production. Then, during the whole life time of the symbol **s** the parameters x, y and z won't change any more whereas the local time t is incremented at each iteration of the animation loop by Δt. The growth functions, however, are evaluated at each time increment of the animation loop. In the growth function expressions the parameters t, x, y, and z refer now to the symbol **s**. The parameter T refers always to the current global time T determined by the animation loop.

In some special force field definition symbols the semantic of some parameters can change. Details are given in the description of the semantic of the corresponding symbols (C, v, e) in appendix C. The next example illustrates the above rules:

The following L-system is defined by an axiom composed of the symbol **a** and one production rule p1.

Axiom:	**a** ()	(2) (1) (3)		(y) (z) (t)			
p1:	**a** (1)	-->	**b** ()	(x2*) (x3+) (xyz**)	(2)	(2y+)	(xz+)

We have the following correspondence. In the axiom for symbol **a**
expr_t = empty,
expr_x = 2
expr_y = 1
expr_z = 3
expr_f1 = y
expr_f2 = z
expr_f3 = t

In the production for symbol **b**
expr_t = empty
expr_x = x2*
expr_y = x3+
expr_z = xyz**
expr_f1 = 2
expr_f2 = 2y+
expr_f3 = xz+

The L-structure at global time T=0.5 is the evaluated symbol
a (0.5) (2) (1) (3) (1) (3) (0.5)
The L-structure at global time T = 1.5 is the symbol
b (0.5) (2*2 = 4) (2+3 = 5) (2*1*3 = 6) (2) (2+5 = 7) (4+6 = 10)
Now we have for symbol **b**:
par0 = t = 0.5
par1 = x = 4
par2 = y = 5
par3 = z = 6

par4 = 2
par5 = 7
par6 = 10.

According to the formal definition given in section 2.1.3 our L-system is conditional. The conditions are logical expressions. Their syntax is given by the following BNF:

LogicalExpression ::=	**LogicalExpression LogicalExpression Lbin_op** |
	LogicalExpression Lun_op |
	() |
	[(**Expr**)**Pred**]
Pred ::= < | p | > | g | = | !	
Lbin_op ::= & | ' |'	(and | or)
Lun_op ::= ! (not)	

Definition 2.1.6: BNF of a logical expression.

Observe that the symbol ' |' is used as logical "or" in the BNF definitions. When it appears as terminal symbol of the BNF, we put it between apostrophes to avoid any confusion. The empty logic expression () always returns 1 (= TRUE). It is used for unconditional productions.

The predicates always compare the result with the numerical expression Expr with 0, which is implicit. The condition 0<x<5, for example has to be written as [(x)>][(x5-)<]&.

2.1.5 The Turtle Interpretation

During the interpretation phase of a derived formal symbol string, some symbols of the alphabet are interpreted by means of a 3D turtle. Its current orientation is given by a local orthonormal coordinate system with the axis **H**(heading), **L**(left) and **U**(up) as shown in figure 2.1.5. Some symbols control the position and the orientation of the turtle's coordinate system, others represent geometrical primitives, drawn in the turtle's coordinate system, and others perform special operations.

The turtle rotations R_i are defined by the following matrices for line vector rotations:

$$R_U(\beta) = \begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (2.1.1)$$

$$R_L(\beta) = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \qquad (2.1.2)$$

$$R_H(\beta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & \sin(\beta) \\ 0 & -\sin(\beta) & \cos(\beta) \end{bmatrix} \qquad (2.1.3)$$

The new turtle orientation is given by

$$\begin{bmatrix} \vec{H}' \\ \vec{L}' \\ \vec{U}' \end{bmatrix} = \begin{bmatrix} H_x' & H_y' & H_z' \\ L_x' & L_y' & L_z' \\ U_x' & U_y' & U_z' \end{bmatrix} = R_i \bullet \begin{bmatrix} H_x & H_y & H_z \\ L_x & L_y & L_z \\ U_x & U_y & U_z \end{bmatrix} = R_i \bullet \begin{bmatrix} \vec{H} \\ \vec{L} \\ \vec{U} \end{bmatrix} \qquad (2.1.4)$$

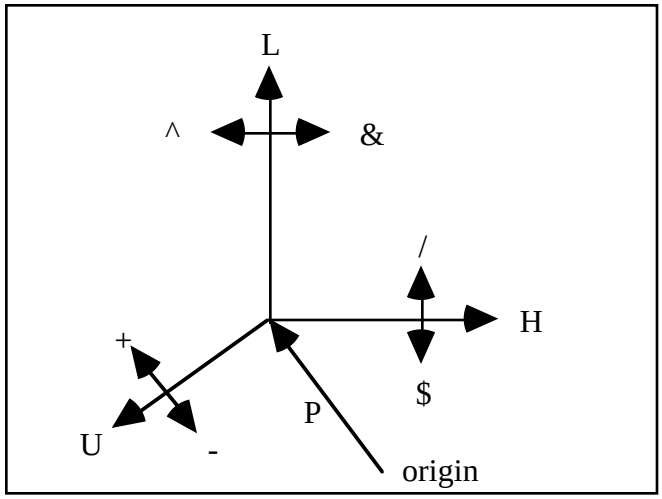


Figure 2.1.5: The 3D turtle as a local orthonormal coordinate system

The following symbols allow to manipulate directly the turtle.

+: turn left
-: turn right
&: pitch down
^: pitch up
/: roll right
$: roll left
f: move forward
[: push the state of the turtle on the stack
]: pop the state of the turtle from the stack

In this and the following sections we only give an informal overview of the symbols and their semantic of the alphabet of the L-system. A complete description of the symbol semantic can be found in appendix C.

The symbol "**$**" has apart from the original “roll left” signification some more meanings determined by its parameter value par2. This kind of parametric overloading of a symbol is used to group certain operations of a similar type in one symbol. We will encounter this symbol overloading for several other symbols described in the next sections. Note, if a symbol uses parameter overloading, then the corresponding parameters should not be used any more in the normal sense for parametric production rules. The semantic change of the symbol has to be taken into account.

To move the turtle without drawing something we can use the symbol **f.** We can choose between relative or absolute movements.

The turtle state is given by its orientation and its position. The symbols **[** and **]** allow to push and pop the turtle's state to and from a stack. They extend the L-system to a bracketed or tree-L-system allowing the definition of branching structures and plants. The semantic of the symbols depends on the parameter values.

Sometimes it is useful to place the turtle's position or its entire state in global variables from which it can be referenced later. They are especially useful for the generation of surfaces.

2.1.6 Camera, Light and Material Control

In an animation system or a film production system it is essential that the camera of the user display can be efficiently controlled. In a L-system based animation system it makes sense to plan a symbol for the camera control. We use the symbol **l** to manipulate the position and look at point of the user camera. By using production rules and growth functions several camera positions and movements can be scripted by the user including key framed camera motion. For interactive games (see section 5.8) or interactive exploring we use a space ball interface and map the spaceball position and orientation via the symbol **l** to the camera. There are also possibilities to display the environment from the point of view of particles or actors and to track any object in the scene by the camera. Further, a light source can be placed and animated.

The L-system interpreter allows the activation of some predefined materials with the symbol **m**. Thus, all geometrical primitives after a symbol **m** in the formal object are colored according to this active material. By using growth functions the diffuse component of an active material can be animated

The symbol **r** permits to activate a rgb color for simple segments in GL rendering.

2.2 Geometric Modeling

In the virtual environment we need a geometric model for modeling the visible objects. In our animation system the geometric modeling is mainly done by symbols of the L-system.

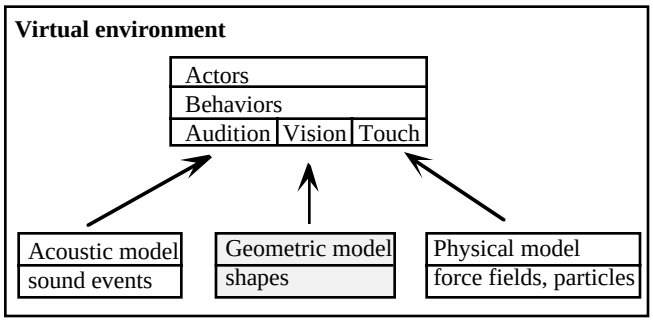


Figure 2.2: The geometric model in the virtual environment

A L-system must provide symbols for geometric shapes which compose a L-system defined object. During the interpretation step these symbols are drawn. Most of the following symbols in this section represent basic geometric shapes drawn in the coordinate system of the turtle and scaled with growth functions. Note that the position of the turtle changes, after having drawn the shape. Typical L-structures like plants and trees are composed of thousands of basic shapes.

2.2.1 Symbols for Basic Shapes

Basic shapes are not necessarily simple shapes. In GL rendering, for example, a sphere can be composed of hundreds of triangles. The following symbols represent geometric shapes as spheres, cylinders or cubes or allow to modify their appearance. The symbol **n** permits to determine the resolution of spheres, cylinders and trunks. In GL rendering such shapes are approximated by polygons. High resolution means a lot of polygons. For the ray tracer,

however, which supports directly these figures, no triangulation is done. The following symbols represent geometric shapes.

R, S, T: line segments
c, d: cubes
O, N: cylinders with ending half spheres
o: scaled cylinders
P, Q: trunks with ending half spheres
a: sphere
s: ellipsoid
p, q: pyramid

According to [PRUS90] we integrated the symbols "{.}" for surface creation by production rules. To generate nested surface definitions we need a stack of polygons. The polygon stack operations are referenced by these symbols.

{ : Push the current polygon on the stack and create a new current polygon.

. : Add the actual position of the turtle as a new vertex to the current polygon.

} : Draw the current polygon and pop a polygon, which becomes the current polygon, from the stack.

Some other special geometric shapes as planes or a tennis court are symbolized by the special symbol **D**, which is treated in the next section.

2.2.2 Special Symbols

As the number of symbols of the symbol alphabet is limited, we used the symbol **D** to represent several special functions depending on its parameters. Most of them are used for the interaction control with the Virtual Life Network, for special geometric shapes, for ray traced film generation, and for some control functions. A detailed description can be found in appendix C.

A versatile animation system makes use of the resources of the computer system and other processes. Therefore, we introduced the symbol **E** that allows to control the execution of c-shell or shell command files through production rules. This symbol considerably enhances the flexibility of the animation system as it enables access to external resources. We pass a standard useful set of parameters to each command file to establish a link with the current state of an animation.

2.2.3 Imported and Exported Data

To increase the flexibility and conviviality of the L-system interpreter some possibilities to exchange data with other applications are included. In an animation system useful data are, for example, geometric surfaces, general animation data, and special actor trajectories.

There exist a lot of 3D shape editors for interactive surface generation. The use of such hand crafted surfaces increases the potential and versatility of an animation system On the other hand, L-systems can create nice plant-like static objects useful in many other applications. Therefore we implemented features for the importation and exportation of surfaces of an interactive surface manager (SM) [SM94].

The importation of general animation data, affecting L-system defined objects, opens the door to interaction with other ad hoc applications written for special purposes. Therefore, we provided for the import of data through channels that are one dimensional parametric data tables stored in external ASCII files. Such files can be created by any external application or

by hand, and they can be read in by the L-system interpreter. Through a special function they can be used in numeric string expressions. For actor animation we introduced a special track file concept that allows to import and to export actor trajectories. In section 6.3.3 we present all these features of the animation system in more detail.

2.3 Physical Modeling

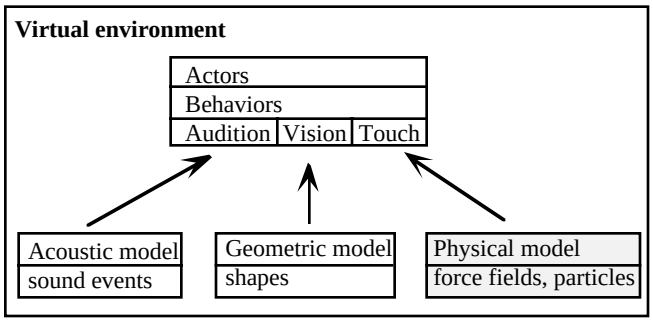


Figure 2.3: The physical model in the virtual environment

To add some realism to our virtual environment we extended it with a physical modeling facility which is based on particle dynamics in force fields based on Newton's equation of movement. Particle dynamics is relatively easy to implement and can be used for physical simulation and for behavioral animation as discussed in [NR93] and [REY87]. Moreover, physics-based particle systems can be extended to powerful modeling tools as shown in [CLH96] or [NCL96] where the authors discuss physical models of loose soils dynamically marked by moving objects, and physical model of a whole mechanical clockwork, where not only motion but also sound is generated by a particle-based modeling technique.

In a force field animation system parts of the 3D world have not only to be modeled geometrically but also by bounding force fields. Some objects will exert repulsion forces on others in case of collisions. Other objects are attractive. There can exist objects being attractive at far distances and repulsive at short distances. Force fields like gravity or wind can influence trajectories of moving particles. For example, in a tennis game simulation we present in later chapters with autonomous actors, the particle dynamics serves to animate the ball. With this approach the ball movement in gravitation and wind fields, including collisions of the ball with the ground, the net and the racket can be treated. In the next section we describe the system of differential equation and its solution technique based on the fourth-order Runge Kutta method. It is a standard method. We present it here for completeness reasons, and for consistency with later used notation. Finally, we show how the particle system is integrated in the L-system interpreter, and how the particles are generated and partially controlled by symbols of the axiom or production rules.

2.3.1 Particle System

In the particle system each particle is treated as a point object with mass m, which can carry a vector force field influencing other objects. Its movement is described by the differential equation (2.3.1)

Definitions:

i: index of particle i

n: number of particles, i = 1, ..., n

m_i: mass of particle i

$\vec{x}_i$: position of particle i

$\dot{\vec{x}}_i$: velocity of particle i

$\ddot{\vec{x}}_i$: acceleration of particle i

t: time

$r_{ik} = |\vec{x}_i - \vec{x}_k|$: distance between particle i and k

$\ddot{\vec{x}} \cdot m = \vec{F}$: Newton' s equation of movement

$$\ddot{\vec{x}}_i = \frac{1}{m_i}\left[\vec{g}_i\left(\vec{x}_i, \dot{\vec{x}}_i, t\right) + \sum_{\substack{k=1 \\ k\neq i}}^{n} \vec{f}_k\left(\vec{x}_k, \vec{x}_i, \dot{\vec{x}}_k, r_{ik}, t, m_k\right)\right]$$

$\vec{g}_i$: individual part of differential equation of particle i, determining its movement in the global force field (2.3.1)

$\sum_{\substack{k=1 \\ k\neq i}}^{n} \vec{f}_k$: global force field which is the sum of the contributions of all other particle

To solve this system of differential equations of order two with the method of Runge Kutta of order four we have to convert it to a system of differential equations of order one according to the following equations (2.3.2).

$$\vec{y}_i = \begin{pmatrix} \vec{x}_i \\ \dot{\vec{x}}_i \end{pmatrix} \quad \text{: position and velocity vector (6 coordinates)}$$

$$\dot{\vec{y}}_i = \begin{pmatrix} \dot{\vec{x}}_i \\ \ddot{\vec{x}}_i \end{pmatrix} = \vec{F}_i(t, \vec{y}) \quad \text{: the time derivation} \qquad (2.3.2)$$

$$\vec{Y} = \begin{pmatrix} \vec{y}_1 \\ : \\ \vec{y}_n \end{pmatrix} \quad \text{: a vector of n * 6 coordinates}$$

$$\vec{F}_i(t, \vec{y}) = \begin{pmatrix} \dot{\vec{x}}_i \\ \frac{1}{m_i}\left[\vec{g}_i\left(\vec{x}_i, \dot{\vec{x}}_i, t\right) + \sum_{\substack{k=1 \\ k\neq i}}^{n} \vec{f}_k\left(\vec{x}_k, \vec{x}_i, \dot{\vec{x}}_k, r_{ik}, t, m_k\right)\right] \end{pmatrix}$$

$$\dot{\vec{Y}} = \begin{pmatrix} \dot{\vec{y}}_1 \\ : \\ \dot{\vec{y}}_n \end{pmatrix} = \begin{pmatrix} \vec{F}_1\left(t, \vec{Y}\right) \\ : \\ \vec{F}_n\left(t, \vec{Y}\right) \end{pmatrix} = \vec{F}\left(t, \vec{Y}\right) : \text{ the linear system of differential equations of order 1.}$$

Now the linear system (2.3.2) can be solved by the fourth-order Runge Kutta method given in appendix H.

In the framework of the L-system theory the particles can be generated and initialized by the symbol **e**. A particle has to be created only once. Therefore, this symbol is ephemeral and it is never transferred in a derived symbolic object as it is exceptionally interpreted in the step function S which iterates the symbolic object at each time step. The step function S creates the particle and initializes it according to the parameters and growth function expressions.

Each time the step function S encounters the symbol **e** with the above mentioned parameters, it acts on a global interaction table (see code C.4.1) containing the trace of all generated particles. The maximum size of this table can be declared in the "Options" section of the L-system definition file. A particle is identified by the identifier of the symbol **C** that follows its generation symbol **e** in a symbolic string. The vector force field pointer and the particle's individual part of the differential equation are initialized with the growth function expressions of the corresponding symbols **e** (see also appendices A, B and C for more information).

The position of the turtle can be mapped to the corresponding particle with the symbol **C** following the symbols **e** of a particle . At the beginning when the local age of the symbol **C** is smaller than a time step, the actual position of the turtle is taken to initialize the particle's position. The velocity corresponds to (par4, par5, par6) of the symbol. The mass of the particle is initialized with the value of par2.

After the initialization the animation loop iterates the numerical solution of the particles differential equation by considering all contributions of the other force fields of the same type. Then, the symbol **C** maps the particle's resulting position to the turtle's position. The orientation of the turtle can be adjusted by using the symbol **$**. The exact meaning of symbol **C** is given in appendix C.

2.3.2 Tropism

Tropism forces as gravity, light, or wind act on the articulations of branching structures [PRUS90] and modify their shape. The bending of branches is simulated by a rotation of the turtle in direction of the tropism forces as indicated in figure 2.3.1.

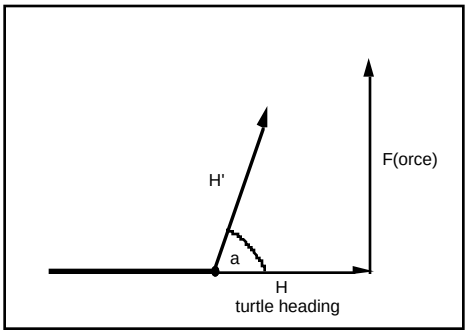


Figure 2.3.1: Tropism force rotates the turtle

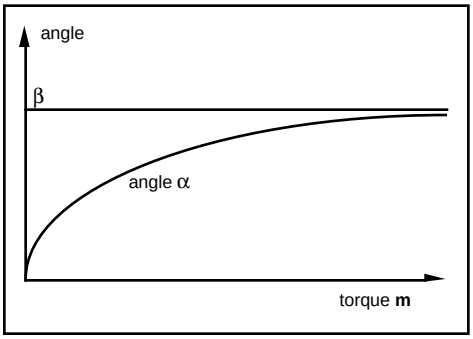


Figure 2.3.2: Rotation angle-torque dependence

In the same way we can apply forces of particles to branching structures by simply adding their contributions to the tropism force. The following equations describe this interaction in detail.

$$
\begin{aligned}
&\vec{F} = \vec{f}_{tropisme} + \sum_i \vec{f}_i && \text{global force field}\\
&\vec{H} && \text{turtle heading vector}\\
&\vec{M} = \left(\vec{H} \times \vec{F}\right) / e && \text{torque vector (e = elasticity of articulation)}\\
&m = \left|\vec{M}\right| && \text{torque}\\
&\vec{A} = \left(\vec{H} \times \vec{F}\right) / \left|\left(\vec{H} \times \vec{F}\right)\right| && \text{rotation vector}\\
&\beta = sin^{-1}\left(\left|\vec{A}\right|\right) && \text{angle between turtle heading } \vec{H} \text{ et } \vec{F}\\
&\alpha = \beta\left(1 - \frac{1}{1+m}\right) && \text{resulting rotation angle}
\end{aligned}
\qquad (2.3.4)
$$

The effective rotation angle is proportional to the torque m, produced by the global force by acting on the turtle's heading vector, but it never exceeds the angle between the turtle heading and the force vector at the turtle's position. Figure 2.3.2 illustrates the rotation angle - torque dependence.

The rotated turtle axes are now given by the following equations:

$$
\begin{aligned}
&\vec{H}' = \left(\vec{H}\vec{A}\right)\vec{A}\left(1 - \cos(\alpha)\right) + \vec{H}\cos(\alpha) + \left(\vec{A} \times \vec{H}\right)\sin(\alpha)\\
&\vec{L}' = \left(\vec{L}\vec{A}\right)\vec{A}\left(1 - \cos(\alpha)\right) + \vec{L}\cos(\alpha) + \left(\vec{A} \times \vec{L}\right)\sin(\alpha)\\
&\vec{U}' = \vec{H}' \times \vec{L}'
\end{aligned}
\qquad (2.3.5)
$$

Tropism forces are global forces and they can be defined via the symbol **v** according to the description in appendix C. The forces act only on the turtle when interpreting certain symbols which represent geometric shapes or particles. The forces can act on the turtle before or after the interpretation of certain symbols. In general, it is not so important whether it acts before or after the interpretation as an object is often composed of a lot of consecutive shapes. But when designing an object the user should know how it is handled. For more "historical" reasons we implemented the following timing. When tropism is enabled, it acts on the turtle after the symbols **c**, **d** (cubes), **p**, **q** (pyramids), **P**, **Q** (trunks) **s** (ellipsoid) and before the symbols **R**, **S** (segments), **O**, **N**, **o** (cylinders). Acting on the turtle **after** the symbol **c**, for instance means that first, the cube is drawn, and then, the turtle is rotated. Acting on the turtle **before** the symbol **R**, for instance, means that first, the turtle is rotated ,and then, the segment is drawn. In a L-system several tropism forces can be defined. Their effects are not summed up. Only the most recent encountered during the interpretation step is active. This allows to define several local tropism forces acting on a branching structure. In our tropism model there is only one instance of a tropism force. Its individual fields are set with the corresponding parameters and expressions of the symbol **v** when encountered somewhere during an interpretation step.

2.3.3 Some Force Field Examples

In this section we present some force field examples that are of general interest or that we used in some simulations. Note that force fields and particles can be used for physical modeling and behavioral animation. In chapter 7 we will show examples in both domains.

The movement of a particle in a global force field is determined by its individual part "g" (see equation 2.3.1) of its differential equation or by a predefined curve. The terms of "g" can depend on the particle's actual position and it's speed. Speed dependent terms can be used to model friction properties. The position variables allow it to make the particle's behavior position dependent. The particle can also carry a force field f (see equation 2.3.1) acting on other particles or as tropism force on branching structures.

General Forces

In the following, we show some elementary force functions we used to model the 3D world's force fields.

$$f_1(r) = a \bullet 3^{-b(r-c)^n} \tag{2.3.6}$$

$$f_2(r) = \begin{cases} 0, & \text{r} < \text{Min} \\ a, & \text{r} \geq \text{Min} \end{cases} \tag{2.3.7}$$

$$f_3(r) = \frac{a}{b \bullet r^n + e} \tag{2.3.8}$$

The exponential function (2.3.6) is well suited to model repulsive components of force fields. The use of the step function (2.3.7), however, should be avoided. It can lead to numerical instabilities. With the parameters a, b, c and n the behavior of the functions can be adapted to a specific animation. Equation (2.3.8) allows the modeling of attractive forces varying with the distance r from the force field center. The small constant e avoids the singularity of the function at r=0.

Another useful function is the cubic Hermite function given in equation 2.3.9.

$$x(t) = \begin{cases} x_{\min}, & t < 0 \\ x_{\min} + 3\left(x_{\max} - x_{\min}\right)\left(\frac{t}{T}\right)^2 - 2\left(x_{\max} - x_{\min}\right)\left(\frac{t}{T}\right)^3, & 0 < t < T \\ x_{\max}, & t > T \end{cases} \tag{2.3.9}$$

This function can be employed as growth function for parametric symbols of an L-system or as part of force field functions. Its growth is zero beyond the interval [0, T] and it gradually increases from x_{min} to x_{max} within the interval [0, T]. With simple parameter transformations illustrated in the figures 2.3.3 to 2.3.6, it can be adjusted to the needs of most applications.

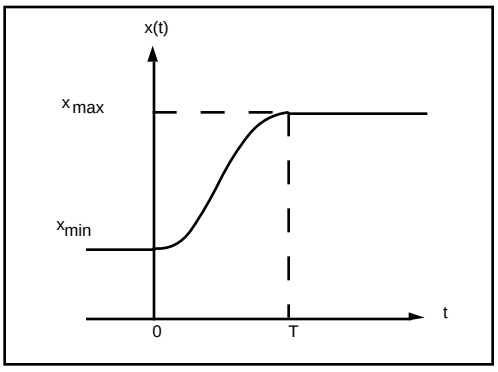


Figure 2.3.3: Growth function

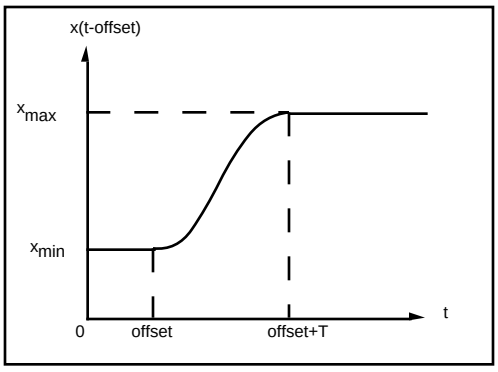


Figure 2.3.4: Transformed function

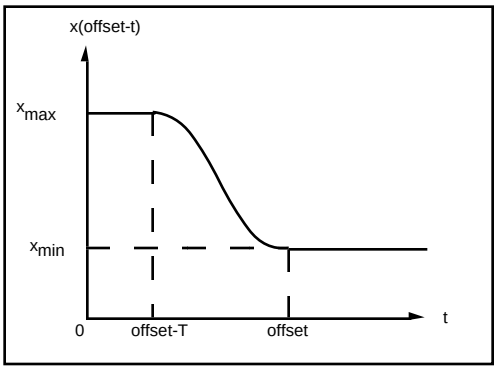


Figure 2.3.5: Transformed function

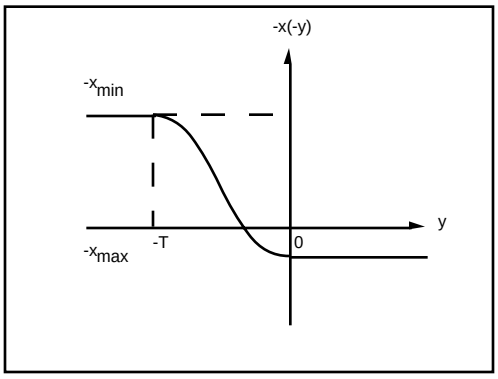


Figure 2.3.6: Simulates gravity and ground

The function of figure 2.3.6, for instance, was used to model the gravitation force and the ground rejection force. The parameter y represents the height of a particle. The ground is situated at $y = 0$. For $y > 0$ the particle feels the gravitation force $-x_{max}$. For $y < 0$, the particle is gradually rejected according to the function parameters.

2.4 The Virtual Acoustic Environment

After the geometric and physical model of the virtual environment we introduce now the acoustic model as illustrated in figure 2.4.

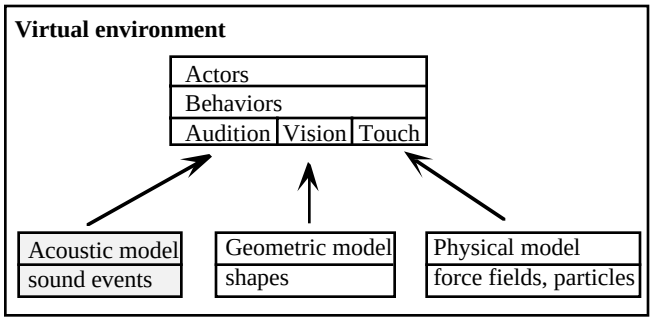


Figure 2.4: The acoustic model in the virtual environment

As already mentioned in the introduction in section 1.2.3, audition represents an important sense for behavioral animation. To enable synthetic audition we first have to model a virtual acoustic environment. Sound rendering increases considerably the quality of such an acoustic environment, but real time constraints lead us to a hybrid approach. For interactive real time applications we use simple 3D sound event frame work, where the actors can directly access the on-going sound events through a sound event handler. For film production, however, we can additionally create a 3D sound track file, where during the animation the sound events and microphone data are logged at each frame. In a second pass this log file is rendered into a synchronized AIFF (Audio Interchange File Format, see section 6.6) sound track file.

Any sound source (synthetic or real) can be converted to the AIFF format and in consequence processed by the sound renderer. The sound renderer takes into account the real time constraints. It is capable to render the sound portion of each time increment in "real time" for each microphone by taking into account the final propagation speed of sound and the moving sound sources and microphones. Thus, the frequency shift of approaching or moving away sound sources, known as the Doppler effect, can be heard.

The sound renderer we propose for this purpose is real time structured, i. e. it renders the sound frame by frame. Simple sound environments can be rendered in real time. The idea of the real time structured sound renderer is to use it in future on parallel processor machines as the Onyx directly for headphone rendering in VR applications. In the next section we discuss in more details this topic before presenting in more detail the actual sound renderer and the 3D sound environment frame work.

2.4.1. General Considerations

Our sound renderer is conceived for real time rendering. But nowadays, real time software-based sound rendering is only possible for very simple sound environments and a few microphones. In synthetic sound sensing, the problems are similar as in synthetic vision. The real time constraints in VR demand fast reaction to sound signals and fast recognition of the semantic it carries. Thus, we can in a first step model a sound environment where the synthetic actor can directly access to position and semantic of an audible sound event. This allows an actor to localize and recognize one or more sound sources in a reliable way and to react immediately. This access to the sound environment representation, however, makes it dependent of it and lets the communication problem with human participants in VR unresolved as they use natural spoken language. For this reason, there is a need for developing really independent actors that can communicate with participants. Such an autonomous actor will accept the same sound signal (digitized) as any other human participant in VR through its headphones. From these sound signals (stereo) the autonomous actor should estimate the position of a sound source and extract its semantic similar to its biological model. How a human localizes a sound in a real 3D environment is described by E. Wenzel in [W92]. By using interaural time differences (ITDs), interaural intensity differences (IIDs) and head-related transfer functions (HRTFs) that are finite impulse response filters (FIRs) constructed from ear-dependent characteristics, combined with head movements people can localize sound sources in space. A good sound renderer for humans in VR should take into account all of the factors previously described to produce a realistic 3D sound. In an ideal case we should use the same sound renderer without modification for synthetic actors to meet our above formulated paradigm about synthetic vision and audition. For computational efficiency, however, we opted for a preliminary version of sound rendering. We propose a real time structured sound renderer that is able to produce a synchronized sound track in AIFF format. In a future extension it should pass the audio signal to a speech recognition module that should pass semantic and noisy position data of sound sources to synthetic actors. With the speech recognition module, an actor should be able to extract some semantic information of some spoken language. By adding a speech synthesizer, the synthetic actor should be able to understand and synthesize a reduced set of vocabulary allowing it to communicate with human participants in VR. Actually, however, we use the sound renderer only for sound track generation for film production and use a sound event based acoustic model for real time applications.

Figure 2.4.1 shows the architecture of a possible (ideal) animation system with a real-time structured sound renderer serving as audition channel for synthetic actors and VR participants. Communication between synthetic actors could also be realized by speech recognition and speech synthesis where audio signals pass by the sound renderer in order to undergo the environmental influences. Of course, such an approach seems to represent a waste of efficiency and computation time as we can pass position and semantic information directly from one actor to another. But on the other hand, it is an approach simulating human communication, and it could represent a standard principle of acoustic perception for synthetic actors of at least scientific and philosophical interest. A future goal of synthetic humanoid modeling is to make them exchangeable with users in virtual environments. Spoken language communication is an essential step toward this goal. If synthetic humanoids can communicate with users through spoken language they can do it also with each other. Such an acoustic perception model can be used for synthetic actors and interactive users.

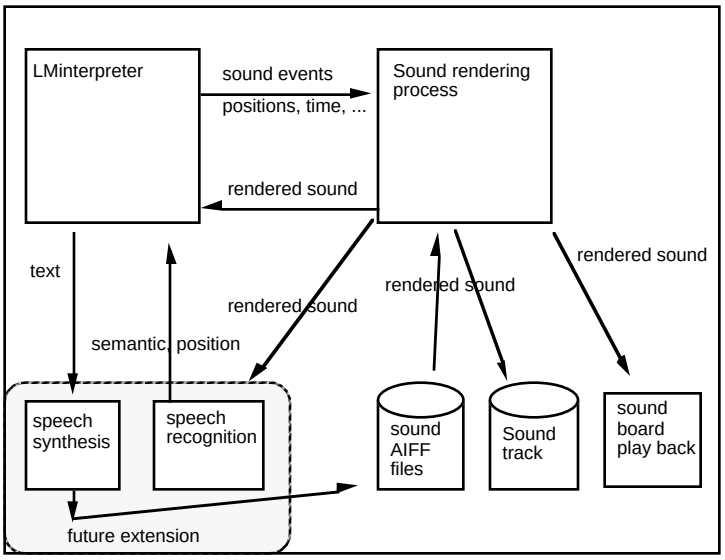


Figure 2.4.1: The architecture of a possible audition model.

2.4.2 Real Time Sound Modeling

For interactive applications a sound rendering approach by two passes has to be modified as simulation and behavioral animation must be in real time. The sound events have to be rendered for each actor at each frame. Thus, the actors can react immediately. Therefore, we propose a real time structured frame work for modeling a 3D acoustic environment with sound sources and microphones (the actors' ears). For real time applications we don't do any special sound rendering. We only treat the sound events.

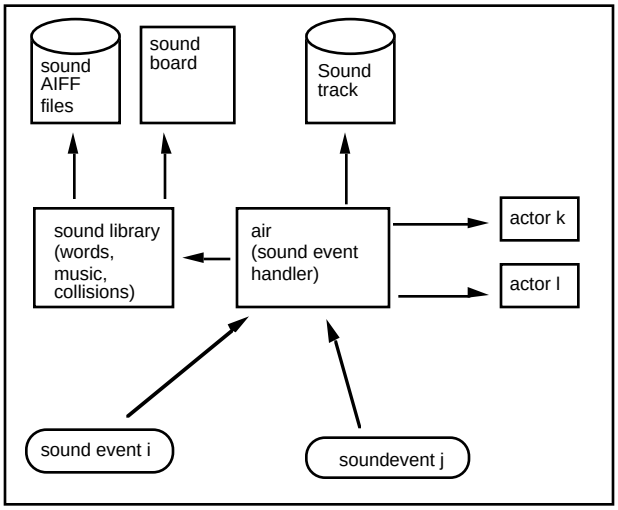


Figure 2.4.2: Architecture of the sound environment

Figure 2.4.2 shows the architecture of the real time sound environment model. The propagation medium corresponds to the sound event handler that controls the sound events and transmits the sounds to the ears of the actors and/or to a user and/or a sound event track file. Each sound event is emitted to the propagation medium, which controls the playback, the soundtrack file updating, the duration of the sound and the update of the sound source position during the activation time of the sound. Each actor can access through a table of the module **air** the current active sounds. We suppose an infinite sound propagation speed of the

sound without weakening of the signal. The sound sources are all omnidirectional, and the environment is supposed to be non reverberant.

An acoustic environment can be modeled by using procedures from the two modules "soundlib" and "air". The module "soundlib" offers the possibility to define a set of sounds and to play them on the SGI hardware. The sound library offers too, the possibility to use it without the play back facility. Thus, it can be used on machines without sound card.

2.4.3 A Sound Renderer for Film Productions

For the sound track generation of actual film productions and future extensions to real time VR sound rendering we developed a real time structured sound renderer, which makes part of the animation system as external process as indicated in figure 2.4.3. After an animation producing a log file of the sound events, this process renders the log file to a final sound track.

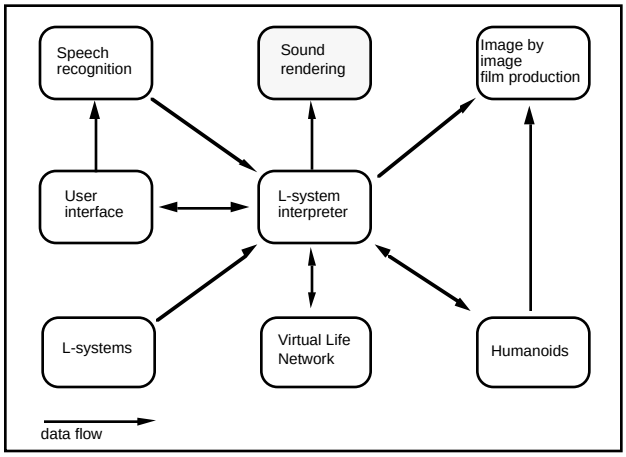


Figure 2.4.3: The sound renderer as external process

A virtual acoustic world is typically composed of sound sources, microphones and some geometric objects modifying the sound during its travel through the environment. The microphones can also be interpreted as ears of the synthetic actors or a soundtrack of a film. The sound sources, microphones and geometric objects can move around in their environment, and they are all characterized by a time dependent position and perhaps by an orientation dependent filter function influencing the emitted, reflected or perceived sound.

The sound sources produce at given times sound events that are determined by a sound identifier, a start time (in 3D object space), the position and the orientation of the object. The sound identifier is in our case the file name of a digital audio file of AIFF format. As the sound rendering is conceived to work in real time for real and synthetic actors, we have to use an image (audio) space algorithm [TAKHA92] that determines the value for each (audio) image buffer position by sampling in object space (3D) at a point given by the inverse transformation.

In order to parallelize sound rendering and geometrical image rendering we separated the sound world completely from the geometrical world. Actually, at each frame the sound renderer reads the position and orientation of all sound sources and the sound events from a 3D sound track text file. The corresponding BNF (SoundItem) and its semantic is given by definition A.1 and table B.7.

A line of this file is associated to a frame and can contain several key words with their corresponding parameters. It is parsed by the sound renderer that updates with this information its internal data model before rendering a frame.

The sound renderer produces for each frame (PAL, 50 Hz) the sound frames of 0.04 sec length for each microphone. According to the application, this sound is directly played on the hardware and/or it is recorded in a AIFF file for later use for a sound track.

For the sound rendering without orientation dependencies of the source and the microphone we use the formula (2.4.1). We suppose the sound source to be omnidirectional forming a spherical field of emission and to be in an environmental context without reflections (e.g., an anechoic chamber).

$$A_m(t) = \sum_{i=1}^{nbr\_sources} A_i\left(t - \tau_{m,i}(t)\right)\left(\frac{d}{c + d_{m,i}(t)}\right)^2 \qquad (2.4.1)$$

A_m is the amplitude of the rendered sound at the microphone m at time. It is the superposition of the amplitude (A_i) of all audible sound sources (i) with a delay of $\tau_{m,i}(t)$ and weakened by the last quadratic term of equation (2.4.1). At long distances when $d_{m,i}(t)$ is large, this term is proportional to the inverse of the square of the distance and conserves thus the energy of the signal. At short distances the singularity of the classical term (d/d(t))2 as described in [TAKHA92] is avoided. The volume can be adjusted by the constants d and c.

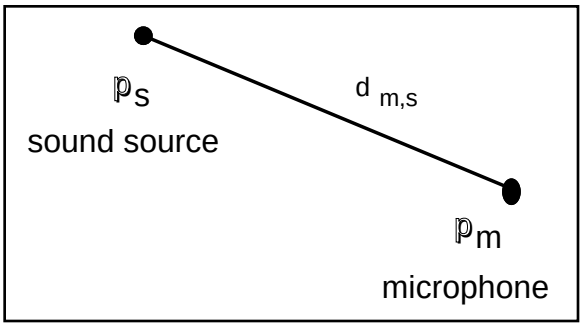


Figure 2.4.4: A sound source p_S and a microphone p_m with distance $d_{m,s}$

As a consequence of the finite sound propagation speed (v_{air} = 330 m/sec) we hear at time t the delayed sound signal A(t-τ). To calculate the effective distance traveled by the sound, we have to know the position of the source at time t-τ. Figure 2.4.4 and equation (2.4.2) show the details.

$$d_{m,s}(t) = \left\|\vec{p}_m(t) - \vec{p}_s(t - \tau)\right\|$$

$$\tau_{m,s}(t) = \frac{d_{m,i}(t)}{v_{air}}$$

t: time

$\vec{p}_m$: position of microphone m

$\vec{p}_s$: position of source s (2.4.2)

d: distance betwenn the source s and the microphone m

v_{air}: sound propagation speed in the air

τ: delay of the sound

In order to calculate the effective path length of the sound, the sound renderer has to memorize the trajectories of the sound sources, at least for some time back to the past. We memorize the trajectory for 1 sec (= 25 positions and orientations), which allows us to treat sources until ~300 meters away from the microphones. The trajectory memories can easily be implemented by cyclic tables. If the relative speed between the microphone and a sound source is lower than the sound propagation speed, the effective path length $d_{m,s}(t)$ and the exact delay $d_{m,s}$ can be determined by the following converging iterative process.

$$
\begin{aligned}
&d_{m,s,0}(t) = \left\| \vec{p}_m(t) - \vec{p}_s(t) \right\| \\
&\tau_{m,s,0}(t) = 0 \\
&d_{m,s,k+1}(t) = \left\| \vec{p}_m(t) - \vec{p}_s\left(t - \tau_{m,s,k}(t)\right) \right\| \\
&\tau_{m,s,k+1}(t) = \frac{d_{m,s,k}(t)}{v_{air}} \\
&d_{m,s}(t) = \lim_{k \to \infty} \left(d_{m,s,k}(t) \right) \\
&\tau_{m,s}(t) = \lim_{k \to \infty} \left(\tau_{m,s,k}(t) \right)
\end{aligned}
\qquad (2.4.3)
$$

It is evident that a recursive determination of the distance for each amplitude value at sampling frequencies of 22050 Hz, 44100 Hz or even 48000 Hz is very time consuming. Therefore, we realized a key frame approach as proposed by [TAKHA92]. The exact positions and orientations are only determined at each frame (1/25 sec). The intermediate values are linearly interpolated according to the following equations. Thus, the sound renderer memorizes at each frame the old position values before iterating the new values.

$$
\begin{aligned}
&t_i = i * \Delta t = i * 0.04, \ \mathrm{i} = 0, 1, 2, 3, \ldots \\
&t \in \left[t_{i-1}, t_i \right] \\
&d_{m,s}(t) = d_{m,s}\left(t_{i-1}\right) + \left(t_i - t\right)\left(d_{m,s}\left(t_i\right) - d_{m,s}\left(t_{i-1}\right) \right)
\end{aligned}
\qquad (2.4.4)
$$

The object positions are also interpolated between consecutive frames. Equation 2.4.5 shows the details. The key values are the positions and orientations memorized in the cyclic trajectory table of the sound renderer.

$$
\begin{aligned}
&k = \mathrm{int}\left(\frac{\tau}{\Delta t} \right) \\
&frac = \tau - k \\
&\vec{p}\left(t_i - \tau\right) = \vec{p}\left(t_{i-k-1}\right) + frac \cdot \left(\vec{p}\left(t_{i-k}\right) - \vec{p}\left(t_{i-k-1}\right) \right)
\end{aligned}
\qquad (2.4.5)
$$

Once the delay τ determined, the source amplitude values $A(t-\tau)$ can be read from the AIFF file. As $t-\tau$ doesn't always correspond to a sampling point, intermediate amplitudes have also to be interpolated using neighboring values. Linear interpolation is sufficient as already stated by [TAKHA92] to avoid audible secondary effects.

A mono sound is a sequence of 16 or 8 bit amplitude values A_k. By giving a start time of the sound and a sampling frequency sampling_freq, the following equations determine the

resulting amplitude at time t-τ. Now, as we have determined all parameters of equation (2.4.1), the resulting amplitude at time t can be calculated.

$$
\begin{aligned}
t' &= t - \tau(t) - t\_initial \\
\Delta t &= {}^{1}\!/_{sampling\_freq} \\
k &= \mathrm{int}\left(\frac{x}{\Delta t}\right) \\
frac &= x - k \\
A\left(t - \tau(t)\right) &= A_k + frac \cdot \left(A_{k+1} - A_k\right)
\end{aligned}
\qquad (2.4.6)
$$

Actually, we use this sound renderer only for sound track generation for film productions. But its real time structure makes it suitable also for rendering the acoustic environment in interactive applications. A stereo sound, rendered frame by frame, would considerably increase the presence for a headphone wearing user as environmental sound information is a precious help in sound event localization. This sound renderer already takes into account interaural time (ITDs) and intensity (IIDs) differences, if the synthetic microphones are placed at the virtual position of the ears of the user. Head-related transfer functions (HRTFs) would improve the localization possibilities. However, such an extension is left to future work, as the actual speed of the present computer generation allows only simple software based sound environment rendering in real time. It is evident that 100% real time sound rendering is demanded as only small delays between sound frames (1/25 sec) would lead to nasty acoustic effects.

2.4.5 Speech Recognition

A considerable part of human communication is based on speech. Therefore a believable virtual humanoid environment with user interaction should include some speech recognition. In order to improve real time user interaction with autonomous actors we extended the L-system interpreter with a speech recognition feature (see figure 2.4.5), which transmits spoken words that are captured by a microphone into the virtual acoustic environment by creating corresponding sound events that are perceptible by autonomous actors. This concept enables us to model behaviors of actors reacting directly on user spoken commands.

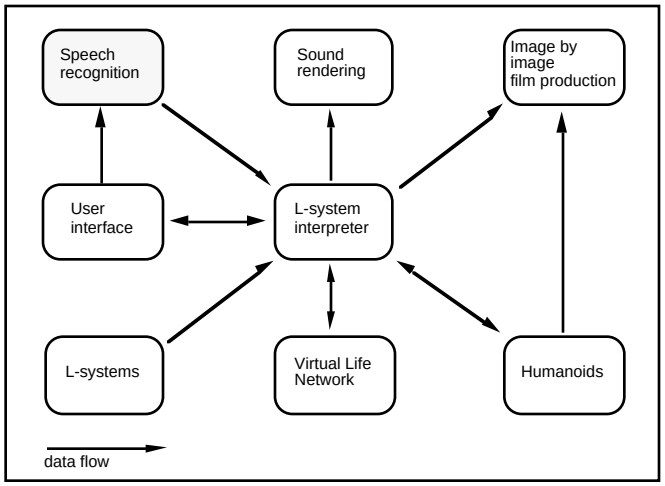


Figure 2.4.5: Speech recognition as natural language interface

For speech recognition we use POST (Parallel Object oriented Speech Toolkit, [HD96], [HTTPpost]), a toolkit developed for designing automatic speech recognition. POST is distributed freeware to academic institutions. It can perform simple feature extraction, training and testing of word and sub-word Hidden Markov Models with discrete and multi Gaussian statistical modeling. We use a POST application for isolated word recognition.

The system can be trained by several users and its performance depends on the number of repetitions and the quality of word capture. This speech recognizing feature was recently added to the system and we don't have much experience with its performance. First tests, however, with a single user training resulted in a nearly 100% recognition rate. A detailed description of the architecture of the speech recognition system can be found in section 6.4.

2.4.6 The Sound Symbol of the L-system

As the whole virtual environment is managed by the L-system interpreter it makes sense to employ a special symbol of the L-system producing sound events and transmitting them into the virtual acoustic environment. Note that sound events can also be produced by actor behaviors, built in force field peak detectors or the speech recognition module. All of them are transmitted into the same virtual acoustic environment. Such a sound symbol offers the possibility to model sound event sequences by production rules. Therefore, the L-system offers the possibility for geometric, acoustic and force field modeling, which represents an original and useful extension for behavioral L-systems.

The symbol **T** allows to create sound events, to play back the corresponding sounds, and in case of sound rendering it writes data into the 3D log file of the sound track. The 3D log file generation of sound events has to be declared in the “Options” section of the L-system definition file by

Sound_event_file *file_name* *list_of_initializations*.

The first argument is the file name of the 3D sound track log file and the second argument contains the list of microphone and sound source initializations. More details can be found in appendix A, B and D.

3. The Actor - Environment Interface

In the above chapters we presented a frame work of the environment of the virtual autonomous agents including geometric objects, force fields and sound events. In this chapter we present models for synthetic sensors for the agents through which they can perceive parts of their virtual environment.

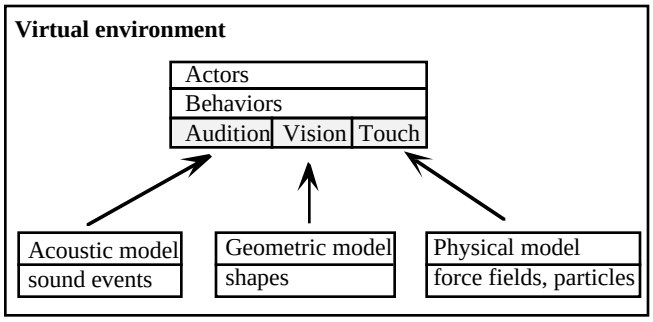


Figure 3: The synthetic sensors in the virtual environment

The actor - environment interface, or the synthetic sensors, constitute an important part of a behavioral animation system. As sensorial information drastically influences behavior, the synthetic sensors should simulate the functionality of their organic counterparts. Because of real time constraints we did not make any attempt to model biological models of sensors. Therefore, the synthetic vision only makes efficient visibility tests by using SGI's graphics rendering hardware to get a Z-buffered color image, representing an agent's vision. A tactile point-like sensor will be a simple function evaluating the global force field at its position. A synthetic "ear" of an agent will be a function returning the on-going sound events. What is important for an agents behavior is the functionality of a sensor and how it filters the information flow from the environment and not the specific model of the sensor.

Another aspect of synthetic sensor design is its universality. The sensors should be as independent as possible from specific environment representations and they should be easily adjustable for interactive users. For example, when we use synthetic vision, the display for an autonomous actor and a user is essentially done by the same Z-buffer based renderer. The user and the autonomous actors perceive the virtual environment through rendered images, without knowing anything about the internal 3D environment representation or the rendering mechanism. The sense of touch plays also an important role for humans. It can be simulated by different techniques. Geometric collision detection, for instance, would be an elegant and precise technique to model sense of touch. As behavioral L-system environments, however, are generally very complex and only defined by production rules, it would be necessary to create at each frame first a database of the geometric primitives to calculate possible collisions. Additionally, the physically response of collisions should also be treated to get realistic looking animations. This approach would only work in real time for simple environments, and it would be difficult to implement. Another possibility to model sense of touch is the use of a physically based force field model. This model is close to reality as the real sense of touch feels also collision forces. By additionally adding a physically based evolution of objects we can extend the force field model to a physically based animation system where touch sensors correspond to special functions evaluating the global force field at its position. This approach solves also the response problem of collisions as they are treated automatically by the physics-based evolution system if both colliding objects (sensor and touched object) exert short range repulsion forces, for example. We opted for a force field based model for the sense of touch realization as it integrates itself naturally in the physically

based particle system we already use for physical and behavioral animation as discussed in section 2.3. Of course, for real time applications the number of sensors and particles must be small as the evolution of the particle system is computationally expensive. Another disadvantage of this approach is that the touch sensors only "feel" geometric shapes which are explicitly bounded by appropriate fore fields. It is up to the user to decide whether objects are only visible or visible and touchable. Another difficulty of a force field model is the fact that the parmeterization of the force fields, the numerical integration of the system of differential equations, the time step and the speed of moving objects are inter dependent, and that the adaptation of all parameters for a stable solution is not always trivial.

As extensively discussed in section 2.4 we use a sound event frame work for the control of the acoustic model of the animation system. The sound event handler maintains a table of the on-going sound events. Therefore, it is immediate to model synthetic listening by simple querying this table of on-going sound events. An interactive user can also produce sound events in the virtual environment via a speech recognition module. Through a sound event an autonomous actor can capture directly its semantic, its position and the source which emitted it. In future extensions we should add some emotional information as voices express also emotions, and as the humanoid library (see section 4.2) we use in our animation system will be extended with a set of facial actions based on the work of P. Kalra [K93]. These facial actions include also emotions such as fear, joy or anger, for example. In the following sections we present synthetic sensor models for vision, audition and touch.

3.1 Vision System

In our implementation of the vision-based approach to behavioral animation the synthetic actor perceives its environment through a small window in which the environment is rendered by the computer from the actor's point of view. The rendering is based on Z-buffer techniques. In [AKE90] Akeley describes in detail the hardware Z-buffer of the IRIS graphics system. The Z-buffer is a set of 24-bit numbers related to the pixels on the screen. During rendering graphical objects are reduced to pixels, each with its own color, x and z coordinates. The z-coordinate effectively specifies the distance from pixel to eye. When rendering a frame, first each Z-buffer location must be initialized to farthest. During rendering, before a pixel's color is written to the frame buffer, its z-value is compared with the value already stored in the Z-buffer. If the new pixel is nearer, its color is written into the frame buffer and its z is written into the Z-buffer. Thus, at the end of the rendering process only the closest pixels are visible. Akeley also gives the formula to calculate the distance of a given pixel to the eye when knowing the projection and the viewpoint transformations. We use this formula only for specialized applications with local vision. In general, we use the conceptually more elegant matrix formulation given in equation 3.1.2 that calculates the 3D world coordinates of a given pixel.

As an actor can access Z-buffer values (distances of the objects' pixels from the observer) of the pixels, the color of the pixels and its own position, it can locate visible objects in the 3D environment. This local information is sufficient for some local navigation. For global navigation, however, a visual memory is useful in order to recognize impasses in global navigation problems such as searching for an exit of a maze. We modeled the visual memory by a 3D occupancy octree grid, where each pixel of an object, transformed back to 3D world coordinates, occupies a voxel. By comparing at each frame the rendered voxels in the visual field, with the corresponding pixel of the vision window we can update the visual memory by eliminating voxels having disappeared in the 3D world. Thus, the visual memory reflects the state of a 3D dynamic world perceived by the synthetic actor.

The concept of synthetic vision with a voxelized visual memory is independent of a 3D world modeling. Even fractal objects and procedurally defined and rendered worlds without 3D object database can be perceived as long as they can be rendered into a Z-buffer based vision window. We use synthetic vision in conjunction with a visual memory for environment recovery, for global navigation, for local navigation optimization and for object recognition through color coding in several behaviors described in chapter 5.

3.1.1 Synthetic Vision

Synthetic vision is the simulated vision for a synthetic actor. The computer renders from the synthetic actor's point of view the virtual environment in a window corresponding to the actors vision image. This image must contain the Z-buffer values and the color of each pixel. In this section we describe the model of the synthetic vision we use and how we extract the position of a pixel in the 3D scene. For local applications it is sufficient to determine the z component of the distance of a given pixel to the observer. For global applications, however, we need the exact 3D position of a given pixel. The following equations show how to extract this information from the Z-buffer values from an image with ROW_LENGTH * ROW_LENGTH pixels.

In the normalized volume of vision, a pixel (i, j) is represented by the normalized point

$$\vec{q}_{norm} = \begin{pmatrix} -1 + 2(j + 0.5) / Row\_length \\ -1 + 2(i + 0.5) / Row\_length \\ -1 + 2z_{screen}/Z_{max} \\ 1 \end{pmatrix} \qquad (3.1.1)$$

where z_{screen} is the Z-buffer value of the pixel (i, j) and with $0 <= z_{screen} <= Z_{max}$

To get the 3D position in the real scene we can inverse the modeling and projection transformation and multiply them with the normalized point.

$$\vec{q}_{real} = \vec{q}_{norm}(MP)^{-1}$$

$\vec{q}_{real}$: point of the real scene

$\vec{q}_{norm}$: normalized point

M: modeling matrix

P: projection matrix (3.1.2)

3.1.2 Object Recognition and Color Coding

In robotics image recognition and extraction of semantic information from images is a difficult problem. In virtual environments using synthetic vision this topic is easier to handle as we don't encounter real world's complexity. In navigation problems all perceived objects are only obstacles. Very often only simple object discrimination is needed. In tennis playing, for example, the actors need to differentiate with their vision system between the ball, the other actors and the rest of the environment. To get semantic information from the vision we use color coding. By associating a certain color to an object. and giving this knowledge to an actor it can localize these objects by the pixel color of its vision image. In reality diffuse reflected light has the same normalized color but different luminance. Thus, the normalized color is best used for color discrimination [GM87] when a diffuse reflection lighting model is used.

Color coding

Color discrimination is simple if a non shading rendering technique is used for the vision system of the actors. In this case each object in the virtual environment needs besides the material attribute, which is used for shading rendering for the user display, an additional color attribute, which is used for the non shading rendering. Thus, the color is preserved, and at pixel level it can be recognized by simple identity tests. We use this approach for time critical applications.

Color coding can also be used if only a shading rendering technique is used. This approach is more universal as we need only material attributes for objects and one rendering technique. However, the results of the color discrimination are not sure any more as the shaded rendering modifies the color of an object. To find a solution to this problem we adopted the following approach.

The normalized color space is given by the x, y and z coordinates of the system of equations (3.1.3). If an actor looks for an object with a given normalized color (r_obj, g_obj, b_obj) representing a point in the normalized color space, it can calculate the chromatic distance between the normalized color of a given pixel (r_pix, g_pix, b_pix) and the object's color. If this distance is smaller than a certain threshold value, the pixel represents a part of the object.

$$\begin{aligned} &\text{sum} = r + g + b \text{ (sum of rgb values)} \\ &x = r/\text{sum} \\ &y = g / \text{sum} \\ &z = b / \text{sum} \\ &\Rightarrow x + y + z = 1 \end{aligned} \qquad (3.1.3)$$

chromatic distance =

$$\sqrt{(r_{pix} - r_{obj})^2 + (g_{pix} - g_{obj})^2 + (b_{pix} - b_{obj})^2} < dist_{max}$$

with

$$r_{obj} + g_{obj} + b_{obj} = r_{pix} + g_{pix} + b_{pix} = 1 \qquad (3.1.4)$$

As an actor interacts with its environment through its vision system, the rendering process should leave the normalized color invariant. We use the GL lighting facilities of Silicon Graphics for the rendering, which has this invariance property when the following rules are followed in choosing materials for objects:

1. Avoid similar colors, the chromatic distance between two objects should be bigger than 0.2 (see equation 3.1.4)

2. The ambient, diffuse and specular components should have the same normalized color.

3. Use only white or gray lights.

4. Use also an ambient light allowing to recognize objects in the shadow.

In the tennis game simulation, for instance, we used the following material definitions for the tennis ball.

ambient	(0.4, 0.2, 0.1)
diffuse	(0.8, 0.4, 0.2)
specular	(0,0,0)

The normalized color of the ambient and diffuse components is (0.57, 0.29, 0.14). By using a chromatic distance threshold value of 0.05 the actors easily recognize the ball in an image. If the ambient rgb component is changed to 0.1, for example, an actor can detect this change.

Position and size of an object

To discriminate between objects we can also use the perceived size of an object as redundant information together with the perceived color. An estimate of the size of an object is a typical feature that can be easily extracted from an image. By counting the pixels of a given color belonging to an object we can estimate the surface size from the actors point of view (see equations 3.1.5, 3.1.6, 3.1.7).

$$x = \tan(viewAngle/2) * dist * 2 \quad (3.1.5)$$

$$Pixel\_area = (x / Nbr\_pixel\_per\_row)^2 \quad (3.1.6)$$

$$Object\_area = Pixel\_area * Nbr\_pixel\_object \quad (3.1.7)$$

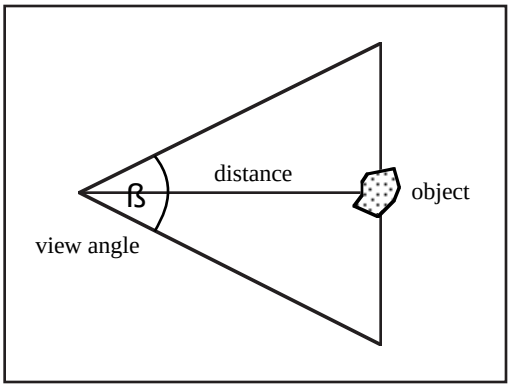


Figure 3.1.1: Object surface estimation

Figure 3.1.1 illustrates the equations describing the estimated surface of an object at a distance "dist" from the actor. Surface size estimation with upper and lower threshold values of each object can be used as additional redundant information to color coding in object discrimination.

Equations (3.1.5, 3.1.6, 3.1.7) give a good approximation of small surfaces centered in the vision image. If the surfaces cover a big part of the image or if they are not centered, the following approach yields better results. We can calculate the size of each pixel and sum them up to the surface size as shown in equation 3.1.8. This last method considers the depth of each pixel. Therefore, we get a better approximation of the surfaces.

In tennis playing, for example, the players have to estimate the ball's actual position. To calculate its position from the corresponding pixels we can invert the modeling and projection matrices and multiply them with the normalized pixel coordinates. To get the ball's position we have to average all the positions of the pixels belonging to the ball according to equation 3.1.9 It is evident that this only works for one ball of the same color per image.

$$z_{eye.i} = \frac{near}{\frac{z_{screen,i}}{F}\left(1 - \frac{near}{far}\right) - 1}$$

$$x_i = \tan\left(\frac{viewAn\lg e}{2}\right) \cdot z_{eye,i} \cdot 2$$

$$Pixel\_area_i = \left(\frac{x_i}{nbr\_pixel\_per\_row}\right)^2$$

$$Object\_area = \sum_{i=1}^{nbr\_pixel\_object} Pixel\_area_i$$

$z_{screen,i}$: z - buffer value of pixel i (3.1.8)

$z_{eye,i}$: z component of the distance of pixel i of object from the observer

F: maximal z buffer value

near, far: near and far clipping values.

$$\vec{p} = \frac{1}{nbr\_pixel\_object} \sum_{k=1}^{nbr\_pixel\_object} \vec{p}_{environment,k}$$

$$\vec{p}_{environment,k} = \vec{p}_{norm,k}(M \cdot P)^{-1}$$

(3.1.9)

$\vec{p}_{norm,k}$: normalised pixel

M: modeling matrix

P: projection matrix

3.1.3 The Octree as Visual Memory Representation

We decided to use an octree space grid as internal representation of the environment seen by an actor. The octree offers several interesting features. With an octree we can easily construct bounding objects by choosing an appropriate maximum depth level of the subdivision of space. For the problem of path searching detailed objects, such as flowers or trees, need not to be represented in all details. It is sufficient to represent them by some bounding cubes of the octree corresponding to the occupied voxels.

The octree adapts itself to the complexity of the 3D environment as it is a dynamic data structure subdividing recursively the space. Intersection tests are easily performed. To decide, whether a voxel is occupied or not, we only have to go to the maximum depth (5-10) of the octree by some elementary addressing operations. The examination of the neighborhood of a voxel is immediate.

An other interesting property of the octree is the fact, that it represents a graph of a 3D environment. You can consider, for example, all the empty voxels as nodes of a graph, where the neighbors are connected by edges. Therefore, we can apply algorithms of graph theory

directly on the octree, and it is not necessary to convert the representation of the octree into a special network suitable for path searching.

Perhaps the most interesting property of the octree is the simple and fast transition from the 2D image to the 3D representation. All we have to do, is to take each pixel with its depth information (given by the Z-buffer value) and to calculate its 3D position in the octree space. Then, we insert it in the octree with a given maximum recursion depth level. The corresponding voxel will be marked as occupied and perhaps with some more additional information as a time stamp, for example. More details about the octree implementation is given in chapter 6.

3.1.4 The Update of the Octree

The octree has to represent the visual memory of an actor in a 3D environment with static and dynamic objects. Objects in this environment can grow, shrink, move or disappear. In a static environment (growing objects are still allowed) an insert operation for the octree is sufficient to get an approximate representation of the world. If there are moving or disappearing objects like cars, other actors or opening and closing doors, we need an elimination operation for the octree that frees occupied voxels. The insert operation is without problems. The elimination of objects, however, is more complicated. We realized the following approach.

At a given instant, each pixel is inserted into the octree and the corresponding voxel is marked with the actual time stamp. After the insertion of all pixels of the image, all the voxels in the volume of vision are tested whether they have disappeared or not. The principle of such a test is shown in figure 3.1.2.

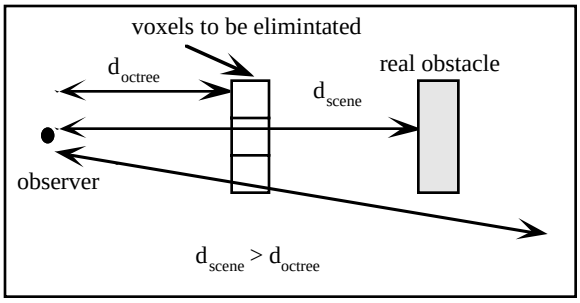


Figure 3.1.2: Occupied voxel removal in the octree.

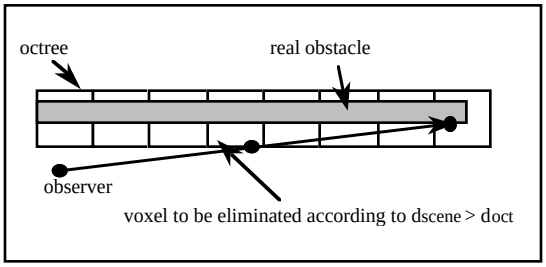


Figure 3.1.3: Imprecision.

An occupied voxel has to be eliminated only, if there is no real object at its position or in front of it. This condition is expressed by equation 3.1.16.

$$d_{scene} > d_{octree} \text{ (distances from the observer)} \qquad (3.1.16)$$

The distance of the voxel of the octree from the observer has to be smaller than the distance of a real object or the background. To get the distance of the voxel from the observer we have to transform the observer into the octree coordinate system and to apply the corresponding modeling and perspective transformation to the voxel. Doing this with all voxels in the vision volume, we get the image of the memory in normalized coordinates and we can compare it directly with the Z-buffer values of the 2D image of the synthetic world.

As the octree is only an approximation of the environment, we have a loss of information that leads to imprecision (figure 3.1.3). Therefore, it is necessary to verify, whether a voxel has really disappeared or not. A first verification consists in checking the time stamps of the voxels susceptible to disappear. If the voxel happens to be inserted, it isn't eliminated, because it recently has been seen. If it has, however, a certain age, it is inserted in an auxiliary octree

X containing all voxels susceptible to be eliminated. In this special octree X, the label of each voxel serves as a counter that is incremented each time the voxel is inserted. The real elimination of voxels in the memory octree is done always after a sequence of n images (typically n = 5). Then, all the voxels of the octree X are tested. If their label is bigger than m (typically m = 3) the voxels are eliminated from the octree. This way, all the voxels have to be noticed as disappeared by the actor at least m times before they are eliminated. After each elimination process, the octree X is reinitialized.

With these verification steps we get a dynamic octree that adapts itself to a dynamic environment with moving, growing, shrinking and disappearing objects. Thus, the actor has a dynamic visual memory. It can learn and forget what it has seen.

3.1.5 The Vision in the L-system

Synthetic vision in the L-system interpreter is an essential and original extension making it to a behavioral L-system. It introduces the concept of individual instantiated actors mapped to symbols of the L-system. These actors (or symbols) with synthetic vision, in contrast to other symbols of the L-systems, maintain like particles an internal state from one time step to the next. In traditional L-systems symbols don't maintain such individual states. Only parameter values of symbols are transmitted from one time step to the next one and can be used in conditions of production rules. If environmental information is used as in environmentally sensitive L-systems [PJM94] during the iteration of the formal object, this information has to be transmitted first to the parameter space by the query symbol **?** during the interpretation of the formal object. As actors and particles, however, have individual internal states in our L-system interpreter, they can directly "sense" environmental information through special functions.

Actors with vision and visual memory have to be declared in the "Options" section of the L-system definition file. After the declaration of the total number of actors by the line

NBR_actors *n*

a vision window tailored to the needs of a given actor can be declared.

For the actor *actor_nbr* the following line

iVisionWinSize_maxColors *actor_nbr win_edge_length maxColors*

declares a vision window of win_edge_length * win_edge_length pixels and the maximum number of recognizable colors for non shading color recognition. The visual memory is initialized and configured through an automata declaration (see table B.6). The four parameters of the "init" command initialize the visual memory. For example, the line

actorNbr_automata 0 0 9 init 10 -20 300 2.0 obs 0 0 50 lat 0 0 0

,for example, with an **init** command creates a cubic octree grid determined by the coin points (-20, -20, -20) and (300, 300, 300) with a voxel size of 2. The first argument 10 of the **init** command determines the repetition rate of the optional octree display (see also section "5.1 Automata Control").

To manipulate the synthetic vision of an actor and to link it with the L-system we use the symbol **M**. It places the turtle according to the actors position and orientation. A complete description can be found in appendix C. In this case the turtle can be regarded as an "intelligent" autonomous actor with vision.

3.3 Tactile Sensors

Ideally, geometric collision detection between surfaces should be used for the modeling of tactile sensors. But, as a typical L-system environment is composed of a large number of objects, and as there is no geometric database of the 3D objects, traditional collision detection is not the best solution for a tactile sensor model. As we already have a force field environment integrated in the L-system, we use a force field approach to model tactile sensor points. All we have to do is to define a function that can evaluate the amount of the global force field at a given position. This amount can be compared with a threshold value that represents, for instance, a collision. With this function even wind force fields can be sensed. Traditional collision detection between surfaces can provide a large number of collisions, and it will not always be easy to model the behavioral response. With the definition of only one or a few sensor points attached to an actor this behavioral response is easier controllable, and the calculation time is reduced, which is important for real time applications. To such a sensor point we can also associate a particle with an appropriate force field that acts automatically on other particles. Thus, an actor can "feel" and manipulate other particles.

In order to use tactile information to model behavior with production rules, the force field sensing function must be usable in the conditions of the production rules during the derivation phase of the symbolic object. During the interpretation of the symbolic object with the query symbol **?** the turtle position can be copied into the parameter space of the symbol, and thus, the turtle position (x, y, z) is available in the parameters x, y, and z of the query symbol for the force field function. This force field function (?40, see table B.5) returns the amount of the force felt at this position (x, y, z), and it can be used in conditions to trigger certain behaviors, represented by the right side symbols of a production rule.

The query symbol **?**, which makes part of environmentally sensitive L-systems [PJM94], can be used also for a geometric collision detection of mathematical surface definitions in the condition function. The condition,

[(y)<] or (y < 0)

for example, is true when the y component of the turtle is smaller than zero. Therefore, this condition detects collision with the ground, if we suppose it to be at y = 0 and parallel to the x, z plane.

3.4 The Audition Sensors

The audition sensor of an actor corresponds to the table of the currently active sounds furnished by the sound event handler representing the propagation medium described in section 2.4. From this table the actor gets the complete information of each event consisting of the sound identifier, the sound source and its position. The semantic information of a special sound identifier is known to the actor and in general included in compiled behavior automata which use sound events.

We introduced also some audition functions usable in the conditions of production rules to be able to model audition dependent behaviors with production rules. The first function (?60, see table B.5) returns the number of the currently on-going sound events. To discriminate several sound events we introduced the function ?61 and ?62. They permit to query the sound identifiers and the sound sources of on-going sound events and enable a more subtle sound dependent behavior modeling.

4. Actor Representation

Actors, as illustrated in figure 4 are important entities in the virtual environment. They maintain an internal state from one time step to the next, and they can sense their environment through synthetic vision, audition and touch. All of them are controlled by the L-system interpreter, partially via external processes. They dispose on a set of versatile behaviors. Actors can be represented in several ways.

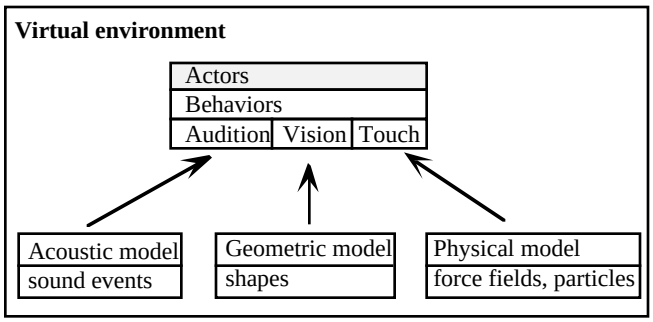


Figure 4: The actors in the virtual environment

In a simulation the actors can have different geometric shapes that range from simple cubes to complicated surfaces or articulated bodies. Simple actors in real time applications can be replaced by complex body surfaces in image by image film productions. Actors can be modeled by production rules or they can be represented by imported SM surfaces. For the humanoid actors we use a special shared memory interface of a parallel server process that communicates through it with the L-system interpreter. In a real time application the humanoids are represented in the L-system with a simple skeleton structure, whereas in film productions the humanoid server produces a deformed body surface in the ray tracer format for each humanoid at each frame.

Another shared memory interface allows communication and synchronization with the virtual life network (VLNET), which allows the autonomous actors to participate in a shared networked environment. In the next three sections we will present in more detail these three different actor types, i.e. L-system modeled actors, imported humanoid actors and networked actors.

4.1 L-system Modeled Actors

The easiest way to visualize an actor in the L-system is to represent it by a geometrical symbol as a cube or a sphere. This symbol, for example, can be placed after the vision symbol **M** that maps the actor's vision system to the turtle. Then, the cube or the sphere are always positioned and oriented according to the vision system.

The example in code 4.1.1 shows a fish model from a force field group animation. After each "fish" particle represented by a symbol **C** we placed a dummy symbol **x** that defines a fish through a production rule. The fish is composed of several scaled cylinders **O** and their periodic rotations **&** simulating the swim movement. These periodic rotations depend on the particle's speed which is taken into account through the functions ?20 and ?34 (see table B.5). Figure I.2 of appendix I shows such L-system modeled fishes.

```
Camera_AnimationParameters ...
RepetitionParameters ...
Options...
Splines ...
Constants ...
Surfaces ...
Axiom ...
[ ()  () () ()  () () ()  /* pushes the turtle state */
x ()  () () ()  () () ()  /* a symbol for a production modeling a fish */
] ()  () () ()  () () ()  /* pops the turtle state */
...
Productions
x (1) () () ()  /* the production of a fish */
1.0  /* the probability of the production application */

n ()  (4) () ()  () () ()  /* sets cylinder resolution */
m ()  (2) () ()  () () ()  /* activates the material 2 (brown) */
^ ()  () () ()  (?20_) () ()  /* phase setting: phase[i] += 0.2 * speed_i * t_interval */
/* the head of the fish */  /* i is the current fish particle */
[ ()  () () ()  () () ()
O ()  () () ()  (0.5) (5) (2)
O ()  () () ()  (2) (4) (1.5)
O ()  () () ()  (2) (3) (1)
/* the eyes of the fish */
[ ()  () () ()  () () ()
m ()  (7) () ()  () () ()
& ()  () () ()  (90) () ()
O ()  () () ()  (1) (1) (1)
^ ()  () () ()  (180) () ()
O ()  () () ()  (2) (1) (1)
m ()  (2) () ()  () () ()
] ()  () () ()  () () ()

- ()  () () ()  (50) () ()
O ()  () () ()  (2.5) (3) (1)
] ()  () () ()  () () ()
/* the body of the fish */
+ ()  () () ()  (180) () ()
O ()  () () ()  (2) (5) (2)
& ()  () () ()  (10?34-$5*) () ()  /* sin(10 - phase[i]) * 5 */
O ()  () () ()  (2) (4.8) (1.5)
& ()  () () ()  (25?34-$5*) () ()
O ()  () () ()  (2) (4.4) (1)
& ()  () () ()  (50?34-$5*) () ()
O ()  () () ()  (2) (3.7) (0.7)
& ()  () () ()  (75?34-$5*) () ()
O ()  () () ()  (2) (3.3) (0.5)
& ()  () () ()  (100?34-$5*) () ()
O ()  () () ()  (2) (2) (0.4)
& ()  () () ()  (125?34-$5*) () ()
O ()  () () ()  (2) (1) (0.2)
& ()  () () ()  (150?34-$5*) () ()
O ()  () () ()  (1) (0.5) (0.2)
& ()  () () ()  (150?34-$10*) () ()
f ()  () () ()  (0.2_) () ()
```

```
/* the tail-fin */
m ()  (8) () ()  () () ()
& ()  () () ()  (180?34-$10*) () ()
+ ()  () () ()  (60) () ()
{ ()  () () ()  () () ()          /* polygon push */
. ()  () () ()  () () ()          /* adds the actual turtle position to the current polygon */
f ()  () () ()  (4) () ()         /* the turtle moves forward by 4 units */
. ()  () () ()  () () ()
- ()  () () ()  (150) () ()
f ()  () () ()  (3) () ()
. ()  () () ()  () () ()
+ ()  () () ()  (30) () ()
f ()  () () ()  (4) () ()
. ()  () () ()  () () ()
} ()  () () ()  () () ()          /* draws the current polygon and pops the next from the stack
*/
EndProb
End
EndProductions
```

Code 4.1.1. An animated L-system fish model

4.2 Imported Humanoids

A sophisticated actor representation is offered by an external process as illustrated in figure 4.2, supporting humanoids with motor programs as walking, grasping, or key framing, for example.

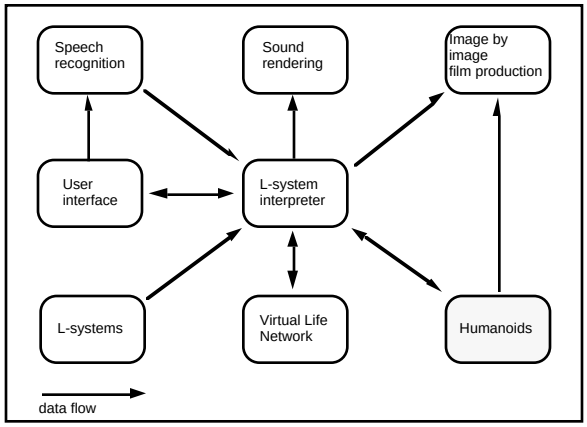


Figure 4.2: Humanoid modeling as external process

For the European ESPRIT project Humanoid 2 [Eagent8125], the computer graphics laboratory (LIG) of EPFL and MIRAlab of the University of Geneva developed the humanoid software library h2_agentlib based on AGENTlib. The humanoid environment for interactive animation of multiple deformable humans is published in [BCH95]. The L-system interpreter supports a shared memory interface to a special process, called h2_process.x, which animates actors based on these libraries according to the protocol defined by the BNF given in definition 4.2.1.

```
ActorCommands ::=
    INIT float float float float float float float int Word |/* initialization */
    KEYFRAMEdef int Word |     /*  imports a TRACK file and associates a number to it */
```

```
RS int |                        /*  ray tracer flag */
RV float |                      /* walking speed */
OV float |                      /* turning speed */
AK int |                /*  activates a TRACK (number = int) playback */
GR int |                /*  grasp an object (number = int) */
MA int |                /*  sets actor material identifier (int) */
SW |                            /*  stops walk action */
SK int |                        /*  stops TRACK action (number = int) */
RK int |                /*  reverses TRACK action (number = int) */
AW |                    /*  activates walk action */
AR float float float |          /*  activates 'reach' action for the right hand */
                                of the given coordinates */
DE |                            /*  detaches the current grasp object */
EXIT |                          /*  stops the humanoid process h2_process.x */

PO int float float float float float float |
/*  picks, positions and orients a grasp object (number = int) */

CA float float float float float float |
/* positions and orients (lookat point) the camera of the humanoid process h2_process.x */
```

Definition 4.2.1: The BNF of the actor commands that are exchanged between the L-system interpreter and the external concurrent humanoid process.

The INIT command must precede at the beginning of a simulation all other commands. It initializes an actor and sets the corresponding parameters. With the command RS the creation of a ray tracer file containing the actor can be switched on or off. It is switched on only for ray traced film productions where the process h2_process.x produces for each actor at each frame a numbered rayshade file which can be included in an animation through the symbol **D** (see appendix B). The other commands allow to guide the humanoids according to the currently implemented features of AGENTlib defined in the Humanoid II project. More details on the importation of humanoid actors are given in section 6.3.2.

4.3 VLNET Interface

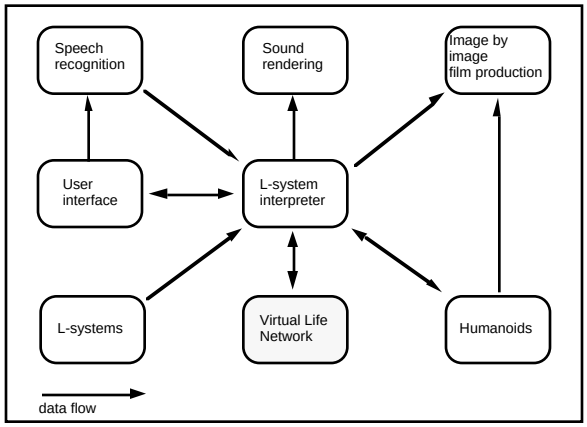


Figure 4.3: The Virtual Network as external process

A networked actor representation is offered by a special shared memory interface to external VLNET client processes as illustrated in figure 4.3. VLNET [PCTT95] was developed by I. Pandzic (MIRAlab, University of Geneva) and T. Capin (LIG, EPFL) and we use it in our animation system through a general interface library for VLNET clients offered by the VLNET system. Through this network autonomous actors of the virtual environment can

participate in a networked shared virtual environment with participants elsewhere in the world, using VLNET.

VLNET, the Virtual Life NETwork [PCTT95], is a shared virtual life network with virtual humans that provides a natural interface for collaborative working and games. It allows multiple users to interact with each other and their surrounding in real time. The users are represented by 3D virtual human actors, which serve as agents to interact with the environment and other agents. The agents have similar appearance and behaviors with the real humans, to support the sense of presence of the users in the environment. In addition to user-guided agents, the environment can also be extended to include fully autonomous human agents which can be used as a user friendly interface to different services such as navigation. Virtual humans can also be used in order to represent the currently unavailable human partners, allowing asynchronous cooperation between distant partners. In the following sections we show how the behavioral L-system animation system can use the VLNET system for networked animations with interactive users and synthetic autonomous actors.

4.3.1 The VLNET System

In this section we describe shortly some features of the VLNET system [PCTT95] before presenting how we use it in our animation system. The shared environment incorporates different media; namely sound, 3D models, facial interaction among the users, images represented by textures mapped on 3D objects, and real-time movies. Instead of having different windows or applications for each medium, the environment integrates all tasks in a single 3D surrounding, therefore it provides a natural interface similar to the actual world. The objects in the environment are classified into two groups: fixed (e.g. walls) or free (e.g. a chair). Only the free objects can be picked, moved and edited. This allows faster computations in database traversal for picking. In addition to the virtual actors representing users, the types of objects can be: simple polygonal objects, image texture-mapped polygons, etc. Once a user picks an object, he or she can edit the object. Each type of object has a user-customized program corresponding to the type of object, and this program is spawned if the user picks and requests to edit the object. In VLNET, three types of real time virtual actors may coexist in the same shared environment, i.e. participant, guided and autonomous actors [TCTP95].

A participant actor is a virtual copy of the real user or participant. His movement is exactly the same as the real user. For a complete representation of the participant actor’s body, it should have the same movements as the real participant body for more immersive interaction. This can be best achieved by using a large number of sensors to track every degree of freedom in the real body, however this is generally not possible due to limitations in number and technology of the sensing devices. Therefore, the tracked information should be connected with behavioral human animation knowledge and different motion generators in order to "interpolate" the joints of the body which are not tracked

A guided actor is an actor completely controlled in real-time by the user. In VLNET, the best example of actor guidance is guided navigation. The participant uses the input devices to update the transformation of the eye position of the virtual actor. From the incremental change of the eye position the rotation and velocity of the center of the actor body are estimated. Then, the walking motor uses the instantaneous velocity of motion, to compute the walking cycle length and time, by which it computes the necessary joint angles. These joint angles represent now the updated state of the actor, and they are diffused through the net to the other VLNET clients where they are used to update the corresponding actor which is of course shared by all clients. Thus, a remote user can see the actual updated state of the guided actor.

In VLNET an autonomous actor is an actor who may act without intervention of the user. An autonomous system is a system that is able to give to itself its proper laws, its conduct, opposite to an heteronomous systems which are driven by the outside. Guided actors are typically driven by the outside. Including autonomous actors that interact with participants increases the real-time interaction with the environment, therefore it contributes to the sense

of presence in the environment. The autonomous actors are connected to the VLNET system in the same way as human participants, and also enhance the usefulness of the environment by providing services such as replacing missing partners, providing services such as helping in navigation. As these virtual actors are not guided by the users, they should have sufficient behaviors to act autonomously to accomplish their tasks. This requires building behaviors for motion, as well as appropriate mechanisms for interaction.

4.3.2 The VLNET - L-system Interpreter Interface

In the L-system interpreter we added a shared memory interface to VLNET client processes which allows an autonomous actor of the L-system to control a "guided actor" of a VLNET client process [NPCTT96]. Code F.1 in appendix F shows some function definitions of the VLNET library, which we used to build the interface of the L-system. Besides the shared memory segment creation, there are body_set functions to guide actors by providing the head and right hand matrices at each frame. The client process will automatically compute the corresponding walk and arm motor parameters of the articulated actor.

With the body_get functions head and right hand matrices of other actors can be imported into the L-system interpreter, where they can be used according to the need of a simulation. The object_pick and object_unpick commands allow to select or drop objects of the shared environment. With the corresponding set commands picked objects can be manipulated.

If a L-system defined simulation uses VLNET clients, they have to be declared in the "Options" section of the L-system definition file. With the keyword VLNETdeclare followed by an initializing string. The BNF of this initializing string is given in definition 4.3.1.

```
VlnetInit ::=  Nbr_exported_actors int KeyId *
               Nbr_imported_actors int int*
               Nbr_objects int int*
KeyId :: = int int int          /* body key, object key, actor identifier */
```

Definition 4.3.1: The BNF of the VLNET initializing string

This initializing string contains the number of exported actors followed by the list of the shared memory keys and the actor identifier. Each actor needs two shared memory keys, one for the body and one for the objects, and an identifier. Then, the number of the imported actors and the corresponding list of the actor identifier has to follow. Finally, the number of the objects with their list of identifiers terminates the initializing string. Of course, all the shared memory keys and the other identifiers have to correspond with the corresponding VLNET definitions. To simplify the start of a VLNET simulation the command file vln.cmd (see appendix F) is executed at the VLNET declaration. This command file in general starts the VLNET server and all the clients exported by the L-system.

As the L-system presently only imports SM surfaces, only this type of objects can be shared with other VLNET clients. Other types of objects supported by VLNET, however, can be associated to L-system internal objects.

Through the symbol **D** the interface functions of the shared memory (see code F.1) can be accessed directly from the L-system definition file. With these functions objects and actors of VLNET can be manipulated or mapped into the virtual environment of the L-system interpreter. Thus, objects and actors of the shared virtual environment can be controlled by production rules. In chapter 7 we discuss as typical example a networked tennis game with a user and an autonomous actor as tennis players, and another autonomous actor as referee judging the game.

5. Behaviors

In the virtual environment modeling it still remains the behavior modeling of the actors (see figure 5), which is the topic of this chapter. The behaviors make the actors autonomous. The behaviors are strongly influenced by the sensory information they get at each time step through their synthetic sensors.

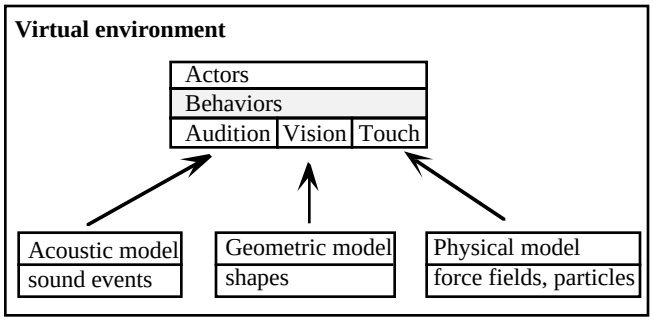


Figure 5: The actor behaviors in the virtual environment

As already mentioned in the introduction, the L-system interpreter integrates three different methods of behavioral animation definition, i.e. animation by force fields, by production rules and by finite state automata. This variety is entirely in the sense of a systemic approach, which opts for variety and warns for oversimplification.

We focus first on the behavior control by automata which is the most flexible of the three approaches as it can be freely programmed by specialized procedures. Note, however, that these automata can not be modified by a user of the animation system as they are compiled behaviors. A user can only adapt force field based and production rule based behaviors to his needs by changing the corresponding L-system definition files before starting a simulation. Behavior control by automata is used in general for complex and specialized behaviors or actions which are employed in higher level scripts.

Generally, a behavior can be composed of other behaviors and basic actions. A basic action is in general a motor procedure allowing an actor to move. A high level behavior employs in general sensorial input and special knowledge. To model behaviors we use a non parallel finite state automata approach. Each actor has an internal state which changes each time step according to the currently active automata and its sensorial input. The next sections illustrate in more detail our behavior control model and some specific automata. Finally we present behavior modeling by production rules, and how an actor can switch between the different control methods.

5.1 The Behavior Control

To control the global behavior of an actor we use a stack of command strings (see also section 6.3.4) which are interpreted by a special command string parser acting on the state variable of the actors. The fact of using a stack based control system allows an actor to get a certain independence by creating its own subgoals while executing the original script. We opted for interpreted command strings instead of a specialized data structure, because they enable us to plan a section in the L-system definition file where a user can simply define an initial script for each actor. Moreover, command string represent a simple interface for future extensions with automatic script generation or possibly concurrent lisp processes with artificial intelligence for autonomous real time actors.

At the beginning of the animation the user provides a sequence of behaviors and pushes the corresponding automata command string on the actor's stack. When the current automata ends, the animation system pops the next automata from the stack and executes it. This process is repeated until the actor's automata stack is empty. Some of the automata use this stack too, in order to reach subgoals by pushing itself with the current state on the stack and switching to the new automata allowing it to reach its subgoal. When this new automata has finished, it pops the old interrupted automata which continues its execution.

At this level the behavior control is sequential, and it reflects the fact that in general an actor executes "consciously" the current behavior (automata). In reality, humans also can concentrate consciously only on one thing at a time. Parallel actions, as tracking a moving object by the vision system and listening to sounds, for example, have to be treated in the corresponding automata procedures.

Through a special automata procedure, named L-system (see table 5.1.1), the actor control can be passed to production rules. How production rules can define behavior is explained later in section 5.10. There exists also a special symbol (**e**(par1 = 0)) of the L-system alphabet which activates an automata specified by the user. Thus, a versatile behavior control switching between production rule control and automata control is possible.

Automata	**Description**
talk	emits some sounds to the sound event handler
listen	captures some sounds from the sound event handler
observe	observes the environment to update the vision system
walk_local	a simple automata for humanoid actors, who try to move to a local goal by using only local vision without visual memory.
watch	the vision system and the visual memory are activated. This automata is only used for tests
walk_continously	navigation and path planning based on vision and visual memory. Wen a collision is detected, the actor doesn’t stop, but it dynamically looks for a new path by taking into account its current orientation and speed. It can be used also for articulated humanoids.
referee	judges tennis games
play_tennis	plays a tennis game
play_tennis_interactively	In a tennis game the user controls one actor with the space ball.
walk_stepworthy	the automata for walking on sparse foothold locations (see section 5.9)
follow_turtle	This automata is used for humanoid actors. They follow unconditionally the current turtle of the L-system interpreter.
L-system	The control is passed to production rules of the L-system.

Table 5.1.1: The actually implemented finite state automata behaviors.

Table 5.1.1 shows the list of the actually behaviors modeled by finite state automata. This set of automata was created in order to test the usefulness of sensor based actor behaviors in virtual environments and to create a basic set of behaviors which can be also called by other new behaviors to be developed. These behaviors can be scripted by the user and they can control most of the virtual actors. Some of the behaviors like the "observe" automata, for example, are simple procedures, others like the "play_tennis" represent complex roles for autonomous actors capable to play a virtual tennis match. Most of these by automata predefined behaviors are explained in more detail in the following sections.

5.2 Global Navigation

Vision based adaptive global navigation and collision avoidance for actors is a complex problem and can not be solved by employing only local vision. Humans use their visual memory to plan paths to given goals. They use their local vision to avoid collisions and to optimize locally their planned trajectories. If the environment has changed, the visual memory is updated with the actually perceived environment. We modeled all these features for global navigation and realized it by the automata "walk_continously". The visual memory, modeled by an octree space grid, is already described in sections 3.1.3 and 6.3.1. It is used for global path searching as described in the next section.

5.2.1 Path Searching

There exists extensive literature about backtracking path searching algorithms in graphs. We adapted basic graph algorithms from [GON85] to our needs by interpreting the voxelized environment of the visual memory of an actor directly as a graph with nodes linked by edges and apply certain graph search algorithms to find an obstacle free path from the current position to a given destination. Each free voxel of the octree space grid, which is not occupied by an obstacle, is interpreted as a node of the graph. All the neighbor voxels are considered to be connected by an edge. Therefore, the octree space grid represents a graph with nodes and edges. The algorithm of path searching uses the principle of backtracking and memorizes all tested nodes in a sorted list. With this list of already tested nodes circuits can be avoided and situations without a path from a given source to a given destination can be detected. In a first approach, a path is represented by a sequence of free nodes. To avoid a combinatorial explosion of the possibilities in graph searching, it is recommended to use a heuristic depth first search. We distinguish between three types of heuristics.

The first type of heuristic is characterized by the choice of the neighbors of the current voxel. For example, if we know, that the search is done in a plane (2D), we will only examine the neighbors in that plane. Therefore, we can reduce the numbers of new voxels to be tested from 26 to 8.

The second type of heuristic is determined by the order of the new neighbors to be tested. If we are searching a path from the current position to an aim, we will sort the list of the new neighbors according to their distance to the aim and we will continue the depth first search with the nearest neighbor to the aim.

The third type of heuristic is determined by some additional conditions on the new neighboring voxels to be examined. If we want, for example, that the actor is bound to the ground (if it cannot fly) in a 3D environment with stairs, mounting ramps, bridges or holes, we can use only neighbors, above an occupied voxel. With these simple condition, the actor is now able to avoid holes, to use bridges and to mount and descend stairs or ramps.

Our current path finding procedure is divided in 3 steps. In the first step (algorithm 1) a path from the point "start" to the aim "end" is searched by a heuristic depth first search. In general, this algorithm will find a path according to the heuristic used, or detect that there is no path.

Algorithm 1: (path searching from the voxel start to the voxel end)

```
path_found = FALSE
if (start == end)
    path_found = TRUE
else
    add_path(start)
    add_candidates_to_candList(heuristic)
    while ((not path_found) and (candList not empty))
        get_next_candidate
        if (candList not empty)
                add_path(pop(candList))
```

```
        if (last(pathList) == end)
                path_found = TRUE
        add_candidate_to_candList
```

add_candidates_to_candList

```
    X = get_neighbors(last(path))
    sort_neighbors(X)
    add label "new_neighbors" to candList
    add all free and still not used neighbors (which are not
        in the already_treatedList) from X to candList and to
    already_treatedList.
```

get_next_candidate

Positioning of the last pointer of the candList to the next candidate to test. Eliminates the current voxel of the pathList, if all its neighbors are tested without success in using the label "new_neighbours" in the candList.

Very often, this path will not be optimized and it will not always correspond to a path a human actor would use. Sometimes it happens, that a whole region is scanned, instead of using a direct way. This happens, when the actor has to go back to find a path to a destination in front of it as illustrated in figure 5.2.1.

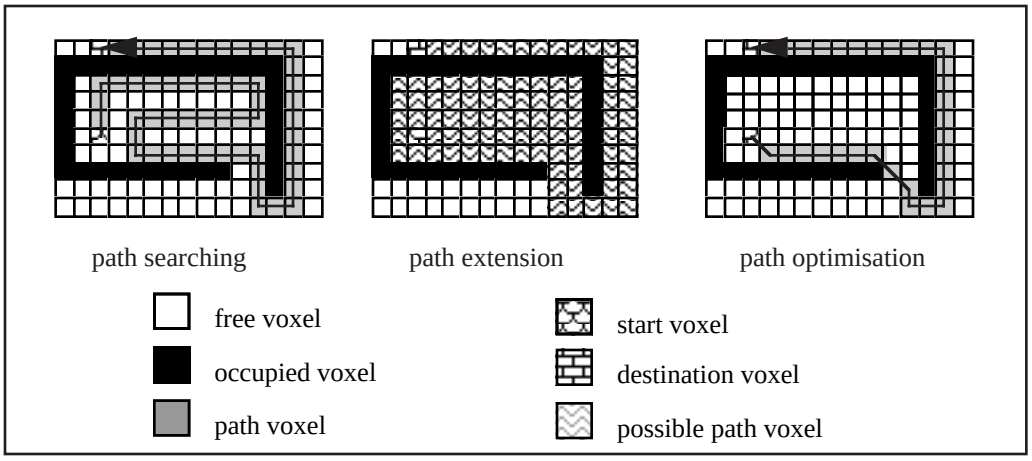


Figure 5.2.1: The three steps of path finding.

In a second step this path is extended. You can create a new octree containing only the voxels of the path and their neighbors according to a given heuristic. This extended path octree (pathOctree) corresponds to the topology of a real road (2D) or channel (3D), where the actor can move without collisions.

Algorithm 2: (path extension according to an heuristic)

```
    Do for all voxels of the pathList
        insert the current voxel into the pathOctree and mark
            it with MAX_LABEL
        add all neighboring voxels according to the heuristic
            to the pathOctree and mark them with MAX_LABEL
```

You can interpret this extended path octree as a second type of inverted graph, where the occupied voxels represent nodes and the (occupied) neighbors are connected by edges. The size of this graph is limited and so we can start a search of the shortest path without having to fear a combinatorial explosion. This final optimization corresponds to the third step in path finding.

Algorithm 3: (shortest path labeling in the pathOctree)

We used an algorithm based on the algorithm de Moore-Dijkstra. It's a search of the shortest path of a voxel "start" to all the other voxels in the pathOctree with positive arc length. Supposing that all arc lengths are 1, we can use a simplified algorithm [GON85]. As a result we get a pathOctree with all nodes labeled with the minimum distance from the start voxel.

From this labeled pathOctree we can easily get a shortest path by a standard algorithm like the following:

Algorithm 4: (shortest path search in marked graph)

```
x = end
S = neighbors(x)
push_path(x)
while (x!= start)
    x = min_label(S)
    push_path(x)
    S = neighbors(x)
```

The extension of a given path produces short circuits of scanning paths and the shortest path search yields a path corresponding to an intelligent human behavior.

5.2.2 Path Exploring

The path searching procedure described above, is a mental process of the actor, which is based on the contents of its visual memory (octree). This means, that during its reasoning on a possible path, it doesn't move. But very often it is placed in an unknown environment, it still hasn't seen and memorized. In this case, it cannot find a path using, for example, some conditional heuristic. Therefore, it is forced to explore its environment guided by its vision and a heuristic. This exploring is an active process and the actor has to walk and memorize what it sees. In this case, the heuristic depth first search (algorithm 1) step can be used to guide the actor to guarantee that it finds a path if there exists one.

If the actor, for example, is in a house with closed doors, there will be no path to a destination outside the house. In this case, it will explore the interior of the whole house which is accessible to it according to its actual heuristic. It will finish its search by having memorized the interior of the house and the statement that it is enclosed. By using the algorithm 1 an actor can avoid turning for ever in a circuit and it can recognize that it has checked all possibilities according to its actual heuristic to find an exit. In this exploring mode the actor executes the reasoning process of the first algorithm in its real environment. This exploring mode, however, is very systematic and each free voxel is visited. It doesn't correspond to a typical human behavior. Therefore, it should not be used for humanoid simulations.

5.2.3 Heuristics

To define the different heuristics in the octree search space, we introduced an arbitrary numeration of the neighbors of a voxel as illustrated in figure 5.2.2. We suppose a horizontal plane to be parallel to the x, z plane and the y axes pointing upwards. A heuristic for the path searching algorithm is defined by a list of numbers of neighboring voxels which are susceptible to be added to the candidate list of the backtracking search algorithm. We can associate a set of conditions to each neighbor voxel. Only when these conditions are satisfied, the voxel is added to the candidate list. The simplest condition is the necessity of a voxel to be empty. Additionally, we can demand for a non flying actor with a certain height, for example, that the voxel below is occupied, and that the voxel above is empty .

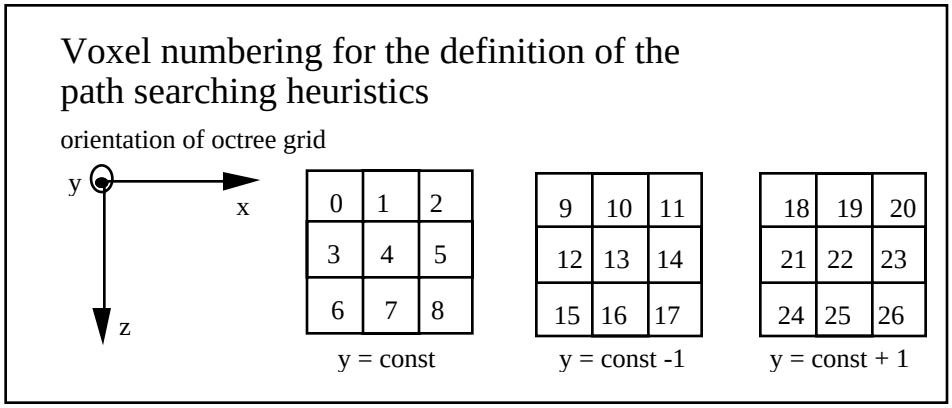


Figure 5.2.2: Numbering of the neighbors of voxel 4.

In appendix G we present the currently available voxel based heuristics we can choose for path searching in two or three dimensions.

5.2.4 The Navigation Automata

Figure 5.2.3 shows the navigation automata based on the octree space grid, the path searching algorithms and voxel based heuristics. It allows an actor to find a collision free path from the current position to any position attainable according to the chosen heuristics.

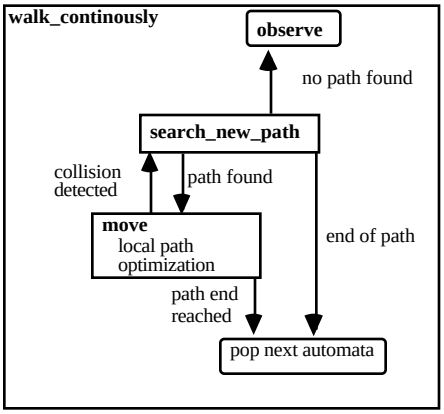


Figure 5.2.3: Global navigation automata of an actor.

Each actor disposes of a goal stack. This goal stack enables a user to script several destinations for the actor by maintaining the possibility for the actor to choose sub-goals during the execution of the original scripts. The aim of this automata is to allow an actor to find a path from its current position to a destination represented by the top goal of the goal stack. The navigation should work in known and unknown environments and it should avoid collisions with static and dynamic obstacles. Moreover, it should make a local vision based optimization of the planned path. The automata "walk_continously", which is supposed to do this, is composed of the two states "search_new_path" and "move". When it starts the first time in the state "search_new_path", it pops a goal from the goal stack. If the goal stack is empty or if the actor is already at this goal the next behavior is popped from the behavior stack. Otherwise a path is searched based on the actual state of the visual memory and the concerned state variables. If no path is found, the automata pushes itself on the stack and switches to the "observe" automata which is described in section 5.3. When a path is found, the automata produces a path spline curve as illustrated in figure 5.2.4 and switches into the "move" state where the actor moves along its actual path. When it reaches the path end, the next automata is popped from the behavior stack. If, however, during the navigation an

unexpected obstacle is perceived on the visible path track, the automata switches to the "search_new_path" state where a new path is looked for to the actual goal.

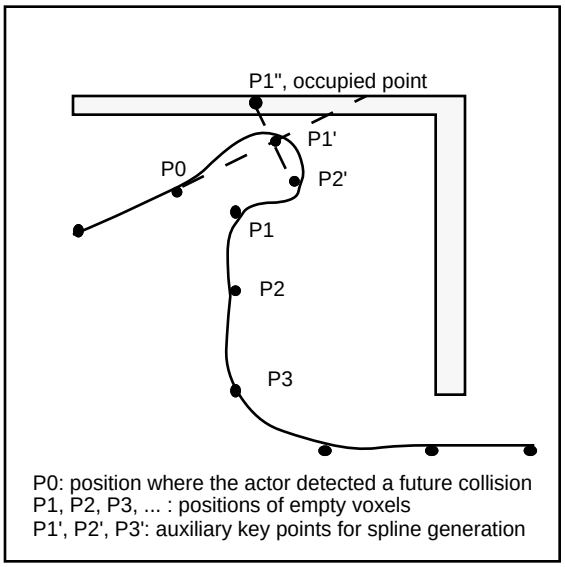


Figure 5.2.4: Path spline generation

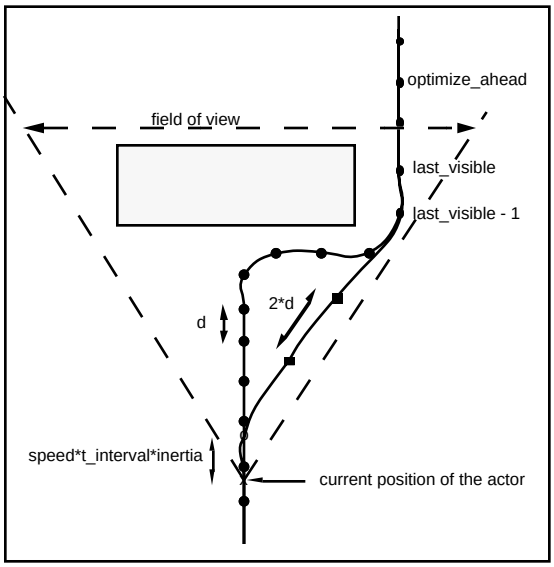


Figure 5.2.5: Local path optimization

In the "move" state, the automata performs periodically a local path optimization of the actual visible part of the path. As illustrated in figure 5.2.5, a certain number of path points are projected into the image of the actual vision window. If these path points are visible and if they are at a certain distance from visible obstacles, the actor creates a new path directly to the last visible path point. The algorithm of this local path optimization based on the actual vision image is illustrated in algorithm 5.

```
For the first n path key points P_i {
   render P_i into the vision image -> P_pixel, i, j
   if P_pixel, i, j is visible {
      for all pixel p_i,k (k = 0, ..., vision window row length) {
         if p_i,k is closer to the actor than P_pixel, i, j {
            calculate the actor size in pixels (n_actor) at the pixel distance of p_i,k
            if abs(k-j) < n_actor   return not visible
            otherwise               return visible
         }
         otherwise return visible
      }
      for all pixel p_k,j (k = 0, ..., vision window row length) {
         if p_k,j is closer to the actor than P_pixel, i, j {
            calculate the actor size in pixels (n_actor) at the pixel distance of p_k,j
            if abs(k-i) < n_actor   return not visible
            otherwise               return visible
         }
         otherwise return visible
      }
   }
   otherwise       return not visible
}
```

Algorithm 5: The local path optimization algorithm for detecting the last visible path point.

This automata, of course, is not perfect. There is no guarantee that it finds the best existing path to a destination because the perception of the environment via synthetic vision is limited and not precise. Even collisions of two actors walking in the opposite direction around an edge can not be avoided. But such a behavior is natural for humanoids as real humans commit sometimes the same errors.

5.3 Observing

At the beginning of an animation or in dynamic environments there is often a need for an actor to update the representation of its local environment by its synthetic vision as several other behaviors relate on the actual state of the visual memory. Therefore, we introduced an "observe" automata (see figure 5.3.1) which looks at the local environment and memorizes it. The vision system turns around by 360 degrees, and thus, the visual memory is updated and visible color coded objects can be found. After turning around once, the next automata is popped from the stack.

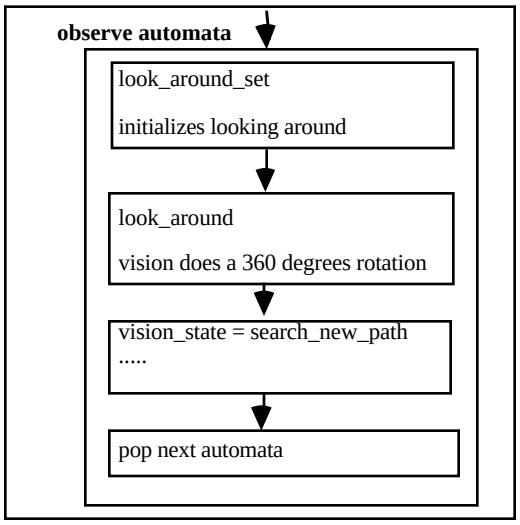


Figure 5.3.1: The observe automata

This automata is also invoked by the "walk_continously" automata described above in case where the actor can not find a path to its destination. Very often a local update of the environment improves the situation, especially, if the environment has changed.

5.4 Talking

Talking is an important feature of humanoids, and some other high level behaviors such as the "referee" automata, for example, which is described later in section 5.7, use extensively this feature. Once a behavior has determined what to say, the talking automata manages the talking procedure by playing sequentially the corresponding sounds and by putting the sound events into the virtual acoustic environment where they are perceivable by other actors. The actor first puts n words into the actor's sound event table "words_to_speak", pushes itself on the stack and switches to the talk automata. The talk automata transmits sequentially these events to the sound event handler by taking into account the duration of each word. When it

has transmitted the last event, it clears the table and pops the next automata from the stack as illustrated in figure 5.4.1.

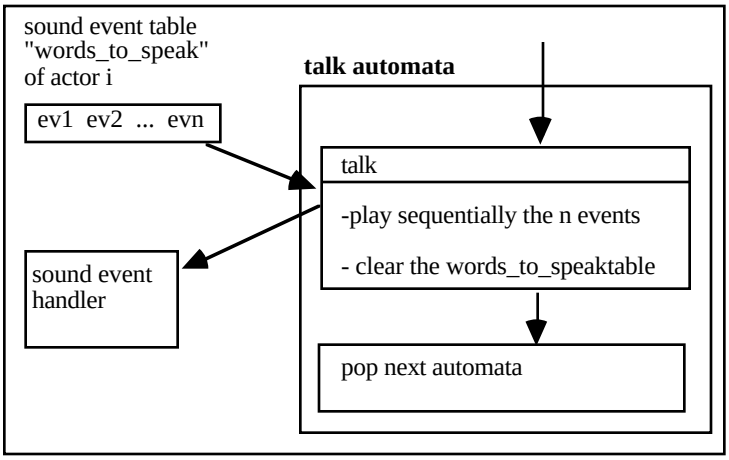


Figure 5.4.1: The "talk" Automata

Actually an actor can only play back prerecorded isolated words (see appendix D). An extension of the animation system with a speech synthesizer would be an interesting future project.

5.5 Listening

Besides talking, listening is another typical humanoid feature for communication. Like talking, listening is a temporal process and there are behaviors which have at a given moment to capture several sequential sound events. The "play_tennis" automata, for example, described in the next section 5.6, has to capture the sound events emitted by the "referee" (see section 5.7) actor in order to decide how to continue the tennis game.

The listening automata illustrated in figure 5.5.1 permits an actor to memorize the next n sound events emitted from this source. It is, for example, used by the "play_tennis" automata of the synthetic tennis players described below when it listens to the referee. To simplify the capture of spoken words we suppose that an actor emits only one sound event at a time. After having captured the n events, the listening automata pops the next automata from the stack.

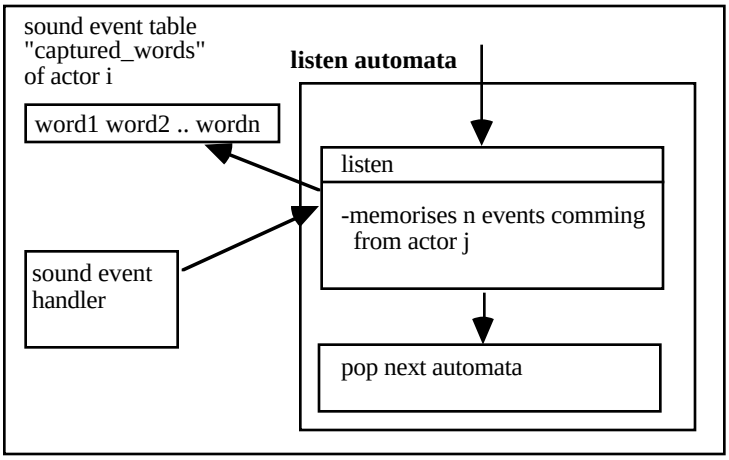


Figure 5.5.1: The "listen" automata.

This automata implies that any other automata which uses it has to know how many sound events of a given actor have to be captured. Also the interpretation of the sound is left to the calling automata.

5.6 Tennis Players

In order to apply the concept of sensor based animation and to show its usefulness we decided to model the complex behavior of a sensor based autonomous tennis player, capable to play against another autonomous player or an interactive user. Whereas behaviors like "talking", listening" or "observe" are implemented with relative simple automata procedures, the "play_tennis" behavior represents already a complex role for an actor, modeled by a sophisticated automata procedure, but which is integrated in the behavior control system in the same way as any other automata. In a user defined behavior script, for example, a tennis game can be included simply by adding the corresponding command line in the "Options" section of a L-system definition file.

At the beginning of this project, the humanoid actors didn’t offer enough flexibility to model the players based on skeleton structures. Therefore, we decided to model the players, first by simple rackets with synthetic vision, and later, by adding interfaces for humanoid actors. An example is given in the description of the VLNET interface of section 4.3. The different behaviors of the actors are mostly modeled by automata controlled by the stack based control system. As the behaviors are based on synthetic sensors being the main channels of information capture from the virtual environment, we obtain a natural behavior, which is mostly independent of the environment representation. By using this sensor based concept the distinction between a synthetic actor and an interactive user merged into the virtual world gets small, and synthetic actors can easily be exchanged as demonstrated by the interactive game facility.

The evolution of a tennis game can be described by a Petri net. A petri net is composed of states and transitions and can be represented by a graph where the arcs link the states and the transitions. Figure 5.6.1 shows the graph of the tennis game's Petri network. At the beginning the player is in an inactive state. Through the transition t1 triggered, for example, through an external event, it switches to the state "at_start" by navigating to its start position. When it arrives there, it passes through transition t2 to the state "look_for_partner" where it waits until its partner has arrived at its own start point. It verifies this of course through its synthetic vision. Then, it passes through transition t3 to the state "follow_ball" where it looks for the ball. When it has seen the ball, it tracks the ball with its vision and estimates speed and position of the ball.

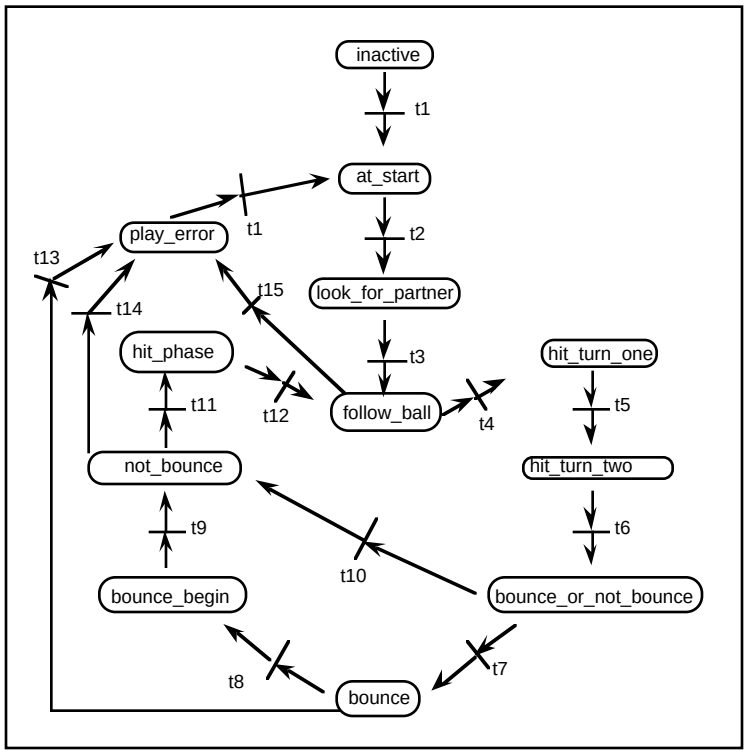


Figure 5.6.1: Petri Net of actor states of a tennis game

Section 3.1.2 describes how the vision system recognizes the ball in the vision image and how it estimates its position at a given frame. In order to estimate its velocity each actor maintains the positions of the last n (=3) frames. The actor derives the velocity of the ball from the memorized positions and the frame to frame time increment. If the actor sees the ball flying towards it, it switches through transition t4 to the state "hit_turn_one". As the velocity estimation of the ball is still not good enough, it waits some time by passing through the intermediate state "hit_turn_two" to the state "bounce_or_not_bounce", where it has to decide whether to let the ball bounce or not. According to its decision it switches to the state "bounce" or "not_bounce". In the state "bounce", where the ball is close to the ground, it passes to the state "bounce_begin". When the ball has collided and starts mounting, it advances to the state not_bounce. If it is close to the racket ball impact point, it enters the state "hit_phase", where it strikes the ball according to the game strategy described in section 5.6.4. After the stroke the player switches to the "follow_ball" state, and a new cycle can begin.

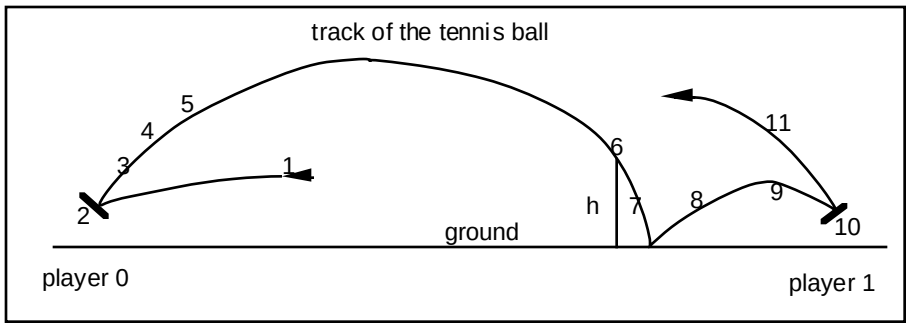


Figure 5.6.2: The tennis ball track.

Figure 5.6.2 illustrates the different states along the ball trajectory. The numbers one to eleven indicate consecutive events important for player 1 on the right figure side. At event 1

the player 1 is in the state "follow_ball", and its vision system focuses the ball. Some time later player 0 hits the ball (event 2), and player 1 sees the ball flying back. Then, it passes through the events 3 ("hit_turn_one") and 4 ("hit_turn_two") into the state "bounce_or_not_bounce", where it decides whether to let bounce the ball or not. Between the events 5 and 7 it is in the state "bounce". Event 6 indicates the ball-racket impact position in case where it lets not bounce the ball. At event 7 it enters the state "bounce_begin". After the collision with the ground when the ball starts mounting (event 8), the player 1 enters consecutively the states "not_bounce" and "hit_phase" indicated be the events 8 and 9. These last two states are equivalent to the case where the actor decides not to let bounce the ball. The actor could enter the state "not_bounce" directly after event 5. Event 10 indicates the hit, and event 11 the state "follow_ball". Now, player one goes back to its start position while continuing to focus the ball. There, it waits until the ball is played back, and the whole cycle is repeated.

5.6.1 Game Automata of an Actor

During a game the "play_tennis" automata has to control the actor's vision system, the audition system and its internal state of the game. Each of the three systems has its own state variables, which can be manipulated by all three systems according to their needs. The audition system control illustrated in figure 5.6.3 checks at each frame the sound events. If the automata detects the sound event "fault" emitted by the referee as described in section 5.7, it pushes the actual "play_tennis" automata on the stack together with the necessary initializations allowing a new game. Then, it activates the "walk_continously" automata in order to move the actor to its start position on the tennis court.

If the detected sound event is "finished", the whole game is finished, and the actor pops the next automata from the stack.

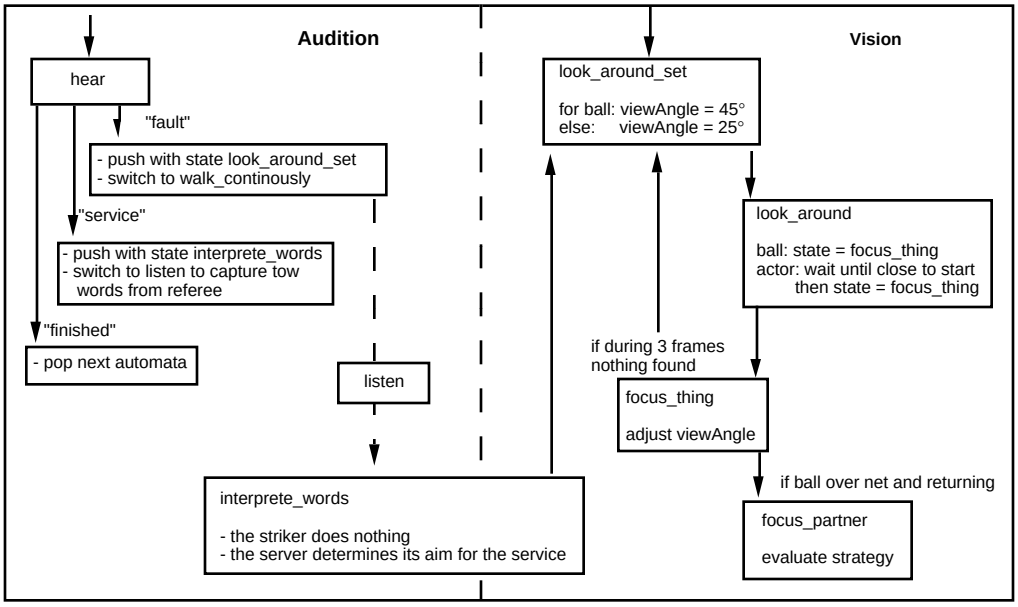


Figure 5.6.3: The "play_tennis" automata.

If it detects the sound event "service" coming from the referee, it pushes the "play_tennis" automata on the stack with an initial state "interprete_words" and it switches to the "listen" automata, which will capture the next two words coming from the referee. When the "listen" automata has captured the two words, it pops the next automata that is of course the "play_tennis" one with the initial state "interprete_words". These two words are the name of the server and the server position that can be “left” or “right”. The actor recognizes whether it is the server or not. If it is the server, it fixes the correct aim point for its service and switches to the "look_around_set" state, and a new game will begin.

Each actor has a state variable designing its object of interest it is looking for with its vision system. In the state "look_around_set" it initializes its view angle according to the size of its object of interest which can be the ball or its game partner and it focuses its eyes on the start position of its opponent. Then, it switches to the state "look_around". If its object of interest is the ball, it switches to the state "focus_thing" when the ball enters its view. If the object of interest, however, is the other player, it waits until it is close to the start point. Only then, it switches to the state "focus_thing". In this state it controls several features. As the name of the state indicates, one task is to focus the vision system on the object of interest and to adjust the view angle. It tries to maintain the number of pixels of the object between 5 and 12 by multiplying it by a factor 1.5 or 1/1.5 respectively. The adjustment of the view angle is necessary as the vision window generally is small (30x30 pixels), and small objects like the ball would disappear at long distances. When the object of interest or the actor move fast, it can happen that the vision system looses it. In this case the state is set to "look_around_set", where the actor starts again to look for the object of interest.

If the actor estimates the impact point to be outside its court during the impact point estimation, it decides to let the ball and to move directly back to its start position waiting there for a new game. Therefore, it pushes the "play_tennis" automata on the stack together with the initializations for a new game and activates the "walk_continously" automata with its start point as goal. If the vision state is "focus_thing" and the game state "look_for_partner", the state is set back to "look_around_set", and the object of interest becomes the ball. When the object of interest is the ball and when the vision state is still "focus_thing", the striker tries to determine its opponent's position to use it for its strategy. That's why it sets its vision state to "focus_partner" when the ball flies over the net. Before, however, the actor selects its opponent as object of interest.

If the vision state is "focus_partner", the actor evaluates the aim point of its next stroke according to its strategy and the actual position of the opponent. Then, it switches back to the "focus_thing" state after having selected the ball as object of interest.

5.6.2 Impact Point Estimation and Learning

During the game each actor should estimate the future impact point immediately after the stroke of its opponent. We designed the actor with the capability to learn from experience and to adapt itself to changing conditions as different masses of the ball, changing air resistance, changing precision of the numerical integration characteristics of the racket and time varying wind force fields. At each frame the actor extracts the position of the ball from its vision image (see section 3.1.2). Every n-th (n=3) frame it derives the velocity of the ball according to equation

$$\mathbf{v}(t) = (\mathbf{P}(t) - \mathbf{P}(t - n * t\_interval)) / n / t\_interval \qquad (5.6.1)$$

It can calculate the future impact point and impact employing this current velocity and the current position of the ball . We suppose that the actor wants to hit the ball at a certain height h. As the vertical velocity component of the ball is in general not too big, we can neglect air damping and assume the ball moving in vertical direction according to the next equation.

$$\begin{aligned} & m \cdot \dot{v} = -m \cdot g \\ & \Rightarrow x(t) = x_0 + v_0 \cdot t - 0.5 \cdot g \cdot t^2 \end{aligned} \qquad (5.6.2)$$

with m: mass of the ball
g: gravity acceleration
v_0: initial vertical speed of the ball
t:: time
x_0: initial vertical offset

The impact time at height h can easily be calculated from the quadratic equation. To estimate the horizontal x and z components of the impact position, we could use the impact time and the solution of the differential equation of the ball movement with linear air resistance given in the following equation .

$$m \cdot \dot{v} = -b \cdot v$$
$$\Rightarrow x(t) = v_0 \cdot \frac{m}{b}\left(1 - e^{-\frac{b}{m} \cdot t}\right) \qquad (5.6.3)$$

with *b*: air damping
m: mass of the ball
v_0: initial speed of the ball
t: time

In this equation, however, the air damping and the mass of the ball are constants, and this would not be sufficient for modeling changing conditions. To be independent of a particular solution we chose a heuristic quadratic function where some coefficients are adjusted during the game according to the actors experience. This heuristic function should be linear at short distances and reduce the extrapolated distance d(t) of equation (5.6.4) at long distances. We chose the distance function shown in equation (5.6.5). The future impact time t_i and the initial velocity v(t) are known at a given time t. The variables x and o model the function's linearity at short distances and the reduction of the extrapolated distance at long distances. Figure 5.6.4 shows this function with the values x = 2 and o = 10000.

$$d(t) = v(t)(t_i - t) \qquad (5.6.4)$$

$$dist(t) = d(t)*(d(t)^2 + o) / (x * d(t)^2 + o) \qquad (5.6.5)$$

d(t): extrapolated distance
ti: future impact time
t: actual time
dist(t): corrected distance
x, o: modeling variables

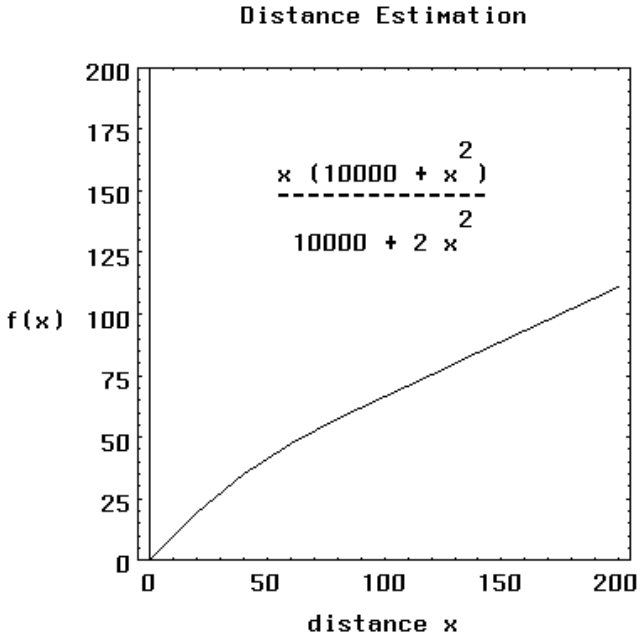


Figure 5.6.4: The heuristic function correcting the extrapolated distance.

Each actor estimates the future impact point every n-th (n=4) frame when it is in a state between "bounce_or_not_bounce" and "hit_phase". If the ball is close enough to the actor (<1 meter) the estimation can be stopped as the impact point is precise enough, and the actor enters the "hit_phase". Thus, at long distances the impact point estimation is rough and it becomes sufficiently exact a short distances to execute the stroke. The variable o = 10000 is constant, and the actor adjusts the value of x during the game. This variable x permits to improve the estimation of the impact point at long distances. The factor x is determined by the system of the next equations.

$$x = (d(d^2 + o) / dist - o) / d^2$$

$$d = v_x(t_1) * (t_2 - t_1)$$

$$dist = \mathbf{P_2}(t_2) - \mathbf{P_1}(t_1) \qquad (5.6.6)$$

During the game when the actor enters the "bounce_or_not_bounce" state, it memorizes at time t_1 the speed v and the position $\mathbf{P_1}$ of the ball. At time t2 when it enters the state "hit_phase" or "bounce_begin" it memorizes the position $\mathbf{P_2}(t_2)$ of the ball. Now it can determine a new value of x. As final value of x, it calculates the average value of two consecutive values. Thus, each actor adapts itself to the current conditions, which is useful if a wind, for example, disturbs the ball movement at long distances.

5.6.3 Impact Point Determination

During the game the striker has to decide whether to let bounce the ball or not after its opponent returned it. Theoretically there exist two possible impact points. The first one, $\mathbf{P_1}$ is estimated according to the previous section before bouncing. The second one, $\mathbf{P_2}$, after bouncing, can be approximated according to equation 5.6.7 and figure 5.6.5. The ball velocity at $\mathbf{P_1}$ is approximated with the actual ball speed. This value is of course too big, but as only a rough approximation is needed at long distances it doesn't matter and at short distances the approximation is good enough.

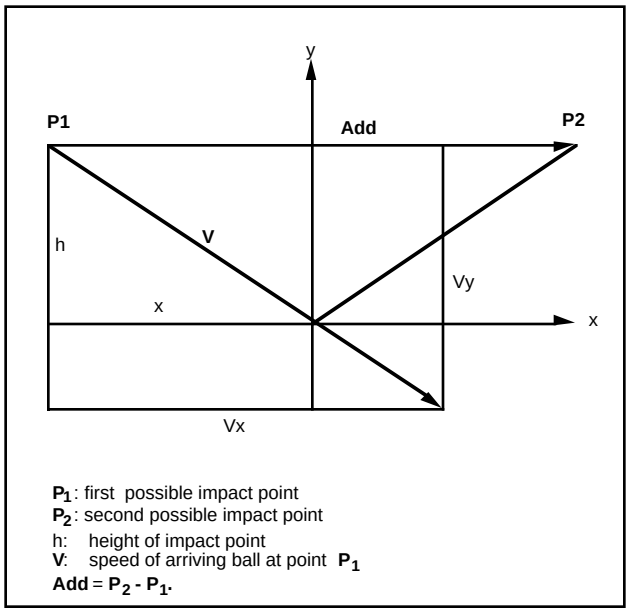


Figure 5.6.5: Estimation of the second impact point $\mathbf{P_2}$.

Now the actor can choose as future impact point the one which is closest to its actual position and according to its decision it enters the "bounce or not_bounce" state.

$$\vec{P}_2 = \vec{P}_1 + a \cdot \begin{bmatrix} \frac{V_x}{|V_y|} h \\ 0 \\ \frac{V_z}{|V_y|} h \end{bmatrix} \qquad (5.6.7)$$

$$a = 2$$

5.6.4 Game Strategy and Stroke Planning

The actors perform a simple game strategy. When the ball flies over the net, the striker glances during one frame to its opponents side. If it can localize it, it chooses as aim point for its next stroke the from its opponent most distant corner of the court area. Now the striker has to determine the speed and the direction of its racket at the moment of the stroke. This situation is illustrated in figure 5.6.6. **v** is the incoming ball speed and **c** the racket velocity at collision time. The racket surface is always perpendicular to its velocity. As the racket moves along a spline, its mass can be supposed to be infinite and the ball reflection becomes purely geometrical. The vector **a** is the wished velocity of the ball after the collision in order to hit the aimed point. This point determines the horizontal direction of the vector **a**. To simplify the strategy we let the vertical inclination of vector **a** constant. Thus, we can estimate the speed of the resulting ball velocity with a heuristic distance dependent function. The direction of the velocity **a** is easily determined from the estimated impact point, the aim point and the vertical inclination of the velocity. The speed is approximated by a heuristic formula shown in equation 5.6.8.

$$\|\vec{a}\| = c1 + c2 \cdot \left\| \vec{p}_{impact} - \vec{p}_{aim} \right\| \qquad (5.6.8)$$

which is proportional to the horizontal distance between the impact point and the aim point. The coefficient c1 is an appropriate constant and the coefficient c2 is adjusted by the actor

itself according to the comparison of the planned aim point and the effective distance traveled by the ball. After the stroke the actor memorizes its impact point and the horizontal distance to the aim point. When it "hears" the ball_racket collision sound of its opponent at a height lower than 1m or the "ball_ground" collision sound, it adjusts the coefficient c2 according to the effective deviation of the effective aim point it extracts from its vision system.

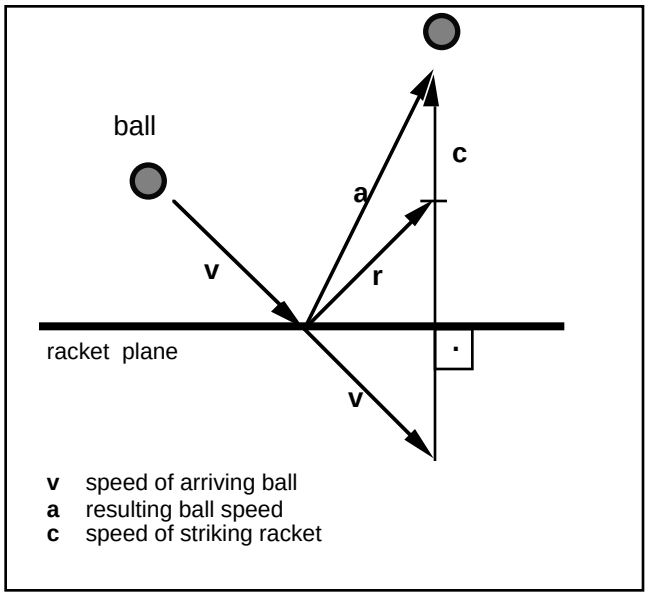


Figure 5.6.6: Ball-racket collision.

Figure 5.6.6 summarizes the geometrical situation of the racket-ball collision. The vector **a** is the wished resulting ball velocity after the hit. The vector **v** is the arriving ball velocity and **r** its reflection vector. In the next section we show how we determine the necessary racket speed **c** by some geometrical considerations.

5.6.5 Hit Determination

According to figure 5.6.6 the velocity **c** of the racket necessary to make the ball leaving with velocity **a** after the collision is determined by the following equations.

$$\mathbf{c} = x^*(\mathbf{a} - \mathbf{v}) \qquad (5.6.9)$$

$$\mathbf{r} = \mathbf{a} - \mathbf{c} \qquad (5.6.10)$$

$$\Rightarrow \mathbf{r} = \mathbf{a} - x^*(\mathbf{a} - \mathbf{v})$$

For $(\mathbf{a} - \mathbf{v}) = \mathbf{0}$ the solution is $\mathbf{c} = \mathbf{0}$, i.e. no stroke is necessary.
For $\mathbf{a} = \mathbf{r}$ it follows $\mathbf{c} = \mathbf{0}$. In this case the racket normal **n** is given by

$$\mathbf{n} = (\mathbf{a} - \mathbf{v}) / |\mathbf{a} - \mathbf{v}| \qquad (5.6.11)$$

For $\mathbf{a} = -\mathbf{v}$ it follows $\mathbf{c} = \mathbf{0}$ and

$$\mathbf{n} = -\mathbf{v} / |\mathbf{v}| \qquad (5.6.12)$$

If the angle between **v** and $(\mathbf{a} - \mathbf{v})$ is between -Pi/2 et +Pi/2, there is a collision like a reflection according to the following equations

$$\mathbf{r}^2 = \mathbf{v}^2$$

$$\Rightarrow \mathbf{r}^2 = \mathbf{a}^2 + x^2((\mathbf{a}-\mathbf{v})^2 - 2x\mathbf{a}(\mathbf{a}-\mathbf{v}) = \mathbf{v}^2$$

$$\Rightarrow x^2(\mathbf{a}-\mathbf{v})^2 - x2\mathbf{a}(\mathbf{a}-\mathbf{v}) + \mathbf{a}^2 - \mathbf{v}^2 = 0$$

with

$$A = (\mathbf{a}-\mathbf{v})^2$$

$$B = -2\mathbf{a}(\mathbf{a}-\mathbf{v})$$

$$C = \mathbf{a}^2 - \mathbf{v}^2$$

we get

$$x = (-B +- \text{sqrt}(B^2 - 4AC)\ ) / (2A) \qquad (5.6.13)$$

For this quadratic equation there exist 2 solutions. One of them is always 1 when the angle between **v** and (**a** - **v**) is not between -Pi/2 et +Pi/2 (no ball reflection). The other solution is

$$x = C/A = (\mathbf{a}^2 - \mathbf{v}^{2)} / (\mathbf{a}-\mathbf{v})^2 \qquad (5.6.14)$$

which allows us to calculate the racket speed at the moment of the impact by replacing x in equation 5.6.9.

5.6.6 Path Planning

During a game the planning of the path of a racket is not a trivial task. When the racket has to strike the ball, it has to be at the impact point at a given time and to move with a given speed in a given direction in order to correctly strike the ball. In the previous sections we have seen how the impact point, the impact racket velocity and the impact time are estimated every n-th frame. From this data the actor creates now a timed 3D spline that it will follow. The racket normal is always aligned with the direction of its movement. Figure 5.6.7 illustrates this spline creation process.

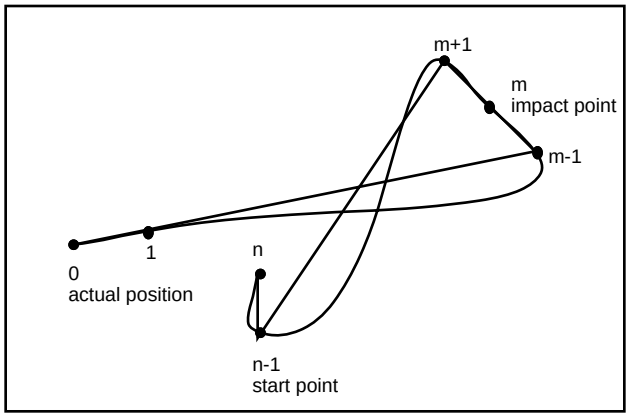


Figure 5.6.7: The planned path of the racket before, during and after the stroke.

The 3D spline has n key points. Its parameter is the time. The first key point corresponds to the racket's position $\mathbf{P_0}$ at the actual time t_0. The middle key point m = n/2 is the estimated impact point $\mathbf{P_m}$ at the estimated collision time t_m. To control well the stroke the racket should move during a certain time $t_{duration}$ in a straight line of length "hit_length" with speed

c through the impact point $\mathbf{P_m}$ in direction of the racket normal **n**. The system of equations (5.6.15) illustrates these relationships.

$$t_{duration} = hit\_length / c$$

$$\mathbf{P}_{m-1} = \mathbf{P}_m - \mathbf{n} * hit\_length$$

$$t_{m-1} = t_i - t_{duration}$$

$$\mathbf{P}_{m+1} = \mathbf{P}_m + \mathbf{n} * hit\_length$$

$$t_{m+1} = t_i + t_{duration} \qquad (5.6.15)$$

We fixed the maximum value of the hit duration at 0.1 sec. If it lasts too long, we reduce the hit length until it meets the above condition. If the speed of the racket c is 0, we set it to a small value in direction of the racket normal **n** to avoid the division by 0 in the above equation. Note that the method described here works also for velocities of the racket moving backwards to slow down the ball movement at collision.

The key points from the actual position of the actor to the beginning of the stroke are simply obtained by linear interpolation to the point m-1. To maintain continuity of the actors direction the first key point is set in direction of the current racket normal.

$$dt = (t_i - t_{duration} - t_0) / (m-2)$$

$$\mathbf{P}_1 = \mathbf{P}_0 + dt * \mathbf{actual\_speed}$$

$$\mathbf{dx} = (\mathbf{P}_{m-1} - \mathbf{P}_1) / dt$$

$$\mathbf{P}_k = \mathbf{P}_{k-1} + \mathbf{dx}, \quad k = 2, .., m-2$$

$$t_k = t_{k-1} + dt \qquad (5.6.16)$$

After the stroke the racket moves in "return_time" seconds to its start point and waits there for the return of the ball. Intermediate points are obtained by linear interpolation from the point m+1 to the start point. The last point of the spline, however, is a point in direction of the middle of the court in order to face the actor toward its opponent.

$$\mathbf{P}_{n-1} = \mathbf{P}_{start\_point}$$

$$t_{n-1} = t_i + return\_time$$

$$\mathbf{P}_n = \mathbf{P}_{n-1} + 0.05*(\mathbf{P}_{start\_point\_opponent} - \mathbf{P}_{start\_point})$$

$$dt = return\_time / m-3;$$

$$\mathbf{dx} = (\mathbf{P}_{n-1} - \mathbf{P}_{m+1}) / dt$$

$$\mathbf{P}_k = \mathbf{P}_{k-1} + \mathbf{dx}, \quad k = m+2, .., n-2$$

$$t_k = t_{k-1} + dt \qquad (5.6.17)$$

5.7 Tennis Referee

Another high level behavior for an actor is that of the role of a referee being capable to judge a virtual tennis game by using synthetic vision and audition and some internal knowledge about a tennis match. Such a behavior is an additional test case for the usefulness of sensor based actors and it integrates itself naturally in the set of behaviors which includes already virtual tennis players. The role of the referee is decomposed into two automata procedures. The first one, the "tennis_error_detection" automata, enables an actor to detect an error during a game. The second one, the "referee" automata, manages a whole tennis match. This decomposition enables the reuse of the "tennis_error_detection" automata for possible future extensions where spectators could be modeled, which watch the game and look for game faults from their perspective.

The automata "tennis_error_detection" illustrated in figure 5.7.1 permits an actor to track the ball with its vision system and to detect player faults. It is used by the referee. This automata controls the vision system and the audition system of the actor.

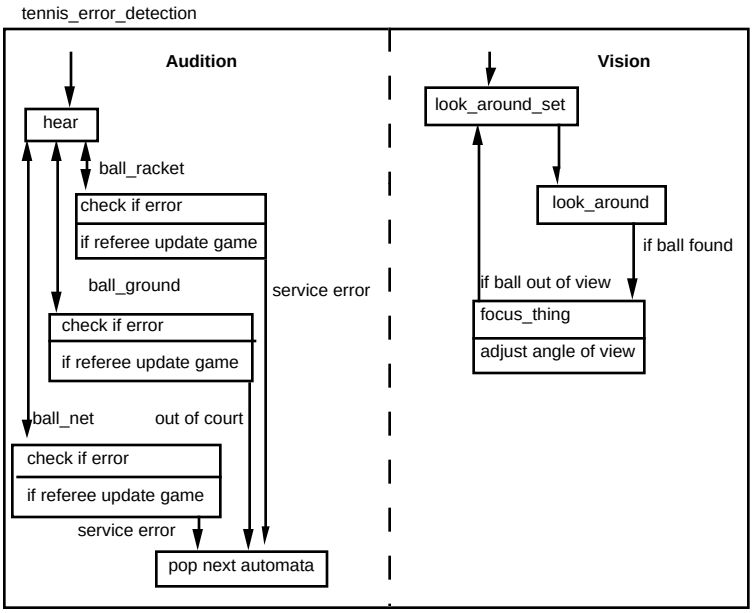


Figure 5.7.1: The "tennis_error_detection" automata

The vision system cycles through the states look_around_set, look_around and fix_thing. In the state look_around_set it sets the view angle to 40 degrees and looks for the ball in the state look_around. When it finds the ball, it switches to the state fix_thing, where it tracks the ball and adjusts the view angle (see also the play_tennis automata). When it looses the ball, it switches to the state "look_around_set". The audition part handles the sound events, controls the game, and if the actor is the referee, it updates also the state of the current game. The audition system has to treat the three collision events ball_racket, ball_ground and ball_net. The ball_racket event only produces errors if the striker doesn't let bounce the ball after the service. If the actor is the referee, the game is updated, i.e. the current player is now the next one, the number of bounces is reset to zero, and if the striker stroke the ball, the service phase is over.

The "ball_ground" sound event produces an error if the ball is out of the corresponding court side or has bounced twice. If the actor is the referee, the game is also updated, i.e. the number of bounces is incremented. The "ball_net" sound event produces an error only after a service stroke. When one of these errors occurs, the next automata is popped from the stack.

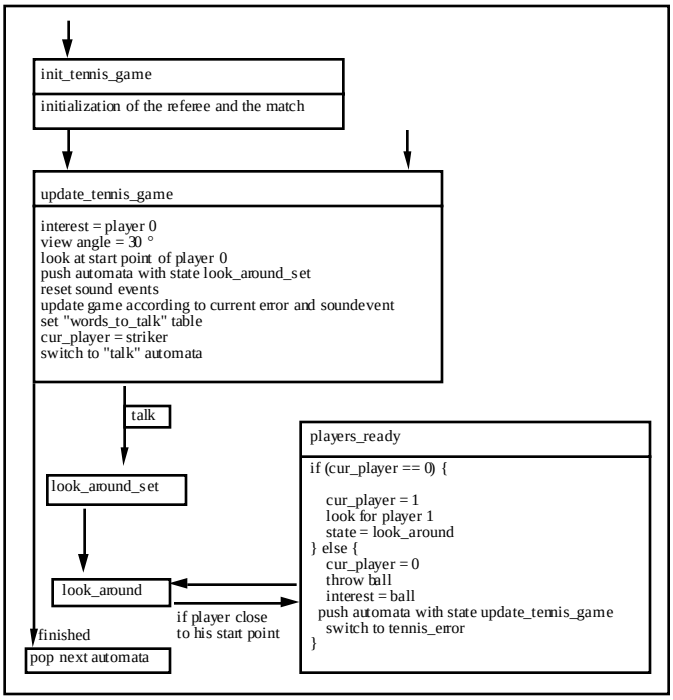


Figure 5.7.2: The "referee" automata.

The referee automata judges and updates a tennis match. The first entry point of this automata is the state init_tennis_game where the match and the referee are initialized. Then, it follows the "update_tennis_game" state. In this state the automata updates the game and the match according to a subset of the international tennis rules and the error that is responsible for the actual activation of the automata. For simplicity, we didn't implement the tie break procedure and the change ends rules. After the update the automata comments the error, announces the actual score and indicates the server the place from where it has to play. Thus, the referee puts for example the words "out, fault green, fifteen all, service blue, right" into the table "words_to_speak" (see the talk automata), pushes the automata on the stack with look_around_set as state and the player 0 as object of interest and switches to the "talk" automata that will "announce" the words in the "words_to_speak_table" to the participants of the game.

As the behavior of the referee is based on its synthetic sensors, it is of course not perfect. It can happen that the virtual referee estimates erroneously the ball position, or that it fails to track the ball by its vision system. In both cases a wrong judgment can result.

5.8 Interactive Tennis Player

If a user wants to play interactively against a synthetic actor by using a spaceball interface, he can do it by taking the role of one player. In this case the actor is controlled by the "play_tennis_interactively" automata that maps the 3D spaceball position and orientation data to the actor's racket. The main display is switched to the actors point of view, which can be done by a camera control symbol of the L-system. With the spaceball a user can control the position, the speed and the orientation of the racket. Its position and orientation determine also the position and look at point of the camera of the main display. As the racket and vision system are linked to a repelling force field of the size of the racket, the user can strike the ball. To facilitate the game the user can choose the resolution of the spaceball movement and he can fix the height at an actual value. A beginner can also fix the racket inclination. Through the racket the user is merged into the virtual environment and can be perceived by the other actors as the referee and his game partner. Nothing has to be changed for these other actors . They behave in both cases in the same way.

During the interactive game the vision system of the manipulated actor is deactivated as it is useless. The "play_tennis_interactively" automata is very simple as the behavior is now determined by the user. If the game is finished, which is determined by the referee, the next automata is popped from the stack, and the interactive user looses control over the actor.

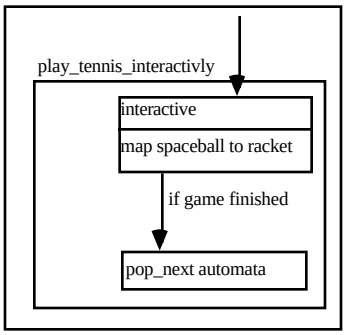


Figure 5.8.1: The "play_tennis_interactively" automata

5.9. Walking on Sparse Foothold Locations

A special application of the synthetic vision is given in [BNT93] and [BNT94]. In a collaborative project we developed the basics for vision based walking on sparse foothold locations and the automatic derivation of curved human walking trajectories. We focus on the role of the vision system in this section and present some aspects of the above mentioned papers. The special automata "walk_stepworthy" which controls navigation on sparse foothold locations was developed. It extends the set of the other automata available for high level actor animation. However, its use is currently limited to vision experiments with cube like actor representation. The adaptation of this automata for walking humanoids is left to future work. In appendix I, the figures I.17 and I.18 illustrate the test environment, and in section 7.3.2 some results are presented.

5.9.1 Introduction

One of the major advantages of legged motion over wheeled motion is its ability to cope with uneven terrain and non continuously supporting surfaces. The first aspect has been extensively studied while non continuously supporting surface has had very few contributions for a biped system. The scope of this section is to propose a vision-based approach of the free-walking planification problem in a planar environment with still and sparse foothold locations. The planification is associated with a cinematic walking motor driven by normalized velocity, curvature and a phase variable. A general methodology associates the two low-level modules of vision and walking with a planification module that establishes the middle term path from the knowledge of the visualized environment. The short term path is accessed by the walking module for fine adaptation of the walking trajectories. The planification is made under the constraint of minimizing the distance, the speed variation and the curvature cost. Moreover, the alternate walking motion is also evaluated in parallel and may be triggered if its benefice in curvature cost is higher than the speed variation cost of the corresponding halt and restart.

Interest in legged locomotion is due to its intrinsic quality of easy adaptation to irregular terrain. This is especially true regarding terrain with a sparse distribution of supporting surface. Such interest can be traced back to the 60th for theoretical studies and first prototype in mobile Robotics while a large family of multi-legged vehicles was being designed in the second half of the 70th [HIR84]. Often associated with vision-based perception, this topic now meets a broader attention of interrelated fields from biomechanics to behavioral animation, via Robotics and Computer Graphics. This section is devoted to a brief survey of

vision-guided locomotion from the different viewpoint of these fields prior to present our specific approach. We stress more results related with the perception and exploitation of the local environment of the legged system, so that it can derive the motion of, at least, the two coming walk steps.

Biomechanics has focused early attention on the analysis of legged locomotion and most of its standard characteristics are now identified[INM81][McM84]. Only recently, the study of visual control associated to locomotion has raised interesting contribution in this field [PAT89] for both walking and running. These authors have shown that subjects are able to rapidly modulate various parameter to successfully modify one step length for early, normal and late cueing times. Noteworthy are two observations. First that, for early cueing time, the subjects could anticipate the realization of the next step by modulating the present step so as to maintain an average velocity of progression during the step length change. Second, the average velocity of progression was slowed for both shorter and longer steps although, for longer step and early cueing time, the change was smaller.

Mobile Robotics is more involved in multiple legged vehicles (>3) and statically stable gait pattern (creeping) for stability reasons [HIR84][OZG84]. The difficulty here is not in the overall control of the structure in order to reach the next foothold, it lies in the selection of the best foothold to maintain the balance and move toward a goal[HOD91]. In [PAL91] and [PAL90] methods have been presented for the generation of a free gait for the straight-line motion and for generalized motion of a quadruped walking machine. By using a heuristic graph search procedure for path planning, local deadlocks and inefficiencies can be avoided. In a kind of mirror effect, some other studies [HOD85] [HOD91], dedicated to the control of a biped structure, are limited to running due to the great complexity of dealing with the double support phase of walking. In their most recent paper these authors propose three parameters to act on for step length adjustment while running (not vision-based). It resulted that the most efficient parameter was the forward speed.

Until now Computer Graphics and Animation papers have addressed the problems of following a predefined continuous path with derivation of the corresponding steps [BEZ92] [KO93a] [KO93b] and animation of a skeleton driven by inverse kinematics from a set of predefined step positions [GIR87]. The approach presented in [KO86a] also considers the problem of building walking trajectories from a succession of steps but these steps were previously derived from the scan of a predefined path by a specialized tool called the "foot print generator". This problem is partly solved in [BEZ92] where a straight line walking pattern controlled by a velocity curve[BOU90] is mapped along a curved path. Each place corresponding to the standing posture of the walking cycle defines a foothold location. Such locations can be tuned interactively by adjusting the velocity profile within an allowed range. Nevertheless, by construction, high curvature path produce noticeable sliding of the foot on the floor. In [KO93b] more general stepping motion are considered. This could be useful to move in a narrow environment.

While the previous Computer Animation approaches do not yet address a vision-based planning of walking, Behavioral Animation and Artificial Life focus more on vision simulation and analysis [REN90][CLI92] and less on the resulting lower level motion.

Our goal is to bridge the gap between vision simulation and analysis for path planning and curved path walking control in a planar environment of sparse foothold locations. The footholds are still and allow the positioning of one foot during walking or both feet at rest. The task to realize is to reach a specified location while minimizing the distance, the speed variation and the curvature of the resulting path.

First, after a normalization stage, the general characteristics of human free-walking are used to set the range of allowed foothold distances for the realization of one step. Then, four ideal velocities are associated with each foothold pair and our methodology naturally emerges at the end of next section. The following sections present successively the vision module, the planification system and the walking module. Some case-studies are evaluated and a discussion stressed the interest and limitations of our approach.

In our current work, we use the synthetic vision system to obtain the topology of the environment, actually seen by an actor. From its visual octree memory the actor extracts close to its current position a map of the step worthy regions (also denoted foothold locations). We suppose that the ground is flat and that the size of foothold locations is the size of a foot. A basic hypothesis is that such foothold locations do not overlap and that the space between foothold locations is empty. Thus, a map of foothold locations is represented by the list of 2D points (called local-map).

5.9.2 Map Extraction

Our map extraction algorithm works in a two step process. In the first step we define the range of interest (neighborhood of the actor) and the plane of the foothold locations in the octree space. The voxelization of the environment is always accompanied by some loss of information and inaccuracies. To improve the reliability of the map extraction, we construct in the region of interest an auxiliary octree (map_octree) containing only the full voxels of that plane and the projections of the full voxels of the neighboring parallel planes (see figure 5.9.1).

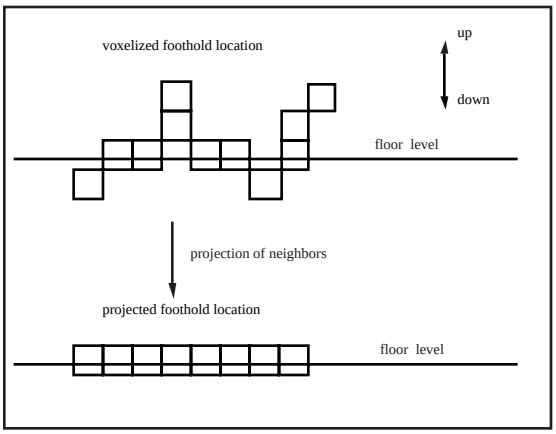


Figure 5.9.1: Auxiliary foothold octree.

Then, the map_octree contains a set of connected components representing the foothold locations. A connected component is defined by its neighboring occupied voxels. In the second step these connected components are determined. The average position of the voxels of a connected component (foothold location) gives the position of a foothold location, and the number of voxels of each components serves as a measure of its size. To minimize descretization errors, the 2D neighborhood of a voxel is defined only by the 4 connectivity shown in figure 5.9.2, and connected components that are smaller than a certain threshold value are not considered as foothold locations.

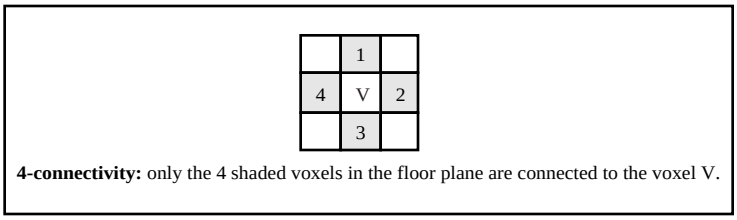


Figure 5.9.2: 4-connectivity of a voxel

Thus, after step 2, a certain number of foothold locations, given by an average position and an approximate size is available. This list or table of foothold locations characterizes the local neighborhood of an actor and serves as the topological base for a path searching algorithm.

5.9.3 Planning

We use a graph search strategy [PAL90], [PAL91] for optimal path planning from a start point to a destination point . A node of our graph corresponds to a walking state of an actor. A walking state is defined by a current foothold location, by the foot on it (left or right) and the previous foothold location. We use the following notation for such a triplet.

node:	s = (previous, current, foot)
the components:	s.previous, s.current, s.foot

Two nodes a and b are connected by an arc if the distance between a.current and b.current is smaller than Df_max and if a.foot and b.foot are not equal, demanding a leg change between connected nodes. In addition, the two nodes a and b have to satisfy the walking model next_step_characteristic, described in detail in the next section. This next_step_characteristic is used to accept or reject candidate footholds and it is derived from a maximum acceleration characteristics. As the equivalent constant foothold distance has been established for constant speed (-> constant step length and so constant foothold-pair distance), we should first evaluate it each time the candidate foothold-pair distance has a different value than the current foothold-pair distance.

The following drawing summarizes the different cases (Figure 5.9.3)

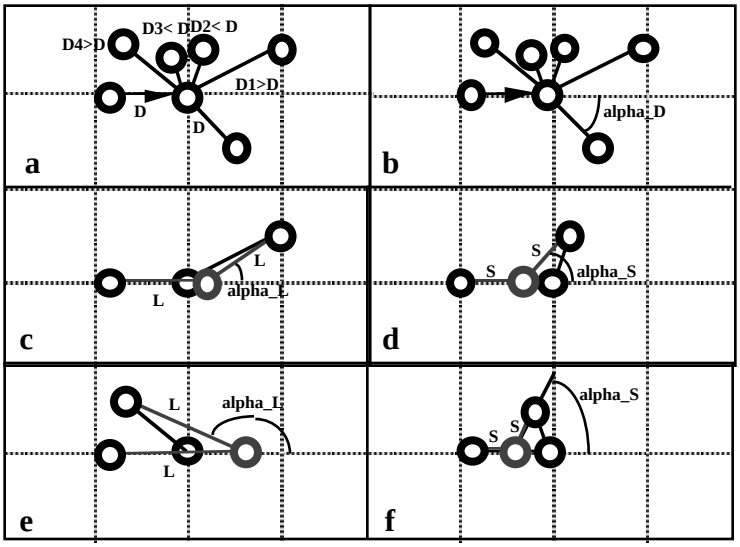


Figure 5.9.3: Candidate footholds.

a) the current foothold-pair is horizontal, others are candidates for next step
b) candidate with same foothold-pair distance D
c) & e) longer distances
d) & f) shorter distances

L, S and D are also called the polar distance while alpha is the polar angle

The arc length or the cost of an arc is a heuristic function h(a, b) of the two nodes. The simplest function, for example, is the distance *h = distance(a.current, b.current)* of the two foothold locations.

Our graph search procedure is based on Dijkstra's algorithm and produces an optimal route from a given node s to a destination node t. A sequence of nodes represents a path. Figure 5.9.4 illustrates a path with halting a) and a path without halting b).

a) (F0,F0,l) (F0,F1,r) (F1,F2,l) (F2,F2,r) (F2,F3,l) (F3,F4,r) ....
b) (F0,F0,l) (F0,F1,r) (F1,F2,l) (F2,F3,r) (F3,F4,l) ....

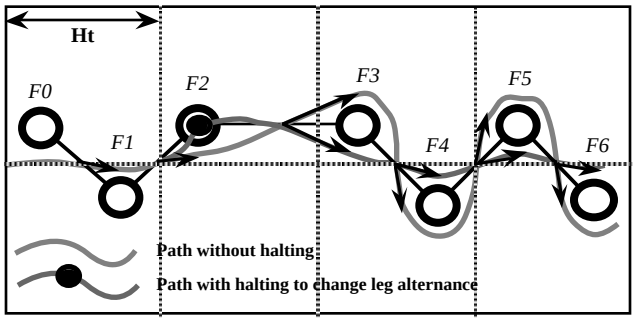


Figure 5.9.4: Paths with and without halting.

A detailed description of an implementation of Dijkstra's algorithm using a graph with an adjacency lists is given, for example, in [TEN90].

Our graph is represented using an adjacency relationship corresponding to the above description of an arc. In general, Dijkstra’s algorithm looks for an optimal path to all nodes from a given start node. If we specify a goal node the algorithm stops when it arrives there, having found all optimal paths to the nodes with lower optimal path cost as the optimal path cost to the goal node. The input of the path searching algorithm are the start and the goal nodes and the list of the foothold locations (local_map). The adjacency relationship of two nodes is procedurally known by the algorithm. During the path search process the algorithm creates and initializes dynamically the adjacent nodes of the currently treated node by using the list of the foothold locations. When it encounters for the first time the goal node, the algorithm stops. By using a procedurally defined adjacency relationship and not an adjacency list representation of the graph, it's not necessary to create the complete set of all possible nodes with their adjacency lists from the local_map. Therefore, only the nodes with a smaller optimal path cost from the start node have to be created, which is useful, if the distance between start and end points is small compared to the total domain of the local map.

The global behavior of an actor is determined by the navigation automata shown in figure 5.9.5. It is a simple automata used for testing the above described features.

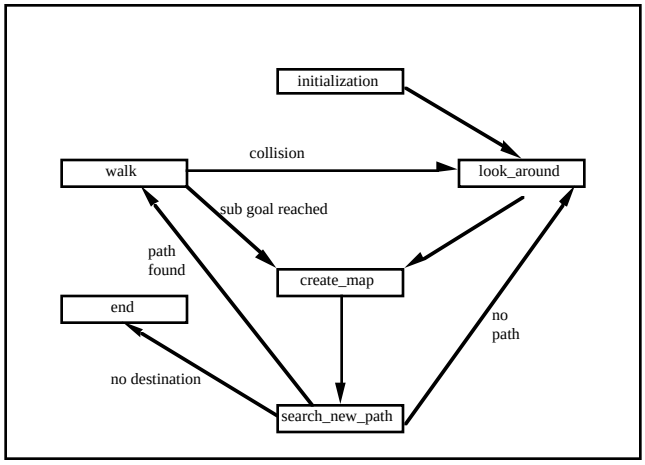


Figure 5.9.5: Navigation automata.

At the beginning of a simulation the user pushes some destinations (3D coordinates of goals) on a destination stack and defines the initial position of the actor. After initializing the actor (state initialization), it starts its simulation by looking around (state look_around) and memorizing what it sees. Then, it creates a local map of the foothold locations in its neighborhood and pops the first destination from the destination stack. If there is no destination on the destination stack, the simulation is terminated (state end), otherwise the path finding procedure continues. As the path searching algorithm (state search_new_path) needs a start and an end node as input, they have to be created first. The start node s is determined by taking the closest foothold location to the actual actor's position and assuming that it has its feet assembled with s.foot = left. Its initial orientation is directed towards its destination. The destination node t is initialized with the closest foothold location (t.current) to the destination, having the feet assembled (t.previous = t.current) and s.foot = left. Now Dijkstra's algorithm can search for an optimal path. If it finds no path, the actor is set to the "look_around" state, otherwise it starts walking (state walking). It continues to walk until it detects an obstacle on its way or until it reaches its destination node. In this case it switches to the look_around state and the simulation continues until there are no more goals on the destination stack.

To detect obstacles on its planned path, the actor checks at each frame (or time interval) whether certain voxels along its path that are within a certain distance from its position are occupied. This way, as the visual octree memory is updated at each frame, the actor is able to detect dynamic obstacles and to react. Of course, this automata doesn't solve the global navigation problem in unknown (unseen) environments, but it allows to find optimal paths in local (or seen) environments where loops and circuits are avoided.

In our present work we focused on a discrete environment of foothold locations that severely limits its application in universal environments. This approach, however, was sufficient to demonstrate its usefulness and feasibility and future work will focus on foothold locations allowing more than one step mixed with discrete locations and uneven terrain by integrating articulated walking humanoids.

Another current limitation is the constant width of the gait that limits the ability to realize optimal path by artificially increasing the total curvature cost. Therefore, future work will first focus on the modeling of a lateral component of walking to overcome the over-curved path limitation.

5.10 Behavioral Control by Production Rules

With the behavioral L-system we can not only define behavior by precompiled automata but also by production rules editable by any user. With the virtual sensor functions usable in the conditions of the production rules and the concept of the query symbol of environmentally sensitive L-systems simple rules can lead to interesting emergent behaviors. The essential elements of the extended L-system interpreter for behavior definition by production rules are:

- individual actor instances maintaining their internal state
- map symbols **M** and **g**, that enable actor - turtle and turtle - actor mapping.
- global variables for turtle states maintaining their content from one time step to the next one
- sensor functions usable in the conditions of production rules
- the query symbol **?** which transmits position data of the virtual environment into the parameter space of the symbols
- the cut symbol **],** which eliminates branches or part of branches of the formal symbol string

- The ephemeral symbol **e**(par1 = 0) allowing to push a behavior command string on the behavior stack and to render the behavior control back to automata procedures
- A special function usable in the conditions of production rules and returning the actual active automata behavior enables behavior control switching to production rules.

With production rules many types of simple behaviors can be realized. In the next two sections we present two examples of behaviors defined by production rules. The first one uses synthetic vision and shows the realization of the Pledge algorithm for an actor trying to escape from a maze. The second example treats random motion of an actor with reaction to tactile collision detection. For simplicity we mention in the following L-system definitions the parameters and growth function of the symbols only where it is necessary for the comprehension.

5.10.1 Escaping from a maze based on vision

The Pledge algorithm [ABE84] describes a sure method for finding the exit of maze.

```
1. Select an arbitrary initial direction, call it "north" and face that way.
2. Walk straight "northward" until you hit an obstacle.
3. Turn left until that obstacle is on your right.
4. Follow the obstacle around, keeping it on your right, until the total turning
(including the initial turn in step 3) is equal to zero.
5. Go back to step 2.
```

Code 5.10.1: The Pledge algorithm

The following L-system pseudo code defines an autonomous actor applying the rules of this algorithm.

```
Axiom: l m(x=1) B [ M f x(z=0) g y ]
Production Rules:
p1: x(z) ==> if    (max_age==0.05) and (path_not_free)
             -(z, f1 = 4)
p2: -(z) ==> if (max_age==0.05) and (path_free)
             x( z + (1+t/dT)*4)
p3: x(z) ==> if(max_age==1.2) and (path_free) and (z>0)
             +(z, f1=4)
p4: +(z) ==> if    (max_age==0.05) and
                ((path_not_free) or (z - (1+t/dT)*4 < 0))
             x(z-(1+t/dT)*4)
```

Code 5.10.2: The Pledge algorithm defined by a L-system

The symbols in the axiom have the following semantic. The symbol **l** switches the camera control to the space ball interface allowing an interactive control of the camera position and orientation. The symbol **m** activates a material referenced by its parameter x. The symbol **B** imports an externally defined triangulated surface, which is drawn in the turtle's local coordinate system and scaled with the growth function values that are omitted here for simplicity. The symbols **[** and **]** push and pop the turtle state and embrace here the actor's behavior. The symbol **M** maps the turtle to the actor's vision system, and the symbol **f** advances the turtle a certain step given by a growth function value in direction of its axis H(eading). The parameter z of the dummy symbol **x** represents the total amount of the turtle's rotation to the left. In the axiom the parameter z is initialized with zero. With the symbol **g** the actor's vision system is mapped to the turtle's actual position and orientation and the symbol **y** symbolizes the geometric shape of the actor. It can be, for example, a simple cube, a complex

figure defined by production rules or an imported articulated humanoid. The axiom lets the actor move straight on in the imported environment defined by the symbol **B**.

Production p1 turns the actor to the left in case of an obstacle in front of it and perceived by its vision system. The symbol **x** with the parameter z is replaced by the symbol **-** if the local age of the symbol **x** is bigger than 0.05 seconds and if the vision sensor function path_not_free indicates an obstacle in front of the actor at a certain distance. The symbol **-** has a parameter z initialized with the value of the parameter z of the symbol **x** and a growth function f1 with the constant value 4 signifying a rotation by 4 degrees to the left at each time step.

Production p2 stops the rotation to the left if there are no more obstacles in front of the actor and updates the parameter z of the new symbol **x** with the total amount of degrees turned to the left. The parameter t is the actual local age of the symbol **-** and dT corresponds to the time step of the animation loop.

After 1.2 seconds production p3 turns the turtle back to the right if there are no more obstacles in front of it and if the parameter z is bigger than 0. The symbol **+**, the right side of the production rule, gets its parameter z initialized with the corresponding value of the symbol **x**, the right side of the production rule. The growth function f1=4 signifies a right rotation by 4 degrees at each time step.

When turning to the right, production p4 stops this rotation if the local age of the symbol **+** passes 0.05 seconds and if the path is free or if the integrated total amount of rotation of the turtle is zero.

5.10.2 Random motion with collision detection

In the following example we define a L-system where an actor moves randomly around until it collides with force field modeled objects such as repelling walls, for example. When it senses a collision, it stops, emits a sound and moves a few steps back. Then, it turns around and continues its random walk. In code 5.10.3 we suppose that the force fields are already defined somewhere in the axiom.

Axiom: ... **[[**(x=1) **Z]]**(x=1) **A** ...
Productions:
p1: **Z** ==> if (max_age==1)
| **[ ]**(x=1) **f** +(f1=random value) **[**(x=1) **?]**
p2: **?**(x,y,z) ==> if (max_age==0.2) and (force(x,y,z)>1)
| **T**(f1=12) **[ ]**(x=1) **f**(f1=-0.02) **[**(x=1) **X]**
p3: **X** ==> if (max_age==0.2)
| **[ ]**(x=1) +(f1=43.2(t-t^2)) **[**(x=1) **? Z]**

Code 5.10.3: Random motion defined by a L-system

In the axiom the symbols **[** and **]** without parameters push and pop the turtle state. They embrace in general one ore more behaviors. Note also the other symbols **[**(x=1) and **]**(x=1) with the parameter x=1. The symbol **[**(x=1) memorizes the actual turtle state in a global variable table indexed by the parameter x. This saved turtle state can be recuperated with the symbol **]**(x=1). This state value is not lost from one time step to the next. Thus, a turtle state at time T can be recuperated by this symbol at time T+ΔT. The normal push and pop operations, however, are only local for one symbolic string at a given time T. The symbol A represents an actor body to be defined by some production rules not given here for simplicity.

For the dummy symbol **Z** there exists the production rule p1 that describes the random walk of the actor. After one second a symbol Z is replaced by the right side of the production rule. The first symbol | is a cut operator. It eliminates the current branch of the old symbol string with all sub-branches including the symbols **[** and **]** containing the branch. The second and last symbols **[** and **]** of the right side of p1 define a new behavior branch. With the symbol **]**(x=1) the old turtle state is recuperated and the symbol **f** moves the turtle forward in direction

of its heading vector. The symbol + rotates the turtle at each time step by a value randomly determined for each application of the production. The symbol **[**(x=1) memorizes the actual turtle position in the global turtle state table and the symbol **?** is the query symbol. Its parameters x, y and z are at each time step initialized with the actual turtle position.

The production rule p2 for the query symbol detects a collision, emits a sound and moves the turtle backwards. When the age of the query symbol is bigger than 0.2, and the amount of the force field at the position given by the x, y, z values of the query symbol is bigger than one, the query symbol is replaced by a cut operator **]**, the sound event symbol **T** creating a sound event referenced by the growth function value f1=12 and a new behavior branch embraced by the symbols **[** and **]**. In this behavior branch the old turtle state is recuperated with **]**(x=1), then the turtle is moved backwards by a step of 0.02 and the new turtle state is saved by **[**(x=1). The symbol **X** represents a germ for the next behavior defined by production rule p3.

After 0.2 seconds this symbol **X** is replaced by the right side of the production rule p3. It cuts the current behavior branch and turns the turtle during 1 second by a total of 180 degrees with a rotational acceleration at the beginning and a deceleration at the end according to the growth function f1=43.2(t-t^2)) of the symbol **+**. The growth function f1 corresponds to the rotation step at each time step (=0.04 seconds) and is a function of the local age t of the symbol. The symbol **Z** represents the germ for the random walk activated by the production rule p1 after 1 second.

6. Implementation

In this chapter we give an overview of the system architecture and we treat some selected implementation details which are complementary to other topics treated in previous chapters. We focus on L-system definition files and some modules of the L-system interpreter as these are essential parts of the animation system. Moreover, we present the shared memory interface for the process “h2_process”, which animates the humanoids. We think the process communication through shared memory is a efficient concept for real time systems on multiprocessor machines. It works also on single processor machines. Additionally, it favors modular programming. Finally, we present the POST speech recognition system, the image by image film production facility and the hardware and software environment in which the animation system has been implemented.

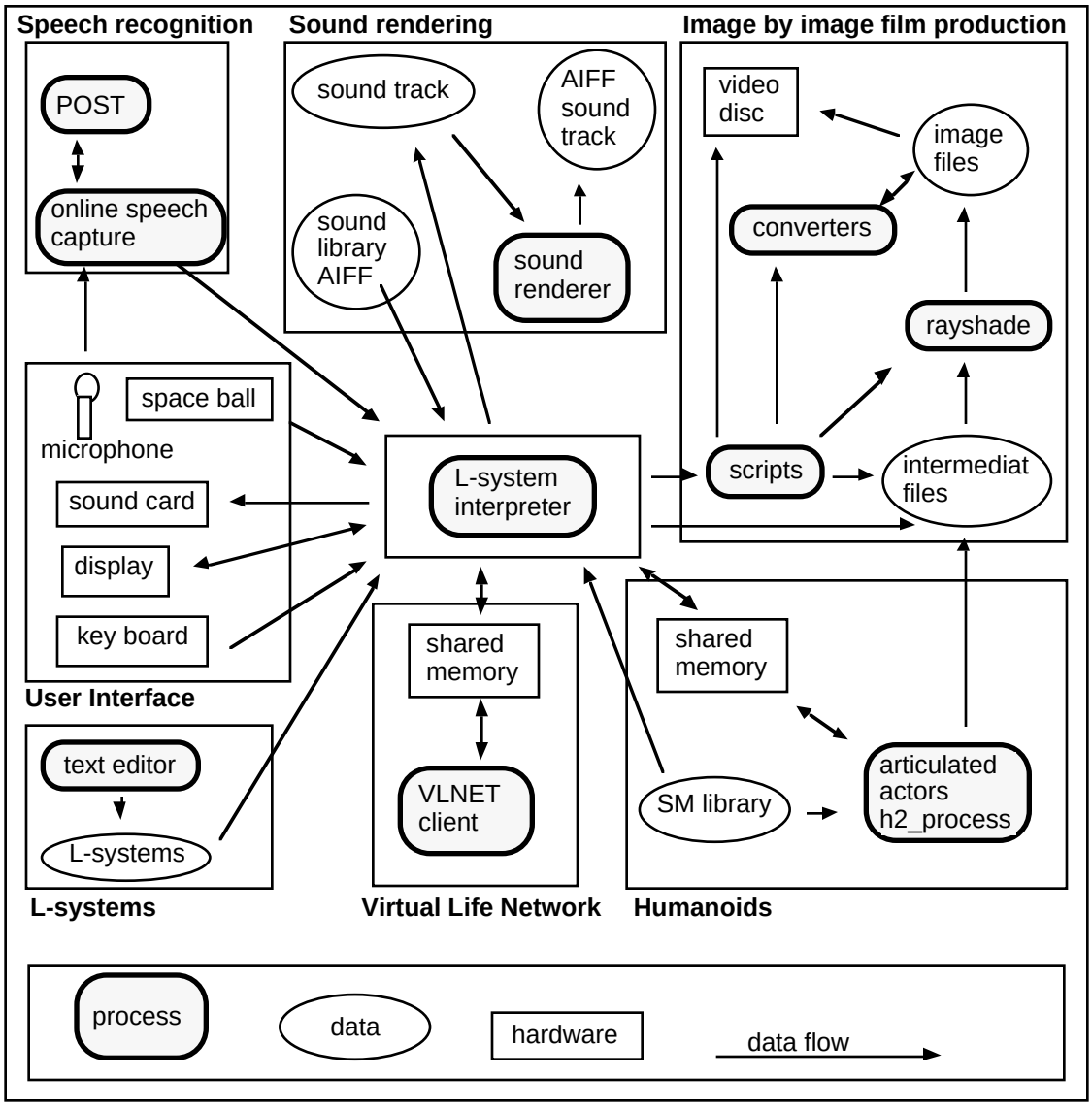


Figure 6: The overall system architecture of the behavioral animation system
VLNET: Virtual Life Network
SM: Surface Manager
POST: Parallel Object oriented Speech recognition Toolkit

6.1 System Architecture

Figure 6 shows an overview of the system architecture containing the hardware configuration, the data libraries and the involved processes with their mutual interactions. The heart of the whole system is the L-system interpreter named LMinterpreter, which models, drives and controls the virtual environment. We introduce the abbreviation LM in honor to (L)inden(m)ayer [PRUS90], the inventor of the L-systems.

This interpreter controls an animation by interpreting one or several L-systems that can be created and edited by any text editor. Simulations can be watched directly on the screen. User interaction is possible through a space ball, a speech recognition system and the keyboard. The computer should be equipped with a sound card playing AIFF files if sound is used in the L-systems. In an animation predefined AIFF files are used as 3D sound sources. When specified, the L-system interpreter can produce a 3D object space sound track file that can be rendered by a special sound renderer producing the final synchronized sound track at the end of the animation. This sound track can be added to the corresponding video film sequence. Video film sequences are in general produced in image by image mode where the L-system interpreter produces intermediate files of the ray tracer's format, containing the complete scene description for a given frame. Then, it starts a script file, which starts the ray tracer and stores the resulting image on a video disc after having converted it to the right format.

When specified, the L-system interpreter controls through a shared memory interface a special process allowing the real time importation of the highly sophisticated articulated actors of the AGENTlib library [EAgent8125]. In real time animation only simplified skeletons are displayed. Actually, only real time simulations including VLNET clients support a body display of actors (see figures I.25 to I.28 of appendix I). In image by image film productions, however, the complete body deformations [S95] are calculated and the resulting triangulated surface is saved automatically in intermediate Rayshade files. The animation and placement of the actors, however, is controlled at a higher level by the L-system interpreter according to the L-systems defined by the user. The L-system interpreter can also import surfaces of the SM format, frequently used and supported in the LIG laboratory. The SM format establishes a link to other applications. This surfaces can be drawn directly in real time applications. In image by image film productions they are automatically converted to the ray tracer format and included in the corresponding environment description file.

The speech recognition system is composed of two processes. The "word capture" process captures the sound signal, makes some signal pre treatment and transmits it to the POST process for the recognition. The POST process communicates the result to the "word capture" process which transmits the semantic of the recognized word to the L-system interpreter through socket communication. These two processes work asynchronously to the L-system interpreter and can turn on different machines.

The Virtual Life Network uses also asynchronous communication. Through the shared memory interface functions which make part of the VLNET system the L-system interpreter can update at each time step the networked shared environment.

The following list indicates which processes of figure 6 have been developed by the author of the thesis, and which processes are commercial products, freeware or common products developed at the Computer Graphics Laboratory (LIG).

- L-system interpreter: by the author
- word capture: by the author
- sound renderer: by the author
- scripts: by the author
- h2_process: written by the author using the AGENTlib library developed at LIG

- VLNET: LIG and MIRAlab
- text editor: commercial product
- converters: commercial products / freeware
- rayshade: freeware
- POST: academic freeware

6.2 The L-system Definition File

In the animation system each animation, real time or image by image film production, is scripted and defined by one ore several L-systems created by a user. The L-system interpreter reads at the beginning these L-systems and initializes the animation. These animations can be of very different nature. The animations range from simple geometric modeling, interactive networked games with autonomous actors to ray traced film sequence productions with rendered sound tracks. In this section we present only a simple L-system for a tree generation showing its main parts. A complete syntactic and semantic definition of L-system definition files can be found in appendix A, B, and C.

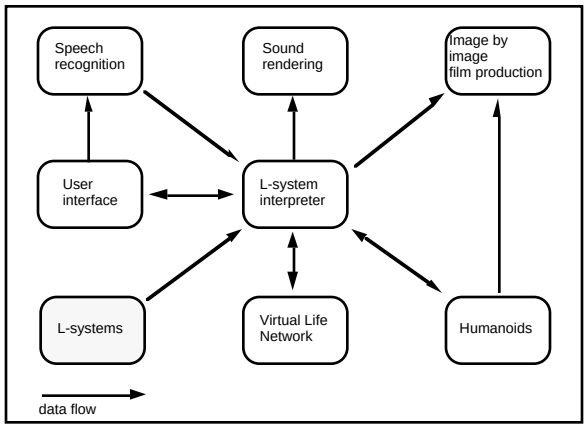


Figure 6.2.1: L-system definition files

We define each L-system by a text file, called L-system definition file. A user can create or edit such a file by any text editor. Code 6.2.1 shows a L-system definition file representing a tree in a tropism force field simulating gravity and wind. Figures 6.2.2 to 6.2.5 illustrate the same tree with different forces. This L-system is an imitation of the parametric L-system example given in [PRUS90, page 59] and illustrates besides its syntax that it corresponds to the cited example. In this text file we can see its different sections. The exact syntax is given by the BNF in appendix A.

The first section "Camera_AnimationParameters" defines visualization parameters of the user's window and some animation parameters. In the section "RepetitionParameters" the object defined by the axiom and the production rules can be stochastically placed, rotated and scaled in a certain area by setting the parameters. In the third section "Options" some options useful or necessary for a particular animation can be defined. The "Splines" section allows to define splines that can be used in string expressions of the axiom or production rules (see also table B1). Triangulated surfaces of format sm are imported in section "Surfaces" that is empty in this example. The last two sections "Axiom" and "Productions" contain the axiom and the production rules of the L-system. Comments can be placed between /* and */. Nested comments, however, are not allowed.

```
/* name of file: cours_tree3.lm defining a tree with tropism forces */
Camera_AnimationParameters
prefpos_x1_x2_y1_y2     0 721  0      576     /* window coordinates on the screen */
observer                0.0    15     130.0   /* position of the camera */
lookatPoint             0.0    15.0   0.0     /* look at point of the camera */
twist                          0               /* the twist of the camera */
viewAngle               500                    /* the view angle of the camera */
aspectRatio             1.25                   /* the aspect ratio of the window */
near                           1.0             /* the near clipping plane of the camera */
far                     10000.0                /* the far clipping plane of the camera */
swapinterval                   1               /* the swap interval of GL */

nbrCalcStep_seed 4      34743
/* the number of intermediate calculation steps during the frame to frame interval T_interval for
evolving the particle system. The second argument 34743 is the seed of the random number generator.
*/

t_final_interval        250.0  0.04
/* an animation always starts from time 0. The two arguments determine the duration of the animation
and the time interval. */

RepetitionParameters
/* in this file the repetition section is empty */
Options
/* in this file the option section is empty */
Splines
/* in this file the spline section is empty */
Constants
/* see table B.5 for the complete syntax. Constants can have two or three arguments */
a  180   0.0          /* divergence angle 1 */
b  252   0.0          /* divergence angle 2 */
r  36 0.0            /* branching angle */
p  1.07  0.0          /* branch elongation rate */
e  1.732 0.0         /* branch width increase rate */
f  0.15      0.0      /* initial width */
Surfaces
/* in this file the surface section is empty */
Axiom
v  ()  () () ()      (0.31_) (0.3) (0.19)   /* tropism force illustrated in figure 6.2.4 */
v  ()  (1) () ()     (2.5) (1) ()           /* stiffness 2.5 for branches */
m  ()  (6) () ()     () () ()               /* material 6 is activated */
n  ()  (4) () ()     () () ()               /* low resolution for cylinders is set */
+  ()  () () ()      (90) () ()             /* the turtle points upwards now */
N  ()  (10) (f) ()   (x) (y) (y)            /* the trunc of the tree is drawn */
z  ()  () () ()      () () ()               /* a germ for the rest of the tree */
Productions
z  (1) () () ()                             /* unconditional production for trunk and three
branches */
1.0                                         /* probability 1 */
N  ()  (5) (fe*) ()  (x) (y) (y)            /* trunk */
[  ()  () () ()      () () ()               /* push turtle state */
&  ()  () () ()      (r) () ()              /* rotation */
N  ()  (5) (f) () (x) (y) (y)          /* first branch */
z  ()  () () ()      () () ()               /* germ for new iteration */
]  ()  () () ()      () () ()               /* pop turtle state */
/  ()  () () ()      (a) () ()              /* rotation */
```

```
[   ()  () () ()        () () ()
&   ()  () () ()        (r) () ()
N   ()  (5) (f) () (x) (y) (y)          /* second branch */
z   ()  () () ()        () () ()
]   ()  () () ()        () () ()
/   ()  () () ()        (b) () ()

[   ()  () () ()        () () ()
&   ()  () () ()        (r) () ()
N   ()  (5) (f) () (x) (y) (y)          /* third branch */
z   ()  () () ()        () () ()
]   ()  () () ()        () () ()
EndProb
End

N  (1) [(z)=] () ()  /* unconditional and context independent production for discrete branch growth */
1.0
N  ()  (xp*) (ye*) ()          (x) (y) (y)       /* elongation and width increase of a branch
*/
EndProb
End
EndProductions
```

Code 6.2.1: A L-system definition file

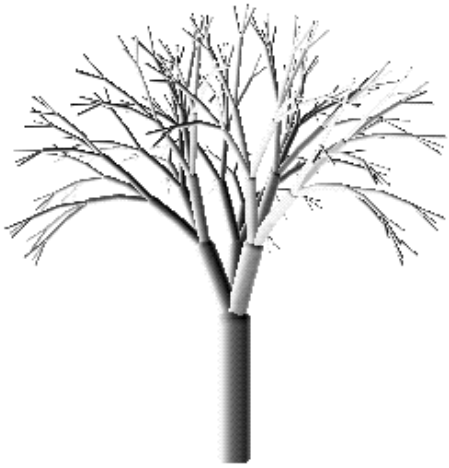


Figure 6.2.2: Tree under gravity influence

Figure 6.2.3: Tree under gravity influence

Figure 6.2.4: Tree with gravitation and a weak wind

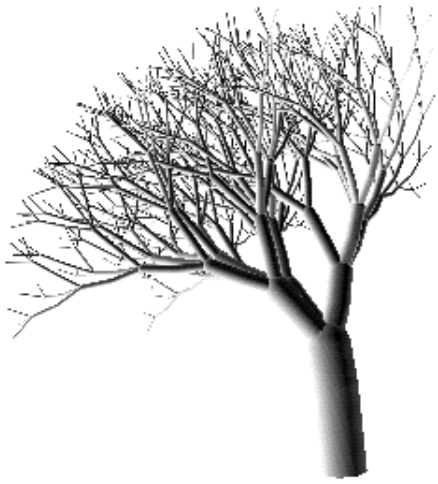


Figure 6.2.5: with gravitation and a strong wind

6.3 L-system Interpreter

The L-system interpreter is the key program of the animation system. It interprets one or several L-system definition files and controls and synchronizes the simulations. Figure 6.3.1 shows its principal modules. The main program consists of the animation loop already explained in section 2.1.1 The module “L-system parser” groups the functions for parsing L-system definition files and the initialization of animations. Another important module permits the step by step iteration of L-systems, and their interpretation by using functions of the “Symbol procedures” module, which interpret symbols. The “turtle” module exports procedures for turtle manipulation during the interpretation of a formal object. The module “Particles” treats the particle system and the force fields, it evaluates string expressions and evolves the system of differential equations of the particle system. The module “Air” manages the sound events of the virtual environment.

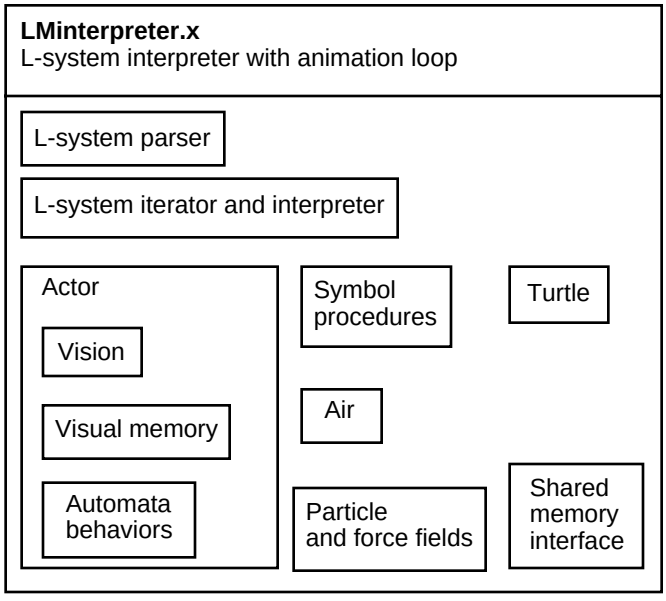


Figure 6.3.1: The principal modules of the L-system interpreter

The actor module uses extensively the “Vision”, “Visual memory” and “Automata” modules. It exports high level actor functions. The “Vision” module implements the synthetic vision features. The “Visual memory” module groups all functions for the environment voxelization through synthetic vision. The behaviors of actors modeled by automata functions, together with behavior control functions, are grouped in the “Automata” module.

Finally the “Shared memory interface” module offers high level functions for inter process communication through shared memory.

6.3.1 Visual Memory Implementation

As already mentioned in section 3.1.3 we implement the visual memory of an actor as an octree voxelizing the environment. Its advantageous features have already been extensively discussed in previous chapters. To implement the octree, we used the data structure illustrated in figure 6.3.2.

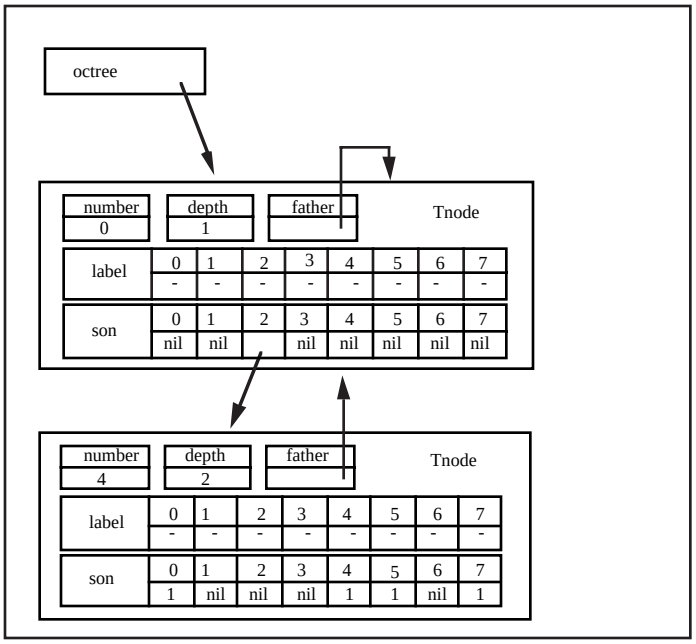


Figure 6.3.2: Data structure of the octree.

A node represents a voxel. Each node in the octree contains a table of 8 pointers (sons) to other nodes, representing the subdivision of itself. The value NIL of a pointer means, that the corresponding voxel is free. The value 1 of a pointer designates an occupied voxel. We suppose that a valid address of a node is greater than 1. If the value of a pointer is a valid address to another node, then the corresponding voxel is subdivided. Each pointer has an associated label, where some additional information of this node can be stored. Moreover, a node contains a father pointer, which allows us to go back to the root of the tree. This father pointer is useful, when we want to simplify the octree, having all the sons of a node occupied. In this case, we can delete the current node and set the corresponding son in the father node to the value 1. This is a recursive process that can propagate to the root. The depth variable of a node indicates the degree of subdivision and facilitates the calculation of the positions of the sons and the size of the corresponding voxel. As the octree corresponds to a discrete representation of the environment, we used a normalized version of an octree. Each position of the scene is transformed by equation 6.3.1.

$$P_{octree}(i) = \text{trunc}((\text{translation}(i) + P_{scene}(i)) * \text{scaling}) \qquad (6.3.1)$$
$$i = 0,1,2$$

Theresult is a point P_{octree}. Its position is determined by three integers. At the initialization of the octree, a maximal range (Range) that has to envelope the whole scene transformed into octree coordinates is defined. Thus, the octree bounds the whole transformed environment by a cube of an edge size of the value of this variable Range. Now, the position of a point in an octree can easily be calculated by simple shift operations on integer variables as indicated in figure 6.3.3.

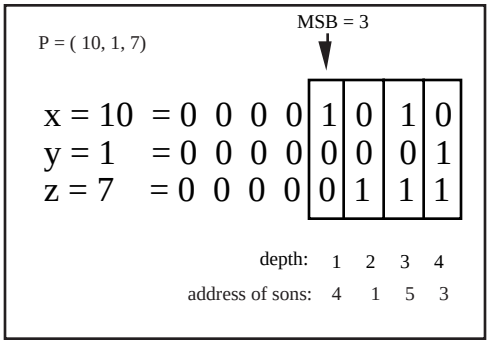


Figure 6.3.3: Addressing of voxels in the octree.

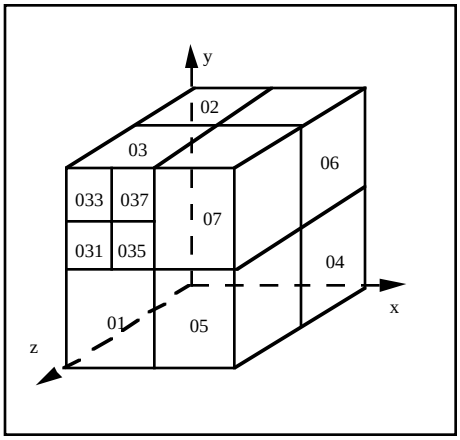


Figure 6.3.4: Octal voxel numbering in the octree according to the calculated addresses of 3D coordinates.

The most significant bit MSB (= Hbit -1) of an address is implicitly determined by equation 6.3.2.

$$range = 2^{Hbit} - 1 \qquad (6.3.2)$$

$$edge\_size = 2^{Hbit} - depth\_level \qquad (6.3.3)$$

$$neighbor(0) = P_{octree}(0) + i * edge\_size \qquad (6.3.4)$$

$$neighbor(1) = P_{octree}(1) + j * edge\_size \qquad (6.3.5)$$

$$neighbor(2) = P_{octree}(2) + k * edge\_size \quad i, j, k = -1, 0, 1 \qquad (6.3.6)$$

The address of a son node at a certain depth level of the octree is formed by the three corresponding bits of the x, y and z coordinates of the point P_{octree}. This addressing mode results in a voxel numbering according to figure 6.3.4. The 26 neighbors of a voxel at a certain level can be calculated by the equations (6.3.4 ... 6.3.6).

6.3.2 Shared Memory Interface

In this section we detail the implementation of the shared memory interface for importing humanoids from the “h2_process” process of figure 6. Figure 6.3.5 illustrates the data structure of the shared memory interface and the communication between the two processes.

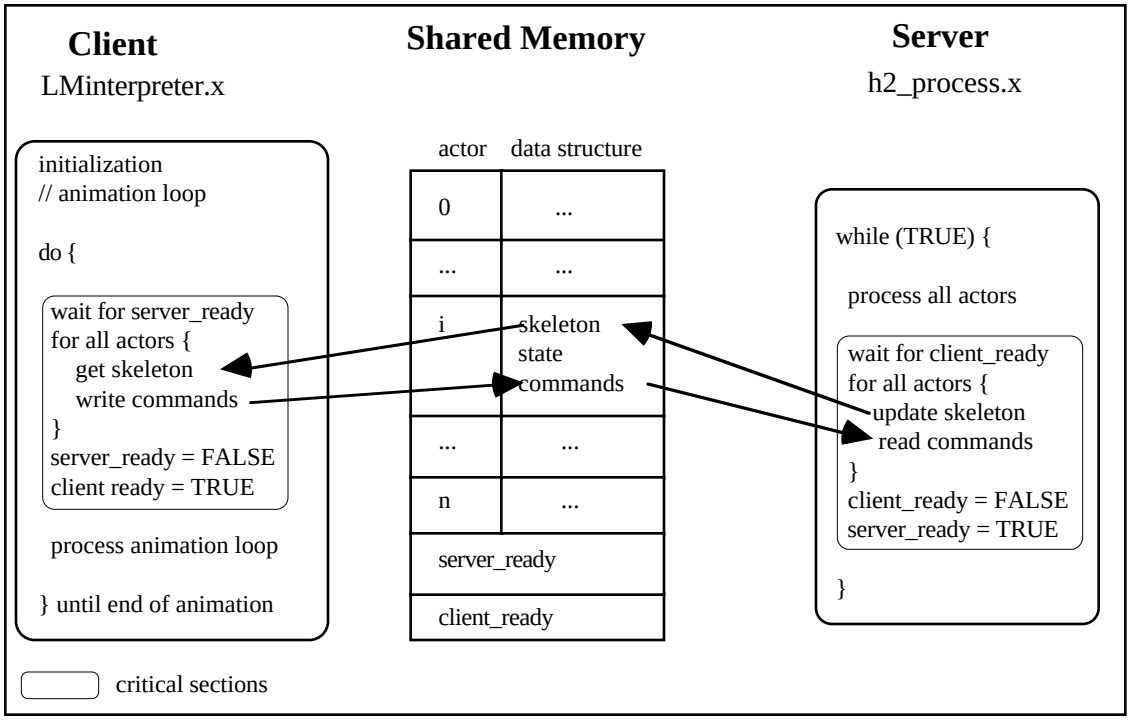


Figure 6.3.5: The shared memory interface of the humanoid process “h2_process.x” and the L-system interpreter “LMinterpreter.x”.

The shared memory data structure contains for each actor a simplified skeleton node table, a state variable and a command string. The simplified skeleton table contains 29 node positions as shown in figure 6.3.6 (see also [Ehum6709]).

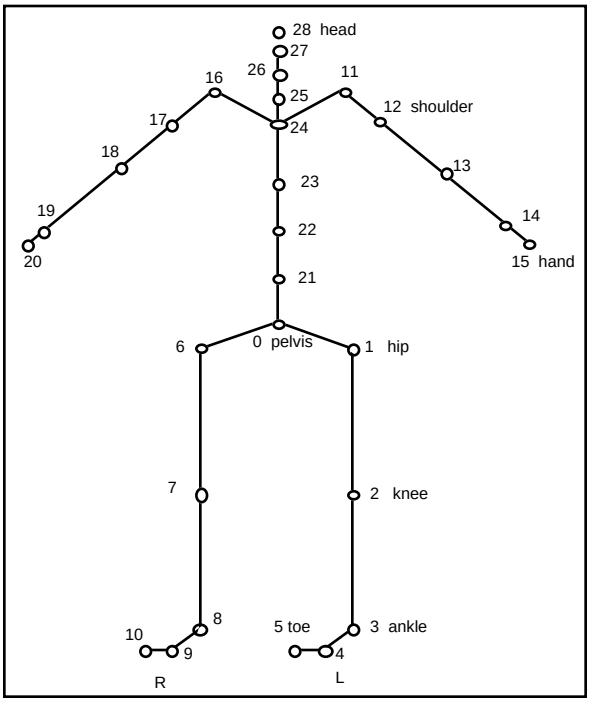


Figure 6.3.6: The simplified skeleton structure with 29 nodes, used for real time display

The access of the shared memory is a critical section of the processes. The shared memory should only be accessed by one process at a time to avoid read and write inconsistencies. We use the two flags server_ready and client_ready to realize the mutual exclusion of the critical section and the synchronization of the two processes. The server process h2_process.x interprets at each frame the commands for each actor and it updates the resulting skeleton positions. The LMinterpreter.x process writes the commands and reads the skeleton positions in order to display them and/or to process them according to the on-going simulation.

The mutual exclusion in the critical section could have been realized also by semaphores which would be advantageous for single processor machines. On multiprocessor machines, however, we can avoid a time consuming context switching by using the “active wait” method.

The declaration of the humanoid actors is done in the "Options" section of the L-system definition file. The lines,

```
NBR_actors 2                                    /* declares two actors */
iVisionWinSize_maxColors 0 40 1 /* vision declaration of actor 0 */
iVisionWinSize_maxColors 1 40 1 /* vision declaration of actor 1 */
NBR_walk_motors 2                               /* starts h2_process.x with 2 humanoids */
actorNbr_automata 0     0 ... motor_init 0.0 300 0   /* automata script for actor 0 */
actorNbr_automata 1     1 ... motor_init 0.0 300 0   /* automata script for actor 1 */
```

for example, declare two humanoid actors with synthetic vision. The motor_init command in the automata command string initializes a humanoid. The three parameters determine its initial speed, the scaling in the L-system, and the ray tracer flag. For real time animations, currently, only a simple display of the skeleton of a humanoid is supported. To draw the skeleton in the turtle's local coordinate system we place the symbol **G** after the symbol **M** which places the turtle according to the actor position and orientation.

6.3.3 Import and Export of Data

As mentioned in section 2.2.3 the animation system offers some features for the import and export of data used or produced by other external applications. Such data exchange has to be declared in the "Option" section of a L-system definition file. The data import is done during the parsing of the L-system at the beginning of a simulation. The export of data into the corresponding files happens during the animation and is triggered by internal flags or symbols of the L-system. The next sub - sections describe the actually implemented features concerning import and export of data.

Surface Manager SM

At LIG the Surface Manager (SM) [SM94] file format is widely used for representing 3D triangulated surfaces. There exist converters to and from other data formats as Wavefront's .obj format, for example. The L-system interpreter supports this format SM. It allows to import one ore several SM files into a L-system and to display them. The importation of the SM files is declared in the L-system definition file immediately after the "Surfaces" keyword by

fig_sm *Nbr path/filename.sm* (ex: **fig_sm** 0 /usr/people/noser/head.sm)

With the symbol **B** this surface can be repeatedly drawn and scaled in the turtles local coordinate system. If the L-system interpreter is started with the option RS followed by a frame number

example: LMinterpreter.x SM 10 flower.lm

a SM file named SM_object.sm is created containing a 3D triangulated surface of the complete scene at frame 10. This feature makes sense only for small scenes or objects and allows to create, for example, simple flowers or trees usable in other applications or other L-systems supporting the SM format. Complex environments, however, will produce too big SM files.

Data Channels

Data channels are ASCII files containing a time step and a sequence of floating point values. These data files are in general produced by external applications used in an L-system defined animation. With the function a special function these data can be used in parameter expressions, growth functions or conditions of production rules. This universal concept is a flexible interface to external applications. During a semester project, for instance, a student wrote a program analyzing music files according to criteria like intensity or rhythm. With these resulting data files parts of a L-system scene could directly be animated allowing automatic video clip generation. Data channels are declared in the “Options” section of the L-system definition file by

NbrChannels_listOfNames Nbr name1 ... nameNbr

The channels are later referenced by their number that corresponds to its list position.

The file format is very simple. The header contains only the time step (float) and the number of values that follow in the data section of the file. At the beginning of a simulation all data are read into a table. In between values are linearly interpolated if the time step of the data file doesn't correspond to the time step of the animation loop. A value of channel number i at time t is obtained through the binary function ?50 that takes the current uppermost stack elements as arguments. A call of this function pops the 2 arguments from the stack and pushes the result on the top of the stack. Thus, an universal application is enabled as the function can be called in parameter expressions, in growth functions and in conditions of production rules. Up to 10 data channels of arbitrary length can be defined.

Actor Trajectories

The L-system interpreter supports importation and exportation of actor trajectories. The ASCII file format of such a trajectory is a sequence of the following record:

Actor Nbr actor state time pos_x pos_y pos_z

The file generation is initiated by the declaration

trace_actor *Actor_nbr* *file_name*

in the “Options” section of the L-system definition file. With the symbol **!** the corresponding record of the actor can be written to the track file at each time step.

The file importation is initialized by the declaration

import_trace *Actor_nbr* *file_name*

in the “Options” section of the L-system definition file. With the symbol **$** the current actor is positioned at each time step according to the imported trajectory.

6.3.4 Behavior Control

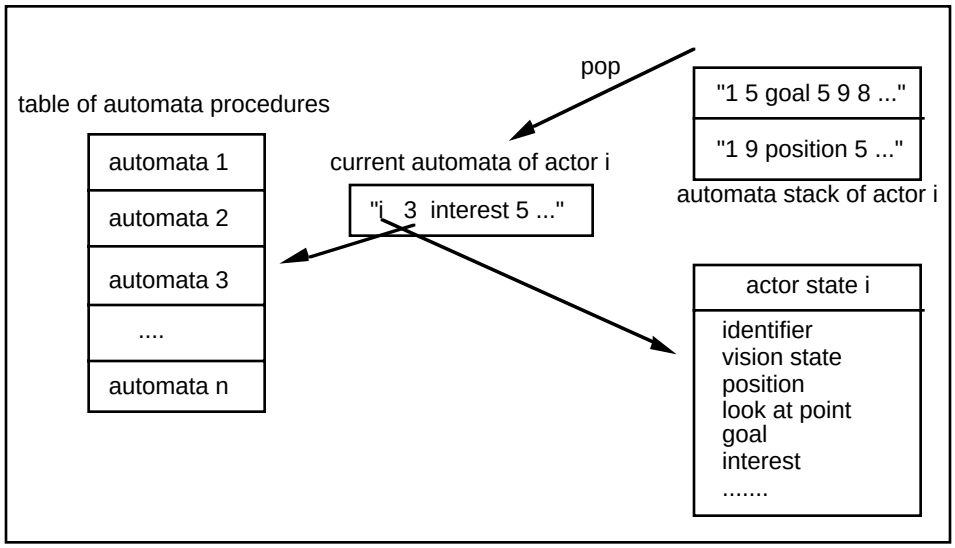


Figure 6.3.7: Architecture of the behavior control.

We implemented the behavior stack by a stack of strings. Each string identifies in its header the actor and the automata. The rest of the string contains commands to modify the actor state variables. An example is shown in figure 6.3.7. If an automata pops a behavior, the corresponding command string is parsed and interpreted. If a state variable is not affected, it maintains the actual value. The BNF of the text string automata is given in appendix A and its semantic in appendix B.

Some times the actor - turtle mapping through the symbol **M** should consider certain constraints concerning the height of the vision system or the horizontal orientation, for example. An actor can choose between five currently implemented turtle types, invoked by the Automata key word "turtle" followed by a number between zero and four.

- The turtle number 0, "turtle_navigate", places itself outside the scene when rendered in its proper vision window, otherwise, it is always placed at the position of the actor. The heading vector H is directed to the look at point of the actor when it is in the look_around state, otherwise it is aligned with the actor direction. The vector L of the turtle becomes horizontal.
- The turtle number 1, called "turtle_foothold", is used for the actor - turtle mapping of the skill corresponding to the walking on sparse foothold location. The mapping works as for turtle number 0, but with an additional material activation corresponding to the actual walk state of the actor. When the actor turns around, the red material is activated. The white and yellow materials are activated to show whether the actor poses the right or the left foot on the current foothold. The vector L is vertically oriented.
- The turtle number 2, called "turtle_racket", is used for the actor - turtle mapping used by the play_tennis and play_tennis_interactively automata (see table 5.1.1). When the 3D path spline for the execution of a hit exists, the actor is positioned and oriented according to this spline. As for turtle number 0, the heading vector will point to the next path point, and the L vector becomes horizontal. Additionally, the planned path is displayed in the user GL window. If no tennis spline exists, the mapping works as for turtle number 0.
- The turtle number 3, called "turtle_show_path", corresponds to turtle number 0, but it additionally displays the planned path in the user window displaying the scene with the actors for the user.

- The last turtle number 4, called "turle_racket_interactive", is used for the interactive tennis game behavior and it maps the interactive user, whose position and orientation can be manipulated with a spaceball, to the turtle. The heading vector H of the turtle points to the look at point of the interactive user, and the vector L becomes horizontal.

6.4 POST Speech Recognition

For speech recognition we use POST (Parallel Object oriented Speech Toolkit, [HD96], [HTTPpost]), a toolkit developed for designing automatic speech recognition. POST is distributed freeware to academic institutions. It can perform simple feature extraction, training and testing of word and sub-word Hidden Markov Models with discrete and multi Gaussian statistical modeling. The system can be trained by several users and its performance depends on the number of repetitions and the quality of word capture.

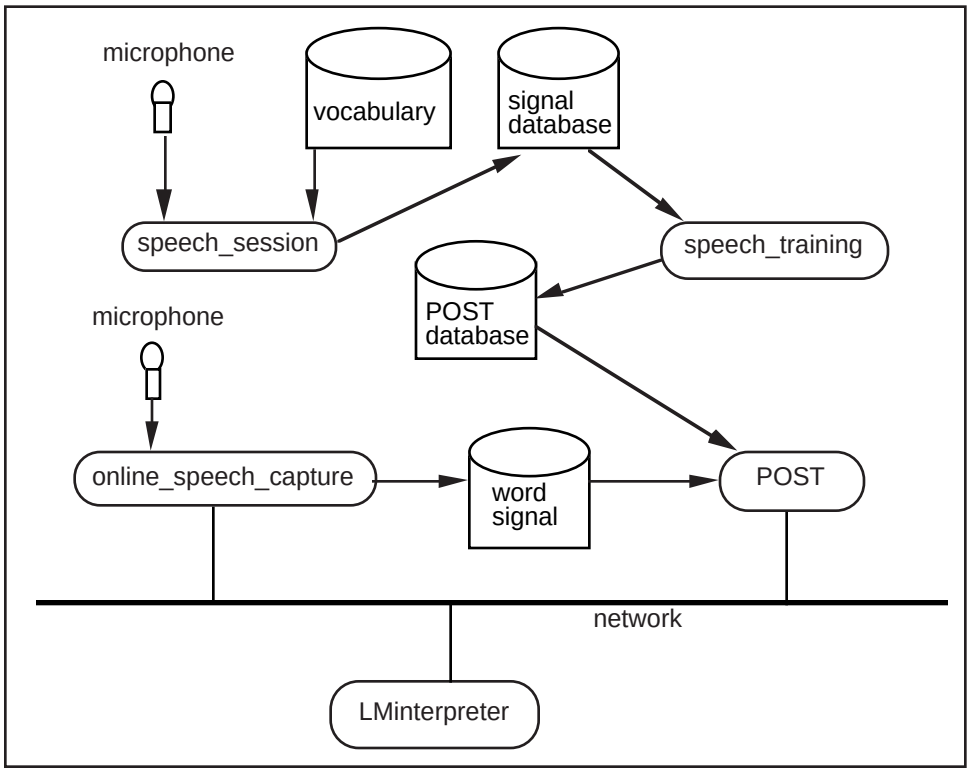


Figure 6.4: Detailed speech recognition architecture

A detailed system architecture is shown in figure 6.4. The program speech_session is used to create the signal database of the vocabulary for the system training. The vocabulary is read from a text file. The begin and the end of a spoken word is automatically recognized. After eliminating eventually present DC signal components, the word is added to a signal buffer. The whole session is written to a binary session file containing the raw signal of each word. For each session the program also creates an index file, containing the word names with the corresponding begin and end positions. For high recognition performances about 20 to 30 session files should be created. The script file speech_training is then executed to train and to test the system. It accesses the signal database containing the session files and produces the POST internal database necessary for on-line speech recognition.

During on line speech recognition the process online_speech_capture captures isolated words and saves them in a binary signal file. Through the net by using PVM (Parallel Virtual Machine) [PVM] it communicates the file name to the POST process that sends back the

recognized word. Then, the online_speech_capture process communicates the word to the L-system interpreter through simple socket communication.

When a user wants to employ the speech recognition features, he has to declare it in the “Options” section of the L-system definition file. With the line

AUDIOport_host int Word

the user specifies the socket port number and the host name of the workstation where the speech recognition module is executed. The L-system interpreter automatically starts then the on-line speech capture process and a POST process for word recognition. A recognized word is communicated to the L-system interpreter. This word is transmitted to the sound event handler after being matched to the actual sound library. Thus, the spoken word of a user is transmitted into the virtual acoustic environment and can be "heard" and "understood" by the autonomous actors. The current vocabulary of the speech recognition system (see table D.2 of appendix D) contains some commands like "move", "turn right" or "stop". We can use these commands to control the user camera. When the camera control symbol **l** (see appendix C) is present in a L-system definition, it filters out automatically these commands and executes them. Thus, a user can interactively manipulate the camera of the display of the virtual environment with spoken commands.

6.5 Image by Image Film Production

In general, image by image ray traced film production is a fastidious job for the user of an animation system, and it employs much computer system resources like disc space and computation time. For an animation system it is essential that it supports image by image film production. Therefore, we included in the animation system some versatile support for automatic image by image video sequence generation as indicated in figure 6.5.1.

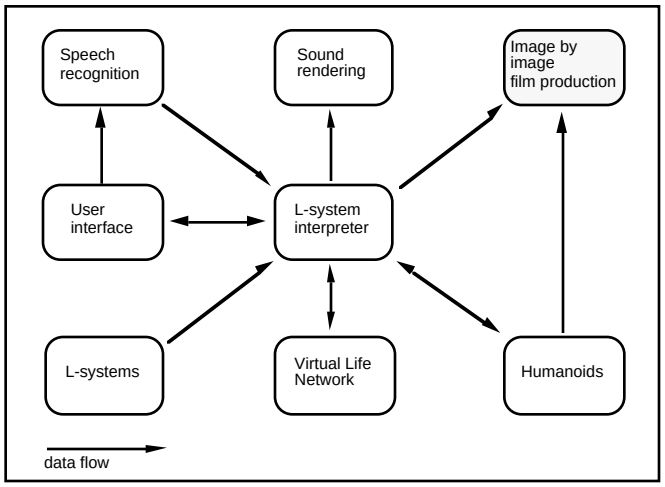


Figure 6.5.1: Image by image film production as external process controlled by the L-system interpreter

Like in traditional film productions the first shot of an animation sequence is in general not satisfactory, and the same sequence has to be recorded several times with changed parameters until the final product satisfies the film makers. Thus, it is preferable to produce first rapidly a flat shaded or Gouraud shaded film before starting the very time consuming ray tracing for the final rendering. When an animation turns in real time, there is no problem to evaluate a sequence directly on the computer screen. But when real time display for complex environments, for example, is not possible any more, the sequence has to be recorded image by image on a video device to get a real time play back for evaluation. With the command file feature of the L-system interpreter program an image by image recording of a screen portion

on a video device supporting this frame by frame facility is possible. First, the program LMinterpreter displays the frame on the screen. Then, in section 5 of the animation loop (see Code 2.1.1) a script file can be executed at each frame. This script file can be configured according to the special needs of a given film project. In appendix E a typical script file, "AFTER_ACCOM.CMD", code E.1, which records the images to an Accom WSD Workstation Disk, is given

The program LMinterpreter starts the C-shell script by passing the frame number to the script. The script saves the corresponding portion of the screen, converts it into a YUV format and copies it to the video disc.

If all the frames are recorded the sequence up to one minute can be played back in real time with the video device. When we have a good sequence, sometimes we want to save each frame on the computer disc for later backup on a tape. The script file "AFTER_RGB.CMD", code E.2, shown in appendix E, shows how to proceed.

If all the animation sequences are satisfactory, the ray traced film production can start. Now we have two possibilities. On the one hand we can produce first all the ray tracer description files with LMinterpreter. In a second pass we can render the files with the ray tracer. On the other hand with an appropriate script file we can produce directly the ray traced images during the animation. In both cases the program LMinterpreter has to be called with the option RS that produces numbered ray tracer definition files. The L-system definition file that is passed as first file argument to the L-system interpreter must contain the line

rs_file_path_name rs_file_path rs_file_name

which allows the user to specify the path and the root of the filename. LMinterpreter produces then the numbered text files rs_file_path/rs_file_name.xxx.ray in Rayshade format where xxxx is the four digit frame number.

Very often we have to render again certain frames or frame ranges of an animation because of errors occurred somewhere or bad image quality. To avoid a repetition of the whole animation we can choose some frame intervals for rendering. With the line

rs_switch min1 max1 min2 max2 min3 max3

placed in the “Options” section of the L-system definition file three frame intervals can be selected for Rayshade file production. Thus, in addition, some snapshots from the animation are possible.

Moreover, if we want to render the whole animation in one pass with the ray tracer, we can execute a script file of type "AFTER_RS_RGB.CMD" shown in Code E.3 of appendix E after each frame. This script file calls the ray tracer and compresses the final image. The now useless Rayshade file is removed to save disc space. We could also directly convert the rgb file to YUV format, transfer it to the Accom video disc and remove the now useless rgb and YUV files as seen in the commented commands.

Generally, however, for big film projects the ray tracer files are generated first. Then, they are distributed for rendering to several machines. The resulting images require often huge disc space. Therefore, they should be stored on intermediate storage devices as optical discs or tapes. Code E.4 shows a script that automatically backs up every set of the numbered rgb image files on a DAT tape. After having stored them, they are removed from the disc. Thus, about 2 GByte of images can be automatically produced with an intermediate hard disc space of only 20 MByte. The last code example of appendix E shows how to incrementally recuperate the image files from a DAT tape.

6.6 Hardware and Software

The L-system interpreter and its modules were developed on Silicon Graphics workstations especially on Indigo 2 (extreme, impact), Onyx, and older IRIS-4D/VGX machines with Z-buffer and hardware graphics engine. For correct operation of the synthetic vision module in double buffer mode, the frame buffer should have 24 bit planes in RGB mode. The graphics part of the L-system interpreter is based on Silicon Graphics "Graphics Library " (GL) supported by ANSI C and C++ programming languages. The standard audio hardware supplied with the IRIS Indigo 2 family of workstations supports 24-bit digital stereo and 16 bit analog stereo sound and a dedicated real-time processor works in tandem with the CPU to ensure that audio timing isn't degraded by other system demands. Some of the modules were written in ANSI C and can be used in other C applications or libraries. The L-system interpreter itself is written in C++.

For the file format of recorded sounds we use AIFF. Apple Computer's Audio Interchange File Format (AIFF) provides a standard file format for memorizing sampled sounds on magnetic media. Any number of channels of sampled sound at a variety of sample rates and sample width can be stored. It conforms to the EA IFF 85 Standard for Interchange Format Files developed by Electronic Arts. Silicon Graphics has adopted AIFF-C (supports compression and is backward compatible to AIFF) as its standard digital audio file format [SGI93].

We produced several ray traced videos with rendered soundtracks. As ray tracer, we used Rayshade from Craig E. Kolb (Yale University) available as freeware (ftp: princeton.edu).

7. Case Studies

The behavioral L-system animation system can be employed in education, research and film production. We used it primarily for research, for the development of sensor based autonomous actors in virtual environments with user interactions. New automata based behaviors can be incrementally added to the system and make the actors more and more sophisticated. We also modeled an interactive tennis game with autonomous actors as game partners. For some film projects we produced film sequences with L-system modeled and animated environments. Moreover, it allowed many students in semester projects to get familiar with L-systems and to design growing plants.

All of these animations are defined by one or several L-system definition files, interpreted by the L-system interpreter "LMinterpreter.x". The program can be started with the following command line:

LMinterpreter.x [options] $L\_system_1.lm$ ... $L\text{-}system_n.lm$

The options are:

- RS: the flag is necessary for image by image ray traced film generation
- N: numbers image file names in image by image mode
- PRINT: prints at each time step the actual formal parametric symbol strings
- SBUF: single buffer display mode
- SM: at frame number i a SM surface of the complete environment modeled by $L\_system_1.lm$ is generated and saved in the file "SM_object.sm"

In the following sections we present and illustrate some typical animations modeled with L-system definition files.

7.1 Force Field Animation

With the concept of tropism forces and the force field based particle system introduced in chapter 2.3 a variety of animation types are possible. In the following sections we show some force field models for physical and behavioral group animation. Code 7.1.1 illustrates how the force fields and particles are defined in a L-system definition file.

7.1.1 Tree - particle interaction

In this animation example, we show the interaction of a particle and a tree. The particle moves towards a tree. As it carries a repulsive "velocity" force field, the tree and the branches bend under its influence and try to avoid collision with the particle. The particle's force field is shown in equation.(7.1.1)

$$\vec{f}_{ball}\left(r, \dot{\vec{x}}\right) = (18 \bullet 3^{-0.005r^2}) \bullet \dot{\vec{x}} \qquad (7.1.1)$$

Here, the force field is always in the direction of the particle's velocity, even behind the particle, so wind, caused by the movement of an object is simulated.

Figure 7.1.1: Ball like particle passing over a tree.

Figure 7.1.2: Airplane passing over a tree.

Figure 7.1.1 illustrates a ball like particle passing over a tree and simulating wind. By activating tropism the tree bends under the particle's force field. Figure 7.1.2 and figure I.29 show ray traced pictures of an animation sequence where the particle's force field envelops an airplane passing over a tree. In this video sequence we additionally used sound rendering, which illustrates the Doppler effect, caused by a fast moving sound source, and the fact that the force field of a particle can influence a branched structure. After the growth of a tree and some flowers an airplane passes very close a tree that bends under the wind caused by the airplane.

7.1.2 Dynamics of a Tennis ball

In a realistic looking tennis game simulation the ball should move according to Newton's laws. We used the following particle definition to simulate the tennis ball in tennis game simulations. The ball's differential equation is given by the system of equations (7.1.2).

$$\vec{x}_{ball} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} : \text{ position of the ball}$$

$$\vec{f}_{ball} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} : \text{ no contribution to the global force field} \qquad (7.1.2)$$

$$\vec{g}_{ball} = -\dot{\vec{x}} * a + \begin{pmatrix} 0 \\ \frac{|\dot{y}|A + B}{1 + C|y|} \\ 0 \end{pmatrix}$$

The first term of g_{ball} simulates an air resistance proportional to the speed of the ball. The second term is responsible for a loss of kinetic energy when the ball bounces on the ground. This loss depends on the speed of the ball and its penetration into the ground ($y < 0$). With the constants a, A, B, and C the ball dynamic can be adjusted to a realistic looking movement. The mass of the ball can be set to an appropriate value. The above equations do not consider ball spin.

7.1.3 Net of a tennis court

In a tennis game simulation, the net in the middle of a tennis court should stop the ball in case of collision by absorbing a big part of the balls kinetic energy. The net exerts always a force on the ball in direction of its normal vector. In our case we placed the net in the y-z plane. Therefore, the repelling force is always in direction of the x axis and in opposite direction of the x velocity component of the ball

$$\text{if } ((\ 0 < y < \text{net\_height}) \ \&\& \ (0 < z < \text{net\_length}))$$

$$\vec{f}_{net} = \begin{pmatrix} \dfrac{(x-a)f - \dot{x}e}{\left(\dfrac{x-a}{b}\right)^{c} + d} \\ 0 \\ 0 \end{pmatrix} \qquad (7.1.3)$$

$$else \quad \vec{f}_{net} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

a: x coordinate of net position

e: damping factor at collision

f, b, c, d: variables for adjusting the force field which simulates the ball - net collision.

7.1.4 Ground and gravitation

In real world gravitation is omnipresent. All objects that are not supported fall down and stop at the ground. If in a virtual world simulation this gravitation effect is missing the simulation is not very convincing for an immersed user. Therefore, we used the following force field model for gravitation and ground repulsion forces for particles which are submitted to physical laws. In a tennis game, for example, gravitation lets the ball fall down. The ground pushes it back in case of collision. Gravitation and ground forces are modeled by equation (7.1.4).

$$\vec{f}_{ground\_gravitation} = \begin{pmatrix} 0 \\ -m_{ball} \cdot g + a \cdot e^{-cy} \\ 0 \end{pmatrix} \qquad (7.1.4)$$

g: gravitation acceleration (9.81 kg * m/sec^2)

m_{ball}: mass of the ball

a, b, c: variables modeling the ground collision force at y=0 acting on the ball.

There is only a vertical force component. Its first term represents gravitation and the second term pushes the ball back with an exponential force proportional to the penetration into the ground. The energy loss of the ball when bouncing is already contained in the balls equation (7.1.2).

7.1.5 Tennis racket

The racket of a tennis game is represented by a disc of radius r. Its movement is not determined by dynamics. It moves along the spline trajectory planned by the corresponding actor (see section 5.6.6). The racket, however, exerts a force on the ball according to equation (7.1.5). Its influence is limited by a sphere of radius r around the racket center. Inside the sphere the force acts in direction of the racket normal. It is a distance dependent exponential function containing a speed dependent damping factor.

r = radius of racket
$\vec{n}$ = normal of racket plane
$\vec{X}$ = racket center
$\vec{x}$ = position of tennis ball
a, b, p = force field modeling factors

$$
\begin{aligned}
&\text{if } \left(\left|\vec{X} - \vec{x}\right| < r\right) \{ \\
&\quad c = -\vec{n} \cdot \vec{X} \\
&\quad d = \left|\vec{n} \cdot \vec{x} + c\right| \\
&\quad \vec{f}_{racket} = ab^{-d \cdot p} \cdot \vec{n} \,/\, \left(1 + \left|\dot{\vec{x}}\right|\right) \\
&\} \\
&\text{else} \\
&\quad \vec{f}_{racket} = (0,\ 0,\ 0)
\end{aligned}
\qquad (7.1.5)
$$

7.1.6 Wind

To simulate wind any appropriate position and time dependent 3D vector force field can be used. Note, however, if in a tennis game simulation, for example, the wind is to strong and too irregularly, the performance of the tennis playing actors will suffer as it affects also the movement of the tennis ball.

$$
\vec{f}_{whirl} = \begin{pmatrix} \dfrac{-z}{(x + 50 - 1.8T)^2 + z^2 + 0.5} + a + b\sin(cT) \\ 0 \\ \dfrac{(x + 50 - 1.8T)}{(x + 50 - 1.8T)^2 + z^2 + 0.5} + d + e\sin(fT) \end{pmatrix} \qquad (7.1.6)
$$

Figure I.1 shows a pictures of an animation where the whirl wind modeled by equation 7.1.6 passes through a forest. This short ray traced sequence starts with about 150 growing trees. Then, a whirlwind passes through this forest and animats the branches and leaves of the tree.

7.1.7 School of Fish and Butterflies

The above force field examples illustrate the use of the particle system for physical modeling. In this section we show its use for behavioral group animation of a school of fish. New fish with mass m=1 are generated at the same place at a regular rate. Their repulsive force field bounding their geometric shape is given by equation.(7.1.7). All these fishes are attracted by a moving object, the bait, with an attractive force field at long distances and a

repulsive one at short distances (equation.(7.1.8)). The movement of each fish is damped by equation.(7.1.9). In figure 7.1.3 and 7.1.4 these functions are illustrated.

$$\vec{f}_{fish}(r) = -2500 \bullet 3^{-0.005r^2} \bullet r \bullet \vec{n}$$

with $\vec{r} = (\vec{X}_{object} - \vec{x}), \quad r = \left\| \vec{X}_{object} - \vec{x} \right\|$ and $\vec{n} = \frac{\vec{r}}{r}$

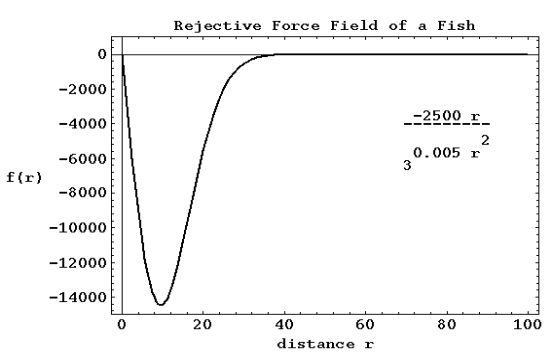
$\vec{X}_{object}$: Position of the object (fish) (7.1.7)

$\vec{x}$: position in the the force field

r : distance from the force field center of the object

$$\vec{f}_{bait}(r) = (4 - 2500 \bullet 3^{-0.005r^2}) \bullet r \bullet \vec{n} \qquad (7.1.8)$$

$$\vec{g}_{fish}\left(\dot{\vec{x}}\right) = -3\dot{\vec{x}} \qquad (7.1.9)$$



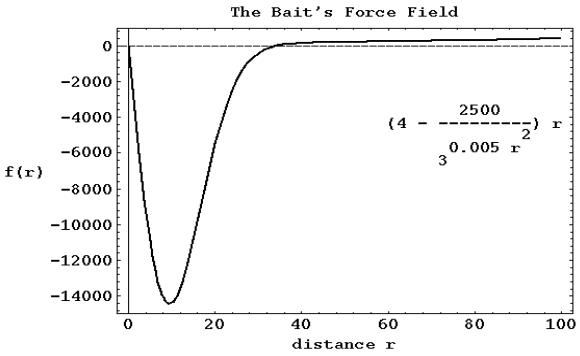


Figure 7.1.3: Repulsion force field of a fish

Figure 7.1.4: The bait's force field

The bait moves along a given 3D path spline. Each new generated fish joins immediately the school, following the bait. The strongly repulsive components of the force fields at short distances of all moving objects, prevent collisions. The modeling of the geometric shape of the fishes and their swim motor are programmed with production rules already treated in section 4.1. In figure 7.1.5 and I.2, we can see some pictures from the whole animation sequence. Figure 7.1.6 and I.3 illustrate a group animation of butterflies flying around a flower by using the same force field principles for group animation as for the school fish. The only difference is that the attracting force field mapped to the flower doesn't move.

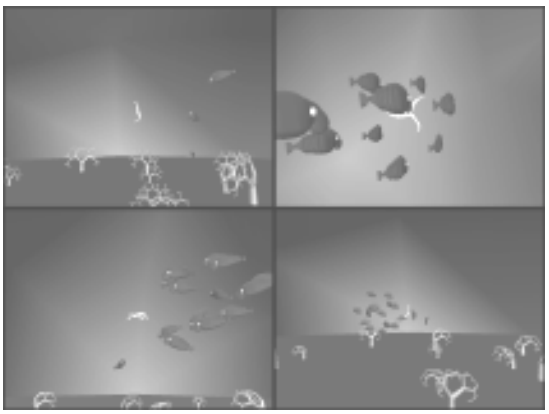


Figure 7.1.5: School of fish

Figure 7.1.6: Butterflies

The following lines show parts of the axiom and the production rules used for the above animation sequence.

```
Axiom
..... /* some symbols, describing the environment It follows the definition of the bait */

f ()    () () (2)       (0) (0) (0)
/* the symbol f places the turtle at (0,0,0) and determines thus the initial position of the bait */

e ()    (10) () ()((4-2500*3^(-0.005r^2) * (X-x))
                   ((4-2500*3^(-0.005r^2) * (Y-y))
                   ((4-2500*3^(-0.005r^2) * (Z-z))
/* Symbol e with parameter x=10 defines the bait's force field. (X,Y,Z) are the coordinates of the
current object (in this case the bait). (x,y,z) is the position of an object, feeling the force field. The
variable r is given by r = sqrt((X-x)^2 + (Y-y)^2 + (Z-z)^2), the distance of a fish to the bait. */

C ()   (1) () ()       (spline_1(t)) (spline_2(t)) (spline_3(t))
/* Symbol C with parameter x=4 places the turtle (= the bait) at the position (spline_1(t), spline_2(t),
(spline_3(t)) determined by a predefined timed 3D spline */

z ()    () () ()        () () ()
/* the symbol z is a germ for the bait. It is an arbitrary geometrical figure, defined by the corresponding
production rule, which is not given here. It is drawn in the coordinate system of the turtle */

x ()    () () ()        () () ()
/* The symbol x is a germ for a fish force field declaration, given in the production rule */
..... /* some more symbols, describing something arbitrary */

Production1
x (maximal_age = 1)      ----------------------> /* symbol x to be replaced by the following symbols */
/* it follow the parameterized symbols, defining the behavior of a fish */

f ()    () () (2)       (50) (50) (50)

/* the symbol f places the turtle (= fish) at the position (50, 50, 50) and thus determines the initial
position of a fish */

e ()    (10) () ()((-2500*3^(-0.005r^2) * (X-x))
                   ((-2500*3^(-0.005r^2) * (Y-y))
                   ((-2500*3^(-0.005r^2) * (Z-z))
/* Symbol e with parameter x=10 defines the fish's force field, felt by other objects. */

e ()    (11) () ()(-3u) (-3v) (-3w)
/* Symbol e with parameter x=11 defines the individual part of each generated fish of the
equation.(7.1.9). The vector (u, v, w) represents the velocity of the current fish. */

C ()   () () ()        (5) (3) (2)
/* Symbol C with parameter x=0 starts the evolution of the generated fish with the initial velocity (5, 3,
2)  */

y (1)  () () ()        () () ()
/* Symbol y represents a germ of the geometric shape and the swim motor of a fish with an initial age
of 1 */

x ()    () () ()        () () ()
/* The symbol x is a germ for a fish force field declaration, given in the production rule */
EndProduction1
```

Code 7.1.1: Pseudo code of L-system for group animation

7.2 Plants, Fractal, Crystals

In this section we illustrate some animations of plant-like, fractal-like and crystal-like objects. These kind of objects are typically in the tradition of L-system applications. Note, however, that the L-systems define not only the topology of such objects, but also their piece-wise continuos development, influenced partially by tropism forces.

7.2.1 Surreal L-structures

Surreal L-structures (surreal means surrealistic) is the title of one of the first demonstration videos (GL rendered, about 4 min. 40 sec) produced with an ancient version of the L-system interpreter. It illustrates some growing and moving fractals and plant like structures and trees. The trees are influenced by gravitation and wind force fields. Leafs are blown away and apples fall down being germs for new trees (see figure I.13). The music was composed by Russel Turner. Figures I.2 and I.4 show pictures from this video.

7.2.2 Plants, Crystals

The L-system interpreter was also used in several semester projects where students had the possibility to get familiar with L-systems and to model growth and topology of certain plants. These L-system defined plants can be employed in video productions to enhance the environment of autonomous actors. The color figures I.6 .. I.14 show some results. Figure I.13 shows a picture from a sequence of a growing forest which starts from a single germ. Germs are symbolized by apples and can fall from the trees. The trees are animated by gravity and wind force fields which can even strip off leaves and blow them away.

7.3 Navigation

Synthetic vision based navigation of autonomous actors in virtual environments is one of the main topic of this work. With the synthetic vision, the visual memory, the heuristic path searching and some navigation automata the L-system based animation system disposes now of versatile tools for high level actor navigation. In the framework of the animation system navigation for actors can be modeled by using the following built in facilities:

- Force fields
- Automata using vision, visual memory and heuristic path searching
- Vision based walking on sparse foothold locations
- Imported actor trajectories
- Local vision and production rules

Force field group navigation for fishes and butterflies was already discussed in section 7.1.7. The Pledge algorithm modeled by production rules and supported by local vision was treated in section 5.10. In the next sections we present some results of the remaining navigation techniques.

7.3.1 Navigation by automata

We produced several demonstration videos illustrating autonomous actors navigating from a given position to a goal position by avoiding collisions and incrementally updating their perceived environment which they completely don't know at the beginning of the animation. We simulated a 2D search, a 3D search and a conditional 3D search. In all 3 cases the actor had at the beginning no knowledge of its environment. In the 2D search, an actor was placed in the interior of a maze with an impasse, a circuit and some animated flowers. The actor’s first goal was a point outside the maze. For the search we used heuristic 1 (see appendix G).

After some time the actor succeeded in finding its goal. When it had completely memorized the impasse and the circuit, it avoided them. After reaching its first goal, it had a nearly complete visual octree representation of its environment and it could find without problems its initial position by a simple reasoning process. In this example it was never obliged to use the explore mode. Heuristic 1 guarantees, that we can always find a path, if there exists one. In the maze there was a wall, which disappeared after some time. When the actor passed there a second time, it eliminated it from its memory.

Figure I.15 shows vision based 2D navigation with heuristic path searching based on the visual memory of a cube like actor. It shows the vision window, the visual memory development with the planned path and the visual memory projected into a window for comparison with the actual perceived environment in the vision window.

In the 3D search illustrated in figure I.16 a flying butterfly searched its way through a flower field, animated by a wind force field. The search with heuristic 2 (see appendix G) was without problems and the complex flowers were represented in the actors visual memory by some bounding cubes of the octree space grid.

In another 3D conditional search, the actor had to avoid a circuit and 2 impasses. Then, it had to use a bridge and to mount a ramp to reach a higher leveled floor, which it had to follow to arrive at its destination (an animated tree). As the floor was not bounded by walls, it had to avoid to fall down. When the actor had reached its goal, it had a nearly complete visual octree representation of its environment and it could find without problems a direct way to its starting point. This last search had to be done in the explore mode. Its systematic character doesn't correspond to a typical human behavior. It can be used as a base for a more sophisticated local behavior of a humanoid actor. Environment exploring for humanoids could be a further research topic.

7.3.2 Vision Based Human Free Walking on Sparse Foothold Locations

We will discuss some features of the automata described in section 5.9, which concerns vision based path searching on sparse foothold locations as illustrated in figure I.17 and I.18 of appendix I. We have seen in the section "Planning" that the behavior of an actor depends locally on the path searching algorithm and globally on the navigation automata. The local path searching is optimal for a given cost function and avoids circuits and impasses in the neighborhood of an actor. The cost function strongly determines this local behavior. We chose the heuristic cost function h given in equation (7.3.1).

It is a weighted sum of the three parameters distance, speed_change and Turning_angle, raised to the corresponding powers n0, n1 and n2.

$$h(a, b) = h(\text{distance}, \text{speed\_change}, \text{Turning\_angle}) = p0*\text{distance}^{no} + p1*\text{speed\_change}^{n1} + p2*\text{Turning\_angle}^{n2}. \qquad (7.3.1)$$

The parameter distance corresponds to the actual distance of the two nodes (see equation (7.3.2))

$$\text{distance} = | a.\text{current} - b.\text{current} | \qquad (7.3.2)$$

Thus, if an actor looks for the shortest path to a given destination, it has to use this term with p0=1 and n0=1 and to eliminate the other contributions by assigning zero to the weights p1 and p2.

During normal walking, people try to keep their speed constant. Therefore, we introduced a penalizing term for speed changes. The parameter speed_change is calculated using equation (7.3.3). As the actor's speed is directly related to the distance of the corresponding foothold

locations, the speed change is proportional to the difference of the 2 foothold location distances.

speed_change = || a.current - a.previous | - | b.current - a.current ||(7.3.3)

Thus, we have a measure of accelerating and slowing down, which can be adjusted by the weight p1 and the exponent n1. This term, for example, penalizes the assembling of the feet on a node and favors a continuous walking with regular leg change. In normal walking, unnecessary direction changes are also avoided. That is why, we introduced the third term with the parameter Turning_angle. This parameter corresponds to the angle between the body directions of two consecutive steps and is described in more detail in the section "Walking Module" of [BNT93]. With this term we can minimize the total curvature of a path. As the current walking model is optimal for walking in a straight line, we use a quadratic Turning_angle term to prevent unnecessary direction changes and to favor leg assembling on a node, allowing a leg alternation to prevent unnatural sequences as indicated in figure 5.9.4 by the path without halting. With the parameter set,

p0 = n0 = p1 = n1 = 1

p2 = 5

n2 = 2

for example, we got satisfying results in our animation.

The behavior of an actor is also influenced by the precision of the vision system. With the voxelization of the environment we only get a more or less rough representation of the foothold locations and it can happen that some theoretically still reachable foothold locations are avoided, because their distance in the "local_map" is bigger than Df_max. In general, such an imprecision is not very disturbing in animation sequences as also real human actors are not perfect and as they don’t use always optimal paths. For a 50 x 50 pixel image as vision window for an actor, we got satisfying results with a voxel size of 0.05 to 0.1 (by Ht normalized units).

We produced a video (GL rendered, about 3 min.), that shows how the vision system creates the map of the sparse foothold environment and how a cube like actor follows the planned path. The actual color of the actor indicates the left or right foot used for the current foothold location (white = right, Yellow = left, red = turn around). Figure I.17 and I.18 show snapshots from this video. The implementation of a corresponding walk motor for humanoid actors is left for future work.

7.3.3 Imported Actor Trajectories

Another video sequence (Rayshade rendered, about 30 sec) illustrates the possibility of the animation system to import actor trajectories realized by other programs and used in the L-system interpreter in a second pass to animate actors in a L-system modeled environment. This video sequence shows three humanoid actors walking on a free place in front of a L-system modeled flower field. The trajectories of the actors have been imported from an application written by Pascal Becheiraz during his diploma project. Figure I.19 shows a snapshot from this film.

7.3.4 Navigation by Production Rules

Several video sequences with navigating humanoids illustrate the modeling of actor behaviors by production rules. Their local vision based navigation is modeled by production rules that simulate the Pledge algorithm (see section 5.10.1) allowing an actor to escape from a maze. In the first navigation sequence we see an actor walking in a maze. In the second sequence it walks through a flower field. The color figures I.6 and I.20 to I.22 show extracts from these videos.

7.4 Tennis

The Tennis game facility illustrates the systemic character of the animation system. It makes use of most of the implemented sub-systems. It can be used for interactive games, for networked interactive games and for image by image ray traced video productions of games with autonomous actors in an L-system modeled environment with sound rendering. It employs several autonomous actors playing different roles (referee, players). The behavior of the autonomous players, for example, is composed of several lower level automata and some production rules. Moreover, the simulation makes use of all parts of the virtual environment, namely the physical, acoustic and geometric modeling. All of these simulations, interactive, networked or ray traced, are defined by similar L-system definition files. We produced several video sequences of tennis games presented in the next sections.

7.4.1 Tennis with Autonomous Actors

In this section we describe a typical ray traced video production with a rendered sound track, where two racket-like actors play tennis. Using force fields, associated to the rackets, they can hit or push the ball, which is submitted to air resistance, gravity, and ground, net and racket force fields (see also sections 7.1.2 to 7.1.5). The vision system of an actor allows to extract and localize the ball and the game partner by color coding and the access to Z-buffer pixel values of the vision window. The reasoning part of the actor is able to elaborate a game strategy, to estimate the hit point and the hit time and to plan its own trajectory in order to hit the ball at the right moment, with the right speed in the right direction. In addition, the actor is able to adjust the hit force according to the last perceived ball exchange by comparing the perceived actual impact point with the planned one. More details about the tennis playing actors can be found in later chapters.

In the numerical integrator and the vector force field functions we have added peak detectors as shown in figure 7.4.1).

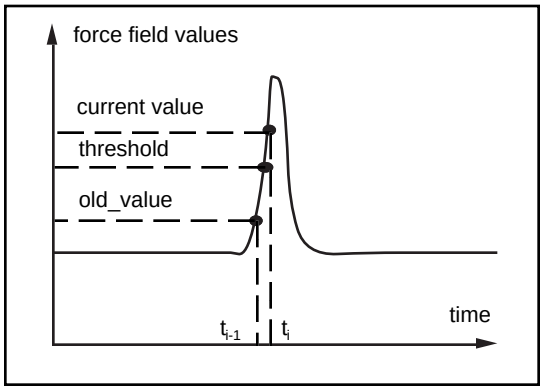


Figure 7.4.1: A peak detector detects a peak in a force field function. When a repelling force passes a threshold value during a collision, a sound event is produced.

Each time the ball touches the ground or the racket, the repelling force field produces a peak that signals a collision. Thus, the peak detector produces a sound event (ball - racket or ball - ground collision) and transmits it to the sound renderer, which mixes the collision sounds to an ambient sound that was started at the beginning of the animation. The sound renderer produces a synchronized sound track, which can be added to a ray traced video sequence of the whole simulation.

The sound rendering was performed using a SGI Indigo2 150 MHz Extreme workstation with an Iris Audio Processor. The rendering, the playback and the writing to an AIFF file could be done in nearly real-time by the same workstation. The sound events were mixed to a background sound and rendered for a moving microphone at the camera's position. The resulting one channel soundtrack lasted 40 sec. We used a sampling frequency of 22050 Hz.

The sound track was stored in a AIFF file containing 882882 sample frames (16 bit inter, big endian or 2's complement) or 1724.5 kBytes.

Figure 7.4.2 shows some pictures from the produced video. The numbering order of the pictures is one to nine from left to right and from top to bottom. Pictures 1 to 4 show the ball passing the net, touching the net, colliding with the net and bouncing on the ground. Picture 5 shows the actors (rackets) moving to their start position controlled by their navigation automata. Pictures 6 to 9 illustrate the game controlled by the tennis game automata.

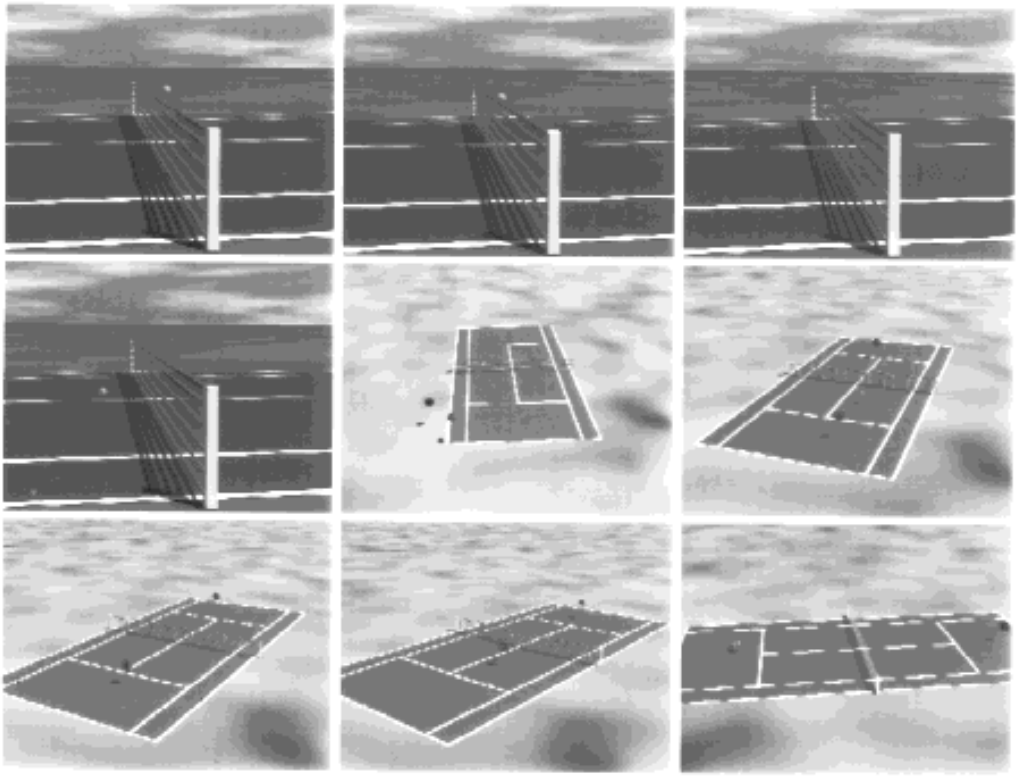


Figure 7.4.2: Some pictures of a tennis game of autonomous racket like actors:
Picture numbering scheme: 1 to 9 from left to right and from top to bottom.
1. the ball passes the net
2. the ball touches the net
3. the ball collides with the net
4. the ball bounces on the ground
5. the two actors (rackets) move to their start position
6 to 9. some pictures illustrating the game

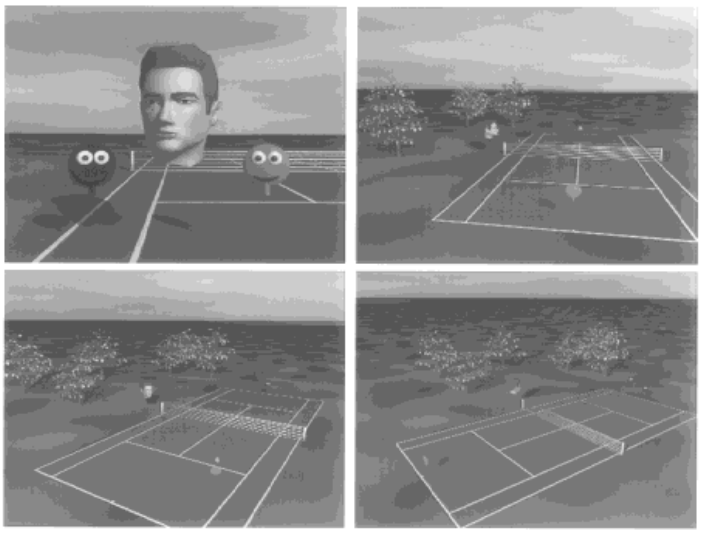


Figure 7.4.3: Some images from a ray traced tennis match with synthetic players and a synthetic referee (see also color figures I.23 and I.24).

Another video (Rayshade rendered, about 40 sec) shows a tennis game with 2 racket like players and an autonomous referee judging the game by using sound events and its vision system. It announces its decisions by spoken words. The sound track is rendered. Next to the tennis court, there are some trees animated by wind like force fields. Figure 7.4.3 shows 4 images from one of the videos where the referee is represented by the head that is a surface imported into the L-system. The blue and the green actors and the rest of the environment are modeled by the L-system.

7.4.2 Interactive Tennis through VLNET

As described in section 4.3 we added to our animation system an interface to VLNET. With this interface the autonomous actors can now participate as any other interactive user of VLNET in shared virtual environments. We wrote an L-system definition file, which branches an autonomous referee and a player to VLNET, and allows a remote user to play against a synthetic partner. With a special command file (see code F.2) the L-system interpreter automatically starts the VLNET server and client processes which are necessary for the game. The two client processes are the referee, judging the game, and an autonomous player, playing against a remote user. Figure 7.4.4 illustrates the resulting process configuration.

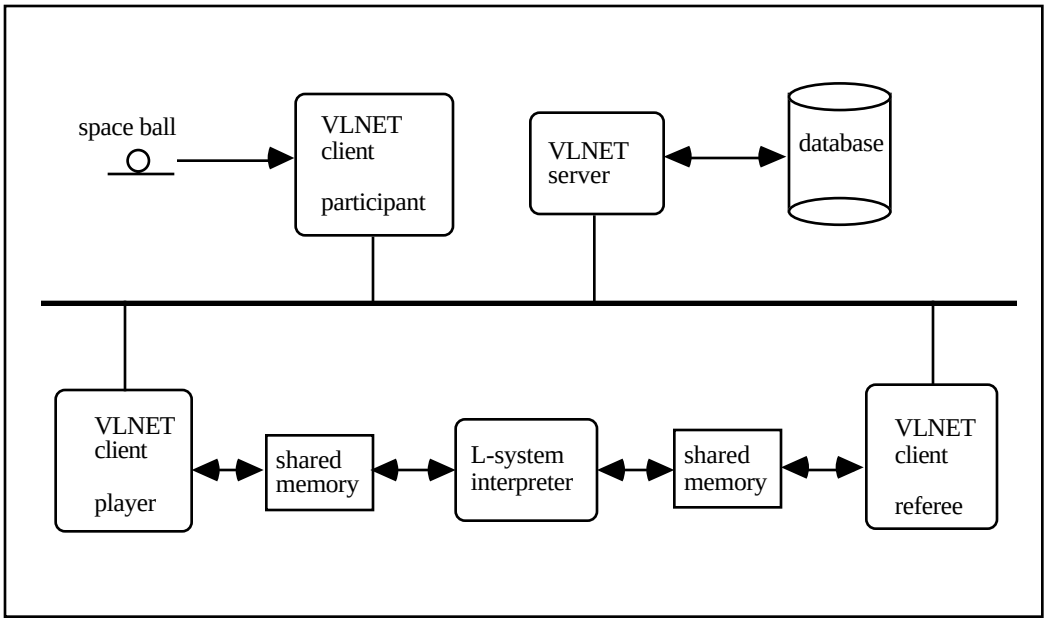


Figure 7.4.4: The process configuration for the interactive tennis game.

In collaboration with Tolga Capin and members of LIG and MIRAlab we produced a video (GL rendered, about 2 min.) that shows a networked interactive tennis game with an interactive user, an autonomous player and an autonomous referee judging the game. Figures I.25 to I.28 show snapshots of the video.

This video shows also the limitations of such applications. First, networked applications with fast moving objects like the tennis ball are not ideal as the update rate is to small. At least 10 to 15 frames per second would be necessary for a believable game speed. On the ONYX from SGI we only got frame rates of about 5 frames per second. Second, when the network is charged we should have some synchronization mechanisms to avoid update delays which lead to locally inconsistent shared environments. Finally, the shared memory interface is to restricted to model sophisticated actor movements like executing a backhand stroke. There are still many topics left for future work.

8. Conclusion

We have presented a versatile behavioral L-system animation system based on force fields and synthetic sensors for autonomous actors. In this system, high level physical and behavioral animation is possible. Vision based actors find their way without collisions in a L-system defined environment to given destinations. Dynamically created objects, moving in complex 3D vector force fields can interact with each other and branched structures, simulating behavior of herds, flocks, schools or some physical effects.

We think, that the behavioral L-system animation system is a convenient environment for simulating behavioral animation. By behavioral animation we normally consider the simulation of the behavior of living creatures and their reaction to their environment. In a larger sense, we can attribute a behavior to each object. Inanimate objects move (behave) in their environment according to physical laws. Plants, such as all other living creatures, obey also Newton's laws (gravitation, wind, ..), but besides, they grow, reproduce themselves and react to their environment (light, temperature, ...). Moreover, the behavior of animals is influenced by emotional and instinctive reactions. A predator, for instance, is attracted by its victim, which on the other hand is repelled by the predator. For most of the humans and a lot of animals their behavior is strongly determined by their vision, audition and sense of touch, which are very important perception channels. Finally, the highest level of behavior includes intelligence that is mainly attributed to humans.

With our current L-system based animation system, simulations in all of the above mentioned domains can be realized and combined in one sequence. As equation (2.3.1) is based on Newton's laws, physical simulations are immediate. Growth and reproduction features are inherent in L-systems. We can realize, for example, animation sequences of a growing tree, producing apples that contain germs for new trees. These germs develop to new trees, after the apples have fallen to the ground. Besides the physical modeling, the force field approach allows as well to simulate instinctive or emotional behavior (attractive , repellent forces) of individuals or groups and to interact physically with plants and inanimate objects. The sensor based features, allow the actors to avoid collisions and to simulate intelligent behavior. The global navigation automata represents 'intelligent' behavior. It allows an actor to find, for example, the exit of a maze, even if there are impasses and circuits, and to memorize the topology of the seen environment. An actor can use this learned information in future path searching. The synthetic vision module represents a very universal and powerful tool for future behavioral models. By designing and combining new automata and including more sophisticated actors using walking, speaking and grasping high level script based animation can be realized in a extremely dynamic environment with autonomous actors.

8.1 Future Work

The L-system interpreter was conceived to be a research tool for sensor based behavioral animation. Its structure is open for extensions, especially for symbol and automata (behavior) complements. The sensor modules developed with the L-system interpreter are also used in the humanoid library AGENTlib [EAgent8125], which will be extended in the next months by a set of new actions as facial actions, grasping and play back of pre-recorded skeleton animation sequences. As there is already a shared memory interface (see section 4.2) to an application of the AGENTlib library, it is obvious to update the system with the new actions and to define new behaviors including these new actions. The same arguments are also valid for the VLNET interface described in section 4.3. In the next subsections we will discuss possible extensions that could be interesting for further research.

8.1.1 Olfaction

After vision, audition and sense of touch it would be interesting to add olfaction as fourth sense for the autonomous actors. This sense plays an important role in human behavior and it could significantly enrich immersive virtual environments. In [BD96] the authors state, that currently, the hardware to produce virtual environments doesn't include olfactory displays, in spite of the fact that a wealth of sensory information is available from olfaction. They review the physiological and psychological aspects of olfaction and discuss the use of olfactory displays for virtual environments. Their paper outlines the information that can be useful for the virtual environment research community to integrate olfactory stimuli into virtual environments.

Based in this information an olfactory environment model should be developed together with synthetic olfactory sensors for the autonomous actors. When designing the olfactory environment model, special attention should be paid to maintain compatibility with future immersion of real actors using some virtual olfactory displays that probably will be developed in the near future.

8.1.2 L-system User Interface

In the current system an L-system is defined by an axiom and a set of production rules written by the user with a text editor. The parametric symbols are represented as proposed in [PRUS90] by alphabetic symbols. This coding of L-systems results in cryptic descriptions that are difficult to read for non experienced user who are not familiar with the semantic of the symbols. In [PRUS90, p 27] the authors proposed to replace the symbols by self explaining identifiers to improve conviviality, like the symbol "[" by the identifier "push" or the symbol "s" by an identifier "sphere". In the current version, we also use parameters of some symbols in two ways. First, they serve as normal parameters used to propagate values to other symbols and second, to modify the semantic of the symbol. Conceptually, it would be better to eliminate the parameter ambiguity and to define for each signification its own symbol. To avoid the run-out of the symbols of the alphabet we could identify the formal symbols by a unique self explaining identifier and associate it to a symbol number used for the implementation. Then, the parameters of symbols could be completely used for their original purpose, the transmission of values. To improve the conviviality, we could also replace the post fixed polish notation for numeric and logic expressions by a normal notation that is easier to read for users. This, however, implies a conversion of these expressions to polish notation during the parsing phase of the L-system to avoid a performance degrading of the evaluation of the expressions during a simulation.

In the current version of the L-system interpreter the number of parameters and growth functions for the symbols is fixed. It allows a fast accessible and relatively simple table based representation of the symbol data structure. With a dynamic symbol data structure the number of the parameters and growth functions could be personalized, which would improve also the syntax in the L-system interface by avoiding the empty parenthesis for the unused parameters and growth function expressions of the parametric symbols.

8.1.3 Evolution of L-systems

L-systems can have a considerable data amplification factor in the design of plants, environments and behaviors. However, it is not trivial to program with rules and to predict the results of a final animation. It would be of interest to have a possibility to evolve automatically L-systems to get better or new results. In [SIM91] Sims proposes that artificial evolution has potential as a powerful tool for achieving flexible complexity with a minimum of user input and knowledge of details. He describes evolutionary techniques to create complex simulated structures, textures and motions for use in computer graphics and animation. With interactive selection, based on visual perception of procedurally generated results, the user directs simulated evolution in preferred directions.

We think that there is an important potential for using genetic algorithms in the optimization and evolution of L-systems. We see applications on three different evolution levels of the L-system. The first level consists in parameter optimization through genetic algorithms. As a L-system is highly parameterized, an optimization by hand is difficult and time consuming. The navigation, for example, depends on all of the parameters of the vision system, the visual memory, and the actors speed. With a genetic algorithm and an appropriate fitness function, which could be the escape time of a maze, for example, a nearly optimal set of parameters could be evolved automatically. Besides the optimization of behaviors, similar to [REY93], we could also optimize plants or other L-structures by parameter variations. In this case, the user could also play the role of the fitness function by choosing his favorite structures for each new population formation. As L-systems are already defined by text files containing all the parameters, it would be straight forward to write a general genetic algorithm system that could create new populations of L-systems with modified parameters. Any subset of parameters of the functions, expressions or automata could be easily optimized by a system, which then, automatically measures the fitness of each member of the population and creates new populations based on the fitness results.

In a second evolution level, the expressions of conditions, parameter expressions and growth function expressions of L-systems could be evolved, leaving a vast field of future research in automatic L-system development.

The third evolution level consists in modifying the production rules itself. With crossover and mutation operators applied on the symbol strings of the axiom or the right sides of the production rules, which can be interpreted as chromosomes, new populations of L-systems could be created. At this level new L-structures with a changed topology and / or new behaviors could be derived. These three levels of evolution could be used separately or together in one evolution cycle. We think the genetic evolution of L-system chromosomes at these 3 levels represents an important and interesting potential for further research.

8.1.4 Particle System

Physical based movements in animation look natural and are appreciated by users of a real time applications or a film public. The currently implemented particle system is only useful for a small number of particles for real time application and film productions as the calculation time increases proportional to the square of the number of particles, and as the force field evaluation is done by the interpreter. An optimized particle system with only certain types of spring and short range force field interaction combined with an octree space grid would considerably increase performance and allow to add simple physically based real time actors like snakes, worms or jumpers. More complex elastic objects could be added. For film sequences complex systems up to 10000 particles could be evolved in a reasonable time.

In this context it would be interesting to extend the L-system with symbols allowing to define the particle system topology, the spring force connections of particles and the short range force fields of particles by production rules. By defining some spring like motor forces between particles in the L-system definition file and the above described genetic algorithm extensions, moving creatures similar to those described in [SIM94] could be evolved.

8.2 Epilogue

A typical feature of a behavioral animation system for humanoids is the fact that it will never be finished. As it tries to model reality there will always be new topics because reality is too complex to be modeled in all details on computers. However, it is a fascinating challenge to model some aspects of reality and human behaviors, and perhaps such work brings us a bit closer to the understanding of notions as life, individuality, mind and self consciousness.

Appendix A: L-system BNF

Appendix A contains the description of the syntax of L-system definition files by Backus Naur Forms (BNF's). In all subsequent BNF definitions all the keywords (as Camera_AnimationParameters, for example) and the following types are terminal:

int: integer type

float: floating point type

Word: a string without blank, carriage return, or tab separators.

Non terminal productions are printed in bold. The symbol ' |' expresses alternatives and the Kleen star * a list of items which can be empty. The upper script $^{+}$ indicates an non empty item list. Productions like **Item*** and **Item**$^{+}$ are recursively defined by

Item* ::= empty | **Item Item***

Item^{+} ::= **Item Item***

Item ::= .....

L-system ::= **Cam_Anim RepPar Opt Spl Const Surf Ax Prods**

Cam_Anim ::= Camera_AnimationParameters |
Camera_AnimationParameters
pref_pos_x1_x2_y1_y2 int int int int
observer float float float
twist int
viewAngle int
aspectRatio float
near float
far float
swapinterval float

RepPar::=
RepetionParameters |
RepetitionParameters
rotate float float float
rotate_max float float float
translate float float float
translate_max float float float
scale float
scale_max float
time_scale float
time_const float
startp_sidel_nbr_dist float float int float

Opt ::= Options **TheOptions** *

TheOptions ::=
MaxPartint |
command_file int Word Word Word |
fConst int float float float float float float float float |
VLNETdeclare **VlnetInit** |

AUDIOport_host int Word |
NBR_actors int |
iVisionWinSize int int int |
actorNbr_automata int **Automata** |
Sound_event_file Word **Sound Protocol** |
rs_file_path_name Word Word |
rs_switch int int int int int int |
NbrChannels_listOfNames int Word* |
trace_actor int Word |
import_trace int Word

VlnetInit ::= Nbr_exported_actors int **KeyId** *
Nbr_imported_actors int int*
Nbr_objects int int*

KeyId :: = int int int /* body key, object key, actor identifier */

Automata ::= int int **Commands** *

Commands ::=
coll_test_func int |
heuristic int |
rel_search_level |
goal float float float |
local_goal float float float |
eps_arrived float |
eps_look_at_goal float |
reset_goal |
turtle int |
mem_displ_cubes |
mem_no_displ |
mem_displ_points |
vis_mem_on |
vis_mem_off |
obs float float float |
lat float float float |
align float float float |
init int float float float |
fov int |
interest int int |
size_actor float |
dist_coll float |
tennis_state int |
state int |
optim_rate int |
speed float |
inertia float |
ahead int |
optim_ahead int |
vision_range float float float |
motor_init float float int

SoundProtocol ::= **SoundItem**$^{+}$

SoundItem ::=

INIT_source int float float float float float float |
INIT_mic int float float float float float float |
PLAY int Word |
SP int float float float |
SO int float float float |
MP int float float float |
MO int float float float |
Time float |
CONST_S int float float float float float float

Spl ::= Splines **Splin***

Splin ::= function time_spline int int float float **FloatPair**$^{+}$ 1e30 1e30

FloatPair ::= float float

Const ::= Constants **Flist** *

Flist ::= **Fbinary** | **Fternary**

Fbinary ::= **Id_binary** float float

Fternary ::= **Id_ternary** float float float

Id_binary ::= a..q, s, A..D

Id_ternary ::= E..S

Surf ::= Surfaces **Surface***

Surface ::= fig_sm int word

Ax ::= Axiom **ParametricSymbol**$^{+}$

ParametricSymbol ::=**Symbol Expr Expr Expr Expr Expr Expr Expr**

Symbol ::=[|] | { | } | / | ^ | & | $ | + | - | a .. z | A .. Z | 0..9 |

Expr ::= () | (**Expression**)

Expression ::= **Expression Expression Bin_op** |
Expression Un_op |
Constant | **Function** | **Parameters**

Parameters ::= t | T | x | y | z | X | Y | Z | u | v | w | r

Function ::= Id_binary | **Id_ternary** | ! | # | ?00..?99

Constant ::= **Float** | **Integer**

Float ::= 0 | 0.**Digit0*** | **Integer**.**Digit0***

Integer ::= **Digit1 Digit0***

Digit1 ::= 1..9

Digit0 ::= 0 | **Digit1**

Bin_op ::= + | **-** | * | **-** | ^ | % | ~

Un_op ::= | | _ | $ | & | : | ”

Prods ::= Productions **Prod*** EndProductions

Prod ::= **LeftSide Expr LogicalExpression LeftContext RightContext RightSidePart** * End

LeftSide ::= **Symbol**

LogicalExpression ::= () | **LogicExpression**

LogicExpression ::= **LogicExpression LogicExpression Lbin_op** | **LogicExpression Lun_op** | [**Expr Pred**]

Pred ::= < | p | > | g | = | !

Lbin_op ::= & | ' |' (and | or)

Lun_op ::= ! (not)

LeftContext ::= (**Symbol**) | ()

LeftContext ::= (**Symbol**) | ()

RightSidePart ::= float **ParametricSymbol*** EndProb

Definition A1: The complete BNF of a behavioral L-system definition file.

Appendix B: Semantic of BNF

In Appendix B we present the signification of BNF productions.

BNF	Comments
L-system	The L-system definition
Cam_Anim Camera_AnimationParameters pref_pos_x1_x2_y1_y2 int int int int observer float float float twist int viewAngle int aspectRatio float near float far float swapinterval float	The camera and animation parameters a keyword The position of the user window on the screen The position of the camera The twist angle of the camera The view angle of the camera int tenth of degrees The aspect ration of the user window The near clipping distance The far clipping distance The GL swap interval of the frame buffers
RepPar RepetionParameters rotate float float float rotate_max float float float translate float float float translate_max float float float scale float scale_max float time_scale float time_const float startp_sidel_nbr_dist float float int float	The parameters that determine how a L-system is repeated a keyword minimum and maximum values of stochastic rotations around x, y and z axes minimum and maximum values of stochastic translation vectors minimum and maximum values of scaling scaling factor for the time if the value is <= 0.0, then the derivation is done normally, otherwise the derivation stops for global times of the animation loop bigger than the value (start point) (side_length) (nbr_of_objects) (max_distance). Defines the square area with a given side length in which the L-system is repeatedly (nbr_of_objects) drawn with a given minimal normalized distance between each object. The area in the x, z plane is defined by the two diagonal vectors (startpoint, 0, startpoint) and (startpoint + side_length, 0, startpoint + side_length)
Opt ::= Options **TheOptions** *	The options section
TheOptions MaxPart int	the different options declares the maximal number of particles

command_file int Word Word Word	(number) (script name) (argument1) (argument2) Declares a shell or C-shell command file that can be executed through the symbol **E** described in section 2.2.2. The number of the script file. When defined, the script file 0 is automatically executed at the beginning of each frame, and the script file 1 at the end of each frame. The current frame number and the last two text strings are passed as arguments to the command file.
fConst int float float float float float float float float	Some of the specialized functions of parameter expressions, growth functions or production rule conditions evoked by "?xx" (xx = 2 digit number) access these constant arrays of 8 float components numbered by the first integer value. For details see table "B.5 Special functions".
VLNETdeclare **VlnetInit** AUDIOport_host int Word	Declares the VLNET interface. (port number) (host name) Declares the use of the speech recognition module by defining the port number and the host name
NBR_actors int	Declaration of the number of actors used in a simulation (upper limit = 6).
iVisionWinSize int int int	(actor_nbr) (window_edge length) (max. colors)
actorNbr_automata int **Automata**	A text string automata (see section 6.3.4) is pushed on the behavior stack of the actor referenced by the integer value
Sound_event_file Word **Sound Protocol**	Declares the sound rendering. All sound events are written into the file named by the first argument.
rs_file_path_name Word Word	Declares ray traced image by image rendering. The arguments are the file path and its name.
rs_switch int int int int int int	Allows to define 3 frame intervals for which the ray traced rendering is done
NbrChannels_listOfNames int Word*	Declares n (= the first int argument) imported data channels (see section 6.3.3). The list of Word arguments is composed of the n corresponding file names
trace_actor int Word	Declares an exported actor trajectory (see section 6.3.3). The first int argument is the actor number. It is followed by the corresponding file name.
import_trace int Word	Declares an imported actor trajectory (see section 6.3.3). The first int argument is the actor number. It is followed by the corresponding file name.
Spl ::= function time_spline int int float float **FloatPair** $^{+}$ 1e30 1e30	The first integer numbers the spline, the second gives the number of float pairs (argument, value) to follow. The two float values scale the argument and the value of the spline.

FloatPair ::= float float	(argument, value) of a spline key point
Const ::= Constants **Flist** *	the section of the definition of function parameters
Flist ::= **Fbinary** | **Fternary**	functions with two or three parameters
Fbinary ::= **Id_binary** float float	functions with two parameters
Fternary ::= **Id_ternary** float float float	functions with three parameters
Id_binary ::= a..q, s, A..D	identifiers of binary functions
Id_ternary ::= E..S	identifiers of ternary functions
Surf::= fig_sm int word	The first argument numbers the imported sm surface and the second argument is the file name of the surface including its path
Ax ::= Axiom **ParametricSymbol** *	The axiom of the L-system
ParametricSymbol	A parametric symbol
Prods ::= Productions **Prod** * EndProductions	The productions of a L-system
Prod ::= **LeftSide LogicalExpression LeftContext RightContext RightSidePart** * End	A production is given by the left side of a production rule, a condition, the left and right context and the right side parts of the production rule
LeftSide ::= **Symbol**	The left side of a production rule is a symbol to be replaced
LogicalExpression	The condition of a production rule
RightContext ::= (**Symbol**) | ()	The right context is a symbol or empty
LeftContext ::= (**Symbol**) | ()	The left context is a symbol or empty
RightSidePart ::= float **ParametricSymbol** + EndProb	A right side part of a production rule is a probability and a string of symbols

Table B1: The semantic of a L-system definition file

Bin_op	**binary operators**
+	addition
-	subtraction
*	multiplication
/	division
^	power
%	modulo
~	is a binary step function returning 0 if (a<b) or (a<0), 1 otherwise with b = top of the stack and a = top-1 of the stack.
<	sets the symbol memory (number stack[top-1]) with stack[top]
?50	is a binary function that pushes the channel value of the argument stack[top] of channel number stack[top-1] on the stack after having popped the two arguments from the stack (see also section 6.3.3

Table B2: Binary numeric operators

Un_op	**unary operators**
|	absolute value
_	minus

$	sinus (argument in degrees)
&	cosine (argument in degrees)
:	$\sin^{-1}$ (returns degrees)
"	tanh (returns rad)
>	puts the symbol memory (number stack[top] on the top of the stack
?03	spline 3 with the top of the stack as argument
?29	returns the Hermite function Hermite(stack[top] + c3) (see section 2.3.3) with the actual top of the stack as input argument and the coefficient array Const[5] with Const[5][**i**] = ci c0: TT (growth duration) c1: min. c2: max c3: offset
?44	returns the symbol's memory value indexed by the top of the stack

Table B3: Unary numeric operators

Pred	**predicates**
<	smaller than 0
p	smaller than 0 or equal to 0
>	bigger than 0
g	bigger than 0 or equal to 0
=	equal to 0
!	not equal to 0

Table B4: Predicates of logic expressions

Function	**some special functions**
a .. q, s, A .. D	linear functions of the global time T, defined by 2 coefficients: a = a0 + a1*T
E .. G	quadratic functions of the global time T, defined by 3 coefficients: E = E0 + E1*T + E2*T^2
P .. S	exponential functions of the global time T, defined by 3 coefficients: P = P0 + P1*e^{P2*T}
K .. O	defined by 3 coefficients: K = K0 + K1*sin(K2*T) (degrees)
H .. J	defined by 3 coefficients: H = H0 + H1*cos(H2*T) (degrees)
!	random number between 0 and 1
#	random number between c0 and c0 + c1
‘	has no effect, can be used as separator in expressions
?00	spline 0 with time as parameter, initialized by the line function time_spline Nbr_spline Nbr_points scale_time scale_y ... dt dy (Options section)
?01	spline 1
?02	spline 2
?08	The function returns the stack top x if x is an element of the interval [0,1], zero if x<0 and one if x>1

?09	is used for force fields. Refers to parameters x, X and r. It returns 1 if r <1e-20, (X-x)/r otherwise. In tennis game simulation, X is the racket center, and x is the ball's x coordinate
?12	- is used for a simulation of the force field of two walls - refers to parameters x and function coefficients of P. -it returns P2 if x < P0 and -P2 if x > P1
?13	- is used for a simulation of the force field of two walls - refers to parameters y and function coefficients of P. -it returns P2 if y < P0 and -p2 if y > P1
?14	- is used for a simulation of the force field of two walls - refers to parameters z and function coefficients of P. -it returns P2 if z < P0 and -p2 if z > P1
?15	- is a specialized function that calculates the racket force field of the tennis game simulation it uses the parameters x, y, z: ball coordinates X, Y, Z: racket center coordinates u, v, w: ball velocity r: ball racket distance Coefficient array Const[0][0..2]: the racket normal Const[0][3]: the racket radius Const[0][4..6]: force field modeling factors a, b, p of equation 7.1.5 Const[0][7]: contains the calculated force - It returns the x component of the force and stores the force in Const[0][7]
?16	returns the y component of the force calculated by ?15
?17	returns the z component of the force calculated by ?15
?18	returns 91 * mass[i], the gravitation force felt by the current particle i.
?19	is a specialized force filed function of the net of the tennis game (see section 7.1.3) - uses the parameters x, y, z: ball position u: ball speed in x direction - the coefficient array Const[1][0..4]: parameters a, b, c, d, e of equation (7.1.3) with f=1.0. - it returns the x component of equation 7.1.3.
?20	is a specialized function used for the speed dependent swim motor of particle i corresponding to the current fish of the behavioral animation described in section 4.1 - it uses the speed of the current particle i, and a global variable phase[i] which is incremented at each iteration according to phase[i] += $0.2*speed_i * \Delta t$ it returns sin(phase[i]) * 5.0
?21	is a function used to shape statistically L-structures that are repeated and statistically placed, scaled and oriented by the animation loop (see section 2.1.1). Note that this function doesn't work for ray traced films as a L-structure is a named object that can only be placed, scaled and oriented, but not reshaped. - it uses the coefficient array Const[3] - it returns Const[3][7] = p0 +° p1*sin(p2*T + p3) with p0 = ((r0 - 0.5 * Const[3][1] + 1.0) * Const[3][0] p1 = ((r1 - 0.5 * Const[3][3] + 1.0) * Const[3][2] p2 = ((r2 - 0.5 * Const[3][5] + 1.0) * Const[3][4] p3 = ((r3 - 0.5 * Const[3][7] + 1.0) * Const[3][6] and ri are stochastic values between 0 and 1 determined by the current L-structure (j) and its repetition (k) (see also section ...)
?22	returns the result calculated and saved by ?21 in fConst[3][7]

?23	calculates and returns the following force field value by using the parameter r and the coefficient array Const[4]. With Const[4][**i**] = c**i** we get: $Const4[7] = c0 \cdot c1^{c2 \cdot r^{c3}} / (r + c4)$
?24	returns Const[4][7] calculated by ?23
?25	returns sin(phase[i]) * 0.5
?26	is used to calculate the ground repulsion forces in case of collision with the y-coordinate of the current particle in the function argument. returns the Hermite function (equation 2.3.9) by using the parameter y and the coefficient array Const[5]: Hermite(TT + offset - y) with Const[5][**i**] = c**i** and c0: TT (growth distance) c1: min. c2: max c3: offset
?27	is used for tennis ball-ground collision. The ascendant ramp of this function produces a ball-ground sound event when passing a threshold value (see section 2.3.3). It returns the Hermite function Hermite(c3-y) (equation 2.3.9) by using the parameter y and the coefficient array Const[6] with Const[6][**i**] = c**i** c0: TT (growth duration) c1: min. c2: max c3: offset
?28	returns the Hermite function Hermite(t + c3) (see section 2.3.3) with the local age t of the symbol as input argument and the coefficient array Const[5] with Const[5][**i**] = c**i** c0: TT (growth duration) c1: min. c2: max c3: offset
?30	is a vision sensor function returning the distance of the center pixel of the vision window of the current actor. No visual memory is used.
'?31	is a vision sensor function, which returns 0.0 if no obstacle is in front of the actor. -1 is returned if the closest obstacles are left of the center pixel, and +1 if the closest obstacles are right of the middle pixel. It takes into account the actor_size and the dist_coll variable, which can be set by automata commands.
?32	is a behavior function (look_at_goal) used in conditions and returns 1.0 if the current actor looks at its actual goal, -1.0 otherwise
?33	is a behavior function (arrived_at_goal) used in conditions and returns 1.0 if the current actor is closer than the threshold value eps_arrived_at_goal to its goal, -1.0 otherwise
?34	returns phase[i] *180 / Pi (see also ?20)
?35	returns the current automata number of the current actor
?36	returns 1.0 if the current thing of interest of the current actor was found, -1.0 otherwise
?37	Vision sensor function: Returns 1 if the thing of rgb-interest was found by the current actor, -1 otherwise

?38	Vision sensor function: Returns 1 if the thing of rgb-interest was found by the current actor, -1 otherwise. When 1 is returned, the local goal number 1 is set with the position of the found object.
?39	returns 1 if the current actor is closer to its goal than a threshold value (eps_arrived_at_goal), -1 otherwise.
?40	is used in a condition of a production rule replacing the query symbol ? and returns the amount of the force felt at the position given by the parameters (x, y, z) representing the actual turtle position.
?41	is used in conditions of production rules and returns the result of the first growth function of the symbol to be replaced.
?42	is used in conditions of production rules and returns the result of the second growth function of the symbol to be replaced.
?43	is used in conditions of production rules and returns the result of the third growth function of the symbol to be replaced.
?60	returns the number n of on-going sound events
?61	returns the sound identifier of the on-going sound event emitted by sound source par1 of the symbol to be replaced. If no sound is emitted, -1 is returned
?62	returns the sound source number, which emits the sound identifier par1 of th symbol to be replaced. If the corresponding sound is currently not activated, -1 is returned

Table B5: Special functions

The functions of the production **Function** are referenced by the alphabetic letters a..z and A..Z or a question mark "?" followed by two digits. The semantic of these functions and their parameters are explained in table B5. Some of the parameters of these functions must be initialized in the L-system definition file as the coefficient arrays declared in the Option section of the L-system definition files by the keyword fConst. These coefficients will be referenced by Const[x][y] in table B5. Other parameters are the local age (t) of the current symbol, the global time (T), the stack top or state variables of the current symbol.

Parameters	**Normal semantic**
t	local current age of the symbol to be replaced
T	global current time
x, y, z	par1, par2, par3 of the symbol to be replaced
	Force field semantic for the symbols v, C, e
x, y, z	the position (x, y, z) of the particle
u, v, w	the velocity (u, v, w) of the particle
X, Y, Z	the position of the distant interacting particle
r	the distance of the current and the interacting particle

Table B5: Parameter semantic.

Automata BNF	**Comments**
Automata ::= int int **Commands** *	The two parameters are the actor and the automata number
Commands ::=	
KEYFRAMEdef int word	reads in the TRACK file (word) and associates a number (int) to it.
coll_test_func int	chooses a heuristic for collision test for the planned path
heuristic int	sets the search heuristic
rel_search_level	specifies the voxel size with respect to the voxel size of the visual memory
goal float float float	pushes the goal coordinates on the goal stack
local_goal int float float float	sets the local goal vector (number = first int) which is used for local navigation
rgb_interest int int int int	sets the actual rgb interest color (first three int, 0..255) and activates or desactivates the interest (last int 1 or 0)
eps_arrived int float	if the current actor is closer than this value, it has reached its local goal (number = first int) (distance in world units, used by function ?33)
eps_look_at float	the threshold value of the scalar product of the actual look at direction and the local goal (number = first int) direction (used in function ?32)
reset_goal	resets the goal stack
turtle int	sets the current turtle procedure for the actor - turtle mapping. 0: navigation 1: footholds 2: tennis racket 3: show path 4: interactive racket
mem_displ_cubes	the occupied voxels of the octree space grid of the visual memory are displayed as cubes
mem_no_displ	the occupied voxels of the octree space grid of the visual memory are not displayed
mem_displ_points	the occupied voxels of the octree space grid of the visual memory are displayed as points
vis_mem_on	the visual memory is activated
vis_mem_off	the visual memory is deactivated. Its current content is not affected.
obs float float float	the actor's position of the vision system is set
lat float float float	the actor's look at point is set
align float float float	sets the point to which the actor will align using the automata 19 (auto_align)
init int float float float	initializes an actor int: the visual memory display modulo float: minimum of world range cube float: maximum of world range cube float: edge size of a voxel of the visual memory octree space grid
fov int	the view angle of the vision system of the actor is set. The value is in tenth of degrees.
interest int int	int: the actor deactivates (0) or activates (1) its interest int: the identifier of the color coded thing of interest. Currently, there exist only the following color codes: 0: material 10 (ball) 1: material 11 (player 0) 2: material 12 (player 1)
size_actor float	sets the actor size in world units. It is recommended to give the maximum edge size of an actor.

dist_coll float	sets the local collision distance in world units which is used to decide whether an obstacle perceived by the vision system in front of the actor, leads to a collision or not.
tennis_state int	sets the state variable of the tennis game. Possible values are: 0: inactive 1: goto_play_position 2: look_for_partner 3: partner_ok 4: follow_ball 5: first_hit_turn 6: second_hit_turn 7: third_hit_turn 8: hit_turn 9: bounce_or_not_bounce 10: bounce 11: bounce_begin 12: not_bounce 13: hit_phase
state int	sets the state of the vision system. Possible values are: 0: off 1: mem_set 2: vis_set 3: end_point_set 4: path_par_set 5: look_around_set 6: look_around 7: search_new_path 8: align 9: accelerate 10: walk 11: explore 12: stop 13: create_map 14: init_communication 15: path_description 16: fix_thing 17: fix_partner 18: interactive 19: init_tennis_game 20: update_tennis_game 21: players_ready 22: init_worte 23: listen_worte 24: interprete_words 25: talk 26: wait_a_bit
optim_rate int	sets the path spline optimization modulo m. Each m frames the path spline is optimized based on the local vision (used in walk_continously automata)
speed float	sets the maximal speed of the actor
inertia float	is a measure of the actors inertia during navigation with the walk_continously automata (used in the path spline generation).

ahead int	sets the number of spline key points to be checked for collision detection along a planned path.
optim_ahead int	sets the number of key points of the path spline to be optimized (used only in the walk_continously automata).
vision_range float float float	sets the near and far clipping values of the vision system and the local range of the visual memory update.
motor_init float float int	initializes the humanoid actors with float: maximum speed in mm/sec float: scale factor int: 1 with ray tracer output, 0 without.

Table B6: The semantic of the Automata production.

SoundItem	**Comments**
INIT_source source_nbr position orientation	initializes and declares a sound source
INIT_mic mic_nbr position orientation	declares and initializes a microphone
PLAY source_nbr file_name.aiff	the source source_nbr starts emitting the sound file_name.aiff at the current time, with the actual position and orientation
SP source_nbr position	Sets the position of the sound source source_nbr
SO source_nbr orientation	Sets the orientation of the sound source source_nbr
MP mic_nbr position	Sets the position of the microphone mic_nbr
MO mic_nbr orientation	Sets the orientation of the microphone mic_nbr
TIME time	Sets the time with time
CONST_S source_nbr position orientation	Sets the position and orientation of the non moving sound source source_nbr
INIT_dist_matrix	has to be used after the declaration of all microphones and sound sources

Table B7: The sound protocol. The above typical commands of the communication protocol illustrate some elements of the communication between the animation system and the sound renderer. Source_nbr and mic_nbr are integers, position and orientation are 3D float vectors and file_name.aiff are file names of AIFF sound files.

VLNET BNF	**Comments**
VlnetInit	The VLNET declaration
Nbr_exported_actors int **KeyId** *	The number of exported actors and a list of key triples
Nbr_imported_actors int int*	The number of imported actors with the list of the actor numbers
Nbr_objects int int*	The number of VLNET objects with the list of the object numbers
KeyId :: = int int int	shared memory key for an actor, shared memory key for an object key, actor identifier number

Table B8: The Virtual Life Network BNF

Appendix C: Semantic of Symbols

C.1 Turtle Manipulation Symbols

+: turn Left: rotation by angle par4 around axis U(z)

-: turnRight: rotation by angle -par4 around axis U(z)

&: pitchDown: rotation by angle par4 around axis L(y)

^: pitchUp: rotation by angle -par4 around axis L(y)

/: rollRight: rotation by angle -par4 around axis H(x)

$: rollLeft: different meanings according to the value of par2
par2=0: rotation by angle par4 around axis H(x)
par2=1: The L-axis becomes horizontal according to the internal gravitation direction (0, -1, 0). The heading vector H remains unchanged.
par2=2: The heading vector is aligned with the gravitation direction by making L horizontal and then aligning the heading vector H with the gravitation direction by rotating it around the L axis.
par2=3: resets the orientation of the turtle to the unity matrix. The position is not changed.
par2=4: The direction of the turtle is aligned with the speed direction of the current particle in the force field (see section 2.3).
par2=5: The turtle is positioned according to an imported trajectory (trace file) referenced by parameter par3. H shows in direction of the path, U points upwards and L is horizontal. A trace file associates to each frame a position.
par2=6: H is projected to the horizontal plane. L becomes horizontal and Z vertical pointing upwards.

f: (forward: the turtle moves)
par3=0: relative move of the turtle in direction of H by par4
$\vec{P}' = \vec{P} + par4 \cdot \vec{H}$
par3=1: relative move of the turtle by the absolute vector (par4, par5, par6)
$\vec{P}' = \vec{P} + (par4, par5, par6)$
par3=2: absolute move of the turtle to the position (par4, par5, par6)
$\vec{P}' = (par4, par5, par6)$
par3=3: relative move of the turtle by the relative vector (par4, par5, par6)
$\vec{P}' = \vec{P} + par4 \cdot \vec{H} + par5 \cdot \vec{L} + par6 \cdot \vec{U}$

[: pushes the state of the turtle on the stack
par1=0: push turtle state (0 < i < 7)
par1=i; par2=0: PositionTable[i] = turtle position
par1=i; par2!=0: StateTable[i] = turtle state

]: pops the state of the turtle from the stack
par1=0: pop turtle state
par1=i; par2=0: (7<i>0) turtle position = PositionTable[i] and adds the turtle position to the current polygon (see figure C.3.6)
par1=i; par2!=0: (7<i>0) turtle state = StateTable[i]
par1=i: (-100<i<0) the cut symbol (see section 5.10)
i=100: the whole branch with all sub-branches is eliminated (see section 5.10)

C.2 Camera, Material and Light Control Symbols

l: controls the camera and light 3
par3=0: par4..6 determine the position of the observer
par3=1: par4..6 determine the look at point
par3=2: par4..6 determine the view angle (given in tenth of degrees), the aspect ratio and twist of the camera
par3=3: par4..6 determine the position of light number 3
par3=4: the position of the current particle determines the position of the camera
par3=5: the look at point is at the distance par4 in direction of the speed of the current particle
par3=6: the position and orientation of the camera is determined by the actor number par4. The position is written to a track file (see also section 2.2.3).
par3=7: the camera looks at actor number par4
par3=8: par4..6 determine the position of the observer and this position is written to the sound environment file for microphone par2
par3=9: The space ball allows an interactive manipulation of the camera. The button numbers of the space ball produce the following effects:
1: faster translation (*1.5)
2: slower translation (/1.5)
3: faster rotation (*1.5)
4: slower rotation (/1.5)
7: the height (y coordinate) is variable
8: the height (y coordinate) is fixed at the current position.
If humanoid actors are attached, the camera of h2_proc.x is placed according to the camera position and orientation of the LMinterpreter.x.
par3=10: the look at point is determined by a variable lookat_global and the position by par4..6. This position is written to the sound environment file for microphone par2. The variable lookat_global can be set with the turtle position of symbol D(par1=40).
par3=11: the camera of the user window is controlled by the speech recognition module in a similar way as by a space ball. The commands of table C.2.1 are recognized.

Command	Semantic
move right	the camera moves to the right
move left	the camera moves to the left
move up	the camera moves up
move down	the camera moves down
move	the camera moves forward
go back	the camera moves backward
turn up	the camera turns up
turn down	the camera turns down
turn left	the camera turns to the left
turn right	the camera turns to the right
faster	the camera moves faster
slow down	the camera slows down

turn faster	the camera turns faster
turn less	the camera rotation slows down
block	the height of the camera is blocked
free it	the height of the camera can move
stop	all movements and rotations are stopped

Table C.2.1: The camera control commands recognized by the speech recognition module

m: sets the material
par3=1: animation of the diffuse component of the current material
par4 = r, par5=g, par6=b
par3=2: switches the lighting model of (usefull for local vision and color coding)

par3	color	par3	color
1	green	16	grey1
2	brown	17	grey2
3	blue	18	grey3
4	red	19	cr_quartz
5	yellow	20	cr_pyrite
6	gray	21	cr_emeraude
7	white	22	cr_grossulaire
8	earth	23	cr_fluorine
9	green2	24	cr_rhodochrosite
10	actor1	25	cr_natrolite
11	actor2	26	cr_celestine
12	ball	27	cr_aigue_marine
13	mirror		
14	skin		
15	grey0		

Table C.2.2: Materials

r: sets the rgb components of segments
par4 = r, par5 = g, par6 = b (values from 0..255)

C.3 Symbols for Geometric Shapes

n: sets the resolution on spheres, trunks and cylinders (symbols O, N, P, Q, a, S)
par1: the resolution, the minimal value = 3 , the default value = 10.

R, S, T: line segments of length par4 in direction of the turtle's heading vector H. The new position of the turtle is at the end of the line.

c, d: Draw unit cubes in the turtle's coordinate system scaled by par4(H), par5(L) and par6(U). The new position of the turtle is given by new_position = old_position + par4 * H.

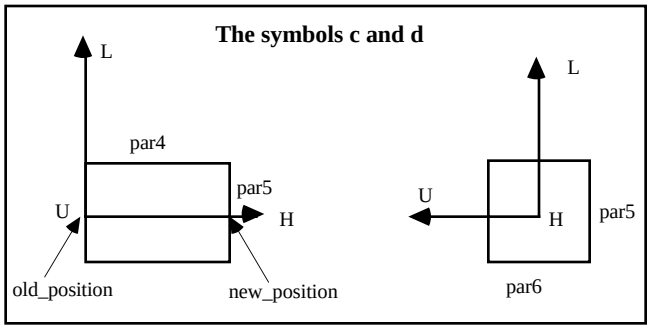


Figure C.3.1: The symbols **c** and **d** represent cubes

O, N: Draw scaled cylinders with ending half spheres. Par4 is the cylinder length and par5 and par6 are the scaling in L and U direction. The new position of the turtle is new_position = old_position + par4 * H.

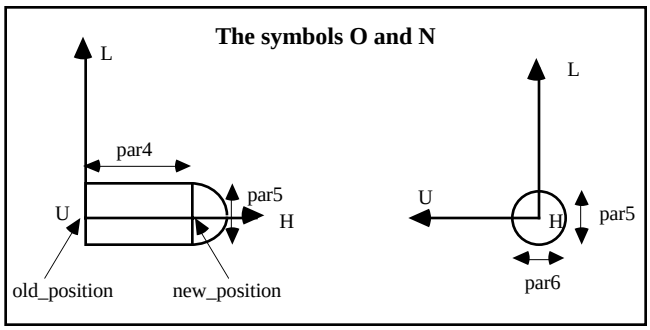


Figure C.3.2: The symbols O and N represent cylinders

o: Draws scaled cylinders with
par4: cylinder length
par5,6: scaling in L and U direction.

P, Q: draw trunks with ending half spheres according to figure C.3.3, where par 4 is the trunk length, par 5 the base thickness and par6 the end thickness.

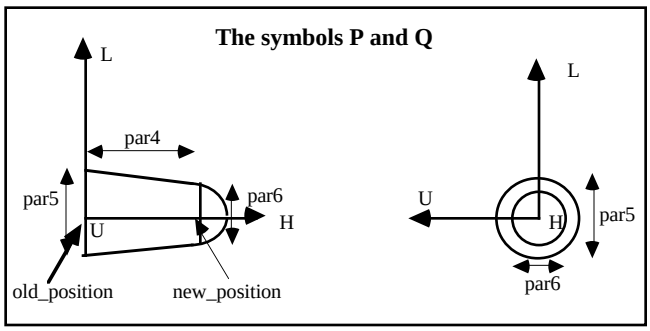


Figure C.3.3: The symbols **P** and **Q** represent trunks

a: draws a sphere of diameter par4 and the turtles position as center. The position of the turtle doesn't change.

s: draws ellipsoids scaled with par4(H), par5 (L) and par6 (U). The turtle is not moved.

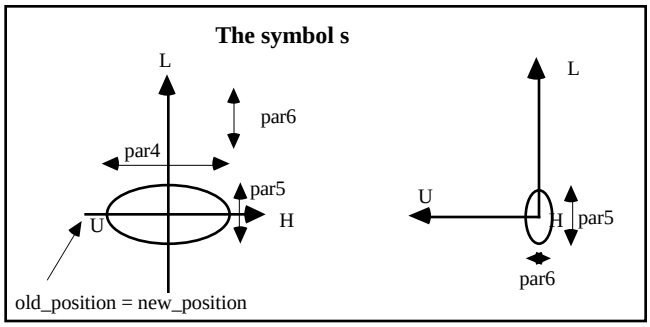


Figure C.3.4: The symbol **s** represents an ellipsoid

p, q: draw pyramids with the peak in direction of the turtle's heading H according to figure C.3.5, scaled by par4 (H), par5 (L) and par6 (U).

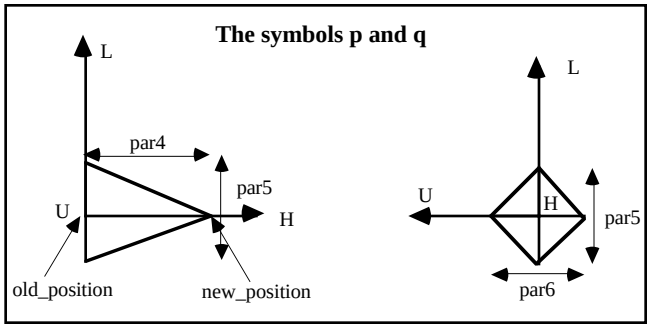


Figure C.3.5: The symbols **p** and **q** represent pyramids

According to [PRUS90] we integrated the symbols "{.}" for surface creation by production rules. To generate nested surface definitions we need a stack of polygons. The polygon stack operations are referenced by these symbols.

{: Push the current polygon on the stack and create a new current polygon.

.: Add the actual position of the turtle as a new vertex to the current polygon.

}: Draw the current polygon and pop a polygon, which becomes the current polygon, from the stack. For GL rendering the semantic of the symbol } depends on the parameters:
par1=1: the polygon normal is inverted
par2=1: a second parallel polygon with an inverted normal is drawn at a distance par4 in direction of the first surface normal
par3=1: By adding the side surfaces to the two polygons the surface is closed.
par3=3: The surface is closed by adding cylinders at the sides of the two polygons.

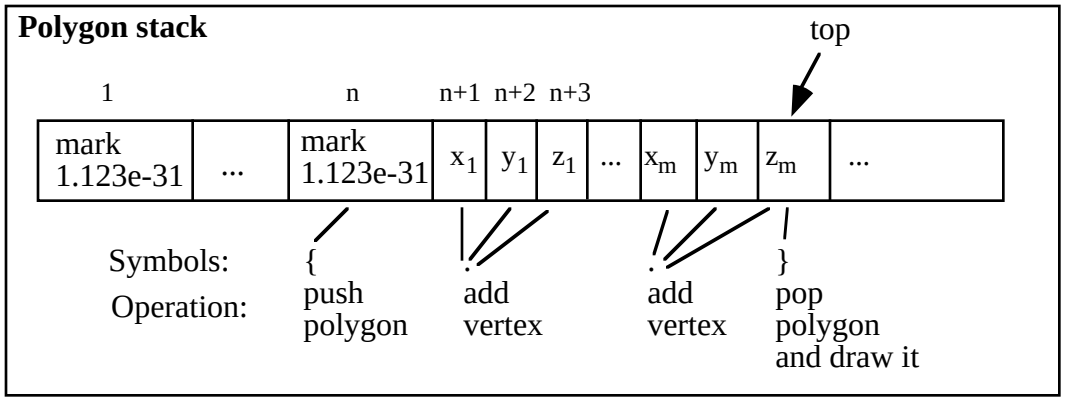


Figure C.3.6: Implementation of the polygon stack.

The symbol **{** increments the pointer "Current" and marks the new position with 1.123e-31 signifying a new current polygon. The symbol "**.**" adds each time the three coordinates of the current position of the turtle and increments the pointer "Current" by three pointing now on the last coordinate. The symbol **}** draws the polygon composed of the vertices between the current position and the last mark and puts the pointer "Current" to the position before the mark.

C.4 Force Field Symbols

v: Tropism force field definition for branching structures

par1=0: The growth function expressions determine the vector field (expr_f1, expr_f2, expr_f3). The parameters x, y, z correspond to the actual turtle position (x, y, z) allowing the use of position dependent forces. Tropism is activated.

par1=1: sets the rigidity = par4 and mass = par5 of the branch. Tropism is activated.

par1=2: Disables tropism for the symbols that follow in the formal symbol string.

e: particle generation and behavior control (see section C.6)

C: Particle - turtle mapping

par1=0: at t_local < t_step mass = par2 and initial velocity = (par4, par5, par6) at t_local >= t_step the turtle position is mapped to the evolved particle position.

par1=1: the particle position is set to (par4, par5, par6) and its velocity is calculated according to its old position and the time step. There is no evolution according to a differential equation.

par1=5: the particles position and velocity are controlled by the tennis rackets of the tennis game simulation. Par4 is the actor number (0 or 1).

par1=6: the particle's position and velocity are controlled by the turtle.

```
class Tinteraction {
    int     id_symbol,      /* a symbol identifier */
            evaluate;               /* an evaluation flag */
    float   posVel[6],      /* table with position and velocity of particle */
            time;                   /* the time */
    char    *force[3],              /* the force exerted on other particles */
            *eqDiff[3],             /* individual terms of the differential equation */
            *type;                  /* a particle type identifier (not used) */
```

```
    float  mass,   /* the mass of the particle */
                k[4][6],              /* used for Runge Kutta evolution */
                y[6]          /* used for Runge Kutta evolution */
};
Tinteraction InteractionTable[MAX_PARTICLES]
```

Code C.4.1: The particle table

C.5 Actor Symbols

M: Enables vision system of actor to turtle mapping
- par1 = 7: maps the position and orientation of the actor par3 to the turtle and makes actor par3 to the current actor.
- par1 = 14: sets the camera position of the octree display of actor par3 with the actual turtle position.
- par1 = 15: sets the camera look at point of the octree display of actor par3 with the actual turtle position.

G: Skeleton symbol
- par1=0: the wire frame skeleton is drawn
- par1=1: the turtle is positioned at the node number par4.

The symbol G with par1=1 positions the turtle at the position of the node number par4 of the skeleton (Figure 6.3.6). Thus, in combination with the query symbol **?** and the function ?40, which evaluates the global force field at the actual position of the turtle, tactile sensor points can be simulated at skeleton nodes.

g: Enables turtle to actor mapping
- par1=0: sets the position of the current actor with the position of the turtle and the actor's lookat point according to the turtle's heading vector H. The walking and the turning speed of the corresponding humanoid is set with the values of the parameters par4 and par5.
- par1=1: the local goal is set with the position of the turtle.
- par1=2: sets the position of the current actor with the position of the turtle and the actor's lookat point according to the turtle's heading vector H. The y coordinate of the actor is overwritten by par4 (projection to a horizontal plane).

H: Humanoid symbol

par1 = actor number

- par2=0: (RV par4) sets the actor speed
- par2=1: (OV par4) sets the actor's turning speed
- par2=2: (AW) activates the walk action
- par2=3: (SW) stops the walk action
- par2=4: (AK par4) activates TRACK action number par4
- par2=5: (SK par4) stops the TRACK action number par4
- par2=6: (RK par4) reverses the TRACK action number par4
- par2=7: (MA par4) sets the actor material identifier par4
- par2=8: (GR par4) grasps object number par4
- par2=9: (DE) detaches the current grasped object from the actor
- par2=10: (AR) tries to reach with the right hand the current position of the turtle
- par2=11: (PO par4) positions and orients the grasp object par4 according to position and orientation of the turtle

C.6 Special Symbols

?: the query symbol
during the interpretation of the symbol string the parameters (x, y, z) = (par1, par2, par3) are initializes with the current position of the turtle.

B: draws the SM surface number par1 scaled by the parameters par4 (H), par5 (L) and par6 (U). The surface has to be imported by the above described declaration.

D: has several meanings according to its parameters.
par1=2: draws a horizontal square (ground) (-1000, 1000, -1000, 1000) at height y = par4 and with the normal (0, 1, 0).
par1=3: draws a square in the z-x plane at height y = par5

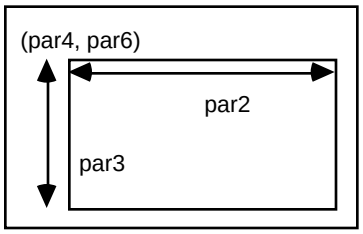


Figure C.6.1: A plane

par1 = 20: initializes the "bounding box" creation.
par1 = 21: creates a L-system definition file containing a cube that envelops the object defined between the symbols **D** () (20) () () () () () and **D** () (21) () () () () ()
par1=30: prints the actual turtle position.
par1=40: sets the user window's look at point at the current position of the turtle. This feature is enabled only by the camera control symbol **l** () () () (10) () () ().
par1=41: draws a tennis court by locally switching off the lighting to maximize the rendering speed in interactive games. If par2=1 the tennis court with an additional surrounding gray ground is drawn.

The tennis court dimensions are proportional to real values. They are shown in figure C.6.2. The court lies in the x, z plane at y equal to zero. The net height is 0.91m. It is parallel to the y, z plane. The regions L and R identify the left and the right parts of the court for a singles game and the regions A, B, C, D are the left and right service areas. As the game is modeled without change of ends, the green actor stays always at the left side and the blue actor on the right side. When the game is played interactively, the user manipulates the green actor with the spaceball . The referee watches the game by default from the point (0,2) close to the net.

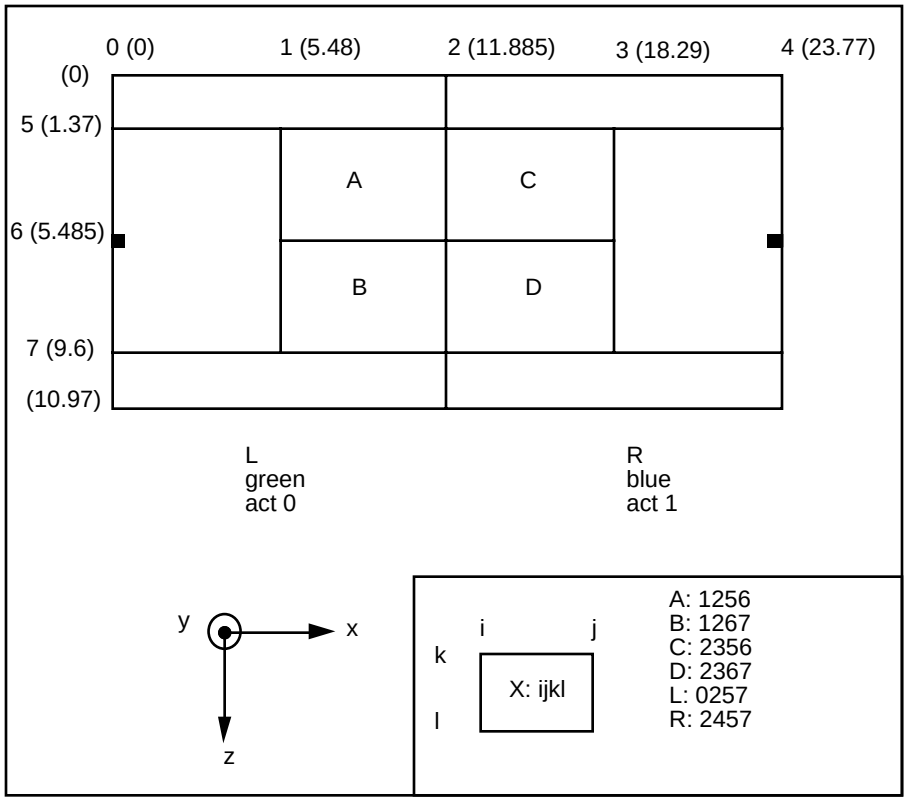


Figure C.6.2: The tennis court

When ray tracing is **enabled,** the symbol **D** has the following semantic:

par1=0: writes "grid par4 par5 par6 " into the Rayshade definition file
par1=1: writes "end " into the Rayshade definition file
par1=2: writes "plane 0 par4 0 0 1 0" into the Rayshade definition file
par1=4: writes "scale par4 par5 par6" into the Rayshade definition file
par1=5: writes "window 0 180 0 144" into the Rayshade definition file
par1=6: writes "name objpar4 list" into the Rayshade definition file
par1=7: writes "object objpar4" into the Rayshade definition file
par1=8: writes "translate par4 par5 par6" into the Rayshade definition file
par1=9: writes "rotate par4 par5 par6" into the Rayshade definition file
par1=10: writes "texture marble" into the Rayshade definition file
par1=11: writes "texture bump" into the Rayshade definition file
par1=12 : writes "window par3 par4 par5 par6" into the Rayshade definition file
par1=13: writes "name objpar4" into the Rayshade definition file
par1=14: writes "list " into the Rayshade definition file
par1=15: writes "sphere ambient 0.9 0.9 0.9 6000 0 0 0 texture sky 1 0.5 2.0 6 0.85 0.45 scale par4 par5 par6" into the Rayshade definition file
par1=16: writes "list sphere ambient 0.1 0.25 0.05 1 0 0 0 texture cloud 1 0.5 2 6 0.1 0.9 10.3 scale 0.5 1 1 end"" into the Rayshade definition file
par1=17: writes "#include "/usr/movie/noser/AVION/plane.ray" " into the Rayshade definition file followed by a transformation corresponding to the turtles coordinate system with a scaling of par4, par5 and par6 in the corresponding H, L, U directions of the turtle.
par1=18: writes "light 0.5 0.5 0.5 point observer[0] observer[1] observer[2] noshadow" into the Rayshade definition file
par1=60: includes the actor file of actor par3 into the Rayshade definition file. The actor file is in general generated by the concurrent process h2_proc (see also section 4.2),

or a corresponding sequence has been generated in a previous pass. The actor is scaled by par4, par5 and par6 in the coordinate system of the turtle.

D: as VLNET symbol
- par1=70: pick the object identified by par4
- par1=71: unpick the object identified by par4
- par1=72: map object identified by par4 with the turtle. If the object is picked, the VLNET object is positioned and oriented according to the turtle state, else the turtle is positioned and oriented according to the VLNET object matrix. The position is scaled with par5 and the object dimension with par6.
- par1=73: sets the head matrix of the exported actor identified by par4. The position is scaled with par5 and the head dimension with par6.
- par1=74: sets the hand matrix of the exported actor identified by par4. The position is scaled with par5 and the hand dimension with par6.
- par1=75: gets the head matrix of the imported actor identified by par4. The position is scaled with par5 and the head dimension with par6.
- par1=76: sets the hand matrix of the imported actor identified by par4. The position is scaled with par5 and the hand dimension with par6.

T: The symbol creates sound events.
- par1=0: the sound number par4 is played back
- par1=1: a sound event is created and par4 is the sound identifier and par5 the sound source. The sound is also played back. The position of the sound source corresponds to the actual position of the turtle.
- par1=2: the position of the sound source par4 is updated with the actual position of the turtle.

E: executes the script file number par1 that has been declared with the "command_file" keyword (see section 2.5) in the "Options" section of the L-system definition file. The arguments passed are

the_current_frame_number, arg2_string arg3_string, (int) par4, (int) par5, (int) par6.

- par1: number of script file
- par2 = 0: the script file is only executed during the interpretation for the user window.
- par2 != 0: the script file is at each interpretation of the symbol executed.
- par4 .. par6: the growth function values are converted to integer values and passed to the script file as parameters.

e: ephemeral symbol for particle generation and behavior control. This symbol is only treated at the derivation of a symbolic object. It does never appear in a formal symbol string as it has to be evaluated only once.

- par1=0: pushes the first growth function text between the parenthesis "()" on the automata stack of the behavior control system and activates the corresponding automata by popping it from the stack.
- par1=10: vector force field definition by (expr_f1, expr_f2, expr_f3) (see equation 2.3.1).
- par1=11: the particles individual part of the differential equation is given by (expr_f1, expr_f2, expr_f3) (see equation 2.3.1).

Appendix D: Sounds

The actually implemented sounds of the sound library are given in table D.1.

Sound event number	File name	Comment
0	ball_racket.aiff	collision sound of tennis ball and racket
1	ball_ground.aiff	tennis ball bouncing
2	cricket.aiff	a cricket
4	frog.aiff	a frog
5	foret.aiff	a tropical forest
6	start.aiff	very short sound used for synchronization
7	vivaldi.aiff	classical music file
8	avion.aiff	a constant airplane noise
9	ball_net.aiff	a tennis ball - net collision sound
10	advantage.aiff	spoken word
11	all.aiff	spoken word
12	deuce.aiff	spoken word
13	fault.aiff	spoken word
14	fifteen.aiff	spoken word
15	fourty.aiff	spoken word
16	game.aiff	spoken word
17	let.aiff	spoken word
18	love.aiff	spoken word
19	server.aiff	spoken word
20	striker.aiff	spoken word
21	thirty.aiff	spoken word
22	blue.aiff	spoken word
23	green.aiff	spoken word
24	out.aiff	spoken word
25	service.aiff	spoken word
26	left.aiff	spoken word
27	right.aiff	spoken word
28	wait02.aiff	0.02 sec silence. Used as pause between two words
29 .. 58	xxx.aiff	the words of table D.2

Table D.1: Actual sound events in the sound library.

The speech recognition system is trained with the vocabulary shown in table D.2. The numbers preceding each word represent their sound event number which is used to recognize the semantic of a word in the virtual acoustic environment.

29: move right	30: move left	31: move up	32: move down
33: move	34: go back	35: turn up	36: turn down
37: turn left	38: turn right	39: faster	40: slow down
41: turn faster	42: turn less	43: block	44: free it
45: stop	46: blue	47: green	48: one
49: two	50: three	51: four	52: five
53: six	54: seven	55: eight	56: nine
57: ten	58: exit		

Table D.2: Vocabulary of the speech recognition system

Appendix E: Scripts

Appendix E contains some c-shell command files useful for image by image film productions.

```
#!/bin/csh -f
# call: AFTER_ACCOM.CMD frame
# the window of the application has to placed at the lower left corner of the screen

set offset = 0
set begin = -1
set end = 1550
set path = /net/ligsg9/usr/export/guests/nosermovie
@ i=$1 + $offset
echo $i
if (($i > $begin) && ($i < $end)) then
    echo frame $i recording
    echo /usr/sbin/scrsave $path/dummy.rgb 13 732 13 588
    /usr/sbin/scrsave $path/dummy.rgb 13 732 13 588

    echo /usr/people/noser/O/bin/toyuv.x $path/dummy.rgb $path/dummy.yuv;
    /usr/people/noser/O/bin/toyuv.x $path/dummy.rgb $path/dummy.yuv;

    echo /usr/bsd/rcp $path/dummy.yuv accom:$i
    /usr/bsd/rcp $path/dummy.yuv accom:$i

endif
```

Code E.1: C-shell script recording images on a Accom workstation disk

```
#!/bin/csh -f
# call: AFTER_RGB.CMD frame actor_name path
# the rayshade file has to be placed in the $path directory
# the filename is given by $path/$actor_name.ray

@ i=$1

set frame=$i
if ($i < 1000) set frame=0$i
if ($i < 100) set frame=00$i
if ($i < 10) set frame=000$i
set actor_name = $2
set path = $3

/usr/sbin/scrsave $path/$actor_name.$frame.rgb 13 732 13 588
echo writing $path/$actor_name.$frame.rgb
/usr/local/pub/bin/gzip $path/$actor_name.$frame.rgb

# gunzip for decompressing
```

Code E.2: C-shell script for saving each frame on the hard disc

First, the frame number is passed to the script together with a path and a filename. Then, the script file produces a numbered and formatted filename path/file_name.xxx.rgb, saves the corresponding screen portion into the file and compresses it with an appropriate compressor program (gzip).

```
#!/bin/csh -f
# call: AFTER_RS_RGB.CMD frame actor_name path
# the Rayshade file has to be placed in the $path directory
# the filename is given by $path/$actor_name.ray

@ i=$1
set actor_name = $2
set path = $3

set Rayshade = /usr/local/bin/rayshade set offset = 0
set begin = -1
set end = 1550
@ j=$i + $offset

set frame=$i
if ($i < 1000) set frame=0$i
if ($i < 100) set frame=00$i
if ($i < 10) set frame=000$i

# call of rayshade on the ONYX
echo /usr/bsd/rsh ligsg1 $rayshade -i -O $path/$actor_name.$frame.rgb
    $path/$actor_name.$frame.ray
/usr/bsd/rsh ligsg1 $rayshade -i -O $path/$actor_name.$frame.rgb
    $path/$actor_name.$frame.ray

# image conversion and transfer to the Accom WSD
if (($j > $begin) && ($j < $end)) then

 echo /usr/people/noser/O/bin/toyuv.x $path/$actor_name.$frame.rgb
$path/dummy.yuv;
 /usr/people/noser/O/bin/toyuv.x $path/$actor_name.$frame.rgb $path/dummy.yuv;

 echo /usr/bsd/rcp $path/dummy.yuv accom:$j
 /usr/bsd/rcp $path/dummy.yuv accom:$j

endif

# elimination of all intermediat files
echo /bin/rm $path/$actor_name.$frame.ray $path/$actor_name.$frame.rgb
/bin/rm $path/$actor_name.$frame.ray $path/$actor_name.$frame.rgb
```

Code E.3: C-shell script for ray traced rendering and transfer on the video Accom disc.

```
#!/bin/csh -f

set sleep_time = 20
set file_name = mary
set begin = 1
set end = 75

mt rewind
set i = $begin
```

```
while ($i != $end)
    echo $i
    if ($i < 1000) set frame = $i\*
    if ($i < 100) set frame = 0$i\*
    if ($i < 10) set frame = 00$i\*
    if ({ (ls $file_name.$frame | grep $file_name > /dev/null) }) then
    echo hallo
    @ j = $i - 1
    if ($j < 1000) set frame = $j\*
    if ($j < 100) set frame = 0$j\*
    if ($j < 10) set frame = 00$j\*
    ls $file_name.$frame
    tar cvf /dev/nrtape $file_name.$frame
    \rm $file_name.$frame
    @ i++
    endif
    sleep $sleep_time
end
mt offline
```

Code E.4: C-shell script for automatic back up of image files on a DAT tape.

```
#!/bin/csh -f
# tar_get_dat.cmd reads periodically files from the DAT, if there only
# remain 9 in the image directory where it has been launched.
# Example: mary.0100.rgb ... mary.0119.rgb are read, if mary.0091 has
# disappeared. The files have to be stored in blocks of 10 on the DAT tape

set sleep_time = 10
set file_name = te3
set begin = 20
set end = 150

tar xvf /dev/nrtape
set i = $begin

while ($i != $end)
    echo ------------------------------------------------------------
    echo start of while : i $i
    set name = $file_name.\?\?\?1.rgb
    echo "test: $name"
    if ({ (test -f $name) }) then
        echo sleep $sleep_time seconds
        sleep $sleep_time
    else
    tar xvf /dev/nrtape
        @ i++
    endif
end
mt rewind
mt offline
```

Code E.5: C-shell script for restoring automatically image files from a DAT tape.

Appendix F: VLNET Interface

```
int vlnet_remote_body_create_shmem( int key, volatile void **buf);
int vlnet_remote_body_set_joints(       volatile void *vbuf, int actor_id,
                                        float *joints_table);

int vlnet_remote_body_get_joints(       volatile void *vbuf, int actor_id,
                                        float *joints_table);

int vlnet_remote_body_set_transformations(      volatile void *buf, int actor_id,
                                                float head_mat[4][4],
                                                float hand_mat[4][4]);

int vlnet_remote_body_get_transformations(      volatile void *buf, int actor_id,
                                                float head_mat[4][4],
                                                float hand_mat[4][4]);

int vlnet_remote_object_create_shmem( int key, volatile void **buf);

int vlnet_remote_object_set_transformation(
                        volatile void *buf,
                        int object_id,
                        float matrix[4][4]);

int vlnet_remote_object_get_transformation(
                        volatile void *buf,
                        int object_id,
                        float matrix[4][4]);

int vlnet_remote_object_pick( volatile void *buf, int object_id);

int vlnet_remote_object_unpick( volatile void *buf, int object_id);
```

Code F.1: Some functions exported by the vlnet_remote module which was used to create an interface with VLNET clients.

```
#!/bin/csh -f
set host=ligsg1

/* eventually existing clients and servers are killed to avoid interferences */
/sbin/killall vlnet.x
/sbin/killall vlnet_server.x
sleep 3

/* the server is started by passing the shared environment definition file tennis_stad9 */
(cd /usr/people/noser/O; /usr/people/capin/LIBVLNET/bin/vlnet_server.x -s
tennis_stad9.scn -b tolgadef.bdy &)
sleep 5

/* the first client with the shared memory keys 80 and 81 gets the actor id 1 */
(cd /usr/people/noser; /usr/people/capin/LIBVLNET/bin/vlnet.x -N -h $host -b
marylin_noface.bdy -r 80 -o 81 &)
```

```
sleep 5

/* the second client with the shared memory keys 82 and 83 gets the actor id 2 */
(cd /usr/people/noser/G; /usr/people/capin/LIBVLNET/bin/vlnet.x -N -h $host -b
superman2.bdy -r 82 -o 83 &)

/* the third client with actor id 3 is in general started on a remote machine */
#sleep 5
#(cd /usr/people/noser/G/BLM; /usr/people/capin/LIBVLNET/bin/vlnet.x -D -b
tolganodef.bdy -h $host -S &)
```

Code F.2: Script file starting VLNET clients and the VLNET server.

```
VLNETdeclare Nbr_exported_actors 2   80 81 1  82 83 2   Nbr_imported_actors 1 3
Nbr_objects 4 0 1 2 3

/* actors: 1: player 2: referee 3: user
   objects: 0: court  1: rack player  2: racket user 3: ball */
```

Code F.3: VLNET declaration of two actor clients in the L-system definition file.

Appendix G: Heuristics for Path Searching

This appendix is a complement of section 5.2.3 treating the heuristic path searching in the voxel space of the perceived environment of an actor. Table G1 shows the actually implemented heuristics for path searching and intersection tests. The first column, "Heu. nbr", enumerates the heuristics. The second column contains the relative y coordinate of the actual x, z plane. The next 7 columns contain the relative x coordinate of the voxels. In these columns the lines in a cell represent the relative z coordinate. The numbers in the table cell indicate the voxel number. In the column "voxel tested" we find the list of the neighbor voxels to be added to the candidate list, with the corresponding condition given in the last column "condition". This condition is a logical expression where a number can be considered as a logical variable which is true, if the corresponding voxel is empty. The symbols "! * +" represent the logical operators "not, and, or".

Heu. nbr	y	Voxel numbering: -> x, ^ -z							voxel tested	condition: x : empty ! : not, *: and, +: or
		-3	-2	-1	0	1	2	3		
0	0			3	2 4	1			1 2 3 4	1 2 3 4
1	0		7	12 3 10	6 2 4 8	11 1 9	5		1 2 3 4	1*5*9*11 2*11*6*12 3*7*10*11 4*8*9*10
2	-1				6					
	0			3	2 4	1			1 2 3 4 5 6	1 2 3 4 5 6
	1				5					
3	0				1				1	1
4				3	2 5 4	1			5	5*1*2*3*4
5	-1			7	6 8	5				
	0			3	2 4	1			1 2 3 4	1*!5 2*!6 3*!7 4*!8
6	-1				1					
	0				2				2	2*!1

7	-1		19	24 15 22	18 14 16 20	23 13 21	17			
	0		7	12 3 10	6 2 4 8	11 1 9	5		1 2 3 4	1*5*9*11*!13 *!17 *!23 *!21 2*11*6*12 *!14 *!18 *!24 *!23 3*7*10*11 *!15 *!19 *!22 *!24 4*8*9*10 *!16 *!20 *!21 *!22
8	-1			7	6 8	5				
	0			3	2 9 4	1			9	9*1*2*3*4 *!5 *!6 *!7 *!8
9	-2		27	34 31 36	26 30 32 28	33 29 35	25		1 2 3 4	1*(*5*9*11*!13 *!17 *!23 *!21 + 5*9*11*17*!13 *!23 *!21 *!25 + !5* 37) 2*(*6*12*11*!14*!18 *!23 *!24 + 5*9*11*18*!24 *!23 *!26 + !6* 38) 3*(*7*10*12*!15 *!19 *!22 *!24 + 7*10*12*19*!15 *!22 *!27 + !7* 39) 4*(*8*9*10*!16 *!20 *!21 *!22 + 8*9*10*20*!16 *!21 *!22 *!28 + !8* 40)
	-1		19	24 15 22	18 14 16 20	23 13 21	17		13 14 15 16	13 * !29 moving down 14 * !30 15 * !31 16 * !32
	0		7	12 3 10	6 2 4 8	11 1 9	5		37 38 39 40	37 *!1 moving up 38 *!2 39 *!3 40 *!4
	1			39	38 40	37				
10	0		7	12 3 10	6 2 4 8	11 1 9	5		1 2 3 4	1*5*9*11*13*14*15*16 2*11*6*12*13*14*15*16 3*7*10*11*13*14*15*16 4*8*9*10*13*14*15*16
	1				13					
	2				14					
	3				15					
	4				16					
11	0			3	2 5 4	1			5	5*1*2*3*4*6
	1				6					

15	-1			1	10	18				
				2	11	19				
				3	12	20				
	0			4	13	21				
				5		22				
				6	14	23			1, .. 26	1, .., 26
	1			7	15	24				
				8	16	25				
				9	17	26				
18	0				1				1	1*2*3*4*5*6
	1				2					
	2				3					
	3				4					
	4				5					
	5				6					

Table G.1: The actually implemented heuristic table.

Heuristic 0, for example, tests the four neighbor voxels (1,2 3 4) in the x, z plane and adds them to the candidate list if they are empty. This yields a simple 2D path searching.

Heuristic 1, adds the same voxels to the candidate list, but only if the three neighbors of each candidate voxel are also empty. Voxel 1, for example, is only added if it is empty, and if its neighbor voxels 5, 9 and 11 are also empty which is expressed by the Boolean expression 1*5*9*11 in the condition column of heuristic 1. Note that the numbers represent logical variables. Heuristic 7 represents as heuristic 5 also a 2D search but with additional conditions for the neighbor voxels of the 4 voxels (1,2,3, 4) to be added to the candidate list. The logic expression for the voxel 1, for example, is 1*5*9*11*!13*!17*!23*!21. It expresses the fact that voxel 1 is added to the candidate list if it is empty together with its neighbors 5, 9 and 11, and if the 4 neighbor voxels 13, 17, 23 and 21 in the lower x, z plane are occupied. Thus, this heuristic yields a 2D path on a support with a distance of at least one voxel edge length from obstacles.

Heuristic 10 can be used for 2D path searching for high objects. It assures that 4 voxels above the path are always free. Heuristics 2 and 15 are 3D search heuristics as they add voxels of the neighbor x, z plane to the candidate list.

Heuristic 9 is the most complex path searching heuristic we tested. It was used for 3D path searching on a support. It allows an actor besides 2D navigation to mount and descend ramps and stairs. Some voxel tests of the heuristics can be also employed for collision detection when an actor is moving along a path. As the environment can be dynamic, a path which has been found by a heuristic search can later be blocked by new or moving obstacles. Therefore, a moving actor always checks at regularly intervals, whether there are new obstacles along the part of the path which is in its field of view. As its visual memory is automatically updated when it moves around, it can project the points of its real 3D path in its octree space and apply a heuristic test of the corresponding path voxel. The heuristics 3, 8, 11 and 18 represent such tests for collision detection along a path. They should be selected according to the heuristic used for the path searching. For example, when the search is done by heuristic 7, then the collision tests should be done with heuristic 8 as it will guarantee that not only the path voxels will be empty but also its neighbors in the x, z plane, and it will also test whether the supporting voxels are occupied or not. The designer of a navigating actor by the L-system can choose the heuristic of the path searching algorithm (keyword: heuristic), the collision test function (keyword: coll_test_func) and the logarithmic relative path search level (keyword: rel_search_level, see table B.6). The relative path search level specifies the voxel size with respect to the voxel size of the visual memory. A relative path search level of 1, for

example, considers voxels with the double edge size length of those of the octree memory, whereas a relative path search level of 0 uses voxels of the same size as those of the visual memory. We made the best experiences with a value of 1. Higher values are useful in nearly empty spaces. A value of 0 can produce paths through small holes in walls due to imprecision of the voxelizing process.

Appendix H: The Fourth-order Runge Kutta method

This appendix is a complement to section 2.3 1. It shows how the Fourth-order Runge Kutta method evolves the force field based particle system of the animation step.

$t_0 = 0$: initial time

$t_n = t_{n-1} + h$: time increment h

$\vec{Y}_0 = \vec{C}$: initial condition at t_0

$$\vec{Y}_{n+1} = \vec{Y}_n + \frac{\vec{K}_1}{6} + \frac{\vec{K}_2}{3} + \frac{\vec{K}_3}{3} + \frac{\vec{K}_4}{6} + O(h^5)$$

$$\vec{K}_1 = h \cdot \vec{F}\left(t_n, \vec{Y}_n\right)$$

$$\vec{K}_2 = h \cdot \vec{F}\left(t_n + \frac{h}{2}, \vec{Y}_n + \frac{\vec{K}_1}{2}\right)$$

$$\vec{K}_3 = h \cdot \vec{F}\left(t_n + \frac{h}{2}, \vec{Y}_n + \frac{\vec{K}_2}{2}\right)$$

$$\vec{K}_4 = h \cdot \vec{F}\left(t_n + h, \vec{Y}_n + \vec{K}_3\right)$$

Appendix I: Color Figures

Literature

[A95] P. Astheimer, *Acoustic Simulation for Visualization and Virtual Reality*, Eurographics '95, State of The Art Reports, R.C. Veltkamp (ed), Maastricht, The Netherlands, Aug. 28 - Sept. 1, 1995, pp. 1-23.

[ABE84] H. Abelson, A. A. diSessa, *Turtle Geometry*, The MIT Press, 1984

[AK84] Aono M., Kunii T. L., *Botanical Tree Image Generation*, IEEE Computer Graphics and Applications, Vol. 4, No. 5, 1984, pp. 10-33

[AKE90] Kurt Akeley, *The Hidden Charms of Z-buffer*, Iris Universe, issue Number 11, 1990

[B83] J. Blauert, Spatial Hearing, The Psychophysics of Human Sound Localization, MIT Press, Cambridge, 1983

[B86] R. Boulic, Conception assistée par ordinateur de boucles de commande avec capteurs en robotique et en téléopération, Thèse, Série A, No. d'ordre 210, No. de Série: 48, Université de Rennes I, 1986

[B96] B. Benes, *An Efficient Estimation of Light in Simulation of Plant Development*, in Computer Animation and Simulation'96, R. Boulic, G. Hégron (eds.), Proceedings of the Eurographics Workshop in Poitiers, France, August 31 - Sept. 1, 96, Eurographics, ISBN 3-211-82885-0, Springer, pp. 153 - 165

[BCH95] Boulic R., Capin T., Huang Z., Kalra P., Lintermann B., Magnenat-Thalmann N., Moccozet L., Molet T., Pandzic I., Saar K., Schmitt A., Shen J., Thalmann D., *"The Humanoid Environment for Interactive Animation of Multiple Deformable Human Characters"*, Proc. Eurographics '95, 1995, pp 337 - 348.

[BD96] Woodrow Barfield, Eric Danas, Comments on the Use of Olfactory Displays for Virtual Environments, Presence, Vol. 5, No. 1, Winter 96, 109 - 121

[BEZ92] Bezault L, Boulic R, Thalmann D, Magnenat-Thalmann N (1992), *A Interactive Tool for the Design of Human Free-Walking Trajectories*,Proc. of Computer Animation 92, Geneva

[BG95] B.M. Blumberg, T.A. Galyean, *Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments*, SIGGRAPH 95, Conference Proceedings, August 6-11, 1995, ACM Press, pp. 47-54.

[BNT93] R. Boulic, H. Noser, D. Thalmann, *Vision-based Human Free-Walking on Sparse Foothold Locations*, Fourth Eurographics Animation and Simulation Workshop, Politechnical University of Catalonia, Barcelona-Spain, September 4-5, 1993

[BNT94] Ronan Boulic, Hansrudi Noser, Daniel Thalmann, *Automatic Derivation of Curved Human Walking Trajectories from Synthetic Vision*, Proceedings Computer Animation '94, IEEE Computer Society Press, Los Alamitos, California, 1994

[BOU90] Boulic R, Thalmann D, Magnenat-Thalmann N (1990) A Global Human Walking Model with Real-Time kinematic Personification, *The Visual Computer*, Vol , No ,December 1990.

[BPW93] Norman I. Badler, Cary B. Phillips, Bonnie Lynn Webber, *Simulating Humans*, Computer Graphics Animation and Control, Oxford University Press, 1993

[C75] E. Catmull, *Computer display of curved surfaces*, Proc. IEEE, Conference on Computer Graphics, Pattern Recognition and Data Structures, also in Tutorial on interactive computer graphics, IEEE Press, pp. 309 - 315, 1975

[CLH96] B. Chanclou, A. Luciani, A. Habib, *Physical Models of Loose Soils Dynamically Marked by a Moving Object*, Proceedings Computer Animation '96, June 3-4, 1996, Geneva, Switzerland, IEEE Computer Society Press, Los Alamitos, California, pp. 27-35.

[CLI92] Cliff D, Husbands P, Harvey I (1992) Evolving Visually Guided Robots, *Cognitive Science Research Papers,* CSRP 220,University od Sussex at Brighton, july 1992.

[CRO85] J.L. Crowley, *Navigation for an Intelligent Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. RA 1, No 1, pp31-41 (1985)

[DB94] Durand R. Begault, *3D-Sound for Virtual Reality and Multimedia*, AP Professional, 1994.

[EAgent8125] Contributors: A. Becheiraz, R. Boulic, L. Emering, H. Noser, Z. Huang, S. Bandi LIG EPFL Lausanne, B. Kalra, MIRAlab, UNI Geneva. Project Directors: C. Thalmann and N. Magnenat-Thalmann, Esprit project P 8125, *Autonomous Virtual Humans, Sensors, User Reference Manual.*

[Ehum6709] R. Boulic, T. Capin, Z. Huang, T. Molet, J. Shen, P.Kalra, L. Moccozet, Directors: D. Thalmann, N. Magnenat-Thalmann, Esprit Project 6709, *Humanoid, Human Data Structure & Parallel Integrated Motion*, User Reference Manual, (at EPFL).

[ELF90] A. Elfes, *Occupancy Grid: A Stochastic Spatial Representation for Active Robot Perception*, Proceeding of the Sixth Conference on Uncertainty in AI, July 1990.

[FL94] J. Françon, P. Lienhardt, Basic Principles of Topology-Based Methods for Simulating Metamorphoses of Natural Object, in Artificial Life and Virtual Reality, N. M. Thalmann, D. Thalmann (edts), John Wiley and Sons Ltd, 1994, pp. 23-44

[FO88] Foster, S. H. (1988). *Convolvotron User's Manual.* Crystal River Engineering. IAC, 12350 Ward's Ferry Road, Grove-land, CA 95321

[FOWER91] Foster, S. H., Wenzel, E. M., and Taylor, R. M. (1991). *Real-time synthesis of complex acoustic environments* [Summary], Proceedings of the ASSP (IEEE) Workshop on Applications of Signal Processing to Audio & Acoustics, New Paltz, NY. Furness, T.A. (1986). She super cockpit and its human factors

[FOWEV91] Foster, S. H. & Wenzel E. M. (1991) *Virtual acoustic environments: The Convolvotron*. Demonstration system presented at the "tomorrow's Realities Gallery. " SIGGRAPH '91, 18th ACM Conference on Computer Graphics and Interactive Techniques, Las Vegas, NV

[GA90] M. Girard, and S. Amkraut, *Eurhythmy: Concept and Process* The Journal of Visualization and Computer Animation, Vol. 1: 15-17 (1990)

[GABO92] F. Gallou, B. Bouchon-Meunier, Systémique, Théorie & applications, Technique & Documentation, Lavoisier, Paris, 1992, ISBN: 2-85206-784-6

[GIR87] Girard M (1987) Interactive Design of 3D Computer-animated Legged Animal Motion, *IEEE Computer Graphics and Applications,* Vol.7, No6, pp.39-51

[GM87] Gary W. Meyer, Donald P. Greenberg, *Perceptual Color Spaces for Computer Graphics,* in Color and The Computer, Chapter 4, ed. by H. John Durret, Academic Press Inc, 1987

[GON85] Michel Gondran, Michel Minoux, *Graphes et algorithmes*, Editions Eyrolles, 1985, page 41, chap 2.2

[HD96] J. Hennebert, D. P. Delacrétaz, *POST: Parallel Object-Oriented Speech Toolkit*, to be published at ICSLP 96, Philadelphia

[HIR84] Hirose S (1984) A Study of Design and Control of a Quadruped Vehicle, *The International Journal of Robotics Research*, Vol 3, No2, pp 113-133

[HOD85] Hodgins JK, Koechling J, Raibert M (1985) Running Experiments with a Planar Biped *Proc of 3th Int. Symp. on Robotics Research*, Cambridge MA:MIT Press

[HOD91] Hodgins JK, Raibert M (1991) Adjusting Step Lenght for Rough Terrain Locomotion, *IEEE Transactions on Robotics and Automation*, Vol 7, No3, pp289-298,June 1991

[HTTPpost] http://circwww.epfl.ch/data/general/jean/post/post.html

[INM81] Inman VT, Ralston HJ, Todd F (1981) Human Walking, Williams and Wilkins, Baltimore

[JO94]Jean-Michel Jolion, *Computer Vision Methodologies,* CVGIP: Image Understanding, Vol. 59, No 1, January, pp 53-71, 1994

[K93] P. Kalra, *An Interactive Multimodal Facial Animation System*, Thèse No. 1183 (1993), EPFL, (Ecole Polytechnique Fédérale de Lausanne).

[KO93a] Ko H, Badler NI (1993) Curved Path Locomotion that Handles Anthropometrical Variety, *Technical report MS-CIS-93-13* Graphics Lab 53, Feb 1993

[KO93b] Ko H, Badler NI (1993) Intermittent Non-Rhythmic Human Stepping and Locomotion, *First Pacific Conference on Computer Graphics and Applications,* Seoul Korea

[L68] A. Lindenmayer, *Mathematical models for cellular interaction in development*, Part 1 and 2, Journal of Theoretical Biology, 18, pp. 280-315, 1968

[L94] H. Lehnert, Fundamentlas of Auditory Virtual Environment, in Artificial Life and Virtual Reality, N. M. Thalmann, D. Thalmann (edts), John Wiley and Sons Ltd, 1994, pp. 161-171

[LR92] R. Lecoustre, P. de Reffy, M. Jaeger, P. Dinouard, *Controlling the Architectural Geometry of a Plants's Growth - Application to the Begonia Genus*, in Creating and Animating the Virtual World, N. M. Thalmann, D. Thalmann (eds.), Springer - Verlag Tokyo 1992, pp. 199-214

[MA82] Marr, *Vision*, Freeman, New York, 1982

[McM84] McMahon T (1984) Mechanics of Locomotion, *Int. Journal. of Robotics Research*, Vol.3, No2, pp.4-28.

[MP96] R. Mech, P. Prusinkiewicz, *Visual Models of Plants Interacting with Their Environment*, SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 397-410

[NCL96] J. Nouri, C. Cadoz, A. Luciani, *The Physical Modelling of Complex Physical Structures: The Mechanical Clockwork, Motion, Image and Sound*, Proceedings Computer Animation '96, June 3-4, 1996, Geneva, Switzerland, IEEE Computer Society Press, Los Alamitos, California.

[NPCTT96] H. Noser, I. S. Pandzic, T. K. Capin, N. M. Thalmann, D. Thalmann, *Playing Games through the Virtual Life Network*, ALIFE V, Oral Presentations, May 16-18, 1996, Nara, Japan, pp. 114-121

[NRT95] Noser H., Renault O., Thalmann D., N. M. Thalmann, *Navigation for Digital Actors Based on Synthetic Vision, Memory, and Learning.* Comput. & Graphics, Vol. 19, No. 1, pp 7-19, 1995

[NRTT93] H. Noser, O. Renault, D. Thalmann, N. M. Thalmann, *Vision-Based Navigation for Synthetic Actors,* SIGGRAPH 93, Course notes n 80 "Recent Techniques in Human Modeling, Animation and Rendering", Anaheim, California

[NT93] Noser H. Thalmann D., *L-System-Based Behavioral Animation*, Proceedings of the First Pacific Conference on Computer Graphics and Applications, Pacific Graphics 93, Aug. 1993, World Scientific Publishing Co Pte Ltd, pp. 133-146

[NT94b] H. Noser, D. Thalmann, *Synthetic Worlds* , Chapter: *Towards Autonomous Synthetic Actors,* edited by T.L. Kunii and A. Luciani, 1994 John Wiley & Sons, Ltd.

[NT94c] H. Noser, D. Thalmann, Artificial Live and Virtual Reality, Chapter: *Simulating Life of Virtual Plants, Fishes and Butterflies* edited by Nadia Magnenat Thalmann and Daniel Thalmann, 1994 John Wiley & Sons, Ltd.

[NT95b] Noser H., Thalmann D., S*ynthetic Vision and Audition for Digital Actors,* Computer Graphics forum, Vol. 14. Number 3, Conference Issue, Maastricht, The Netherlands, pp. 325 -336, August 28 - Sept. 1, 1995

[NT95] Noser H., Thalmann D., *Complex Vision-based Behaviors for Virtual Actors*, CEIG'95 V Congreso Espanol De Informatica Grafica, Palma de Mallorca, 28 29 30 Junio, pp 269-284, 1995

[NT96] H. Noser, D. Thalmann, *The Animation of Autonomous Actors Based on Production Rules*, Proceedings Computer Animation'96, June 3-4, 1996, Geneva Switzerland, IEEE Computer Society Press, Los Alamitos, California, pp 47-57

[NTT91] H. Noser, R. Turner, D. Thalmann, L-structures, Travail de diplôme à l'EPFL, Computer Graphics Lab, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland, 1991

[NTTU92] Noser H, Thalmann D, Turner R, *Animation based on the Interaction of L-systems with Vector Force Fields*, Proc. Computer Graphics International '92, pp. 747-761

[OZG84] Ozguner F, Tsai SJ, Mc Ghee RB (1984) An Approach to the Use of Terrain-Preview Information in Rough-Terrain Locomotion by a Hexapod Walking Machine, *Int. Journal. of Robotics Research*, Vol.3, No2, pp.134-146

[PAL90] Pal PK, Jayarajan K (1990) A Free Gait for Generilized Motion, *IEEE Transactions on Robotics and Automation*, Vol 6, No5, pp597-600, October 1990

[PAL91] Pal PK, Jayarajan K (1991) Generation of Free Gait - A Graph Search Approach, *IEEE Transactions on Robotics and Automation*, Vol 7, No3, pp299-305,June 1991

[PCTT95] Pandzic I., Çapin T., Magnenat-Thalmann N., Thalmann D., "*VLNET: A Networked Multimedia 3D Environment with Virtual Humans"*, Proc. Multi-Media Modeling MMM `95, World Scientific, Singapore

[PJM94] P. Prusinkiewicz, M. James, R. Mech, *Synthetic Topiary* , SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, 1994, pp. 351-358.

[PRUS90] P. Prusinkiewicz, A. Lindenmaer, *The Algorithmic Beauty of Plants* (1990), Springer Verlag

[PRUS93] P. Prusinkiewicz, M.S. Hammel, E. Mjolsness, *Animation of Plant Development*, Computer Graphics proceedings, Annual Conference Series 93, SIGGRAPGH 93

[PVM] A. Geist and al., *PVM: Parallel Virtual Machine*, A Users' Guide and Tutorial for Networked Parallel Computeing. MIT Press, 1994

[R83] W. T. Reeves, *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*, acm Transactions on Graphics, V2 #3, April 83, and reprinted in Computer Graphics, V17 #3, July 1983, (acm SIGGRAPH '83 Proceedings), pp. 359-376.

[R88] Craig W. Reynolds, *Not Bumping into Things*, SIGGRAPH course 27, notes: Developments in Physically-Based Modeling, 1988, G1-G13

[RC90] G. Ridsdale, T. Calvert, *Animating Microwrolds from Scripts and Relational Constraints*, Proc. Computer Animation '90, Geneva, 1990

[REF88] P. de Reffye, C. Edelin, J. Françon, M. JaegerC. Puech, *Plant Models Faithful to Botanical Structure and Development*, SIGGRAPH ‘88, Computer Graphics, Volume 22, Number 4, August 88, pp. 151-158

[REY87] Reynolds C (1987), *Flocks, Herds, and Schools: A Distributed Behavioral Model*,Proc. SIGGRAPH 1987, Computer Graphics, Vol.21, No4, pp.25-34

[REY93] C.W. Reynolds, *An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion*, in: Meyer JA et al. (eds) *From Animals to Animats*, Proc. 2nd International Conf. on Simulation of Adaptive Behavior, MIT Press, 1993.

[RID90] Ridsdale G. *Connectionist Modelling of Skill Dynamics*, Journal of Visualization and Computer Animation, Vol.1, No2, pp.66-72, 1990

[ROT89] Y. Roth-Tabak, *Building an Environment Model Using Depth Information*, Computer, June 1989, pp 85-90

[RTT90] O. Renault, N.M. Thalmann, D. Thalmann (1990), *A Vision-based Approach to Behavioral Animation*, The Journal of Visualization and Computer Animation, Vol 1, No 1, pp 18-21

[S84] Smith A. R., *Plants, Fractals, and Formal Languages*, Computer Graphics, Vol. 18, No. 3, July 1984, pp 1-10

[S95] J. Shen, D. Thalmann, *Interactive shape design using metaballs and splines*, Implicit Surfaces '95, Grenoble, France, April, 1995

[SB93] C. Streit, H. Bieri, *Modellierung mit Lindenmayer-Systemen in der Computergrafik*, Diplomarbeit, Betreuer Prof. Dr. H. Bieri, November 1993

[SGI93] Digital Media Programming Guide: *Audio, MIDI, Video, and Compression*, Document Number 007-1799-010, Silicon Graphics, Inc. Mountain View, California, 1993

[SIM91] K. Sims, *Artificial Evolution for Computer Graphics*, ACM PRESS, Computer Graphics, Vol. 25, Number 4, July 1991, pp. 319 - 328

[SIM94] K. Sims, *Evolving Virtual Creatures*, SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, 1994, pp. 15 - 22.

[SM94] Surf Man, A Triangular-Mesh Sculptor Program, Version 1.2, Sept. 26, 94, Esprit 2, Project 6709: A Real Time and Parallel System for the Simulation of Virtual Humans., 1994

[TAKHA92] Tapio Takala, James Hahn, *Sound Rendering*, Computer Graphics, proceedings, SIGGRAPH '92, Vol 26, No 2, July 92, ACM Press

[TCTP95] Thalmann D., Capin T., Magnenat-Thalmann N., Pandzic I.S., *" Participant, User-guided, and Autonomous Actors in the Virtual Life Network VLNET "*, Proc. ICAT/VRST '95, Chiba, Japan, November 1995, pp.3-11.

[TEN90] Tenenbaum Aaron M., Langsam Yedit Yah, Augenstein Moshe J, *Data structures using C*, International Editions, 1990, sect. 8.3.

[TJ95] Tunbridge A., Jones H., *An L-system Approach to the Modelling of Fungal Growth*, The Journal of Visualization and Computer Animation, Vol. 6: (1995), pp 91 - 107

[TNH96] D. Thalmann, H. Noser, Z. Huang, Chapter: *How to Create a Virtual Life ?*, in Interactive Computer Animation, eds. N.M. Thalmann, D. Thalmann, Prentice Hall Europe, 1996, pp 263 - 291

[TR95] Terzopoulos D., Rabie T. F., Animat Vision: Active Vision in Artificial Animals, Proc. of the Fifth Int. Conf. on Computer Vision (ICCV'95), Cambridge, MA, USA, June, 1995, pp 801-808

[TTA94] X. Tu and D. Terzopoulos, *Artificial Fishes: Physics, Locomotion, Perception, Behavior*, Proc. SIGGRAPH '94, Computer Graphics, pp.42-48.d

[TTP94] X. Tu and D. Terzopoulos, *Perceptual Modeling for the Behavioral Animation of Fishes*, Proc. Pacific Graphics '94, World Scientific Publishers, Singapore, pp.165-178

[W92] Wenzel, E. M., "*Localization in Virtual Acoustic Displays*", PRESENCE: Volume 1, Number 1 (1992), pp. 80-107

[WF90] Wenzel, E. M., S. H. Foster, 1990, "*Realtime Digital Synthesis of Virtual Acoustic Environments*", Computer Graphics, Vol. 24, No. 2

[WG94] Peter Wavish, Michael Graham, *Roles, Skills and Behaviours*, Proceedings of the ECAI-94 workshop on Agent Theories, Architectures and Languages, eds. Woolridge & Jennings, Amsterdam, August 1994.

[Z82] D. Zeltzer, *Motor Control Techniques for Figure Animation*, IEEE Computer Graphics and Applications, 2 (9), 53-59, 1982

Curriculum Vitae

Currently, Hansrudi Noser (1955) is a Ph.D. student and assistant at EPFL. He received 1981 his diploma in Physics (Dipl. Phys ETHZ) from the ETHZ (Swiss Federal Institute of Technology in Zuerich, Switzerland). After working for one and a half year as a researcher for the Institute of Applied Physics at the ETHZ (C-MOS process developing), and for five years as a researcher in the department of Thin Film Electronics of the Balzers AG, Balzers (FL), he received 1992 the diploma in Computer Science (ing. info. dipl. EPFL) from the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL). His research interests include behavioral animation, sound rendering and L-systems.

Publications:

[BNT93] R. Boulic, H. Noser, D. Thalmann, *Vision-based Human Free-Walking on Sparse Foothold Locations*, Fourth Eurographics Animation and Simulation Workshop, Politechnical University of Catalonia, Barcelona-Spain, September 4-5, 1993

[BNT94] Ronan Boulic, Hansrudi Noser, Daniel Thalmann, *Automatic Derivation of Curved Human Walking Trajectories from Synthetic Vision*, Proceedings Computer Animation '94, IEEE Computer Society Press, Los Alamitos, California, 1994

[NPCTT96] H. Noser, I. S. Pandzic, T. K. Capin, N. M. Thalmann, D. Thalmann, *Playing Games through the Virtual Life Network*, ALIFE V, Oral Presentations, May 16-18, 1996, Nara, Japan, pp. 114-121

[NRT95] Noser H., Renault O., Thalmann D., N. M. Thalmann, *Navigation for Digital Actors Based on Synthetic Vision, Memory, and Learning*. Comput. & Graphics, Vol. 19, No. 1, pp 7-19, 1995

[NRTT93] H. Noser, O. Renault, D. Thalmann, N. M. Thalmann, *Vision-Based Navigation for Synthetic Actors*, SIGGRAPH 93, Course notes n 80 "Recent Techniques in Human Modeling, Animation and Rendering", Anaheim, California

[NT93] Noser H. Thalmann D., *L-System-Based Behavioral Animation*, Proceedings of the First Pacific Conference on Computer Graphics and Applications, Pacific Graphics 93, Aug. 1993, World Scientific Publishing Co Pte Ltd, pp. 133-146

[NT94b] H. Noser, D. Thalmann, *Synthetic Worlds* , Chapter: *Towards Autonomous Synthetic Actors*, edited by T.L. Kunii and A. Luciani, 1994 John Wiley & Sons, Ltd.

[NT94c] H. Noser, D. Thalmann, Artificial Live and Virtual Reality, Chapter: *Simulating Life of Virtual Plants, Fishes and Butterflies* edited by Nadia Magnenat Thalmann and Daniel Thalmann, 1994 John Wiley & Sons, Ltd.

[NT95b] Noser H., Thalmann D., S*ynthetic Vision and Audition for Digital Actors*, Computer Graphics forum, Vol. 14. Number 3, Conference Issue, Maastricht, The Netherlands, pp. 325 -336, August 28 - Sept. 1, 1995

[NT95] Noser H., Thalmann D., *Complex Vision-based Behaviors for Virtual Actors*, CEIG'95 V Congreso Espanol De Informatica Grafica, Palma de Mallorca, 28 29 30 Junio, pp 269-284, 1995

[NT96] H. Noser, D. Thalmann, *The Animation of Autonomous Actors Based on Production Rules*, Proceedings Computer Animation'96, June 3-4, 1996, Geneva Switzerland, IEEE Computer Society Press, Los Alamitos, California, pp 47-57

[NTT91] H. Noser, R. Turner, D. Thalmann, L-structures, Travail de diplôme à l'EPFL, Computer Graphics Lab, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland, 1991

[NTTU92] Noser H, Thalmann D, Turner R, *Animation based on the Interaction of L-systems with Vector Force Fields*, Proc. Computer Graphics International '92, pp. 747-761

[TNH96] D. Thalmann, H. Noser, Z. Huang, Chapter: *How to Create a Virtual Life ?*, in Interactive Computer Animation, eds. N.M. Thalmann, D. Thalmann, Prentice Hall Europe, 1996, pp 263 - 291