

Diss. ETH No. 16083

Behavior Modeling and Real-Time Simulation for Autonomous Agents using Hierarchies and Level-of-Detail

A dissertation submitted to the
Swiss Federal Institute of Technology, ETH Zurich

for the degree of
Doctor of Sciences

presented by
Christoph Beat Niederberger
Dipl. Informatik-Ing. ETH
Swiss Federal Institute of Technology, ETH Zurich
born April 21st, 1975
citizen of Dallenwil, NW, Switzerland

on the recommendation of
Prof. Markus Gross, examiner
Prof. Daniel Thalmann, co-examiner

2005

Das Glück im Leben hängt von den guten Gedanken ab, die man hat.

Marc Aurel (121-180)

A C K N O W L E D G E M E N T S

This report on my PhD thesis is the result of a long process which would not have been possible without the help, support, direction, and encouragement of many people. Therefore, I'd like to thank all these who have given me any support to achieve this goal.

Especially, I'd like to thank Prof. Markus Gross for being the supervisor during the past years, taking care of me and of course offering me the opportunity to research in this fascinating area even though not being part of the computer graphics core topics. It was a great pleasure and experience to be part of the GGL with absorbing insights into cutting-edge research in one of the best computer graphics research groups world-wide. Additionally, I would like to thank Prof. Daniel Thalmann for being my co-advisor.

Being part of that group would not have been such a excellent experience without my PhD colleagues even though our research areas had quite often nothing in common. Therefore, a triple chapeau goes to all members of the CGL that made the group to what I experienced during the time I spent here. Within this group, Stephan Würmlin has been a special guide, advisor, and discussion partner all over. The possibility to make my Master's Thesis for him was the starting point and ignition for these past years. Beside the professional advice his floorball capabilities have been very challenging during our seasons at Dietlikon. Special thanks also to Bruno Heidelberger for a humorous office member in the last years although his soccer preferences are distinct from mine. Tim Weyrich needs to be pointed out as an additional special friend since his diversified knowledge and remarks after presentations have been precious and are unforgettable. Also Martin Roth, Daniel Bielser, Matthias Zwicker, Mark Pauly, and Reto Lütolf are beared in remembrance as the guys who already have been here when I started at CGL. They were one of the strong reasons to chose this opportunity.

Without the student projects, my thesis would still be unwritten by today. Therefore, David Bannach, Adam Moravansky, Ruedi Arnold, Bruno Heidelberger, Marcel Wassmer, Dejan Radovic, Christoph Hauser, Jan Rihak, and Daniel Knoblauch (in chronological order) deserve this acknowledgement of my gratitude.

Over all, life at Zürich would have been a far worse experience without the two years with Michael Grossniklaus as my room mate at Weinbergstrasse, the innumerable lunches at "Amore Mio" with Oscar Chinellato and Oliver Bröker, the

discussions with Lisa von Böhmer, and the rise of FCB concurrent to the nemesis of GC.

The last and most important thanks are dedicated to my private vicinity, especially all members of my expanding family and the colleagues Basel. In particular my parents for not being only the financial foundation of my life up to now but also for providing the environment for me to evolve into what I am today – you are the best parents I can imagine! Finally, my life would be worth nothing without the presence of my dear Isabelle! Her love, gratitude, cordiality, endorsement and smiles make me happy every day until forever (and a little bit longer... ;-).

A B S T R A C T

The generation of believable behavior for synthetic characters is a major problem in many products of the entertainment industry. While the film industry can spend hours on designing and modeling the behavior, the time to generate an appropriate behavior is restricted to a few milliseconds in an interactive computer game. Due to this time constraint, the behavior of computer game characters is often very restricted such that the resulting performance is mostly not authentic and the underlying rules are transparent to the user after only a few trials.

In this thesis, we propose a framework for the simulation of artificial behavior for synthetic characters in a dynamic real-time environment, for example a computer game. The character's behavior model is based on the concept of agents. This generic approach relies on a repeated sense-think-decide-act cycle where the middle steps can provide different forms of sophistication. The novelties presented in this thesis are manifold: First, a simple method to generate heterogeneous characters is presented. Secondly, an improved generic and two-dimensional path planning algorithm has been developed which overcomes different problems and yields shorter paths in less time than traditional approaches. And thirdly, a level-of-detail approach for behavior simulation is presented and analyzed which improves the visual quality of the simulation by a smart distribution of the available resources.

A first approach implements purely reactive behavior, thus, the characters only react on the stimulus given by the environment, on internal states, or a combination of both. The underlying reactive agent model consists of a component-based knowledge base which allows to easily compose complex behavior out of simple basic behavior patterns by defining and combining base components into new characters. The combination of multiple patterns can be weighted such that particular base behaviors are preferred to others.

The reactive agent model is then extended to support additionally proactive behavior, hence, the character is enabled to enhance its behavior by exploring possible future states and by selecting the sequence of actions that lead to the most promising state with respect to a goal definition. However, the generation of these plans is computationally expensive and, thus, poses several demands on the system to enable concurrent planning of multiple characters while maintaining a sufficient frame-rate. We use interruptible anytime planning algorithms to search for the currently best partial plan and present different mechanisms that allow for computationally feasible planning in a dynamic environment.

The framework provides a quality of service to the environmental simulation with respect to the time required for the generation of the behavior. This allows the over-all simulation to maintain a practically constant frame-rate independent of the number of agents simulated. The time available to the simulation is split up by a scheduling algorithm according to the needs and the importance of the characters. The importance of the characters is determined by a level-of-detail approach which distinguishes between visible, nearly visible, and invisible agents. This classification makes the visual appearance remain qualitatively high without neglecting any character.

Finally, we demonstrate the functionality and usefulness of our approach by several examples and measurements.

Z U S A M M E N F A S S U N G

Das Erzeugen von glaubwürdigem Verhalten von synthetischen Charakteren ist ein wichtiges Problem von vielen Produkten der Unterhaltungsindustrie. Während die Filmindustrie mehrere Stunden damit verbrauchen kann, spezifisches Verhalten zu erzeugen, ist die zur Verfügung stehende Zeit in interaktiven Computerspielen auf wenige Millisekunden beschränkt. Wegen dieser Einschränkung ist das Verhalten der Charaktere meist sehr beschränkt und deswegen auch nicht authentisch und können vom Spieler nach wenigen Versuchen durchschaut werden.

In der vorliegenden Arbeit wird ein Framework zur Simulation von künstlichem Verhalten von synthetischen Charakteren in einer dynamischen Echtzeitumgebung, wie zum Beispiel Computerspiele, präsentiert. Das Verhaltensmodell der Charaktere basiert dabei auf dem Konzept von Agenten, welchem ein sich wiederholender Wahrnehmen-Denken-Entscheiden-Ausführen Zyklus zu Grunde liegt. Dabei können die beiden mittleren Schritte verschiedenste Formen von Raffinesse aufweisen. Mit der vorliegenden Arbeit werden zugleich verschiedene neue Ansätze präsentiert: Erstens, ein einfaches Verfahren zum Erzeugen von heterogenen Charakteren. Zweitens, ein verbesserter generischer Pfadplanungsalgorithmus für zweidimensionale Probleme der vorhandene Ansätze in Sachen Laufzeit und Ergebnisse übertrumpft. Und drittens verbessert ein Level-Of-Detail Verfahren für Verhaltenssimulation den visuellen Eindruck durch geschicktes Verteilen der vorhandenen Ressourcen.

Ein erster Ansatz realisiert rein reaktives Verhalten, bei dem der Agent nur auf umgebungsbedingte Impulse und interne Zustände reagiert. Das darunterliegende Modell besteht dabei aus einzelnen Wissensbausteinen, welche einfach zusammengesetzt werden können um ausgeklügelteres Verhalten zu erzeugen. Die Kombination der einzelnen Bausteine kann zusätzlich gewichtet werden, so dass einzelne Bausteine gegenüber anderen bevorzugt werden.

Danach wird das reaktive Verhaltensmodell in einem zweiten Schritt erweitert, so dass die Agenten auch noch zusätzlich proaktives Verhalten unterstützen. Infolgedessen kann der Charakter sein Verhalten verbessern, indem er möglichst viele zukünftige Zustände erzeugt und danach die Sequenz von Aktionen, welche zum Besten der erzeugten Zustände führen, auswählt. Die Bewertung der einzelnen Zustände beruht dabei auf einer Formulierung des aktuellen Zieles der Figur. Die Erzeugung einer solchen Sequenz ist rechnerisch sehr aufwändig und stellt deswegen verschiedene Ansprüche an das System, so dass gleichzeitiges Planen von

mehreren Agenten möglich wird und die Bildwiederholungsrate ein interaktives Steuern erlaubt. Dazu werden unterbrechbare Anytime Planungs Algorithmen verwendet um den aktuell besten Plan zu finden und wir stellen dazu verschiedene Mechanismen vor, welche die Veränderungen im dynamischen Umfeld respektieren und dennoch eine rechnerisch nicht zu aufwändig sind.

Das Framework bietet der Simulation der Umgebung einen Service, bei welchen die Qualität des Verhaltens abhängig von der zur Verfügung stehenden Zeit ist. Dies erlaubt es, eine praktisch konstante Bildwiederholungsrate zu halten, unabhängig von der Anzahl Agenten, welche simuliert werden. Die der Simulation zur Verfügung stehenden Zeit wird von einem Scheduling Algorithmus so aufgeteilt, dass den Ansprüchen und der Wichtigkeit eines einzelnen Charaktere Folge leistet. Dabei wird die Wichtigkeit eines Agenten durch einen Level-of-Detail Ansatz bestimmt, welcher zwischen sichtbaren, fast sichtbaren und unsichtbaren Agenten unterscheidet. Diese Klassifikation behält die visuelle Qualität der Simulation hoch, ohne dass ein einzelner Charakter völlig vernachlässigt wird.

Die Funktionalität des präsentierten Ansatzes wird durch mehrere Beispiele und Messungen unterlegt.

C O N T E N T S

1	Introduction	1
1.1	Motivation	2
1.2	Goals of this Thesis	3
1.3	Contribution	4
1.4	Outline	5
2	Virtual Worlds	7
2.1	Artificial Intelligence & Characters	7
2.1.1	<i>Agent Definition(s)</i>	10
2.2	Autonomous Characters	11
2.2.1	<i>Simple Reactive Agents</i>	14
2.2.2	<i>Reactive Agents with Internal State</i>	15
2.2.3	<i>Goal-based Agents</i>	16
2.2.4	<i>Utility-based Agents</i>	17
2.3	Adaptive Agents	17
2.4	Human-like Characters	20
2.5	Behavior Modeling and Architectures	23
2.5.1	<i>Reactive Agents</i>	24
2.5.2	<i>Proactive Agents</i>	27
2.5.3	<i>Summary</i>	29
2.6	Game Agents	29
2.7	Environments	32
3	Reactive Agents	35
3.1	Reactive Agent Model	35
3.1.1	<i>Requirements</i>	36
3.1.2	<i>Model Overview</i>	37
3.1.3	<i>Extensible Agents</i>	38
3.2	Behavior Composition	39

3.3	Knowledge Base	41
3.3.1	Concept	41
3.3.2	Agent Description File	42
3.3.3	Components	42
3.3.4	Agent Generation	44
3.3.5	Group Generation	46
3.4	Behavior Model	51
3.4.1	Perception	51
3.4.2	Situation Recognition	52
3.4.3	Action Execution	53
3.5	Navigation	57
3.5.1	Introduction	57
3.5.2	A* Algorithm	58
3.5.3	Preprocessing	60
3.5.4	Dynamic A*	66
3.5.5	Postprocessing	68
3.5.6	Results in Path-Planning	71
3.6	Results	73
4	Proactive Agents	81
4.1	Proactive Behavior	81
4.1.1	Definitions	82
4.2	Inference Mechanism	84
4.2.1	Goals	85
4.2.2	Goal Selection	86
4.2.3	Goal Evaluation	86
4.3	Search Algorithms	87
4.3.1	Uninformed Search Algorithms	88
4.3.2	Informed Search Algorithms	91
4.3.3	Discussion of Search Algorithms for One-person Problems	92
4.3.4	Two-person Problems	93
4.3.5	Applicability	95
4.3.6	Major Problems	96
4.4	Concurrent Real-Time Planning	98
4.4.1	Anytime Algorithms	98
4.4.2	Anytime Planning	99
4.5	Dynamic Environment	100
4.5.1	Planning Proxies	104
4.6	The Proactive Agent Model	105
4.6.1	Spatio-Temporal Planning	106
4.6.2	Planner Actions	107
4.6.3	The knowledge-base components	108
4.6.4	Blackboard Extension: Time-consuming Subsystems	112
4.7	Results	114

5 Hierarchies and Level-of-Detail	119
5.1 Introduction	119
5.2 Related Work	120
5.3 Hierarchies	121
5.3.1 <i>Hierarchical Path-Planning</i>	121
5.3.2 <i>Hierarchical Flocking</i>	123
5.4 Level-of-Detail	125
5.4.1 <i>Idea</i>	126
5.4.2 <i>Setup</i>	127
5.5 Agent Scheduling	128
5.5.1 <i>Process Scheduling</i>	129
5.5.2 <i>Agent-Scheduling</i>	132
5.5.3 <i>Time Accounts</i>	133
5.5.4 <i>Priority Queue Agent Scheduler Algorithm</i>	134
5.5.5 <i>Results</i>	137
5.6 Hierarchical Control	139
5.6.1 <i>Algorithm</i>	140
5.7 Summary	142
6 System Overview	151
6.1 Introduction	151
6.2 Simulation	152
6.3 Interfaces	153
6.3.1 <i>Control Interface</i>	153
6.3.2 <i>Sensor Interface</i>	154
6.3.3 <i>Effector Interface</i>	155
6.4 Core Components	155
7 Conclusions & Outlook	159
7.1 Summary	159
7.2 Outlook & Future Work	161
A Software Components	163
B User Commands	165
C References	169
D Curriculum Vitae	181

F I G U R E S

Figure 2.1	The basic action-perception cycle according to [Mal97] shows three different types of interaction.	13
Figure 2.2	A simple reactive agent.	14
Figure 2.3	A reactive agent with internal states.	15
Figure 2.4	A goal-based agent.	16
Figure 2.5	An utility-based agent.	18
Figure 2.7	The layered cognitive architecture for synthetic characters by Burke [BID+01].	25
Figure 2.6	The architecture of Blumberg’s interactive creatures [BG95].	25
Figure 2.8	The interaction mechanisms of Funge’s cognitive model in [FTT99] is very similar to Chen’s model in [CBC01]	27
Figure 2.9	The CogAff Architecture as presented by Sloman in [Slo99].	28
Figure 2.10	A classification of different agent architectures.	30
Figure 3.1	Overview of the main components of the reactive agent model.	37
Figure 3.2	The blackboard setup for a reactive agent.	39
Figure 3.3	Combining two basic behaviors into a new agent.	40
Figure 3.4	The agent description file format.	42
Figure 3.5	The attribute component.	44
Figure 3.6	The agent component	44
Figure 3.7	The sensor component	44
Figure 3.11	Adapting an existing agent.	45
Figure 3.8	The situation component	45
Figure 3.9	The action component	45
Figure 3.10	The condition component	45
Figure 3.12	Combining multiple parents into a new agent.	46
Figure 3.13	The description of a group of agents.	47
Figure 3.14	The description of a structured group.	48
Figure 3.15	Modulo rules in the instance section of the description allow for regular patterns within the group.	48
Figure 3.16	A recursive group example	49
Figure 3.17	The description of a recursively defined group.	50
Figure 3.18	The instances generated by the definition in Figure 3.17.	51
Figure 3.19	The action queue mechanism.	53
Figure 3.20	The action execution mechanism as code.	54

Figure 3.21	The checkPreConditions method of the action executor.	55
Figure 3.22	The checkPostConditions method of the action executor.	56
Figure 3.23	Effect of underestimating the distance to the goal	60
Figure 3.24	Effect of overestimating the distance to the goal	61
Figure 3.25	Four different approaches to the discretization problem.	62
Figure 3.26	Discretization into obstacle-free regions.	63
Figure 3.27	Generation method for contours of lakes	64
Figure 3.29	Limitations of different approaches for building the graph	65
Figure 3.28	Building the Graph	65
Figure 3.30	The dynamic A* uses two different strategies	67
Figure 3.31	Path Optimization: The Cone-of-Sight algorithm	69
Figure 3.32	Path adapted to the slope of the terrain	70
Figure 3.33	Comparing the path length of traditional approaches with the dynamic A* approach.	72
Figure 3.34	Comparing the run time of traditional approaches with the dynamic A* approach	73
Figure 3.35	Two different sceneries.	74
Figure 3.36	Two different rendering modes	74
Figure 3.37	The structured group example	75
Figure 3.38	The modulo group example	75
Figure 3.39	The hierarchical group examples	76
Figure 3.40	Reynold's herding algorithm	76
Figure 3.41	A full scene example with over 1500 agents	77
Figure 3.42	The average frame-rate depending on the number of agents	78
Figure 3.43	Absolute timings for the different stages of the reactive behavior model.	79
Figure 3.44	Relative timings for the different stages for the same scenarios.	79
Figure 4.1	The two-layer architecture for proactive agents.	84
Figure 4.2	The goal component	85
Figure 4.3	A general search algorithm.	87
Figure 4.4	A tree generated by the minimax algorithm	94
Figure 4.5	A simple example for repeated states	96
Figure 4.6	A general anytime search algorithm.	100
Figure 4.7	Proxies represent the real agent in the static planning environment	104
Figure 4.8	The planner actions representing the goto-action	107
Figure 4.9	The anytime iterative deepening algorithm core loop	109
Figure 4.10	The extended goal component	110
Figure 4.11	The planner component workflow	111
Figure 4.12	The blackboard mechanism extended for proactive agents	113
Figure 4.13	Sequence of an agent planning	114
Figure 4.14	Multiple agents planning concurrently	115
Figure 4.15	An elephant collecting food	115
Figure 4.16	Sheep-dog behavior	116
Figure 4.17	The average time to achieve a full plan for different depths and fanouts.	117
Figure 4.18	The average planning depth with anytime algorithms in different setups.	118
Figure 5.1	References to other agents within a recursively defined group of agents.	122

Figure 5.2	The leader-follower situation that is used for hierarchical path-planning.	123
Figure 5.3	The hierarchical herding approach based on Reynold's algorithm	124
Figure 5.4	Analysis of the hierarchical herding algorithm	125
Figure 5.5	The extended computer graphics modeling hierarchy with level-of-detail	126
Figure 5.6	The setup scheme for the level-of-detail	127
Figure 5.7	Schematic view of the priority queue algorithm	131
Figure 5.8	The priority queue agent scheduling algorithm	136
Figure 5.9	A comparison of the average activation frequency	138
Figure 5.10	The time needed to resort the queues and calculating the time quanta.	138
Figure 5.11	The average time account for each priority queue in a special setup	139
Figure 5.12	The hierarchical control algorithm	142
Figure 5.13	Overview of the solution	143
Figure 5.14	Screenshots of scenes similar to Table 5.4	146
Figure 5.15	The behavior of the whole solution over time for a particular scenario.	148
Figure 5.16	The behavior of the whole solution over time for another scenario.	149
Figure 6.1	The tasks and interaction of the simulation environment and the agent engine	152
Figure 6.2	The main components of the agent engine.	156
Figure 7.1	A screenshot of the run-time control interface.	164

T A B L E S

Table 2.2	The characteristics of a believable character in an artificial environment as proposed in [HR97].	9
Table 2.1	A summary of proven AI techniques as given in [HR97].	9
Table 3.1	The characteristics of the maps used for comparing our approach with others.	71
Table 4.1	Comparison of the uninformed search strategies.	91
Table 5.1	Comparison of the reactive phase and the proactive mode.	133
Table 5.2	The three different types of presented behavior.	144
Table 5.3	Frame-rate achieved by restricting the total time available.	145
Table 5.4	Comparing the resulting behavior in different setups.	145
Table 6.1	Comparison of different data-structures used for local neighborhood queries.	154

C H A P T E R

1

INTRODUCTION

community has begun to look for new areas of interest. Now, with hardware capable of processing billions of instructions per second and rendering millions of triangles at interactive frame-rates, the focus has begun to change to other levels, for example physically based effects and the behavior of the characters simulated by the computer.

Related to the latter and during the same period, the area of artificial intelligence has been one of the most discussed topics with various promises that either had been successfully achieved or remained the dreams they were. Unlike the above mentioned topics in computer science, the improvements and research advances of the early days in artificial intelligence became less and less when the dream of artificial humanoids has been recognized as very hard to achieve.

In recent years, the research community has become increasingly interested in the use of concepts from artificial intelligence in simulations of humanoid characters. Not only the game industry used the achieved results to improve the character's capabilities and behavior in their products but also the film industry. Completely computer-generated feature movies such as *Toy Story 1&2*, *A Bug's Life*, *Monsters Inc.*, *Finding Nemo*, *Shrek*, *Final Fantasy*, and various others have become very popular and created tremendous economical gains. Beside the known commercial interests, the military has funded the development of simulation environments that can be used to train soldiers in artificial situations and spent millions on development and research of such systems. Of course, there is a noteworthy difference between computer animated films and computer games or military simula-

tions: While the former genre can spend hours on large clusters of powerful computers to generate appropriate behavior and renderings, the latter has to provide an interactive frame-rate and has to generate the behavior of the characters on the fly according to the user's game-play.

According to Hawes [Haw03], we use the term computer game as a virtual world in which one or more users interact, i.e. the player(s). This world is usually populated with artificial characters that either help, hinder or are neutral to the player. The interaction of the player occurs in *real-time*, meaning that each action taken by the player is immediately executed, thus, such a game is said to be *interactive*. This may not cover all existing computer games, for example turn-based strategy games such as chess, but is sufficient for the cases this thesis is dealing with. When comparing this rather loose definition of games to the real world, we see that these features also match to our environment. Therefore, the research results achieved in such environments can eventually be later applicable in the real world with robots.

The simulation and modeling of the behavior interactive of artificial characters in real-time environments has therefore become a challenging topic in computer science where many different research fields merge into a collaboration of different expertises. Artificial intelligence, psychology, biology, and of course traditional computer science build a basis for believable and sophisticated artificial behavior – be it artificial, animal, human, or even super-human. Computer games form an ideal domain to pursue such research by providing a character-based environment. The modeling of these artificial characters is based on the concept of *agents*. The term agent has many different definitions as will become clear in the next chapter of this thesis. Basically, an agent is anything that can perceive and act in an environment. But how it decides to act upon its perception is not defined yet and left open.

This thesis will present the research in the area of agent-based behavior modeling that has been done during the last years at the Computer Graphics Laboratory at ETH Zürich. The aim of this thesis is to devise a agent-model for real-time environments such as game worlds. Additionally, the behavior generated by these characters should provide a level of sophistication that exceeds the one usually experienced in current games. This goal is very vague as the term “intelligence” can not be defined precisely and strongly depends on the domain and type of character on which it is referred to. We will focus on the generation of goal-oriented behavior [RN96] which allows for flexible characters that show an awareness of their current situation and how to use their possibilities to improve it. Since the simulation of goal-oriented behavior is computationally expensive and therefore contradictory to the real-time claims of games, methods to break down this complexity are covered with respect to different aspects of our approach.

The remainder of this chapter starts with the motivation of this thesis before the different goals are specified and explained. A preview on the contributions of the thesis as well as an outline of this report conclude the chapter.

1.1 MOTIVATION

In most of today's computer games, the opponents and background characters often lack a truly sophisticated behavior. Usually based on predefined reactions or scripts,

their manifoldness is very limited and, thus, the provided behavior is predictable, dull and gets boring after some hours of game-play. The common approach to resolve this situation is to add more reactive and scripted behavior routines to give the characters more possibilities and make them appear more flexible. However, this protracts the tedious experience only a little more. It would be desirable to experience an adaptive character whose behavior is based on its experience and will change during the game-play as the world, the characters, and of course the player evolves and gains novel knowledge and competencies.

The lack of more sophistication in computer games has several reasons. First, the computational cost to create an intelligent performance is much higher than just reacting to special events with predefined behavior routines or scripts. The selection of the appropriate routine can be done by comparing the different possibilities and selecting the most promising which is very straight-forward. Second, since the resulting behavior is predefined in all situations, the emerging determinism provides many advantages when debugging a game during development. An adaptive character's behavior is expected to change during its lifetime and it is not obvious how this change will affect the presented behavior – in the worst case, the character will not act sophisticated at all but rather confusing or puzzling. This might be the major reason why commercial game companies have not yet decided to push such a technology, however, when comparing with the progress of physical simulations, this argument might become feeble one day. However, the former reason has become more and more obsolete during the last decade. In a first period, the increased CPU speed and decreased cost of hardware has been used to the last bit in order to increase the details in graphics. But the last years have shown that modern graphics cards got increased capabilities such that many calculations can be swapped out to the graphics processing unit (GPU) resulting in equal or even better visual results and a decreased computational load on the CPU.

The now available computational resources on the CPU have been spent rather on physically-based effects, for example rigid or deformable body simulation. We expect that sooner or later, these calculations will be done on special hardware, too, as the graphics cards do today. Not later than then, the resources on the CPU could be available to an improved behavioral simulation. When the computational resources are available for an additional game experience such as elaborated behavior of the characters, the foundations of such a technology have to be available to the computer game industry. Therefore, we anticipate this development and present (partial) solutions to the core problem of sophisticated characters and related topics.

1.2 GOALS OF THIS THESIS

One major goal of this thesis is to investigate on how to improve a character's behavior by incorporating enhanced decision-taking algorithms into the simulation. And, as known from graphics, it is not only necessary to be able to render a triangle mesh on the screen but also to provide suitable data-structures and algorithms that allow to further reduce the complexity of the computation which is the other major goal of this thesis. We want to investigate methods and models that allow already today to simulate a sophisticated behavior on current hardware and improve these methods to surpass a straight-forward approach.

Therefore, the implementation of a framework suitable for dynamic real-time environments, such as games, is necessary to further investigate these methods. We envision a rather simple world which is populated by agents that act individually but also group together in order to build herds. The characters simulated in this environment should provide fast reactions to environmental changes as well as long-term behavior which tries to achieve one or more goals. Therefore, the generation and execution of plans to achieve such high-level goals has to be integrated into the characters decision-making mechanism. The planning system has to be interruptible to fulfill the requirements proposed by the real-time aspects of the environment.

Hierarchical approaches are considered to be a key to break down complexities in some of the required methods. Such hierarchies can be applied to several different aspects of the overall system. A hierarchical composition of complex agents can reduce the work to be done when designing sophisticated behavior. Also, hierarchically organized groups can be used to break down complex herding behavior.

In order to provide a quality of service, the time spent on behavioral simulation should be restricted such that a constant frame-rate of the rendering mechanism can be achieved. Therefore, the time available for the simulation of the behavior is variable and should be split up between all simulated entities without neglecting any one but such that the visual appearance is coherent and remains appealing. A level-of-detail approach that assigns the available computational resources to these agents that are visually important or at a high level of a hierarchy is expected to achieve this goal. Then, the high-level goals of the complex behavior can be accomplished with such restricted resources.

1.3 CONTRIBUTION

In this thesis, the results of the above mentioned work are presented. The contribution of our work can be summarized as follows:

- The agent model presented in this thesis allows for the weighted combination of simple basic behavior mechanisms into complex and sophisticated agents. The weights can be used to favor some basic behaviors over other by assigning a higher weight. The composition of agents is achieved by a component-based knowledge where components can be reused and cloned. The model also supports the easy generation of heterogeneous groups of agents in order to provide an diversified appearance and, thus, an impression of personality. The agents are capable of perceiving the environment through a sensory system and of acting on the environment by the use of an action system. Both of these are easily adaptable to different environments.
- *Reactive and Proactive Behavior*
A two-layered approach is used to model the agents behavior where the lower level maintains a correct state by reacting to situations in a short-term manner and the higher level tries to simultaneously achieve goals by planning into the future. The planning mechanism is interruptible and allows for simultaneous planning of multiple agents.

- ▶ *Level-of-Detail*
The proposed framework supports a level-of-detail approach that classifies the agents according to their visibility and distance to the viewer. Using this classification, a special scheduling algorithm distributes the time available for the simulation to the agents such that visually important agents receive more time than others.
- ▶ *Hierarchies*
The hierarchical composition of agents as well as a simple mechanism to generate hierarchical groups achieves a simplification in two aspects. The hierarchical composition reduces the work of a behavior designer to achieve complex behavior by reusing elementary and parametrizable components for different types of behavior. The hierarchically organized groups allow for the decomposition of complex tasks such that the overall work-load can be reduced.
- ▶ *Path-Planning*
The path-planning system which is used by the agents to move to a goal destination uses a novel approach which is based on the traditional A* algorithm. This algorithm achieves shorter paths by implementing a dynamic variation of the traditionally static A* algorithm.
- ▶ *Framework*
A framework is presented which allows for the simulation of a large number of both reactive and proactive agents in a dynamic real-time environment. The framework supports constant frame-rates by allowing the behavior simulation to run only for a restricted amount of time. Therefore, the behavior can be made adaptive depending on different resources of the underlying hardware.

1.4 OUTLINE

In the next chapter, an introduction to the field of virtual worlds and their inhabitants will be given. First, the definition of agents is presented which is the fundament of almost all artificial characters. Different types of autonomous characters rely on this concept and are discussed subsequently. Autonomous agents act independently of human interaction and can be extended to adaptive agents which change their behavior based on their experience. Additionally, some missing characteristics and abilities towards human-like behavior are presented before going deeper into modeling issues and agent architectures. This leads to a discussion on game agents which are the target of this thesis. A discussion on the classification of environments concludes this chapter.

The third chapter is devoted to our model of reactive agents whose behavior is determined by reactions to external or internal stimulus. The possibility to create sophisticated agents by composing them out of simple base types influences the model on which our reactive agents rely. The main part of this chapter is devoted to the components which make up such an agent as well as the perception and the action execution mechanism. The navigation system which allows the characters to find a route to a particular destination completes the chapter.

Then, the fourth chapter extends the model from a reactive to a proactive agent which plans ahead in order to find the best actions to take in order to be more suc-

cessful than purely reactive agents. Therefore, a different inference mechanism is needed which relies on searching through possible states in the future. As mentioned before, this is a very time-consuming task which calls for a solution that respects the real-time requirements of the environment. A thorough investigation on the according topics will be followed by the presentation of the proactive agent model.

The fifth chapter uses the afore developed methods and models and deals with approaches to reduce the complexity of the overall system. Hierarchical solutions for groups of characters that break down the computational cost are presented in the first part. The second part introduces a level-of-detail approach that allows to distribute the time available to those characters that need it most to present their behavior without neglecting or preventing others from being simulated. An according scheduling algorithm and control mechanism rely on this approach and further extend the system.

The sixth chapter is denoted to a short system overview with respect to implementational issues. The simulation environment as well as the interfaces between the different components are discussed and presented.

The last chapter provides a conclusion on all these topics and an outlook to possible future work.

VIRTUAL WORLDS

This chapter presents an introduction to virtual environments and their inhabitants. In such environments, the presence of virtual characters is essential for the impression of an alive ambiance. But how can living creatures or even humans be modeled to get the impression that these are like their real counterparts? And what are the requirements that should be met with such characters? How can they be made look intelligent and, more important, which forms of intelligence can be achieved?

This section gives a classification of different forms of intelligence and how characters can be modeled that bear some form of sophisticated or even human behavior. We will start by giving a short introduction on artificial intelligence before presenting the concept of agents as a model for artificial characters. Then, different types and models of agents are characterized before discussing their usefulness in real-time environments such as games. Such environments, though, can be very different, therefore a classification for environments is given in the last part.

2.1 ARTIFICIAL INTELLIGENCE & CHARACTERS

Russel and Norvig provide a broad overview on topics concerning *artificial intelligence* (AI) [RN96]. They classify different forms of intelligent systems by distinguishing their behavior and the way they behave:

► **Acting humanly**

This category is basically the area where the *Turing Test* can be applied to test the human characteristics of a program. This test consists of four main aspects: natural language processing, knowledge representation, automated reasoning, and machine learning. Note that this test does not test the physical abilities. This is done by the *Total Turing Test* which only works in connection with computer vision and robotics.

► **Thinking humanly**

This category is covered by the field of *cognitive science*, which brings together computer models from AI and experimental techniques from psychology. Here, the goal is to imitate human thinking as closely as possible. It is not only the solution of a problem which is interesting, but how the program achieves this solution.

► **Thinking rationally**

This field of AI is based on Aristotle who was one of the first to codify “right thinking”. His famous *syllogisms* lead later to the field of *logic*. There are two main obstacles pointed out here. First, it is not easy to handle uncertain things (which obviously do exist in our world) and, second, there is a big difference between being able to solve a problem “in principle” and doing so in practice.

► **Acting rationally**

Acting rationally means acting so as to achieve one’s goals, given one’s beliefs. The last category leads therefore towards a *rational agent* approach. The reader can imagine an agent as something that perceives and acts. Correct inference from the last category is only a part of a rational agent. A rational agent also needs possibilities to make decisions upon not provably correct things and on events which cannot be reasonably said to involve inference (e.g. reflexes, etc.).

This thesis presents a framework that allows for the simulation of rationally acting characters in a real-time environment. As stated above, these characters are modeled as rational agents which will be defined later in this chapter. The intelligence of such characters is related to several different fields of research in computer science. Jon Doyle et al. give an extensive list of primary areas that present-day AI research covers [DD+96]:

- Knowledge representation and articulation
- Deliberation, planning, and acting
- Autonomous agents and robots
- Multi-agent systems
- Cognitive modeling
- Learning and adaptation
- Manipulation and locomotion seeks
- Speech and language processing
- Image understanding and synthesis
- Mathematical foundations

Out of these topics, not all are directly related to the research presented in this thesis. While the first ones are more important for our needs, the latter are not necessary in our environment. For example, image analysis can be omitted in an artificial environment since we can directly access the world and the objects without understanding the contents of a rendered image of the view of an agent. However, the knowledge representation and the ability to act are indispensable.

In [HR97], Hayes-Roth presents a summary of the proven AI techniques that have been developed. He distinguishes four areas of techniques such as representation, inference, control, and problem-solving architectures as shown in Table 2.1.

TABLE 2.1 A summary of proven AI techniques as given in [HR97].

Representation	Languages, Domain Modeling and Knowledge Engineering Rules, frames, classes, cases, hierarchies, propositions, constraints, demons, certainty factors, fuzzy variables
Inference	Theorem-Proving, Heuristic Reasoning, and Matching Techniques Forward and backward-chaining, unifications, resolution, inheritance, hypothetical reasoning, constraint propagation, case-based reasoning
Control	Goal and data directed, messaging, demons, focus, agenda, triggers, metaplans, scheduling, search algorithms
Problem-Solving Architectures	Rule based, object oriented, frame based, constraint based, blackboard, heuristic classification, task-specific shells

Such fundamental thoughts on AI techniques have been used to specify the characteristics of artificial characters that live in artificial environments. Van Lent and Laird conclude that an effective artificial intelligence engine should support characters that are reactive, context specific, flexible, realistic, and easy to develop [vLL99] as presented in Table 2.2. Reactive means that the character should respond quickly to changes in the environment while the context specific attribute enables the character to act based on prior experiences and activities. Flexible characters should have the possibility to select a tactics on a high level to achieve a goal and also to choose from different forms of low-level behavior that implements a particular tactic. Their description of realistic characters is directly related to Russel and Norvigs first classification type at the beginning of this paragraph. According to van Lent and Laird, a realistic character behaves like humans. Furthermore, a realistic character should have the same strengths as a human but also the same weaknesses. The last characteristic – simplicity in development – builds on a simple but powerful knowledge representation and reusable components, a fundamental requirement also stated by Hayes-Roth in [HR97].

The characteristics of a believable character in an artificial environment as proposed in [HR97].

Reactive	Respond quickly to changes in the environment.
Context specific	Consider prior sensory information and actions.
Flexible	Usage of high-level tactics with multiple low-level implementations
Realistic	Acting like humans with same strengths and weaknesses.
Easy to develop	Simple knowledge representation and reusable components.

With these characteristics in mind, some basic entity has to be found that fulfills all requirements proposed in this paragraph. The next section will present an approach which is able to meet these.

Agent Definition(s)

This chapter introduces a generic approach that is widely used to model artificial characters in virtual environments: the concept of *agents* can fulfill the requirements in the last section. The word *agent* is widely used in computer science but when trying to find a general definition for an agent, we have to consider Wooldridge's remark in [WJ95]:

“ (...) the question *what is an agent?* is embarrassing for the agent-based computing community in just the same way that the question *what is intelligence?* is embarrassing for the mainstream AI community. The problem is that although the term is widely used, (...) it defies attempts to produce a single universally accepted definition.”

It seems that is not straightforward to define what an agent is. And there exist almost as many definitions as researchers have tried to define it. Within this section, we provide several possible definitions from different contexts.

The online dictionary Merriam Webster [MW05] states:

agent

1 : one that acts or exerts power

2 a : something that produces or is capable of producing an effect : an active or efficient cause **b** : a chemically, physically, or biologically active principle

3 : a means or instrument by which a guiding intelligence achieves a result

4 : one who is authorized to act for or in the place of another: as **a** : a representative, emissary, or official of a government <crowns *agent*> <federal *agent*> **b** : one engaged in undercover activities (as espionage) : *spy* <secret *agent*> **c** : a business representative (as of an athlete or entertainer) <a theatrical *agent*>

Thus, we can conclude that an agent represents another entity and can somehow achieve a result by acting alone. This does not define an agent as it is needed in our environment but gives a starting point for a further refinement. Wooldridge considers the question and presents his own notion of agency [WJ95]:

“The perhaps most general way in which the term *agent* is used is to denote a system, be it hardware- or software-based, with the following properties:

Autonomy

Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal states.

Reactivity

Agents perceive their environment and respond in a timely fashion to changes that occur in it.

Pro-Activeness

Agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

Social Ability

Agents interact with other agents (or possibly humans) via some kind of agent-communication language.”

This is a very general definition that can be applied to both hardware and software systems. Nevertheless, all these properties are really needed in order to generate characters that act rationally: Since we do not want to guide each character at any moment and tell it what to do, it needs some autonomy that enables to act as a single entity. Obviously, the need for a repetitive algorithm that models the characters decision process is essential and will be covered in the next section. Note, that learning is also a part of autonomous behavior according to Wooldridge's definition.

In the target environment of this thesis, the reactivity of an agent is very crucial. The characters should maintain a correct state at every time and need therefore to have a short response time to external events. But not only the reaction to certain events is necessary but also the ability to take the future into account to achieve long-term goals gives a character some sort of sophistication: A person that first walks into a door before realizing that it is necessary to open it is not considered to be intelligent. After discussing the agents repetitive mechanism in the next section, we will present generic models for different types of agents.

Furthermore, we may want the characters to act in groups and to interact with each other. The ability to act together, depend on other's decisions and to communicate is summarized in the section *social agents* just before concluding this section by characterizing *game agents* which populate the target environment of this thesis.

2.2 AUTONOMOUS CHARACTERS

As stated in Section 2.1.1, a character or an agent is something that perceives and acts in its environment based upon the perceived information and goes with the psychological model shown Figure 2.1. Therefore, an agent is basically a mechanism that has a mapping from percepts to actions. But the way the resulting action is determined upon the perceived information differs for different types of agents.

Mallot approaches this topic from a psychological point of view [Mal97] since the human being is the most autonomous character we can imagine. He refers to the term *cognition* which refers to a wide variety of mental processes including attention, recognition, planning, reasoning, thinking and language understanding, as well as memory and recollection.

Russell and Norvig split an agent into an architecture and an agent program, which maps the percepts to some actions. In this section we only consider the latter while postponing the agents architecture to a later section. First, we have to think about what kind of mapping is needed and how this can be achieved.

A *rational agent* is considered to be an agent that does the right thing [RN96]. But what is the *right thing*? Russel and Norvig define the right thing as something that causes the agent to be successful with respect to a *performance measure*. This measure determines how successful an agent is. Obviously, there is no globally applicable measure for all types of agents. According to Russel and Norvig, a rational decision depends on four things:

- ▶ The *performance measure* that quantifies the success.
- ▶ The *percept sequence* that the agent has perceived through its sensors.

- The *knowledge* about the environment.
- The *actions* that can be performed by the agent.

This leads to the definition of an *ideal rational agent* who always takes the action that is expected to maximize the agents performance measure, given the percept sequence it has seen so far and all the knowledge of the agent. This means that the ideal agent is perfect with respect to its knowledge and experience. Of course, this definition does not prevent an ideal agent from making mistakes. Nevertheless, such an agent will not repeat its errors if its performance measure is perfect.

An agent is considered to be *autonomous* to the extent that its action choices depend on its own experience, rather than on knowledge built in by the designer [RN96]. Note that this does not require the ability to learn from experience but rather the possibility that the resulting behavior is not totally pre-scripted within all details. An agent is also autonomous if its decisions rely on the perceptions from the environment or from other agents, too.

In principle, the basic action-perception cycle is composed of the same tasks as shown in Figure 2.1. These tasks are executed repeatedly. Thus, any program of an autonomous agent basically consists of the same four steps:

1. **Perception:** The agent gets actual information from its environment through sensors in order to know about its new situation after the last action taken.
2. **Inference:** The agent infers about the world and what has to be done with respect to its percepts.
3. **Selection:** The agent selects one or more actions based on the outcome of step 2.
4. **Acting:** The selected actions are performed.

These four steps together form a mechanism that allows to design different types of agents. The complexity of each step depends on either the architecture of the agent or its environment. For example, in a game environment, perceiving other agents is not as crucial as for a real-world robot that uses a camera and has to interpret the contents of an image. Obviously, perceiving the environment strongly depends on the type of environment and its properties. On the other hand, the inference mechanism and also the selection of the appropriate action are purely agent-dependent since it relies only on the knowledge of the agent which has been updated in the previous step. Acting, however, depends both on the agents architecture and its environment. The way the agents actions are modeled and how they can be applied to the environment has a strong influence on the possibilities of an agent.

Mallot also considers the basic human action-perception cycle and points out three different types of feedback from effectors to sensors as shown in Figure 2.1 [Mal97]:

- a. The internal regulation of the organism is called *homeostasis*. It does not include any type of behavior because all feedback is internally. The adjustment of the body temperature or the blood pressure are examples of this category.
- b. Visible behavior without any changes to the environment is called *acquisitory behavior*. It is used to improve the acquisition of information. Examples for this type are eye movements, active vision, etc.

2.2 Autonomous Characters

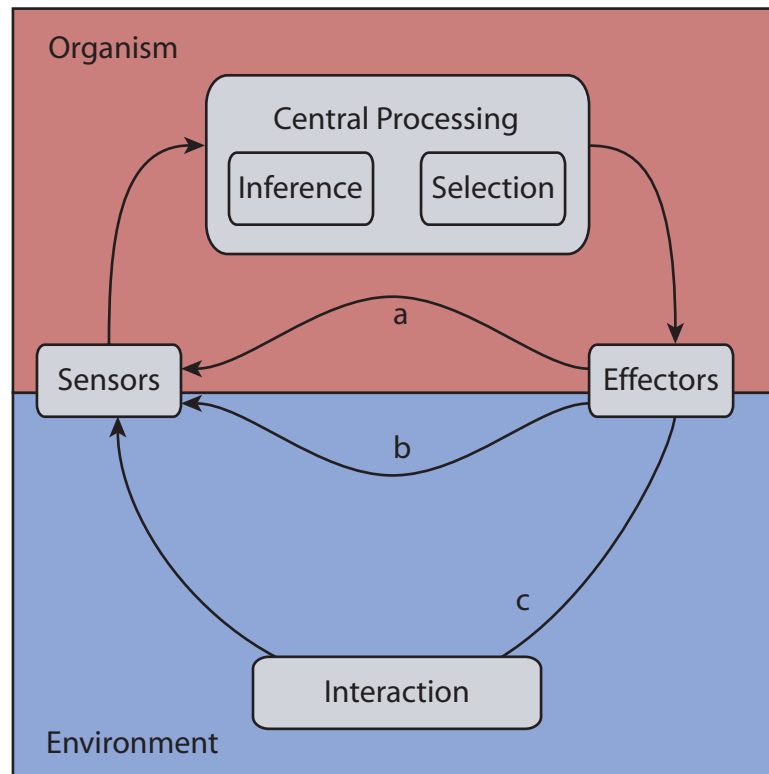


FIGURE 2.1 The basic action-perception cycle according to [Mal97] shows three different types of interaction.
a) The internal regulation, e.g. body temperature, blood pressure
b) Acquisitory behavior, e.g. eye-movement
c) Interactions with the environment

- c. The most complex feedback loop is *interactions with the environment*. These interactions have an impact on the environment. Besides all actions, social behavior and communication are also part of this category.

Thus, a human being can use all these three feedback loops to adjust its behavior to its needs. Therefore, an autonomous agent should also provide these three forms to reconsider its behavior and adapt it with respect to its performance measure.

Lent refers to the inference mechanism as the central component of a AI engine because it sets forth constraints that the other components must meet. Further, the job of the knowledge machine is to apply knowledge from the knowledge base to the current situation to decide on internal and external actions. The most characteristic details of an inference machine are how it implements the think step of the decision cycle and any internal actions of the act step [vLL99].

The most simple way to describe the mapping from percepts to actions is a lookup-table – but this approach fails immediately since such a table would require a huge amount of memory and the time needed to build the table can get infinitely long. Furthermore, and most important, such an agent is not autonomous at all since all actions are predefined. If the environment changes somehow, the agent is completely lost. The next sections introduce different types of agents which make

2.2.1 Simple Reactive Agents

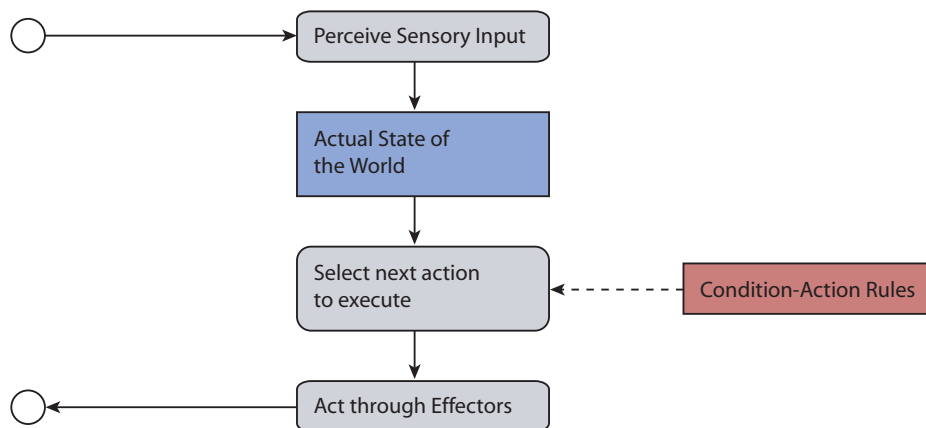


FIGURE 2.2 A simple reactive agent. The decision relies only on the condition-action rules which can be implemented very efficiently.

2.2.2 Reactive Agents with Internal State

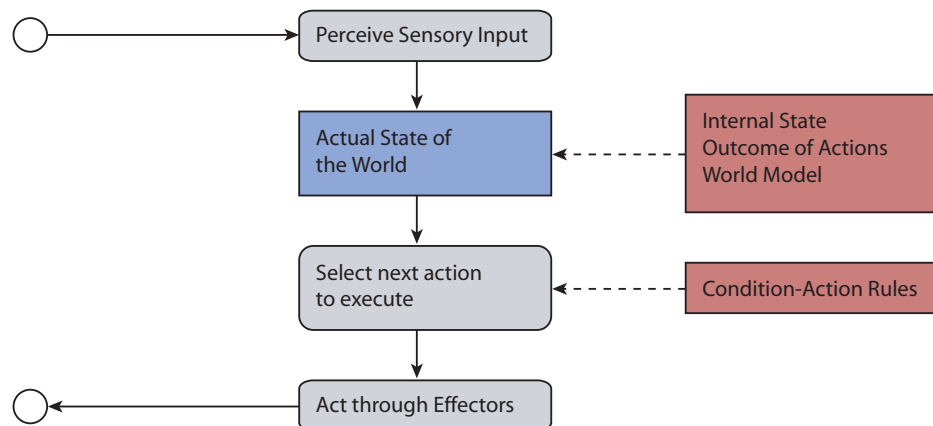


FIGURE 2.3 A reactive agent with internal states. Here, the decision is based on the condition-action rules, too, but also on the internal state, the knowledge about the outcome of possible actions, and a world model.

2.2.3 Goal-based Agents

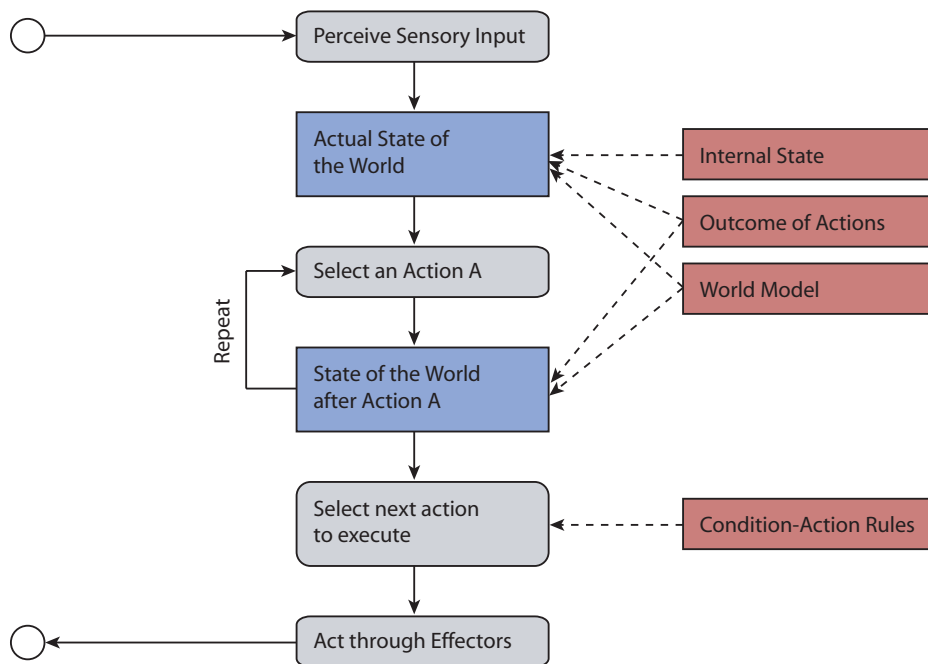


FIGURE 2.4 A goal-based agent. In this architecture, the agent tries to figure out what states can be reached when executing different actions.

2.2.4 Utility-based Agents

2.3 ADAPTIVE AGENTS

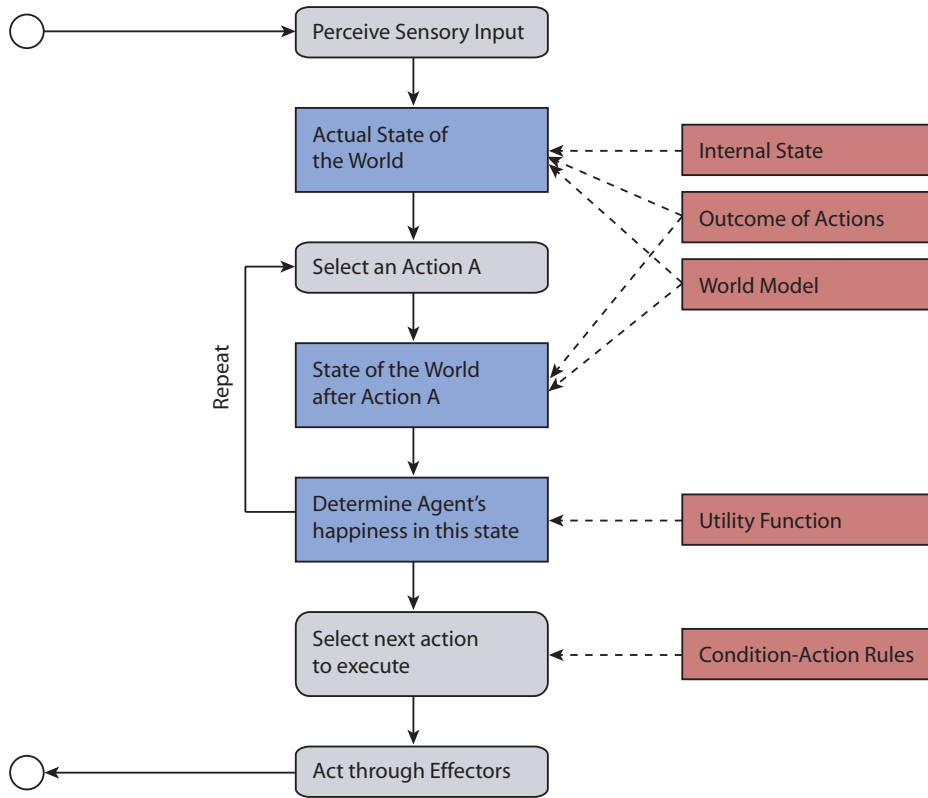


FIGURE 2.5 An utility-based agent. In contrast to the goal-based agent, this approach compares different possible ways to achieve the same state and selects one based on a utility function.

ble of sequential behavior and learning [YB94]. However, recent academic research with artificial characters has moved away from genetic algorithms.

Another form of adaptation is . Learning can be broken down into learning from examples or reinforcement learning. While the first approach, for example neural networks (NN) or support vector machines (SVM), needs a set of samples and their corresponding output, the latter is based on the interaction with an expert who suggests success, failure, reward, and punishment as indicators of the result.

Neural Networks

NNs are derived from the functionality of the brain and try to rebuild neurons, synapses and dendrites in the computer. Several layers of so called units are fully connected by weighted joints and each unit uses an activation function to generate a continuous or discrete output signal [MP69, RN96]. Then, learning is considered as a backtracking process which adapts the weights between the units such that the output of the whole system for a specific input stimulus matches the known output of training examples. Afterwards, the NN can be applied to novel input patterns. The main difficulty when designing NNs is the structure of the network, be it the number of input/output units or the number and size of hidden layers in between [Rip96].

Outside agent-related research, NN have been applied in various fields, for example pattern recognition [Rip96], handwritten character recognition [LBD+90], or controllers for nonlinear plants [Son92]. Lin and Reiter go the opposite way and discuss a theory about forgetting of facts that are no longer true [LR94].

Related to the field of artificial characters are the driving of vehicles [Pom89], back-parking of a truck with a trailer, landing of a space shuttle on the moon surface, or even an animated dolphin learning to swim [GTH98]. Faloutsos et al. present an approach for composable controllers for physics-based characters where each controller learns its preconditions not only manually but also by using a Support Vector Machine [Bur96, OFG97] such that the character can select the appropriate controller by itself [FvdPT01]. As stated above, the commercial computer game *Creatures* uses heterogeneous NN to simulate the brain of the virtual pets which determines the presented behavior [GCM97]. Terzopoulos et al. implemented a virtual fish tank with fishes that navigate upon an artificial perception [TRG96]. These fishes use a reinforcement learning that is based on an objective functional that expresses the success of the current learning state to learn complex motor skills such as turning while swimming, jumping out of the water, or even performing tricks during the jump-phase.

Reinforcement Learning

Reinforcement learning [KLM96, RL95] needs a supervisor who reviews the result and gives a feedback to the system as an expert. Then, the system adapts its interior decision process according to this feedback such that the triggering action (sequence) will be likely to get activated after a positive feedback or fewer in the case of negative feedback.

Conde et al. provide basic information about reinforcement learning for behavioral animation of autonomous agents [CTT03]. In their approach, the virtual agents learn by themselves how to find a route to a destination, i.e. the feedback is given by the environment. In recent years, Blumberg et al. investigated on learning of artificial animals in a virtual environment [TB, YBS00, BDI+02]. In one setup, the animal's motivation system provides reinforcement feedback to the behavioral action selection mechanism such that it adapts its behavior during lifetime. In another approach, the user can interact with a dog and teach it to accomplish certain tasks using a method called "clicker training" which is applied successfully on real dogs. The clicks give the dog a reinforcement feedback who associates certain sequences of actions to commands given by the user. In order to find such sequences, it analyzes the states, actions, and state-action space in between the command and the feedback. After a few training sessions, the dog is able to accomplish simple tasks such as sitting or begging.

Multi-Agent Learning

Among the approaches for multi-agent learning methods, Makar et al. presented an approach for hierarchical multi-agent reinforcement learning [MMG01] where agents learn how to speed up cooperative tasks in a decentralized fashion. The cooperation takes place on the level of sub-tasks rather than on primitive actions. Matsuno et al. present a reinforcement learning method for multi-agent setups and test it with partially observable competitive games, for example "Hearts" [MYIM01]. Tumer et al. focus on individually learning sequences of actions in collective environments such that the global reward is maximized [TAW02].

Although learning is very important for a sophisticated behavior, we do not consider adaptive behavior to be crucial for first results. We believe that the process of learning is rather independent and can be modeled and added to the system at any time. Thus, we decided to focus on the characters decision capabilities rather than adaptive ones.

2.4 HUMAN-LIKE CHARACTERS

Logical reasoning and adaptive behavior are only a part of multiple characteristics which constitute human behavior, thus, the traditional AI techniques alone will not help achieving this goal. Although not a key issue of this thesis, a short but dense overview of the area is given here.

As stated in Section 2.1, human-like behavior goes beyond purely rational behavior as shown by other research fields such as psychology, biology, and others. Whereas rational behavior has a clear and precise scientific foundation, human behavior has not. In this section, we will present some topics where research has become more and more interested in during recent years in order to generate believable agents.

Reilly describes the term *believable characters* as coming from arts and describing characters that "work" [Rei96]. He states that believable characters seem to be alive and that the audience has emotions for or about them. In his Ph.D., Reilly presents three lessons from the arts about the fundamental nature of believability:

1. While AI research is devoted to generate intelligent agents performing difficult tasks, it is sometimes desirable to have stupid characters (e.g. Forrest Gump, Al Bundy).
2. Some areas of AI research try to reproduce natural behavior with cognitive modeling [MDBP95, FTT99, SLL02] as one example. In animated films, however, the characters are not at all realistic, although often very believable.
3. Traditional AI research is not particularly interested in personalities or variations across characters. Nevertheless, personalities are very important for characters with which the observer can identify itself.

We consider different fields related to this topic: Personality, motivation, emotion, and social or individual behavior. Within this section, we will discuss these during the next few paragraphs.

Personality

Personality seems to be one key to human-like agents. One approach to something like personalities has been done by Blando et al. [BLM99]. Their approach builds upon the idea to compose behavior out of basic components like in object-oriented programming. A personification is then the task to generate a character by re-implementing particular behavioral routines. But personality is more than that – is rather bound to emotions than pure individualism. Recently, Kshirsagar and Magnenat-Thalmann developed a model based on the Five Factor Model (FFM) known from psychological studies that uses Bayesian Belief Networks in a layered approach [KMT02]. The FFM defines five basic dimensions of a personality space: Extraversion, agreeableness, conscientiousness, neuroticism, and openness. They demonstrate the usefulness of their model in a text-based dialogue-system. A simpler model for personality is presented by Wilson where the personality of a character is defined in a 3D cartesian space with the axis being extroversion, fear, and aggression [Wil00].

Motivation

Motivations are widely used to influence the characters behavioral or cognitive processes. For example, Canamero uses motivations in the behavior selection process and based on arousal and satiation of an autonomous creature with learning and problem-solving capabilities [Can97]. Burt complements the behavior of animated agents in virtual worlds with motivational constructs in order to better distribute the available time with respect to the agents current goals [Bur98]. The already mentioned work by Yoon et al. creates characters with a transparent motivational behavior based on the behavioral action selection mechanism of Blumberg et al. [YBS00]. The motivation system has strong correlations to the behavior system and consists of a drive system which depends on internal states and an affect system.

Emotions

When reducing the interest to emotions, much research has been done in recent years. Although emotions and computers have very few in common, there exist some models that can be applied to model emotional states. Wehrle presents

research on the motivation for modeling emotions and distinguishes two kinds of modeling approaches: black-box models and process models [Weh98]. Unuma et al. generate emotion-based human figure animations by analyzing empirical periodic movements. A Fourier-series expansion of the data is used to generate a model that produces variations of these movements. The work of Rose et al. attributes verbs (actions) with adverbs (parameters) that allow for expressive behavior [RCB98]. After Reilly's emotional agents [Rei96], Canamero reports on autonomous creatures with emotions triggered by particular events that affect the intensity of behavior [Can97]. Velasquez [Vel], Chown et al. [CJH02], as well as Gmytrasiewicz and Lisetti [GL01, GL02] present architectures where emotions have an influence on the process of decision-making. These approaches all model emotions that can not be directly influenced by the agents actions but rather by environmental feedback. Recently, Wilson presented the Artificial Emotion Engine that generates emotional behavior in form of gestures, motivational actions, and internal states [Wil00]. It divides emotions into three layers: Momentary emotions (e.g. reactions), moods, and personality (as described above), with increasing temporal extent. Tomlinson and Blumberg enable their characters to have and express emotional states using emotional memories based on three variables describing pleasure, arousal, and dominance [TB]. The influence of emotions in multi-agent systems have attracted researchers in recent years, e.g. [BB01, MG03], but are out of the scope of this work.

Social Behavior

Multi-agent systems lead us directly to social, collective, and collaborative behavior – another core property of human-like beings. Simulating the interaction of different characters has been research since the early days of computer science. The field can be divided into *individual-based characters* and *centralized behavior* where a special unit controls the other individuals, however, we will focus on the former approach. Dautenhahn argues that the individual-based approach is based on observations of human behavior [Dau00]: Humans need to pay attention to others and their interactions individually. Opposite to social insects, humans live in individualized societies and need to communicate with each other effectively. While insects recognize each other as group members rather than individuals, humans tend to select particular members of the society as special group members, i.e. friends. Therefore, we have to further differentiate between approaches for simple social animals where the agent does not build special relationships and relationship-based ones. While the former approach is widely used, the latter seems to find its way into research only during the past years.

The seminal paper by Reynolds in 1987 presents a particle-based method to simulate a natural emerging flocking behavior in a group of individuals with only a few simple rules [Rey87]. This model is still used for animated movies and, as mentioned before, will be the basis for a part of this thesis as well. Also, many other approaches were inspired by natural and biological observations. Kube and Zhang studied the controlling of multiple autonomous robots inspired by social insects [KZ92]. Similar to Reynolds boids, their approach is based on a small number of rules: a common goal, non-interference, herding, environmental cues, group detection, and self-facilitation. However, not all approaches are promising as Zaera et al. show. They tried to find an evaluation function to implicitly generate schooling behavior with neural-network controlled agents, but fail [ZCB96]. Another

decentralized approach for real robots has been presented by Martinoli and Mondada who use a bio-inspired model similar to Kube and Zhang's to make robots collecting items [MM98]. Wilkins and Myers centralized approach goes one step further: A multi-agent planning architecture with a central repository for plan-related information makes it possible to explore cooperative problem-solving strategies [WM98]. Lita et al. coordinate goal-driven mobile agents in uncertain environments like in the Canadian Traveler Problem [BNS91].

Musse and Thalmann present a hierarchical model for simulating virtual human crowds [MT01]. Various degrees of autonomy on different levels of individuality allow for a distinctive behavior simulation of crowds. Guided, programmed, and autonomous groups form the basic levels. Similar, Ulicny and Thalmann deal with large numbers of virtual humans with scripted and autonomous behavior that are used to either populate virtual heritage environments and games or simulating emergency situations in real-time [UT01, Tha02]. Recently, Bayazit et al. presented a road-map based approach to flocking for complex environments [BLA02]. O'Sullivan et al. present a level-of-detail approach for conversational and social behavior [OCV+02] which will be discussed in greater detail later.

As mentioned before, a part of Reilly's PhD thesis deals with believable social agents [Rei96]. He points out that social behavior is closely related to individual properties of the character and emphasizes the need for a proper model of other characters. As examples, his focus lies on two social behaviors: negotiation and making friends. Tomlinson and Blumberg's work deals with wolves that exhibit social interactions [TB]. They use context-specific emotional memories (CSEM) to remember previous interactions with other wolves with respect to a emotional experience. Other than the previously mentioned "clicker-trained" dogs, the wolves are not trained based on reinforcement of a human player but rather by the reward that emerges the social environment. Then, this positive or negative experience is associated with the previous activity.

Individual Capabilities

Other than the above mentioned properties of human behavior, there are many individual capabilities that make characters more human-like, such as respect [BDH+01], anticipation [Lai01, KB00], or even creativity [Sau02] and many more which are clearly out of the scope of this thesis.

2.5 BEHAVIOR MODELING AND ARCHITECTURES

After having discussed the characteristics of artificial characters and having shown some rough outlines of control mechanisms for rational agents, this section will go into the details of behavior modeling and control architectures by presenting selected approaches from research. The design and underlying models of these systems vary very much since there is obviously no correct way to implement an agent. First, the low-level behavior selection processes in reactive agents will be discussed before increasing the complexity towards hierarchical or layered approaches, proactive, and high-level cognitive processes.

2.5.1 Reactive Agents

Concerning individual behavior, research has begun to examine how to model the locomotion of animals [Mil88, RH91, GT95, TT94]. With moving characters, the next step was to model the selection of different behaviors that are executed using such locomotion systems. This process is usually located in between the perception and action selection mechanism and extends very simple methods such as Reynold's boids [Rey87].

The game *Creatures* presents a very open and biologically inspired approach [GCM97] which consists of two subsystems: the attention lobe directs the creatures attention towards an object of interest and the decision making lobe consists of a perception, concept, and decision part where each part is essentially a extended neural network. This approach reduces the set of possible actions to "verb object" tuples instead of "subject verb object" triples because of the reduction of possible subjects to one object of interest. The perception part combines the sensory input with the internal state and influences the concept part where event memories are stored and activated based on this perceptive input. The decision part stores concept-action relations that determine the executed behavior. Although very simple, this approach lacks the possibility to model behavior explicitly.

Tu and Terzopoulos' artificial fishes [TT94] already introduce a very simple intention generator which influences the selection of appropriate predefined behavior routines in reactive agents. The intention generator has a limited set of intentions available: avoid, escape, school, eat, mate, leave, and wander. A predefined intention selection mechanism selects one out of these with some persistence in order to prevent the fish from switching intentions rapidly. Additionally, they extend the basic intention generator such that three different types of artificial fishes result: predators, prey, and pacifists.

Based on Blumberg's work and PhD thesis [BG95, BM97] a whole series of consecutive systems have been presented to simulate animated characters based on a modular and biologically inspired approach, mainly with animals such as dogs or wolves. The layered architecture allows the characters to follow high-level goals and select appropriate sub-level behaviors to achieve these goals, although the behavior selection process is reactive rather than goal-based as presented in Section 2.2.3. Some part of the sensory input is achieved by a synthetic vision system. A character is able execute multiple actions at the same time as long as they do not allocate the same resources, for example turning the head can be executed concurrently to wagging the tail.

The architecture of Blumberg's original behavior selection mechanism is composed of five layers as shown in Figure 2.6. The underlying geometry of the character is governed by the physical properties which can be manipulated by changing the values in each degree of freedom. Articulated atomic movements are generated by motor skills that affect the degrees of freedom directly. Then, motor controllers map the motor commands of the behavior system to particular motor skills. The primary task of the behavior system is therefore, to select the "right" set of control signals at each time-step based on the internal state and the state of the environment perceived. Therefore, the system recalculates the importance of all available behaviors during each cycle without restricting the execution to only one – allowing for secondary behaviors and meta-commands. However, the winning behavior has pri-

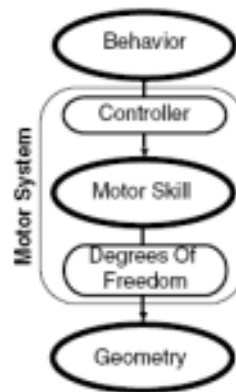


FIGURE 2.6 The architecture of Blumberg’s interactive creatures [BG95].

ority over all other behaviors. The value of importance is influenced by *releasing mechanisms*, which are some sort of precondition, that also generate *pronomes*, a data-structure that allows to use abstract behaviors independent from the object it is instantiated with. *Inhibition* and the *level of interest* are used to arbitrate among competing behaviors. Each behavior has a level of interest that decreases while the behavior is active such that other behaviors get activated. Inhibition is used to provide a robust winner-takes-all arbitration among the behaviors and is explained in detail in [BG95].

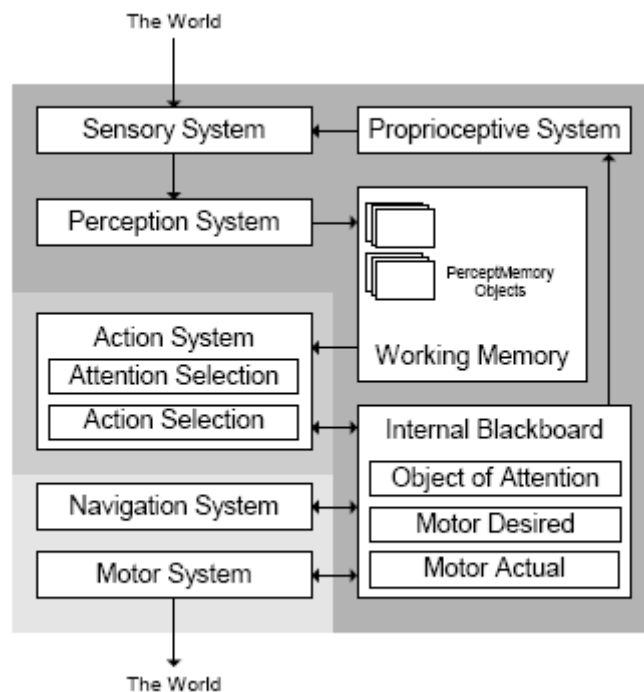


FIGURE 2.7 The layered cognitive architecture for synthetic characters by Burke [BID+01].

Isla et al. refine the behavior system of Blumberg's approach into several distinct systems that communicate through an internal blackboard [IBDB01]. Burke et al. extend this approach again by the proprioceptive channel resulting in a system as shown in Figure 2.7 [BID+01]. This channel is used to emulate self-perception, accept direct self-influence and include self-awareness.

The character's sensory system is used to filter incoming data and events in order to maintain the creature's virtual sensation honest, for example invisible objects are rejected and absolute positions are converted into relative ones [IBDB01]. After sensing a stimulus, it is forwarded to the perception system which uses a *percept tree*, a hierarchical data-structure that classifies the sensory input by calculating matching probabilities. Partial data of the stimulus is passed down the hierarchy only if the parent has recognized the data itself – leading to a more efficient classification. *Innovation* is the process of changing the topology of the percept tree on a reward-driven base. Generated by the perception system, the working memory is filled with objects that represent the sensory history of the objects in the world. A confidence value is associated to each memory object and decayed at every timestep the object is not recognized anymore. The temporal information of these objects allows the character to predict the future and, therefore, to show surprise when prediction and perception do not match.

Having gathered all percepts and collected them into the working memory, the action system has to decide about the actions to take by querying the relevance of all action-tuples in its collection. The relevance corresponds to the amount of reward that is expected when executing a particular action and is calculated by taking into account the intrinsic value of the action, its precondition state and duration. Similar to Badler et al.'s parametrized action representation [BBB+98], each action-tuple addresses fundamental questions:

- *What do I do?* – defines what is actually executed. It can be for example an update of the working memory, an event to the environment, or an instruction to the motor system.
- *When do I do it?* – defines the triggering context that represents the relevance with respect to the current working memory.
- *What do I do it to?* – defines the target of the action which is – during activation – given by the current object of interest or attention.
- *How long do I do it for?* – defines when to end this particular action which can be either a scalar timer value or a postcondition function.
- *What is it worth?* – defines the intrinsic value of the action. This value can be modified through a learning process.

After having selected the current action, the internal blackboard is updated with the appropriate information as depicted in Figure 2.7. The navigation system can override navigational motor commands, e.g. “approach”, until the respective condition is satisfied. This is done before the motor system is able to execute a command.

Another approach based on the reactive agent design has been presented by Kuffner [Kuf98, Kuf99]. His work concentrates on a synthetic vision system, goal-directed navigation, and manipulation tasks but restricts the definition of high-level behavior to simple scripts. However, his characters are articulated human avatars

with a spatial extent and a high number of degrees of freedom. Therefore, navigation as well as manipulation tasks, e.g. grasping, are not trivial to compute. The presented synthetic vision system relies on a flat shaded low-resolution rendering of the current view of the character with each object having a different color. When analyzing this synthetic sight the character can determine all objects currently visible. To build up and maintain the characters own model of the world a simple algorithm is presented which works in static as well as dynamic environments. Then, the task of exploring an unknown environment is discussed and a solution is presented.

2.5.2 Proactive Agents

While the behavior presented by reactive agent models can be very impressive this approach still has its limitations when concerning adaptive or goal-directed behavior. Some researchers argue that human behavior is based on the execution of learned “scripts” rather than planning and interference. Nevertheless, goal-directed behavior offers a possibility to let the agent decide itself how to achieve a goal by taking into account its possible actions and the current state of the (dynamic) environment. This makes it easy to enhance the character’s possibilities by introducing new actions without having to adapt all the before mentioned scripts. Thus, it is not necessary to tell the agent *how* to achieve something but just to tell *what* to achieve.

In 1999, Funge et al. addressed this topic and extended the computer graphics modeling hierarchy by a new top-layer they refer to as *cognitive modeling* [Fun98, FTT99, Fun00]. Their approach introduces a cognitive modeling language (CML) that hides the underlying first-order logic from the user and allows to define the knowledge of the agent, primitive actions and the goals it should achieve. A method to generate nondeterministic scripts is given, too, and in order to deal with uncertainty about earlier sensed information interval-valued epistemic fluents are used. The overall interaction of the system is depicted in Figure 2.8. The high-level rea-

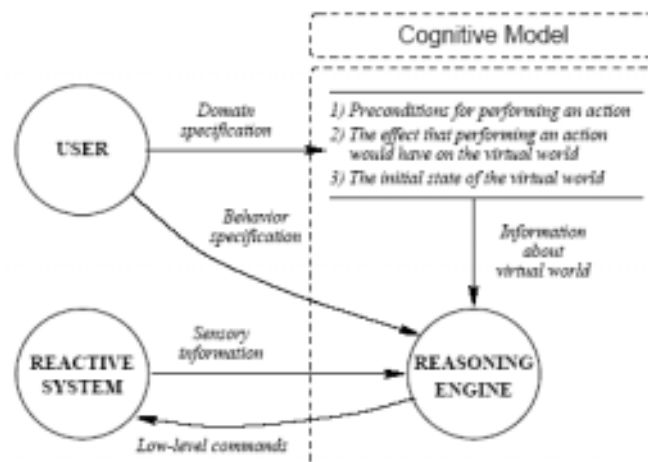


FIGURE 2.8 The interaction mechanisms of Funge’s cognitive model in [FTT99] is very similar to Chen’s model in [CBC01].

soning engine generates a pruned tree of possible action sequences and selects the currently best sequence for execution which is passed to the reactive system resulting in characters that can plan several steps ahead. An automatic camera selection mechanism, a prehistoric world with a T-Rex preying Raptors, and a physically based undersea world with sharks and a merman demonstrate the usefulness of this approach. While this work is based on *situation calculus*, a similar approach presented by Chen et al. relies on *event calculus* and defines a behavior specification language (BSL) [CBC01]. They argue that event calculus, although sharing the same basic ontology¹, has the advantage of representing actions with duration rather than situations which makes it possible to achieve a more narrative modeling language.

More specific towards computer games, Hawes presents an approach for goal-oriented behavior in real-time environments [Haw00, Haw01]. His goal is to provide fast yet flexible responses. His work extends the CogAff architecture [Slo99], a three-layered approach as shown in Figure 2.9. The reactive layer handles fast

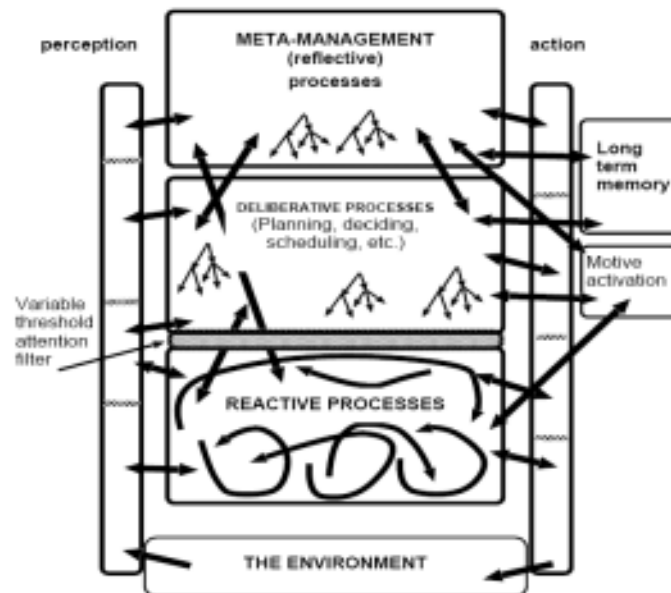


FIGURE 2.9 The CogAff Architecture as presented by Sloman in [Slo99]. The reactive processes are influenced by deliberative processes that generate adaptive behavior. The meta-management layer on top are used for self-monitoring, e.g. to improve the quality of the overall behavior.

decisions based on the current state while the deliberative layer infers on how to achieve goals and how to distribute the available time onto the running processes. The meta-management layer on top handles self-monitoring mechanisms that adjust and influence the deliberative layer in order to increase the overall success and behavior. Hawes transfers the concept of anytime algorithms [ZR95, Gra96] to planning algorithms and presents *anytime planning* [Haw01], a class of interruptible

1. Ontology: A particular theory of the nature of being or existence. In case of knowledge representation, ontology is a formally defined system of things and/or concepts and relations between those.

and qualitatively progressive planning methods which will be used and discussed in-depth later in this report.

The *Belief - Desire (or Goal) - Intention* model has been introduced in the late 1980s based on philosophical sources [Bra87], is often referred to as the BDI model and has been researched thoroughly in the last years but often lacks a concrete implementation. Theoretical formalizations [RG91, Rao96] have laid out the foundations for implementations of rational agents that rely on the BDI model. In this logic, the beliefs represent the agents knowledge about the environment, from perception mechanisms, as well as internal states. The desires represent the state which the agent likes to achieve and the intentions are the means that can be used by the agent to achieve the desires. These intentions are usually implemented as plans with pre-conditions and post-conditions. This may sound similar to classical planning systems such as STRIPS [FN71] but such systems are significantly more sophisticated since the system has to consider reactive behavior as well as proactive behavior concurrently in a dynamic environment which does not allow to predict the actual behavior exactly. Recently, Thangarajah et al. address the gap between theory and practice and present an explicit representation for desires [TPH02]. Geiger and Latzel present an multi-agent system for agent oriented prototyping which relies also on the BDI model and can handle hierarchical plans [GL00]. The agent model of Caicedo et al. uses a similar approach and presents communicative characters and an according behavior engine [CMT01]. Broersen et al. go one step further and introduce the Beliefs-Obligations-Intentions-Desires (BOID) architecture [BDH+01]. They consider obligations as external motivational attitudes whereas desires are their internal correspondence. Their architecture considers all effects of an action before committing it – by taking into account fifteen different types of either internal or external conflicts.

2.5.3 Summary

The presented modeling techniques and architectures differ widely with respect to controllability and adaptability. The former characteristic is based on the amount of influence, the designer has on the resulting behavior while the latter denotes the flexibility of the resulting behavior to changes in the environment. Figure 2.10 shows a summary of the main approaches on a plane that is spanned by both properties. The horizontal axis denotes the amount of influence by the designer where self-contained systems lay on the left side in opposition to user-defined ones on the right side. The vertical axis represents the adaptability of the model where inflexible systems are on the bottom while the ones which can alter the behavior based on the environment are placed on top.

2.6 GAME AGENTS

When looking at the target platform of this thesis, real-time environments such as games, we find that there exist many different roles for characters in a virtual environment. Laird and van Lent provide a list of such roles [LL01]:

- *Tactical enemies*, where many general AI problems have to be solved: Navigation, path planning, spatial reasoning, and temporal reasoning. They also need a perception system with the same capabilities as humans.

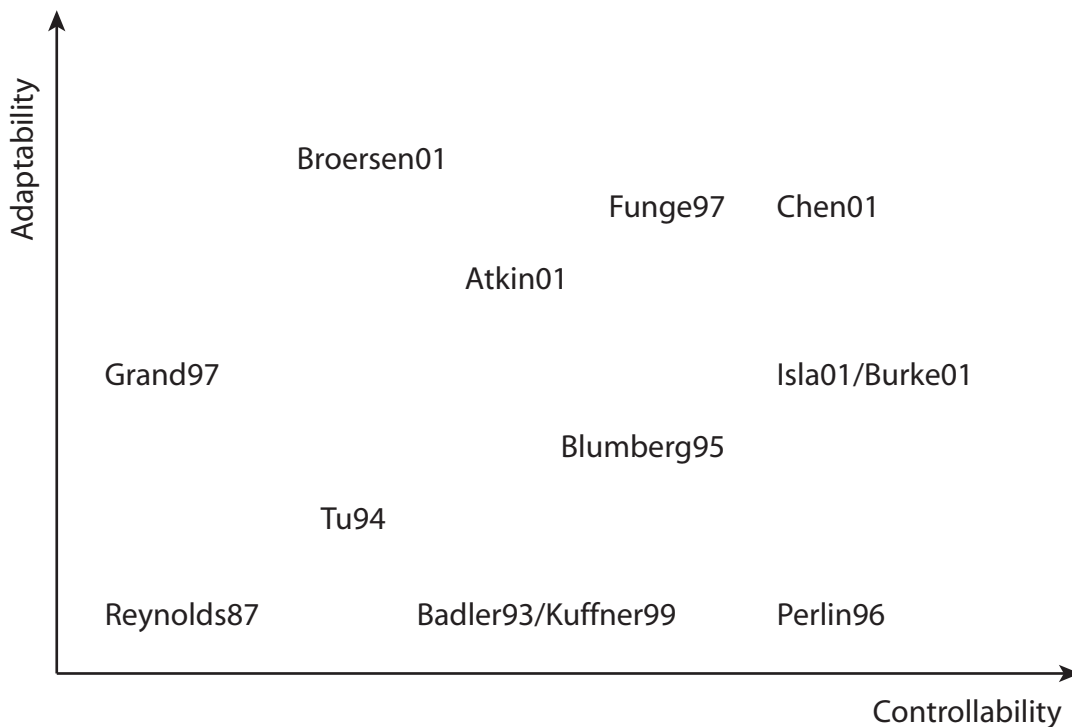


FIGURE 2.10 A classification of different agent architectures. The horizontal range denotes the level of controllability of the simulated agent. The more control is feasible by the designer the more to the right. The vertical axis indicates the level of adaptability. The possibility to adapt to changes in the environment is considered to allow for more flexibility rather than learning novel behavior.

- *Partners* involve similar AI techniques as tactical enemies. However, while enemies emphasize autonomy, partners emphasize effortless cooperation and coordination between both the player and the partner. The partner AI must coordinate its behavior, understand teamwork, model the goals of the human, and adapt to his style.
- *Support characters* are usually some of the least sophisticated AI characters, but they have the most promise to improve games. Since these characters need to exist in a virtual world and generally play a human role in this world, they provide a useful first step towards human-level AI. They must interact with and adapt to human players and provide human-like repossessions.
- *Strategic opponents* often have advantages because most game developers resort to cheating to obtain a challenging opponent. Even with these advantages, most strategic opponents are predictable and easily beaten once their weaknesses are found. In team sports games, these opponents style of play must match a real world team about which the human player is likely to be very knowledgeable. The tasks of strategic opponents can be divided into two categories: allocating resources and issuing unit control commands.

- ▶ AI-driven *units* are used in strategy games, god games, and team sports games. Generally, a high-level command of the player has to be carried out. Because of the large number of simulated units, their computational needs must be kept very low [AWC99]. Therefore, they are often controlled by FSMs and augmented by some path-planning and path-following.
- ▶ The role of *commentators* is to observe the actions and to generate natural language comments suitable to describe the action [Fra99]. The obvious challenge for a commentator is to create a natural language description of the on-going action in the game. The description may include both the moment to moment action as well as key tactical and strategical events that can require complex plan recognition and a deep understanding of the game.

We will focus on the type of tactical enemies and AI driven units. These characters should present behavior with specific characteristics in order to be identified as human-like and accepted by human players. The artificial opponents often have superhuman capabilities in order to provide a challenging antagonist. Therefore, the AI requirements to be fulfilled are different from those of traditional games such as checkers or chess. Nareyek discusses this topic and presents a list of features of modern computer games [Nar00]:

- ▶ **Real-Time:** The time available for taking decisions is very limited.
- ▶ **Dynamics:** The environment the characters live in is highly dynamic.
- ▶ **Incomplete Knowledge:** The character's knowledge of the world is limited.
- ▶ **Resources:** The character's resources (computational as well as memory) may be restricted.

Subsequently, Nareyek states that in modern computer games the goal-directed behavior of the character is often implemented in predetermined behavior patterns or – in more sophisticated approaches – as neural networks. He points out that any-time planning is needed in order to have characters capable of easily adapting to a changing environment. In this case, planning has to take into account the temporal and spatial development of the world and the character should be able to deal with incomplete knowledge.

In computer games, the inference mechanism is usually divided into hierarchically ordered levels. For example, Lent's AI engine for game agents uses three levels: At the top level, there are goals or modes of behavior, the second level represents the high-level tactics to achieve the top-level goals, and the lower level contains the steps and sub-steps, called behaviors, used to implement the tactics [vLL99]. Although Atkins' hierarchical agent control is designed for military purposes rather than games it can be used in real-time environments [AKW+01]. Therein, the sensors as well as the goals can be defined in a hierarchical fashion in order to reduce the complexity.

Isla and Blumberg point out a list of challenges for modern computer games ranging from perception over anticipation to emotions and learning [IB02]. Sensory honesty seems to have a great influence on the authenticity of a character. For example, the opponents should not see through walls or outside their field of view. Imagination is pointed out as a key component of anticipatory behavior. Frustration and curiosity as two examples of emotional responses might affect the believability of an opponent, too. An counterpart who always makes the same mistakes fails to

appear authentic – a behavior that can be circumvented by adaptive characters. Furthermore, communication and coordination skills are often required to achieve a realistic game play. In team sports, for example soccer, ice-hockey, or basketball, good coordination skills are essential for a believable game.

We think that it is not advantageous to concentrate on too much challenges. Thus, we restrict the skills of our target agents to only the necessary but try to find an architecture and model which allows for later enhancements. A simple sensory and action system should be enough to provide a working agent. The key is the inference mechanism whose architecture seems to be promising when using different layers. First, an extendable reactive agent model should provide the basics and will be enhanced in a second step to provide goal-oriented behavior. When designing the over-all architecture carefully, adding adaptive or other human-like behavior should be feasible. We also consider hierarchies as a very promising concept to break down many complexity issues but will apply such only on parts where others did not before.

2.7 ENVIRONMENTS

A character cannot live without an environment. The interaction between the agent and the environment is two-sided. First, the agent's perception is based on the information the environment provides and, second, the agent's actions are executed on the environment and may affect its state.

Russel and Norvig give a general classification scheme for environments [RN96]:

- **Accessible vs. inaccessible**

If the agent has full access to the environmental information it is said to be accessible to that agent. If all aspects necessary to the choice of an action are detectable, it is said to be effectively accessible.

- **Deterministic vs. nondeterministic**

If the current state and the choice of an action completely determines the next state of the environment, then the environment is deterministic. In an accessible and deterministic environment, uncertainty has no effect on the behavior. An inaccessible environment may *appear* as nondeterministic since not all aspects that have an influence are likely to be recognized.

- **Episodic vs. nonepisodic**

In an episodic environment, the agent's experience is based on *episodes*. Each episode is independent, thus, has no influence on other episodes. It consists of perceiving and then acting. Since the episodes are independent, the agent does not need to think ahead.

- **Static vs. dynamic**

An environment is considered to be dynamic if it can change during the deliberative phase of the agent, thus, it has to keep looking at the world while deciding on an action. On the other side, static environments remain unchanged during this phase and are therefore much more easy to deal with.

- **Discrete vs. continuous**

If the number of percepts and actions is limited and they are clearly defined the environment is termed discrete. For example, chess is discrete while a game

environment is continuous. Of course, each environment becomes discrete at some very fine level of granularity but the agent probably won't deliberate on this level.

Based on this classification, the target environment of this thesis is considered to be accessible, nondeterministic, nonepisodic, partially dynamic, and continuous. Partially dynamic because it won't change during one agent cycle but when the deliberative phase takes more than one cycle it will most probably change.

This has several implications: First, the implementation of perception seems not to be challenging due to the accessibility. Second, the nondeterministic and nonepisodic behavior and the dynamics of the environment make it hard to plan ahead without having to consider uncertainty. This has no direct impact on the purely reactive design but has to be considered later when approaching a goal-oriented solution.

REACTIVE AGENTS

This chapter presents the work on the simple characters that only react to external stimulus and their internal state. First, we characterize an appropriate agent model that meets our needs. Then, the composition of different behavior patterns is discussed. The knowledge base of the agents as a basic requirement is described before the implemented behavior model is presented. The navigation facility is thoroughly investigated and presented. The chapter concludes with the results that have been achieved with reactive agents.

3.1 REACTIVE AGENT MODEL

As stated in Section 2.2, reactive agents react upon information that is available from internal states or that is perceived through sensors. As a consequence, this type of agents is restricted to a reduced set of possible behaviors. Within this paragraph, we will discuss the requirements for a reactive agent including a discussion on the expected behavior. Also, the necessary components of an agent that should be able to generically represent all different kinds of reactive agents. At the end, we will present our agent model which is used for the simulation.

Moreover, we want our agents to be composable out of different basic agents which is described in Section 3.2. After having constructed different agents, we want them to build groups such as herds, families, and so on. They should be able to act not only individually but also in a collective of two or multiple agents. In order to not restrict our approach to only the necessary components, the agents use an extensible architecture which allows for easy replacing or adding novel components.

3.1.1 Requirements

Before defining an agent program, the designer has to know about the possible percepts and actions, the goals the performance measure tries to reach, and what kind of environment the agent acts in as described in Section 2.7. Russel and Norvig propose a classification scheme called PAGE (percepts, actions, goals, environment) [RN96]. In order to achieve a proper classification the target scenarios that should be feasible with these agents as defined in Section 1.2 has to be considered first.

In our target framework, an agent's necessary percepts are the current position, orientation, and information about the local environment such as the neighboring agents. Since the environment is purely artificial, it will impose only few restrictions on the agent's perception system as will be discussed thoroughly. The actions an agent can take have to be at least walking, turning, stopping, using objects lying around, e.g. eating food, or interacting with other agents. Most of these actions require the agent to move in the artificial environment. Since the lowest level of instructions is on the level of "goto" or "use", it needs a path-planning mechanism and the ability to follow such a precomputed path in a separate sub-system. The underlying path-planning system that has been designed will be devoted a large section in the second part of this chapter. When talking about the goals the reactive agents should achieve, the ability to stay alive should be emphasized first. This includes avoiding enemies and finding food. Furthermore, the agents engaged in groups should be able to keep with their herd without losing contact.

Because of the generic approach that is intended to be used for a hierarchical composition of simple behavior blocks, the underlying model should support the generation of instances and the composition of existing instances into new base agents. Therefore, each agent has to consist of the same components and support an arbitrary number of specific components, for example sensors or actions. In order to easily define this process, we need an agent specification language that allows for the specification of the simple base agents as well as the complex composition of that ilk. Furthermore, the aggregation into hierarchical and heterogeneous groups has to be considered, too.

In summary, our environment should provide characters with a simple reactive behavior model that form groups but present some individuality. The characters are depicted as animals – mainly elephants or tigers – without the intention to present a behavior specific to these animals but due to the lack of three-dimensional and animated models. Rather, we use animals instead of humans since the behavior model is not the key issue of our approach. We assume that there are behavior models that can generate a specific attitude of a particular animal with some reactive rules. However, in order to provide a simulation of behavior, our model consists of strolling around, avoiding lakes and enemies, staying on the ground, possibly keep in groups, and moving to certain locations if ordered by the user. Of course, this is a very simple model, but the focus of our work is not to generate a novel behavior model but to provide mechanisms to create characters based on particular models and to simulate them appropriately.

3.1.2 Model Overview

The main components of an agent and behavior model have been discussed in the previous chapter. This chapter presents an assemblage of these models that meets the requirements proposed in the last section.

Our model for a reactive agent is simple. Figure 3.1 shows an overview where

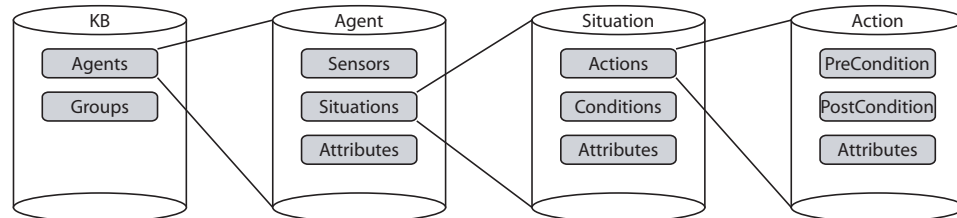


FIGURE 3.1

Overview of the main components of the reactive agent model.

The knowledge base (KB) consists of agents that are composed of sensors, situations, and attributes. Sensors and attributes are simple components. The situations are recognizable upon their conditions and attributes respectively. They provide several actions possible to take. Each action has its pre- and postcondition as well as additional attributes. The groups also shown in the knowledge base are the same components as agents but additionally contain other agents as well.

the main components are depicted. Each agent stores its private knowledge in a knowledge base (KB). As we know from Chapter 2.1 the agent life cycle starts with sensing the environment and updating the internal state, then selecting an appropriate action and finally executing the action.

Consider the example of an animal in a natural environment. It walks around and while nothing happens, it will keep strolling. Once it perceives another animal the reaction to this event depends on the kind of animal encountered. If the animal has the same race and a different gender then a mating behavior might be appropriate. If the animal has different race different reactions are possible. The animal might flee, attack or just ignore the meeting which depends on a second consideration of the situation. Therefore, we propose a two-phase mechanism to determine the appropriate action to take. First, a situation has to be found that matches best. Then, this situation can provide several different possible actions which are determined in a second step.

This scheme is reflected in the components of our agent model. Each agent has a set of sensors that are activated first. Their task is to deliver actual information about the environment to the agent. Afterwards, the agent goes through a set of situations from which the one is selected that has the largest possibility to hold. Therefore, each situation has to provide a function that can estimate the possibility that this particular situation actually holds. Then, each situation can have a set of different actions that could be necessary to resolve the situation. One of these is selected and gets executed afterwards. To allow individual customization, every component used by this model can hold a set of attributes that specify the exact behavior of the component.

A more concise description of the model will be presented in the first part of this chapter. For the moment, we just have to know that each agent is a container of different sensors, situations, actions and that these components can be customized using attributes.

3.1.3 Extensible Agents

In order to allow for an extensible agent mechanism which acts not only reactively but later also proactively as proposed in Section 1.2 our solution is based on the blackboard architecture as presented in [IBDB01]. Blackboards [HR85] were introduced first as an exchange platform for hierarchical planning processes and later as communication mechanism among competing agents. A blackboard can deal with multiple cooperating and competing processes. It allows for different levels of abstraction and provides a simple interface to access the information stored in it. This approach seems to best suit the needs posed in Section 3.1.1 since it can be easily adapted by changing single components or extended by adding a new one. In other terms, the blackboard is used as an intermediate storage for the current results of different processes that influence the overall behavior of the agent and to store fluent internal states. The mentioned processes can access specific information on the blackboard and alter others to influence related processes. Of course, the order in which these processes access the blackboard is very important.

Thus, each blackboard component can have input and output units that are restricted to the basic components of the agent such as sensors, situations, actions or attributes. The blackboard assures that the desired inputs are given and connects each unit to the appropriate component.

For the reactive agent model, the blackboard setup is shown in Figure 3.2. Each component represents a step of the reactive agent cycle and uses information produced by the preceding components. The agent cycle presented in Section 2.2 can be converted directly into several processes. First, the agent senses its environment and updates its knowledge according to the perceived information. Then, we break up the condition-action-rules proposed by Russel and Norvig into a process that identifies the current situation before deciding on the action to take in this situation. At the end, the action system will execute the current action and the process begins again.

The according blackboard information units these processes have an influence on are the following:

- The sensory mechanism updates the knowledge base of the agent directly and, thus, has no direct influence on the blackboard.
- The situation recognition process will select the *current situation* based on the current knowledge of the agent due to its sensory input. This decision might be affected by the currently executed action, too. This process should provide a measurement about the *probability* of this situation for further decisions.
- The reactive action selection mechanism relies on the current situation and evaluates the according *current reaction* and maybe an *object of interest* the reaction is associated to.

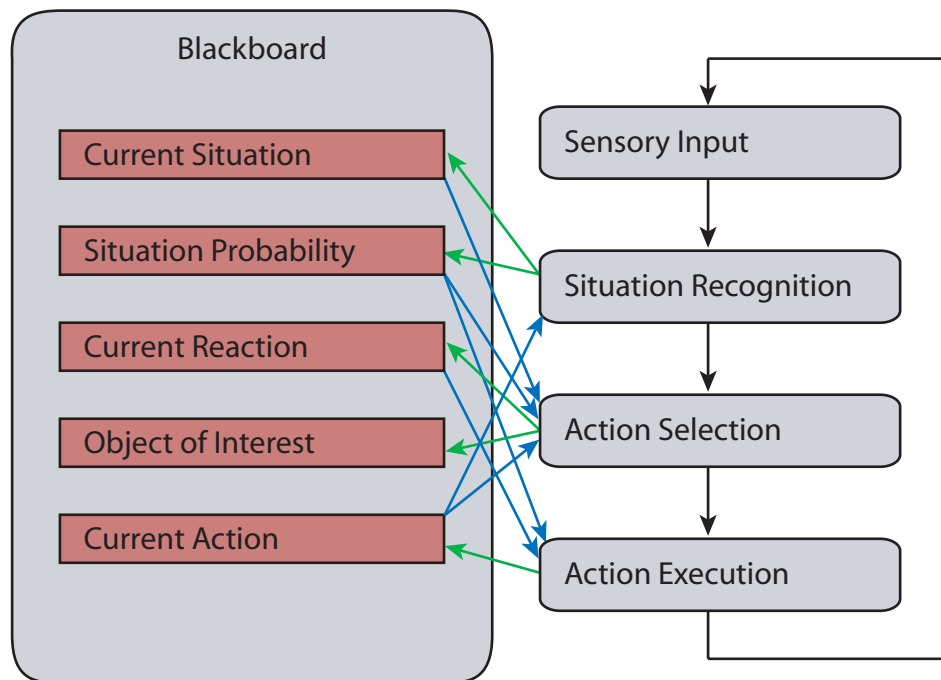


FIGURE 3.2 The blackboard setup for a reactive agent. Four processes read and write five units as described in the text.

- The action system will take the current reaction and decide on the currently executed action and the probability of the recognized situation whether this reaction is more important and should be executed immediately. The feed-back of the current action to the situation recognition can be useful to take decisions there. Therefore, the action system will provide its decision about the *currently executed action* back to the blackboard.

This process will be described thoroughly in the second part of this chapter after the base components of the agent model have been introduced.

3.2 BEHAVIOR COMPOSITION

Like a child using bricks to build a complex building or a software engineer using reusable components and object orientation to construct complex software, we want to use basic behavioral components or agents to build a sophisticated behavior for our agents. Therefore, we intend to compose different types of agents rather than building special ones for every type. However, since many characters should manifest subtle differences in order to display their own personality, we'd like also to weight the inheritance to magnify or scale down the inherited basic behavior. In order to generate additional individual knowledge, we use randomized attributes to further personalize individuals of the same type [NG03].

Blando et al. [BLM99] presented a system which models behavior by using hierarchical inheritance to specialize instances by composing them from basic behavior

types. Subtle differences, however, can only be achieved by defining multiple unique base components. This is the only article known to the author that deals with such a topic.

The principal idea is to first generate basic behavioral patterns that deal with special situations. For example, we can imagine patterns to avoid lakes or enemies, to collect food or to follow another character. When having several of these basic patterns, we can generate more sophisticated characters by combining these patterns and adjusting their attributes. While most of the before mentioned patterns serve the individual skills of a character, the possibility to follow another agent implies collective behavior that depends on another character. Collective behavior can be observed within groups of characters and therefore we present a mechanism to build heterogeneous groups with dependencies that allow for collective behavior.

In our approach, we want not only to collect behavioral elements together but also weight the composed elements with their importance. This means, we can generate characters that behave differently, even though they consist of the same basic patterns. The assigned weights have an influence on the selection of the currently best possible action within the sense-decide-act cycle of an agent. In order to be able to compose characters, we take advantage of the fact that our agents are composed of different components as described in Section 3.1.2. Using that, combining two basic patterns is just the union of the components used in each of the two basic units. When applying weighted inheritance, some components of each base component have to be updated according to the weight when generating the new character.

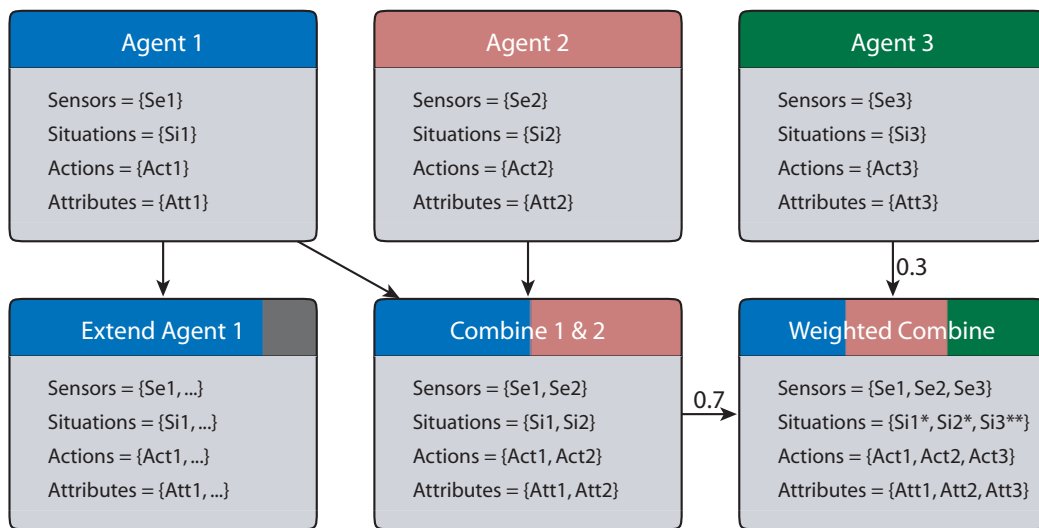


FIGURE 3.3 Combining two basic behaviors into a new agent. A simple extension of an agent is shown on the left. In the middle, the combination of two different agents into a new one. On the right, the weighted combination is shown that has an additional influence on the situations of the novel agent type.

Figure 3.3 shows the three basic composition operations. On the left, Agent 1 is extended by adding new components. In the middle, Agent 1 and Agent 2 are com-

bined in to a new agent which possesses the components of both its parents. On the right, this new agent and Agent 3 are combined using weighted inheritance. This has the same effect on the components as the before mentioned unweighted combination, namely the collection of all components. But this process will additionally affect the probability of the situations in the novel agent.

Within the next sections, the base components of the reactive agent model are introduced and explained. Afterwards, the generation of agents or groups of agents will be explained, including the weighted inheritance mechanism.

3.3 KNOWLEDGE BASE

As almost every agent-based approach we use a Knowledge Base (KB) to store the knowledge of our agents. This is one of the central parts of the whole architecture and separates the individual knowledge of each agent from the common agent model which is the same for all.

First, we discuss different approaches that are possible before describing each KB component more precisely than in the model overview. The process of generating single agents, or groups of agents concludes this section.

3.3.1 Concept

When designing a knowledge base we can distinguish between an agent-centric design and a global one. In the agent-centric design, each agent has its own knowledge which is completely independent of the others knowledge. On the other hand, the common knowledge base allows for common knowledge which is shared among several agents.

The first approach has the advantage that every agent is independent of the others but this approach is disadvantageous when dealing with collective behavior. While the common knowledge base allows easily to exchange information or synchronize several agents, a communication protocol has to be maintained for the agent-centric approach. Nevertheless, when dealing with a concurrent access to the data, e.g. when using threads, one has to be very careful with writing the data.

Our approach uses a common knowledge base, since we do not have a distributed simulation and we would like to easily simulate collective behavior. Nevertheless, each agent has associated its own knowledge components which are separated from the others. Only a few attributes are shared with others. Additionally, we allow for references within the knowledge base that can be used to create dependencies.

The knowledge base distinguishes between *abstract agents*, *agent instances*, and *group agents*. While the first only describes a template, pattern or a skeleton of an agent the second is an agent that exists and can be simulated in the world. The last type is also an agent instance but is connected to a whole group of agent instances.

The *initialization* process of the engine takes an agent description file as input and first constructs a (usually small) set of abstract agents. Afterwards, these abstract agents are used to generate multiple, probably different, agent instances and groups. This process is described later in Section 3.3.4.

3.3.2 Agent Description File

The agent description specification is stored in a XML file and describes the rules how agents are generated during the initialization process. This file contains all information necessary to build the agents for the simulation. The basic format of an agent description is shown in Figure 3.4.

```

<agent name="ID" [type="abstract"]>
  [<parents>
    {<parent type="ID" [value="float"] />}
  </parents>]
  [<sensors> ... </sensors>]
  [<situations> ... </situations>]
  [<attributes> ... </attributes>]
</agent>
<agentgroup name="ID" [type="abstract"]>
  [<parents>
    {<parent type="ID" [value="float"] />}
  </parents>]
  [<sensors> ... </sensors>]
  [<situations> ... </situations>]
  [<attributes> ... </attributes>]
  [<group> ... </group>]
  [<instances> ... </instances>]
</agentgroup>

```

FIGURE 3.4 The agent description file format. The upper part shows the declaration of a single agent while the lower part specifies an agent group.

As stated in the above section, the system distinguishes between abstract agents and instances where abstract agents have to be regarded as templates which generate instances according to rules. Furthermore, agents and groups of agents have to be defined differently. While the first part of Figure 3.4 displays an agent definition, the second part shows an agent group definition. Both the agents and the groups can specify one or more parents from which their knowledge is inherited. Additionally, the sensors, situations, and attributes can be extended. For an agent group, the specification of the instances is necessary and components only needed in the group instance can be declared separately.

The following section provides an overview of the different components and presents their declaration specification.

3.3.3 Components

As described in the above sections, the knowledge base stores everything that makes a difference between two agents, namely *components* that determine the agents behavior. This includes a representation of the agent itself, sensors, situations, actions, attributes, and also conditions.

Attributes. First of all, the most basic entity in the knowledge base is the *attribute* depicted in Figure 3.5. Every other component is a container of attributes and can store as many attributes as desired which are used to specify the component's exact behavior. Attributes can hold either a string, integer, float, boolean or 3D vector value. All attributes except strings allow to specify the value exactly or by the use of

random values within bounds. Random values are described with the range of possible values and are uniformly distributed within this range. Special attributes like the *AttrPtr* or *AgentPtr* reference to other attributes or other agents and can be used for collective behavior. The values of these special attributes are specified with the name of the target which is linked during initialization.

All attributes have an evaluation type which specifies the way of determination of the value. While a value of type *fix* remains constant for all times, the other types are used with rather randomized attributes. The *init* type attributes determine the value once for each instance whereas the *any* type attributes are reevaluated every time a value is requested which is especially useful with random-based variables that rely on an interval of possible values. The *init* type attributes are needed when creating a group in which every instance should have an individual randomized value.

Agents. The *agent representation* stores all data specific for one particular agent as shown in Figure 3.6. This agent representation is a container of any other component except the actions.

An agent can be specified either by creating a totally new one from scratch, by cloning and extending an existing one, or by multiple inheritance that can be extended by weights as described in Section 3.2. In the `<parents>` section, such base agents can be specified. The `<sensors>` section can be used to add additional sensors, just as situations in the `<situation>` section.

Each agent has a predefined set of attributes that can be specified and extended in the `<attributes>` section. The current position, orientation, and velocity as well as the maximal velocity are the base attributes which every agent has access to.

Sensors. A *sensor* is a very simple component as shown in Figure 3.7 which is used to gather external information during the first agent cycle phase. It has a frequency which determines how often the sensor should be activated and a method which starts the sensing process and collects the information. The frequency is very useful for time-consuming sensors such as determining neighboring objects and agents because it allows to reduce the computational load.

Situations. The *situation* component is depicted in Figure 3.8 and basically stores possible actions that can be taken in this situation. Therefore, the situation is an action-container. Additionally, it stores conditions which must hold in order to activate the situation. When testing a situation, first all conditions are tested and when none of them fails then the situation can evaluate its probability. Then, the situation with the highest probability is asked to provide the appropriate action.

Actions. The *action* component is stored in the situations action container. As depicted in Figure 3.9, each action has containers for each the preconditions and the postconditions. Additionally, the action has a *duration* attribute that describes the maximal temporal length. An action can only be executed when all of its preconditions hold and is executed until either the duration is reached or all the postconditions are true. Therefore, it is possible to design actions that have an intrinsic duration, e.g. taking an object, or a variable one, e.g. walking forward for a certain amount of time or to a specific location.

Furthermore, an action can contain sub-actions such that simple predefined sequences of actions are possible. Such hierarchical actions form a container of actions as the situation does. Only the leaf actions can be executed but the inner

container actions also have an influence on the behavior since their pre- and post-conditions can be altered, too. Usually, such multi-actions are not commonly used within the reactive system since all responses to external triggering events can be handled with the provided atomic actions. The multi-actions are used rather in the proactive agent model to store a plan or sequence of actions that is executed.

Conditions. The *condition* component returns a boolean value on demand that describes the state of the condition it represents. As shown in Figure 3.10, the condition can be customized by the use of attributes as all other components, too.

```
<attribute
  [type="{String, Int, Float, Bool, Vect3, AttrPtr, AgentPtr}"]
  name="ID"
  value="{VAL, set, random, randomness}"
  [evaltype="{fix, init, any}"]
/>
```

FIGURE 3.5 The *attribute* component. It is specified by a type, a name, a fixed or random value, and the evaluation type.

```
<agent name="ID" [type="abstract"]>
  [<parents>
    {<parent type="ID" [weight="float"] />}
  </parents>]
  [<sensors> ... </sensors>]
  [<situations> ... </situations>]
  [<attributes>
    <attribute type="Vect3" name="Position" />
    <attribute type="Vect3" name="Orientation" />
    <attribute type="Float" name="Velocity" />
    <attribute type="Float" name="MaxVelocity" />
    ...
  </attributes>]
</agent>
```

FIGURE 3.6 The *agent* component. The parents from which some knowledge is inherited can be specified as well as the weight of the inheritance. Additionally, the agent can be extended by adding particular sensors, situations, or attributes.

```
<sensor type="ID" [frequency="milliseconds"]>
  <attributes> ... </attributes>
</sensor>
```

FIGURE 3.7 The *sensor* component. The frequency determines the rate of activation and is given in milliseconds. A value of zero will activate the sensor every time the agent is activated. The attributes can be used to change the behavior of the sensor, for example the search radius to find neighbors.

3.3.4 Agent Generation

The generation of new agents can be categorized into three different scenarios. First, a new abstract agent can be created by either adapting an existing agent's setup


```
<situation type="ID">
  <attributes> ... </attributes>
  <actions> ... </actions>
</situation>
```

FIGURE 3.8 The *situation* component.
Any situation can be customized by attributes as any other component and stores a container of possible actions to take in this situation.

```
<action type="ID">
  <attributes>
    <attribute name="Duration" />
    ...
  </attributes>
  <preconditions> ... </preconditions>
  <postconditions> ... </postconditions>
</action>
```

FIGURE 3.9 The *action* component.
The actions length can be specified either by a specific attribute named *Duration* or by postconditions. The preconditions must hold before the action can start.

```
<condition type="ID">
  <attributes> ... </attributes>
</condition>
```

FIGURE 3.10 The *condition* component.
It can only be customized by attributes.

into a new one or by composing two or more abstract ones together. Second, agent instances are generated by selecting one or multiple and possibly abstract base agent representations and generating a fully operational agent. Third, setting up groups of agent instances allows for heterogeneous and also hierarchical herds or flocks.

The process of adapting an existing abstract agent representation into a new one is the most simple case:

```
abstrAgent adapt(abstrAgent agent, Description desc) {
  abstrAgent new = agent.clone();
  new.load(desc);
}
```

FIGURE 3.11 Adapting an existing agent.
First, the base agent is cloned, then the description of the new agent is loaded such that new components are added.

First, a clone of the existing agent is constructed by recursively cloning or copying the components into the new one. In a second step, the description of the new agents either overwrites attributes of the existing agent or adds new components to the clone. Therefore, each component has to provide a method which returns a recursive clone of this component or copies its member components into an exist-

ing equal component. The container pattern used throughout the components allows for an efficient way to do that.

During the very first initialization steps, the system provides an ‘empty’ abstract agent that is used to generate the new base agents. Afterwards, these new agents can be used as parents in order to further specialize or enhance additional ones. In the agent specification, these parents are declared within the <parents> section. If this section appears, at least one *default parent* has to be specified.

When combining two or more agents together, the <parents> section will contain more than one entry. Where the first remains the default parent, subsequent parents are declared using the <parent> tag as depicted in Figure 3.6.

As described in Section 3.2, our system allows for weighted inheritance. As shown in Figure 3.6, it is possible to specify a weight to each parent from which knowledge will be inherited. This weight will be used to alter the importance of the situations. Thus, the probability of all situations inherited from a weighted parent will be multiplied by this weight.

```

abstrAgent adapt(abstrAgent default,
                 abstrAgent parents[],
                 Description desc)
{
    abstrAgent agent = default.clone();
    for each parent in parents {
        if (parent.hasWeights) {
            abstrAgent tmp = parent.clone();
            tmp.adjustSituations(desc);
            agent.copy(tmp);
        } else {
            agent.copy(parent);
        }
    }
    agent.load(desc);
}

```

FIGURE 3.12 Combining multiple parents into a new agent. The default agent is cloned and will be extended by the components of all further specified parents. If a parent is inherited with weights, a temporary clone will be adapted according to this specification else the parent is simply copied into the default’s clone. At the end, the additional and individual descriptions are loaded.

The according algorithm is described in Figure 3.12. First, a clone of the default parent is generated. Then, each parent’s components are copied into this new agent. When dealing with a weighted parent, the algorithm first generates a temporary clone of the parent and adjusts the importance of all situations according to the weight. Obviously, the order of the parents is very important, since some parents might have the same components but with different values of their attributes. After having collected all the components of the parents, additional specifications are loaded.

3.3.5 Group Generation

The generation of groups of agents is very similar. Nevertheless, it provides some mechanisms to create heterogeneous and hierarchical groups using very simple

rules. Each group is also represented with an agent that has the same possibilities as an individual. In order to provide a generic mechanism, the members of the group and the group agent can have common parents but can be further specialized using the mechanisms explained in the previous section.

The most simple group pattern is a *flat group* that consists of all the same agents. Figure 3.13 shows an example for the specification. As above, the `<parent>` section defines the parents of all the agents in the group including the group agent. The `<group>` section can be used to further specify the group agent itself, while the `<instances>` section only applies to the real agent instances. The *Count* attribute in the `<instances>` section defines the number of instances in the group excluding the group agent itself.

```
<agentgroup name="Flat">
  <parents>
    <default type="baseagent" />
  </parents>
  <group> ... </group>
  <instances>
    <attributes>
      <attribute name="Count" value="10" />
    </attributes>
    ...
  </instances>
  ...
</agentgroup>
```

FIGURE 3.13 The description of a group of agents. It has the same components as a single agent but additionally the number of instances can be specified in the `<instances>` section. This example will create a group that consists of ten members of type *baseagent*.

A more advanced group pattern is a *structured group*, for example a family. In a structured group, the instances can be individually specified. Every family consists of a father, a mother and possibly several children. The first two instances are pre-defined while the number of children is not. Such a pattern is represented by an *abstract agent group* which can be instantiated by declaring the final count as shown in Figure 3.14. This approach allows for easily creating multiple groups with a different number of members using the same abstract group pattern.

Regular patterns within a group are supported, too. Instead of specifying a distinct member of the group (e.g. the father as first instance), it is possible to specify rules based on modulo calculus. For example, a large herd should consist of one half females and the other half males. Additionally, one out of five might have some additional capabilities. A group description using these modulo rules is depicted in Figure 3.15, where every second ($0_{\text{mod}2}$) is female, every other second ($1_{\text{mod}2}$) is male and one out of five ($4_{\text{mod}5}$) gets some additional behavior.

Even more complex groups can be achieved by using *recursive group* patterns. Using these, tree-formed hierarchical groups can be built. For example, consider the specification in Figure 3.17. The first definition yields an abstract agentgroup where some of the members are instances of the group itself. In order to obtain a recursive definition, at least one instance has to specify the group itself as a parent.

```

<agentgroup name="Family" type="abstract">
  <parents> ... </parents>
  <instances>
    <!-- first agent -->
    <instance number="1"> {declaration of father} </instance>
    <!-- second agent -->
    <instance number="2"> {declaration of mother} </instance>
    <!-- every other agent -->
    <instance number="n"> {declaration of child} </instance>
  </instances>
  ...
</agentgroup>
<agentgroup name="RealFamily">
  <parents>
    <default name="Family">
  </parents>
  <instances>
    <attributes>
      <attribute name="Count" value="5" />
    </attributes>
  </instances>
</agentgroup>

```

FIGURE 3.14 The description of a structured group. First, an abstract pattern is defined which is then instantiated with a certain number of members. In this example, a family consisting of 5 members is created.

```

<agentgroup name="Regular" type="abstract">
  <parents> ... </parents>
  <instances>
    <instance number="0mod2"> {female} </instance>
    <instance number="1mod2"> {male} </instance>
    <instance number="4mod5"> {additional behavior} </instance>
  </instances>
  ...
</agentgroup>

```

FIGURE 3.15 Modulo rules in the instance section of the description allow for regular patterns within the group. Here, every second instance is a female, while every other is male. Additionally, one out of five is also further specified.

Others might be not and will therefore be leaf nodes in the resulting tree of members. Obviously, recursive groups can not have an unlimited number of instances. Therefore, the recursive group definitions must specify the number of instances. Of course, these definitions can contain a default element as presented in Figure 3.6 which is applied to all members of the group. When constructing a recursive group during initialization, the tree is built from top in a breath-first manner. For each instance to generate, the system determines the level in the hierarchy and which rules have to be applied.

The topology of a given recursive group is determined by two parameters: First, the number of allowed agents per group, a , and second, the number of recursive agents per group, r . Note, that the recursive ones are assumed to be the first agents in the group. Figure 3.16 shows an example with $a = 5$ and $r = 3$. In order to determine the correct group, level, and parent some calculations are necessary since

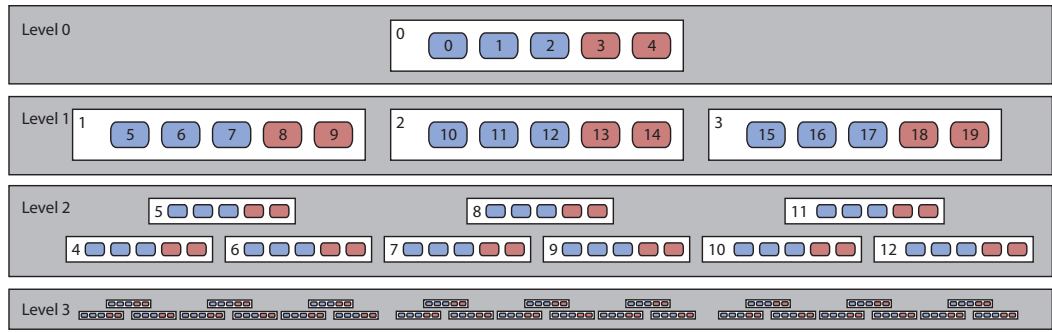


FIGURE 3.16 A recursive group example. The number of allowed agents per group is $a = 5$ and the number of reactive agents is $r = 3$. The numbering of agents, groups, and levels in this figure correspond to the formulas given in the text.

only the current agent counter, a , and r are known. The case with $r = 1$ is trivial and, therefore, the formulas given here assume $r > 1$.

First, the group of the current agent can be determined with

$$id_{group} = \left\lfloor \frac{id_{agent}}{a} \right\rfloor, \quad (3.1)$$

where id_{agent} is the agent's ID and id_{group} the group's ID as given in Figure 3.16. Then, the level l of this group is given by

$$l = - \frac{\log(r) - \log(r \cdot id_{group} - id_{grp} + 1)}{\log(r)} + 1. \quad (3.2)$$

Once the level of the current group is known, the ID of the first group on this level is

$$id_{firstgroup} = \frac{r^l - 1}{r - 1}. \quad (3.3)$$

With that, it is easy to calculate the position of the current group on the current level as

$$pos_{group} = id_{group} - id_{firstgroup}. \quad (3.4)$$

Then, the ID of the parent group is given by

$$id_{parentGroup} = \left\lfloor \frac{id_{group} - 1}{r} \right\rfloor, \quad (3.5)$$

and the ID of the first agent in this group is therefore

$$id_{parentFirstAgent} = id_{parentGroup} \cdot a. \quad (3.6)$$

Based on the previous calculations, the ID of the parent agent in the hierarchy defined by a and r is

$$id_{parentAgent} = id_{parentFirstAgent} + pos_{group} - \left(\left\lfloor \frac{pos_{group}}{r} \right\rfloor \cdot r \right). \quad (3.7)$$

Thus, we can determine the current agent's parent and add a reference to each one. This is necessary when using the hierarchy of the group for collective behavior routines such as herding. If an instance has an attribute which refers to the recursive group itself, this reference will be updated accordingly as the next example points out.

```
<agentgroup name="Recursive" type="abstract">
  <parents>
    <parent type="base" />
  </parents>
  <instances name="R.%i">
    <attributes>
      <attribute name="Count" value="5" />
      <attribute type="AgentPtr" name="ParentAgent"
        value="Recursive" />
    </attributes>
    <instance number="1-2">
      <parents>
        <parent type="Recursive" />
      </parents>
    </instance>
  </instances>
  <attributes> ... </attributes>
</agentgroup>

<agentgroup name="RecursiveInstances">
  <parents>
    <parent type="Recursive" />
  </parents>
  <instances name="Rec.%i">
    <attributes>
      <attribute name="Count" value="23" />
    </attributes>
  </instances>
</agentgroup>
```

FIGURE 3.17 The description of a recursively defined group.

On top, the definition of the abstract pattern is defined. All instances will be derived from the base-type. The `<instances>` section defines the number of agents on each recursion stage and how many of these agents will apply this pattern recursively. In this case the first two of five agents on each stage will reimplement the recursive pattern. Note the attribute *ParentAgent* which is a reference to the recursive group definition. This reference will be automatically updated during agent instantiation such that each instance has a reference to its parent.

Below, the instantiation of the group takes place. A group consisting of twenty agents will be built using the above scheme. The resulting abstract agents and hierarchy is shown in Figure 3.18.

Figure 3.17 shows the definition of an abstract group of five agents where only two apply the pattern recursively. Below, a group of 23 instances of this pattern is generated. The resulting instances to generate such a hierarchical group are shown in Figure 3.18. On the left, the instances of the abstract recursive group are shown and on the right the 23 instances inside the new group. Note, that the abstract group definition on top yields four instances. One for the recursive group itself, and one

for each defined instance, i.e. the two recursive and the default instance. The instances of the abstract group get an attribute `ParentAgent` of type `AgentPtr` with the value being the name of the group itself. During instantiation, this reference is updated dynamically such that it always points to the superior agent as shown in Figure 3.18.

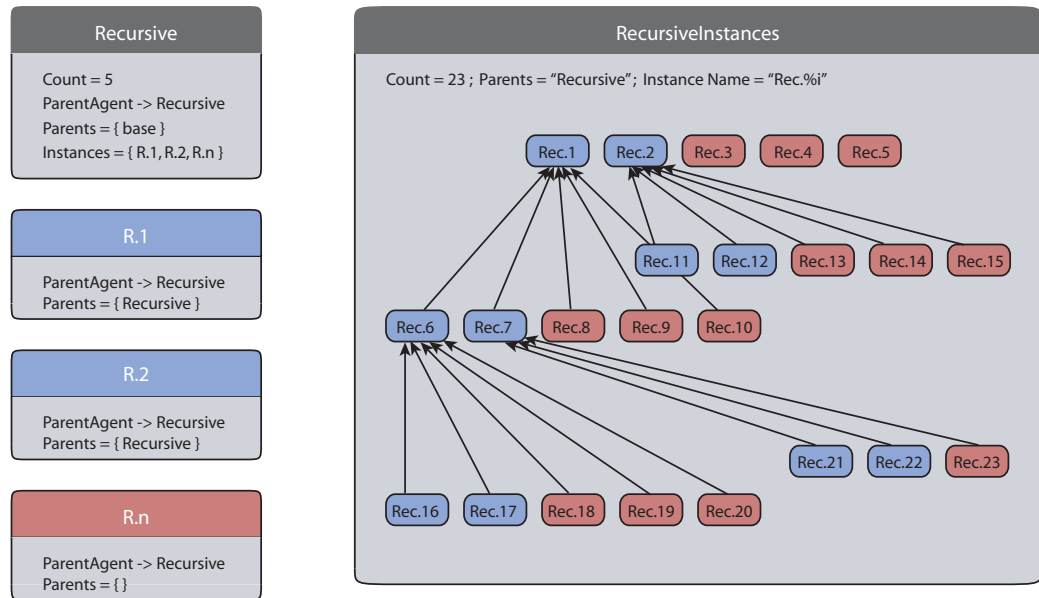


FIGURE 3.18 The instances generated by the definition in Figure 3.17. The abstract group and the according abstract instances defined with the code on top of Figure 3.17 are shown on the left. The instances generated by the code below are shown on the right. The arrows on the right side denote the agent-reference attribute *ParentAgent* given in the definition.

3.4 BEHAVIOR MODEL

So far, we presented the components which are used to build reactive agents. And, the mechanism to generate structured, heterogeneous groups has been presented, too. This chapter discusses the functionality of these components and how they work together in order to provide a generic model for reactive agents. Each component and its specializations should provide a large amount of flexibility which makes them reusable for different solutions.

The agent life cycle described in Section 2.2 determines the overall structure of this section. We start by describing the perception functionality which is then followed by the situation-recognition process which possibly yields an action for execution before the cycle restarts the next time the agent is activated.

3.4.1 Perception

The first step in the agent life cycle is the perception of internal and external information. Since agents in our system have always access to all internal states, the process of determining internal information is not necessary during this step. But

requesting information not directly accessible or computational intensive queries, e.g. neighborhood search, are done during this phase.

Usually, we want the agents to get the most actual information each time they are activated. But the sensory environment provides functionality which allows the user to specify the frequency of activation for each sensor individually. So, each time, the sensor system is activated, it checks each sensor if it has a frequency value specified. Sensors without such a value get activated every time while the other agents remember the last time they were activated. Therefore, the specific value of the frequency is usually slightly higher than the actual frequency of activation which depends on the activation frequency of the agent itself.

Different types of sensors have been implemented and used:

- The *position sensor* retrieves the actual position of the agent's avatar in the simulated world. It accesses the simulation by using a sensor interface to the environment which will be described in Chapter 6. The position sensor is very fast and has therefore the highest frequency.
- The *orientation sensor* and the *velocity sensor* act the same way as the position sensor and provide the current orientation and velocity.
- The *neighborhood sensor* is more complex. It provides a list of other agents within a certain distance or the k nearest neighbors. In order to use this sensor, the simulation environment has to provide such a functionality. Since such an operation is rather time-consuming and the neighborhood might not change that much, this sensor is usually not activated every time.
- The *vision sensor* is a special sensor because it relies on the output of the above mentioned neighborhood sensor. It takes its output and tests each agent from the neighborhood whether it lies within the visible field in front of the agent or outside. Using this sensor, an agents restricted visible field can be simulated.

3.4.2 Situation Recognition

After having perceived the environment, the agent can be considered up-to-date and is ready to decide what action could be taken. This process is fairly simple since each agent has a collection of situations in the knowledge base. These situations can access the agent's updated knowledge and decide about the probability that this situation actually holds.

Therefore, each situation has to provide a method which returns a probability value in $[0,1]$. Each situation is tested and the one returning the highest value is selected as the actual situation. Therefore, only one situation can be considered during one cycle.

After having selected the current situation, there might be multiple actions that should be considered. Therefore, each situation component has a collection of possible actions. The situation decides which action seems to be most appropriate and returns a reference which is passed to the action execution system.

As stated in Section 3.3.4, weighted inheritance has an influence on the situation recognition process. The weight associated with the parent agent has been multiplied with the base weight of each situation. This weight acts as a final multiplier on the evaluation of these situations. Therefore, the estimated probability of a specific

situation is increased if the situation originates from an agent with a weight above 1.0, and on the other hand, the probability is decreased when the weight is below 1.0. Thus, multiple weighted inheritance allows to adjust the probability of a base behavior relative to another. For example, one base agent could provide the ability to avoid lakes while another enables the agent to stay away from enemies. Depending on the weights when combining these two base agents an instance that is near a lake *and* an enemy would decide to either avoid the water *or* the enemy when both situations have the same probability.

3.4.3 Action Execution

The action provided by the situation recognition process should be executed afterwards. This is a crucial part of the whole system since it is possible that each time the agent is activated, another action is appropriate. Then, no action would last more than a few milliseconds resulting in a unnatural behavior. Therefore, we should allow some actions to be fully executed such that the actual situation can be resolved. On the other hand, some situations need a really short response time – for example when approaching a dangerous situation.

Our approach considers two types of actions. First, *reactions* are usually short in execution and need a very short response time after having been selected. Second, *actions* usually have a duration up to some seconds but can be interrupted. Hence, these have to be handled differently in the action system.

An *action queue* in the action system handles this task. As can be seen in Figure 3.19, the queue of pending actions in the action system puts reactions on top for immediate execution. Only one reaction can be in the action queue at any time. If the action queue already has a different reaction on top of it, the reactions are compared and selected with respect to their importance value which is assigned by the situation that returned the action for execution.

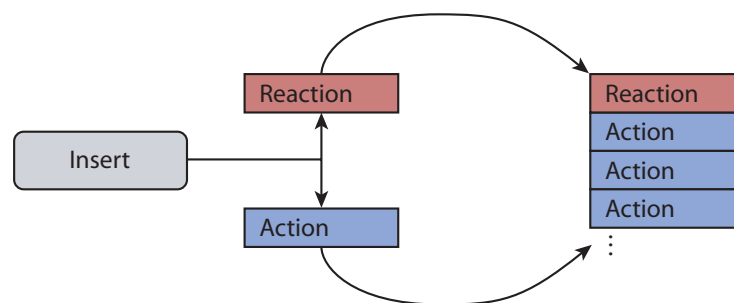


FIGURE 3.19 The action queue mechanism. Reactions are placed on top while actions are appended at the end of the queue in order to provide a temporal order.

For actions, the mechanism is slightly different. Actions are placed at the end of the action queue. If the action to be inserted is already in the queue nothing has to be done. This queue mechanism provides temporal order for different actions that should be executed. For example, the agent first selects an action that approaches an object and when getting near the object, it decides to grab the object. The second

action will wait until the first action has been fully executed and the agent is just next to the object. Afterwards, the grab-action is executed to take the object.

The actions in the queue get dispatched in the *action executor*. The mechanism of the executor is displayed in Figure 3.20. The executor first starts by asking the

```

1: Action _current = 0;
2: if (queue.hasNewReaction()) {
3:     _current = queue.getReaction();
4:     if (_current.checkPreconditions())
5:         mCurrentActions.push(_current);
6: }
7: if (mCurrentActions.empty()) {
8:     _current = queue->getAction();
9:     if (!_current)
10:        return 0;
11:    mCurrentActions.push(_current);
12:    _current = checkPreConditions();
13:}
14:if (!mCurrentActions.empty()) {
15:    _current = mCurrentActions.top();
16:    if (_current.isActive()) {
17:        _current = checkPostConditions();
18:        if (_current)
19:            _current.execute(locale);
20:    } else {
21:        checkPreConditions(locale);
22:    }
23:}
24: return _current;

```

FIGURE 3.20 The action execution mechanism as code. First, the queue is checked for a new reaction (1-6). In case of no reaction and an empty stack of current actions, a new action is fetched from the queue and tested for possible execution (7-13). If the current actions postcondition hold, thus active, and the postconditions are not reached yet, the action is executed.

queue for a new reaction (2). If there is one on top, it is dispatched and its preconditions are checked. In case of possible execution it is placed on top of the *current action stack* (3-5). Using a stack allows for holding back an actually executed action in case of a high-priority reaction. If the current action stack is empty, there might be a new action in the queue (7-8). The algorithm stops if there is actually no action being executed and queued (9-10). Otherwise, the action is pushed on the current action stack (11) and its preconditions checked (12). If the preconditions hold, the action's starting mechanism is executed and it is marked as activated.

After these steps, we can expect an action on the stack whose preconditions have been tested and that might be activated. It is fetched (15) and if active, the action can be executed. But first, the postconditions are verified (17) and if these are not met yet, the action is finally getting executed (19). If the action is not yet active, its preconditions are checked again. The postcondition check mechanism ensures that an finished action is removed from the stack and the queue and returns the next possible action for execution on the stack.

Since actions are hierarchical and can contain multiple subactions (Section 3.3.3), the mechanisms for checking pre- and postconditions are not

straightforward. The `checkPreconditions()` method recursively traverses the actions and checks the preconditions of each action in a depth-first manner as described in Figure 3.21. First, the topmost action of the current action stack is fetched (2). If its preconditions hold, the action is started and activated (8-9). If this action contains subactions (10), these are put on the stack (12), too, the subactions

```

1: function checkPreconditions() {
2:     Action _c = mCurrentActions.top();
3:     bool _finished = false;
4:     bool _condition = false;
5:     while (_c && !_finished) {
6:         _condition = _c.checkPreconditions();
7:         if (_condition) {
8:             _c.startAction();
9:             _c.setActive();
10:            if (_c.isMultiAction()) {
11:                _c = _c.getNextAction();
12:                mCurrentActions.push(_c);
13:            } else {
14:                _finished = true;
15:            }
16:        } else {
17:            return 0;
18:        }
19:    }
20:    return _c;
21:}

```

FIGURE 3.21 The `checkPreConditions` method of the action executor. It handles hierarchical actions that contain subactions.

are traversed recursively and checked for preconditions. When there is no subaction anymore, we have reached a leaf action of the hierarchy which is executable (14, 20). An action that is currently not executable because of unfulfilled preconditions will remain in the queue. Each time the preconditions are not satisfied, a counter is incremented and when a threshold is reached, the action will fail and its `onFail()` method is called. This mechanism provides the possibility to extend an action with some kind of failure analysis mechanism to correct the cause of the failure. It has access to the situation which has generated the action and can call the situation's `onFail()` method, too, in order to provide a direct feedback mechanism. Of course, the action could also change a particular internal variable such that a resolving situation can be activated soon.

The `checkPostConditions()` method checks the postconditions of each subaction in a similar way. It is shown in Figure 3.22. The current action ends if its postconditions are reached or its duration has exhausted (5). Then, it is deactivated and a termination method is called (6-7) before trying to remove the action from the queue (9). This is done only if it is the toplevel action and no subaction. If the current action is a reaction (11), another action might have been interrupted and is waiting to be carried on (14-20). Else, the current action is also popped from the stack and the next action on the stack is taken as the current one. Then, there are two cases to distinguish: First, this action could have an additional subaction which is fetched and tested for execution using the `checkPreCondition()` method (34-

```

1: Action checkPostConditions(ActionQueue queue)
2: {
3:     Action _c = CurrentActions.top();
4:
5:     while ((_c) && (_c.checkPostconditions()
        || _c.checkDuration()))
        {
6:         _c.setInactive();
7:         _c.endAction();
8:
9:         queue->removeAction(_c);
10:
11:         if (_c.isReaction()) {
12:             mCurrentActions.pop();
13:
14:             if (!mCurrentActions.empty())
15:             {
16:                 _c = mCurrentActions.top();
17:                 _c.restartAction();
18:                 return _c;
19:             }
20:             return 0;
21:         }
22:
23:         mCurrentActions.pop();
24:
25:         if (!mCurrentActions.empty()) {
26:             _c = mCurrentActions.top();
27:         } else {
28:             return 0;
29:         }
30:
31:         Action _action = _c.getNextAction();
32:         if (_action)
33:         {
34:             mCurrentActions.push(_action);
35:             _c = _action;
36:
37:             return checkPreConditions();
38:         } else {
39:             _c.setInactive();
40:             _c.endAction(locale);
41:         }
42:     }
43:     return _c;
44: }

```

FIGURE 3.22 The `checkPostConditions` method of the action executor. It checks the hierarchical actions recursively for reached postconditions and returns the next possible action if any.

37). Second, it could have no further subactions, which means that it can be inactivated and ended (39-40). Then, the algorithm traverses up the tree of actions until the action is either entirely executed or another action with executable subactions appears.

The navigation of an individual agent is part of the action execution system in the agent engine. As in [IBDB01], the navigation system is used when an agent decides to move toward a certain location. A `goto` action is the lowest level of abstraction of agent-internal actions. Therefore, when initiating a `goto` action, the agent has to find out if there exists a path that leads to the goal location and how to get there. This topic is discussed in the next chapter.

3.5 NAVIGATION

Open terrain navigation in static environments can be considered as a path-planning problem where the task is to find a sequence of waypoints from a start to a goal location. Additionally, the line segments connecting the waypoints have to be collision-free with respect to obstacles. In most computer games, path-planning is regarded as a graph-search problem where the graph represents the terrains connectivity and spatial extension [DeL00, DeL01]. Usually, the A* algorithm is then used to search the graph which is fast but nevertheless can achieve optimal results. The resulting path is a sequence of collision-free straight-forward movements that lead the agent to a particular position.

This chapter presents a solution to path-planning in static environments by first discussing the requirements of such an approach and by giving an overview of the overall process. In the first section, an introduction to the A* algorithm is given and it is shown how this algorithm can be used to find paths. Then, the preprocessing of an arbitrary map containing obstacles is described which mainly consists of a discretization of the map and the setup of a graph for later search. In Section 3.5.4, the graph search algorithm derived from A* and optimized for path-planning is introduced. The last sections cover the postprocessing steps that reduce the number of waypoints and the results when comparing our approach to comparable conventional methods.

3.5.1 Introduction

For real-time applications such as games, the key to efficient path planning is the spatial decomposition of the environment. Therefore, a human level designer who defines a graph of landmarks is still needed. The task to find a lean and correct graph representing the topology of the landscape is crucially, since the smaller the graph the faster will be the search on the graph. On the other hand, a smaller graph introduces an approximation error by reducing large areas to single points or lines.

A generic path planning algorithm needs to meet several requirements [Rad03]:

- ▶ The resulting paths should have the *lowest possible cost* to prevent any indirection.
- ▶ It should be *fast* to not thwart the simulation process, it should be *correct*, i.e. no collisions occur, and it should be *robust*, i.e. the same request generates the same path.
- ▶ An *automatic* approach is desirable to assure that no human interaction is necessary.
- ▶ Last but not least, the algorithm should be *generic* with respect to different maps, i.e. it should not be optimized for a specific map.

This chapter presents a novel approach to fast path planning in generic terrains that meets the above mentioned requirements. The presented algorithm is a deviation of A* and processes static maps that contain polygonal obstacles. Our solution finds shorter paths which connect arbitrary start and goal locations than traditional approaches.

Generally, *graph search* based approaches use the vertices of a graph to represent feasible points in C_{free} , e.g. landmarks. Variations include methods based on Voronoi decomposition [OY85], and *cell decomposition* methods. For static, two-

dimensional environments with convex polygonal obstacles and a point sized agent, these approaches afford efficient solutions, as surveyed by Schwartz et al. [SSH87]. Exhaustive graph search algorithms, such as A*, constitute the only known optimal algorithms [HKR93]. But the optimality is bound by the approximation error of the decomposition, since the vertices of the graph always reduce an area or line to a single graph node.

For general environments, the most efficient and complete approach is the *roadmap* (or *silhouette*) method [Can88], or its variant, the probabilistic roadmap (PRM) [KSLO96, KLMR95]. Roadmaps reduce the agent's free configuration space C_{free} to a skeleton R_{free} which can be used to search a path from a given start to a goal location in C_{free} . The PRM is very useful in high-dimensional C -spaces. It searches randomly for configurations in C_{free} and connects them to a roadmap.

When dealing with real-time applications, the path planning process has to be as fast as possible. Therefore, many simplifications are made. This often leads to more approximation errors. For example, path planning for the real-time strategy game Star Trek®: Armada is presented in [Dav00]. It applies a quad-tree based decomposition of the playing field to reduce the number of cells. However, their rubber-band algorithm used to make the paths looking natural does not even achieve local optimality.

In order to let an object or character move inside a scene from one location to another, a path has to be planned that guarantees a collision-free translation from the start to the goal position. Hence, the whole task of path planning is usually broken down into four subproblems:

- First, one has to find a suitable discretization of the ground on which one can build a graph. This can be done offline in a preprocessing step. The resulting graph should be as lean as possible to allow a fast search. If the graph is too large, the search will be significantly slowed down. On the other hand, the discretization should be as fine as possible so that the areas corresponding to graph nodes are not too large. This would lead to an approximation error which ends up in suboptimal paths.
- For a specific path request, the task of point location determines the corresponding graph nodes for each the start and goal position. This depends heavily on the chosen discretization.
- Then, the graph has to be searched for a solution which connects the nodes found in the previous step. For static environments, as expected, the A* algorithm is commonly used.
- Afterwards, the resulting sequence of graph nodes needs to be transferred back to the original environment.

Therefore, the main problem seems to find an optimal trade-off between graph nodes representing spatially small areas (less approximation) and a small number of nodes (faster search). Additionally, it will be shown that unintuitive and suboptimal results can occur since the graph is fixed and cannot be adapted to a specific request.

3.5.2 A* Algorithm

The A* Algorithm is a directed breadth-first search used to solve the *shortest path problem* in a graph. It combines the advantages of uniform-cost and greedy searches

by wisely choosing its search-direction. It is the most common search algorithm adopted by the AI community in this context but can be applied to many problems due to its generality and efficiency. This section explains how it works on a rather informal level. Many implementational details which are crucial for its efficiency are omitted for now [Sto00, Rab00].

Given a graph $G(N, E)$ with a set of nodes N and a set of edges $E = (N \times N)$ and a distance function $d: E \rightarrow \mathfrak{R}^+$ that measures the cost of traveling between two neighboring nodes, a so-called *fitness function*

$$f(n) = g(n) + h(n), \quad n \in N, \quad f: N \rightarrow \mathfrak{R}^+ \quad (3.8)$$

is defined, where

- $g(n)$ is the accumulated cost to get from the start node to node n and
- $h(n)$ is a heuristic estimation of the remaining costs to get from node n to the goal node.

In order to know which nodes have been considered during an ongoing search, the A^* algorithm manages two lists: the *open-list* and the *closed-list*. The *open-list* contains the nodes that have to be considered next; they form the front-line of the search. The *closed-list* contains the nodes already visited. At the beginning, only the start node is in the open-list and the closed-list is empty.

The A^* algorithm consists of *expanding* the node p from the open-list whose fitness function is minimal. *Expanding* a node means that its neighbors $n_i \in N$ are inserted in the open-list and that their fitness function is evaluated. Their accumulated cost $g(n_i)$ is computed by simply adding the cost to reach them to the accumulated cost so far:

$$g(n_i) = g(p) + d(p, n_i) \quad (3.9)$$

After p has been expanded, it is inserted in the closed-list. The algorithm stops when it tries to expand the goal node. In this case, a path was found. When there are no more nodes to expand, i.e. the open-list is empty and the goal node has not been expanded so far, there is no path leading from the start node to the goal node.

Note that there might be more than just one possibility to reach the same node. Therefore, the A^* algorithm has to keep track of the whole node sequence constituting the solution up to a specific node. For this purpose, every node inserted in the open-list obtains a pointer to its antecedent node, i.e. the node being expanded. This way, the complete path can be reconstructed by following these pointers from the goal node back to the start node. If two paths are found leading to the same node n , i.e. a node is inserted a second time into the open-list, the path with higher cost $g(n)$ is discarded since it is evidently longer.

The heuristic estimation of the remaining costs to get from an arbitrary node to the goal node impacts both the quality and the efficiency of the search. As long as the heuristic underestimates the real cost to the goal, the shortest path in the graph is guaranteed to be found. Considering Equation 3.8, it is obvious that in this case the fitness function never overestimates the total cost of a path. Since A^* always chooses the node with a minimal f , the search cannot end by expanding the goal node of a path that is longer than the shortest path (see [RN96] for a formal proof).

It is crucial that the heuristic estimate is fairly accurate [Sto00]. Figure 3.23 illustrates the effect of having a heuristic function that underestimates the real costs too much. The left figure shows the state of the search when deploying an exact estimation of the distance to the goal. The grey nodes are the start and end node respectively. The blue ones denote the front-line of the search, i.e. the nodes in the open-list. The red nodes are expanded nodes, i.e. the nodes found in the closed-list. As can be seen, a minimal number of nodes has been expanded. The right figure shows the same search but this time using a heuristic function that underestimated the distance to the goal by a factor of 0.625. This time, four times more nodes have been expanded.

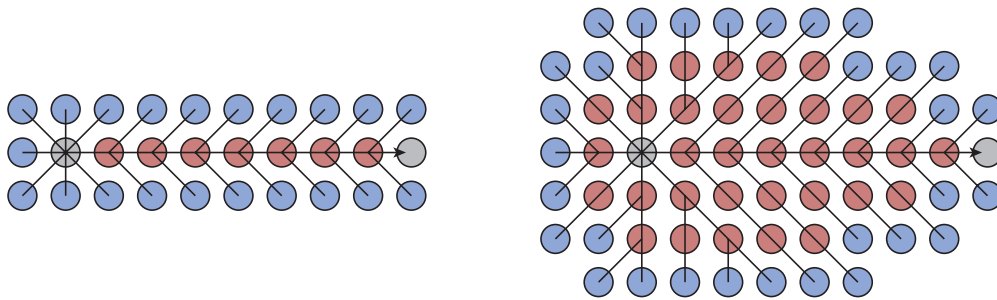


FIGURE 3.23 Effect of underestimating the distance to the goal. On the left, the result of the exact estimation is shown while the right shows what happens when the distance is underestimated. Grey nodes depict the start (left) and goal (right) location. The blue nodes are the front-line of the search, the red nodes have been expanded and are now in the closed-list.

If the heuristic is allowed to overestimate the real distance to the goal, even better results can be obtained [Rab00]. Consider Figure 3.24 where obstacles have been added to the scenario. As before, the state of the search using the exact distance to the goal is depicted on the left hand side where the search struggles to overcome the large obstacle. Many nodes have to be expanded since an *optimal* solution might dodge the obstacles on either side. If overestimating the distance to the goal, the A* algorithm tends to expand nodes that lie on the direct path to the goal before trying others. This effect is shown in the right figure. Comparing both, the non-overestimating heuristic explores three times more nodes than the overestimating heuristic.

A disadvantage of overestimating distances is that the solution might be suboptimal. Additionally, the search is slowed down significantly if the final path contains directions that lead away from the goal. Thus, it is the application domain that decides if an overestimating heuristic makes sense. As will be seen later, this technique is a vital mechanism to improve efficiency for open terrain navigation. Note that some experimentation might be necessary in order to assess the optimal quantity of overestimation.

3.5.3 Preprocessing

The following section discusses various aspects of our solution. First, the chosen discretization method is presented and the given algorithm is improved to better fit

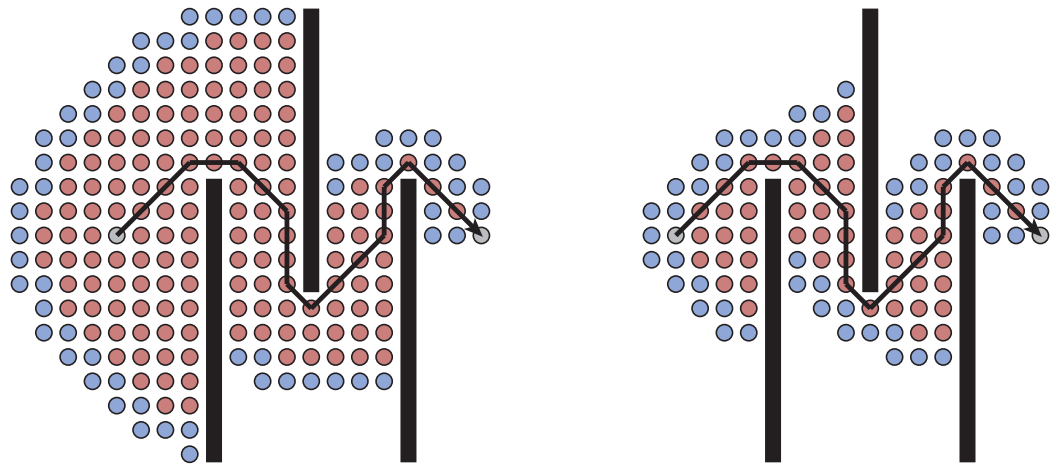


FIGURE 3.24 Effect of overestimating the distance to the goal. On the left, the situation with an exact distance measure is shown. When overestimating the distance to the goal, as shown on the right, less nodes get expanded. Grey nodes depict the start (left) and goal (right) location. The blue nodes are the front-line of the search, the red nodes have been expanded and are now in the closed-list.

the needs. Second, various approaches to build a graph on such a discretization are discussed and it is shown that the traditional approaches fail to find optimal and intuitive solutions.

Discretization

As explained before, the first task is to discretize the scene into obstacle-free regions. Different approaches to this problem are proposed in [DeL00] and shown in Figure 3.25. The first and trivial idea is to use a regular grid of some arbitrary resolution as shown in Figure 3.25i). This approach obviously leads to a very dense graph. Additionally, it can exhibit loss of connectivity, since some cells are only partially empty. A second approach is a quad-tree, depicted in Figure 3.25ii). The quad-tree displays a better approximation of the scene and a leaner graph, but still has a major drawback: At the borders of the obstacles there are many very small cells. Since many paths follow the borders of an obstacle the expected speed-up is partially lost. The problem with partially occupied cells remains, too. The third approach is to tessellate the ground into convex polygons, shown in Figure 3.25iii). Convex polygons have the useful property that any straight path inside the polygon can not collide with its border. Additionally, since the obstacles are polygonal, a partition of the walkable ground can be obtained without the loss of connectivity. Finally, a very lean graph can be achieved when using the points-of-visibility approach in Figure 3.25iv). For each obstacle corner, all other visible corners are connected to build a graph with a small number of nodes. However, the determination of the nearest graph node for an arbitrary location is rather difficult since a visibility-check is necessary for every potential node.

Based on this insight, the tessellation into convex polygons seems to be most promising due to its simplicity. To the end, the algorithm of Seidel [Sei90] has been used. This algorithm tessellates the ground into trapezoids with horizontal borders.

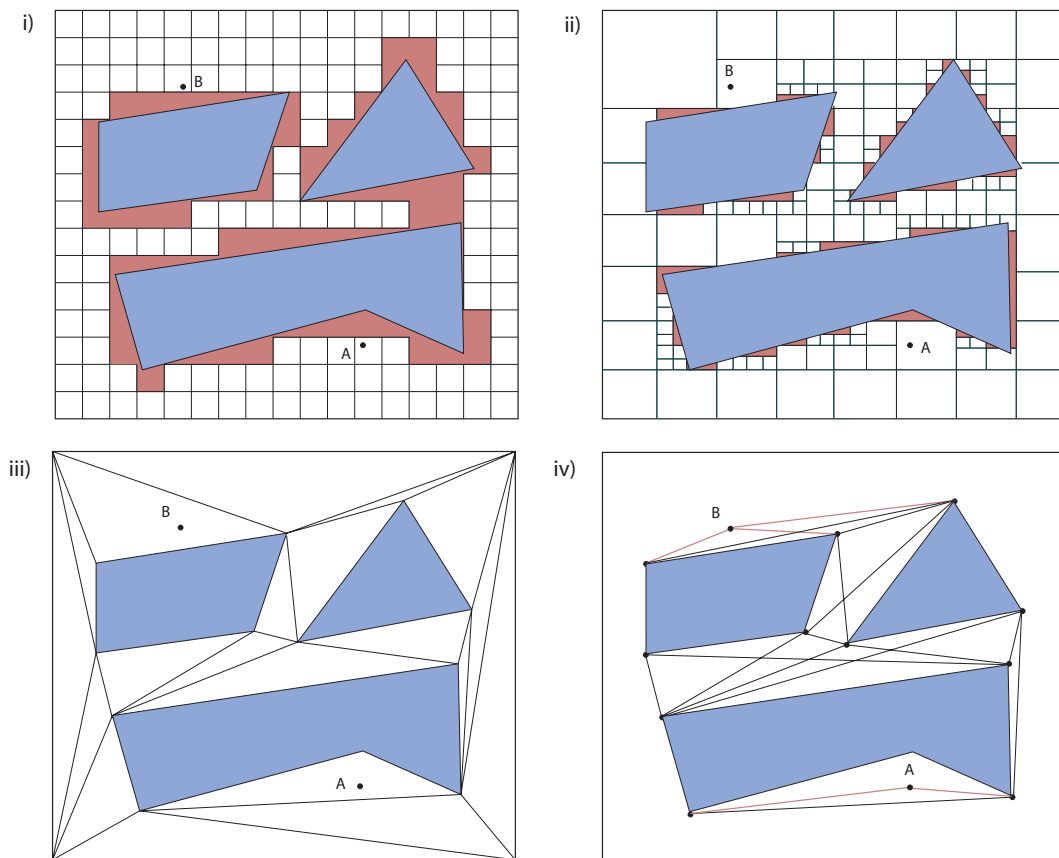


FIGURE 3.25 Four different approaches to the discretization problem. The Rectangular Grid (i) is based on a uniform grid which does not fit exactly to the obstacle borders. Therefore, a rather large area remains unwalkable (red). The Quadtree (ii) provides a better approximation of the obstacle-borders and less graph nodes but still has some drawbacks. The Convex Polygon approach (iii) yields a lean graph and no loss of connectivity. A very lean graph is created with the Points of Visibility approach (iv). However, for a specific path request, the start and goal node have to be added to the graph as denoted by the red edges.

These allow for very fast line intersection calculation which makes this approach very interesting. Additionally, the algorithm can also deal with polygons containing holes. This is very important since in our scenario the large polygon that defines the border of the map contains other smaller polygons representing the obstacles. Additionally, the tessellation process automatically yields a query-tree that allows to efficiently handle the task of point location. Given an arbitrary point, its corresponding trapezoid can be found in $O(\log N)$ with N being the number of cells.

But the resulting tessellation of Seidel's algorithm is still suboptimal as depicted in Figure 3.26. The need to merge neighboring trapezoids into arbitrary convex polygons wherever possible emerges in order to reduce the number of nodes in the graph. This algorithm is due to Hertel and Mehlhorn [O'R94, TA02]. This process is not negligible as can be seen in Figure 3.26ii). Tests on sample maps have shown that on average 50 percent of the trapezoids are eliminated. When allowing for slightly concave polygons by introducing a tolerance parameter even more trape-

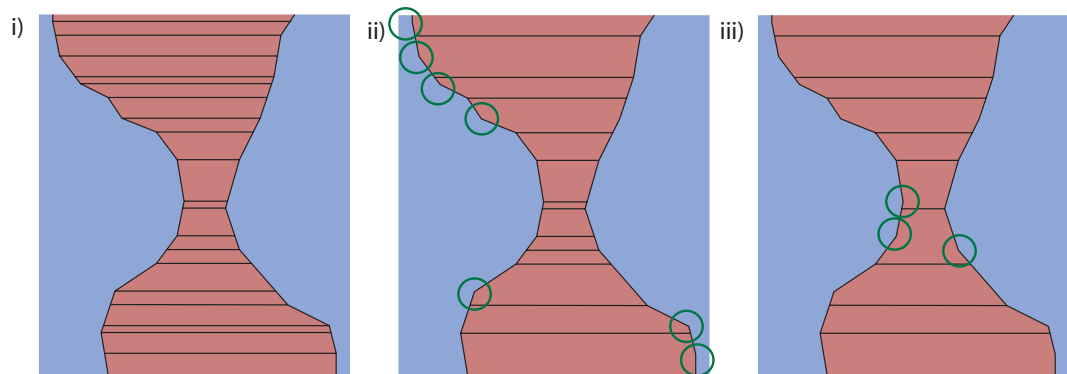


FIGURE 3.26 Discretization into obstacle-free regions. The red area denotes the walkable area while the blue one depicts the polygonal obstacle.

- i) The tessellation using Seidel's algorithm is not optimal with respect to the pathfinding problem.
- ii) Merging neighboring cells into larger ones that still remain convex (green circles).
- iii) Allowing for slightly concave polygons further reduces the number of polygons (green circles).

zoids can be eliminated as depicted in Figure 3.26iii). The consequence is that paths can potentially intersect the border of the polygon. If this is inadmissible, one can simply expand the contours of the obstacles in a preprocessing step as described in [You01].

Lakes and Contours

In our environment, obstacles are either objects or lakes. While the position and extension of the objects can be derived from the object itself, the contours of the lakes depend on the landscape's heightfield and need to be generated during initialization. All areas below the height of zero are considered to be lakes. Such a situation is depicted in Figure 3.27i).

In order to find the contours of the lakes, a binary map of the environment is generated distinguishing between walkable area (green) and underwater area (blue) as shown in Figure 3.27ii). During the next phase, each disjunct lake is found using a floodfill algorithm. This algorithm uniquely labels connected areas known to be underwater. For each lake, the contour points are the ones that belong to the lake but have at least one neighbor on the walkable area. The underlying algorithm first finds one contour point and then follows the contour in clockwise direction and is described in detail in [Nie01]. The result is shown in red in Figure 3.27iii). However, this sequence of pixels can be coarsened into a sequence of line segments that approximates the contour line sufficiently using a loose-contour algorithm [Nie01] which results in Figure 3.27iv) where the contours of the lakes are shown using red line segments. Note, that the contours are closed sequences that may overlap the map borders.

Building the Graph

Building a graph on this tessellation leads to several possible approaches. One could use the polygon centers as graph nodes as shown in Figure 3.28ii). This strat-

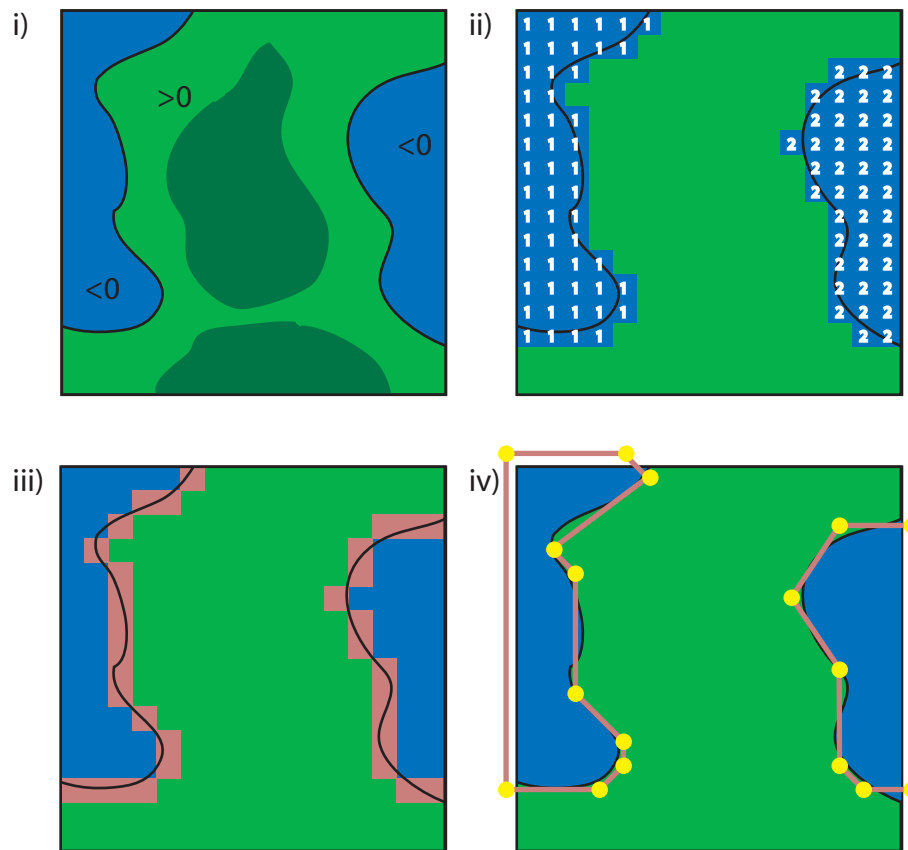


FIGURE 3.27 Generation method for contours of lakes.
 i) All areas below the height of 0 are considered to be underwater.
 ii) A binary map is generated which distinguishes between walkable (green) and underwater (blue) areas. Using this map, each disjunct lake is labeled uniquely.
 iii) The contour points shown in red of these areas are determined.
 iv) The resulting contour lines are closed sequences of line segments that approximate the lake's real contour.

egy fails since the corresponding portals of each pair of adjacent polygons have to be found during the search. Instead, the portal centers could be used directly as nodes, as Figure 3.28iii) shows. While the resulting graph contains more nodes than the first approach, two advantages can be achieved: First, the step to find portals is omitted. Second, the nodes represent more accurately the geometric locations relevant to the final path which allows for a better cost estimation between the nodes.

The so far presented solution works fine but has some major limitations. Consider the situation depicted in Figure 3.29i) where a path from A to B has to be found. A* will find the path leading around the obstacle instead of the expected straight path because the graph displays a shorter route around the obstacle. Anyway, choosing the portal centers as node locations is a rather arbitrary choice. Since most paths dodging an obstacle will follow its contour, one could try to anticipate this by placing the nodes on the portal end-points. This solves the situation in Figure 3.29i), however, it fails in other situations as shown in Figure 3.29ii) where

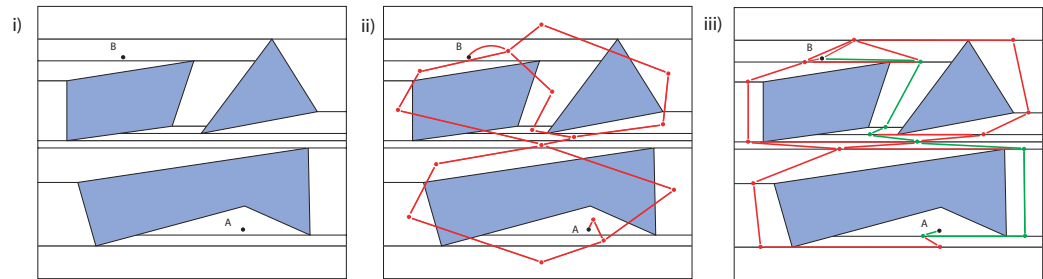


FIGURE 3.28 Building the Graph.
 i) The tessellation using Seidel's algorithm.
 ii) The resulting graph using the trapezoid centers as nodes.
 iii) Using portal centers as nodes. The resulting node sequence connecting A and B is shown in green.

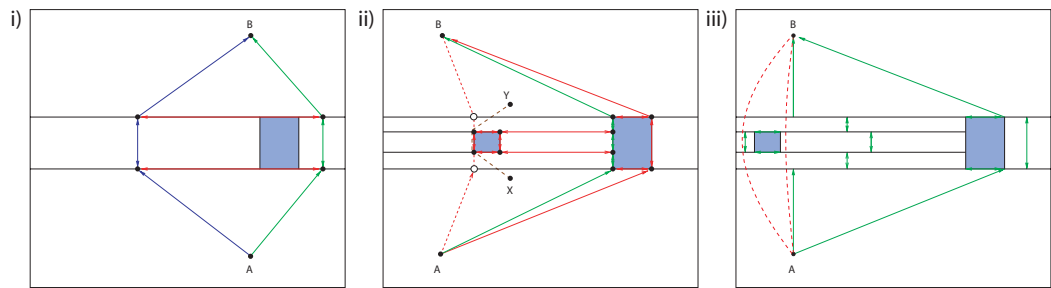


FIGURE 3.29 Limitations of different approaches for building a graph.
 i) Using portal centers: Unintuitively, the green path will be chosen instead of the blue one when connecting A and B using a graph built upon portal centers.
 ii) Using portal end points would resolve the situation in i) but fails when adding a second obstacle where the green path is chosen instead of a direct link connecting A and B. Adding even more nodes (shown in white) would result in the correct path (dashed red) but will fail when connecting X and Y (dashed black).
 iii) Abandoning the idea of fixed node locations and using the whole portals instead leads to undecidable situations. When connecting A and B, both red dashed lines show paths of equal length.

an additional obstacle has been inserted. Additionally, the resulting graph has twice as many nodes as the previous one.

All problems encountered so far stem from the fact that long segments are reduced to just one or two points in the map and make it impossible to get a correct heuristic for A^* . Introducing a maximal portal width and splitting up broad polygons in order to reduce the deviation could resolve that problem. However, this would lead to an even denser graph because of the additional interconnections between horizontally neighboring polygons. Also, note, that no matter how small the portals will be, there are always counter-examples that produce unintuitive results.

When abandoning the idea of graph nodes representing precise locations, the idea to use the whole portals themselves as nodes ends up in a very lean graph again

as shown in Figure 3.29iii). In order to construct the graph, a distance measure between portals is needed – the minimal distance between two segments seems to be appropriate which is in most cases the vertical distance between two portals as shown as green arrows. This distance measure ensures that the total length of the final path is never overestimated and therefore the optimal path has to be contained in the set of possible solutions of the A* search.

When reconsidering the above examples the first situation is solved correctly. Anyway, no assertive answer can be given in the second situation, since both paths around the small obstacles have identical costs. As a consequence, the problem can not be decided and in fact, the outcome will depend on the implementation of the algorithm. Therefore, a modification to makes this approach robust is required.

One possibility could be to tweak the A* heuristic function in Equation 3.8 which estimates the remaining cost to the goal node. Since the function can be arbitrarily chosen it can be changed to prefer portals that lie on a straight line between the start and goal location. Adding a term that enforces the A* search to expand nodes that lie near to the connecting line also considerably improves the efficiency. This technique is vital to open terrain navigation since in most cases the paths are straight or deviate only little from the straight connection between start and goal. However, the search is significantly slowed down when the path is forced to lead away from the goal location. Also, the computational expense to calculate such a heuristic function further slows down the search. A further limitation is that the search direction is always attracted by the straight line connecting the start and goal location. When the path leads away from this line it will tend to return rather than moving ahead from its current position to the goal since the heuristic is only globally defined. Instead, a local search that depends also on the current position is more promising.

When looking for a solution that unifies the advantages of the above approaches while eliminating the incorrect solutions, it should provide a small number of nodes and accuracy with respect to distance measures. The fundamental problem of the whole approach so far is that the graph is built offline in a preprocessing stage. It stays fixed for its lifetime with the exception of the start and goal node that are introduced for a search. Therefore, there is no way to bring in the information of a specific pathfinding request into the structure of the graph.

3.5.4 Dynamic A*

As has been shown, the major drawback is that no specific information of a request can be included into the graph since it is built offline. The solution to this problem presented herein is not to specify the exact location of a node until necessary [NG04]. The location is set according to the previous course of the path using two different strategies.

When moving away from the goal location, the direction in which the final path will lead is not known exactly. Therefore, a lazy strategy is applied which selects the closest possible location on the next portal. This results in a minimal deviation from the final path. Here, an approximation error is introduced since the final path will most likely not traverse this exact location. This situation is depicted in Figure 3.30i) where the red as well as the blue path both seem promising to lead to the goal. When moving towards the goal location, a greedy strategy is used since the

direction where to go is known. This means that the node can be set as close as possible to the straight line connecting the actual location with the goal. An example where this strategy is used can be seen in Figure 3.30ii) and iii).

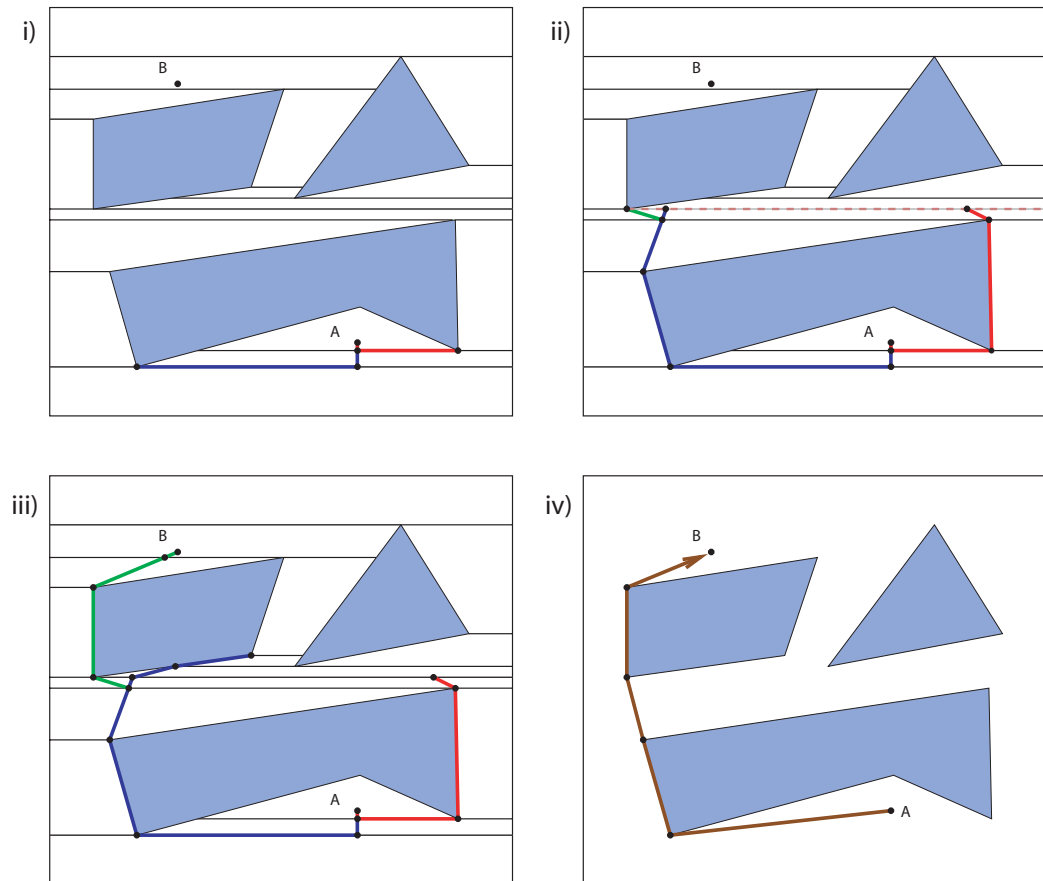


FIGURE 3.30 The dynamic A* uses two different strategies.

- i) The lazy strategy is applied when moving away from the goal location. The nodes are then placed on the next portal as near as possible to the previous position.
- ii) The greedy strategy is deployed when moving towards the goal location. The nodes are placed as near as possible to the straight line to the goal.
- iii) The shortest path finally reaches the goal location.
- iv) The resulting path after postprocessing. Note, that even though the obstacles are the same as in Figure 3.28, the solutions found differ where this one is actually shorter.

Using these two strategies, several facts provide advantages to traditional approaches. First, the graph is still small and has a minimal number of nodes with respect to the tessellation. It is not necessary to introduce additional nodes that slow down the entire process. Second, a reduction of approximation errors is achieved since the greedy strategy prevents from spatial approximations. Only the lazy strategy introduces an overestimation of the path length which is bound by the factor $\sqrt{2}$. This worst case occurs when moving vertically instead of diagonally within a square. Third, the heuristic has not to be tweaked as presented in the last section

since every situation can be decided entirely. Considering Figure 3.30ii), the red and blue path both lead to the dashed red portal. Assuming that both of them have the same cost up to this point, A^* could not decide which one to favor since both are indistinguishable. The presented approach allows for an online distinction because both paths reach different locations on the portal, hence, their predictions for the remaining cost differ and make both paths distinguishable.

Of course, this approach has also some drawbacks. It seems to be slower than the conventional approach because of the additional evaluation of the distance between two successive nodes. Also, the calculation of the intersection between the straight line to the goal and the next portal is a disadvantage. But since the portals are always horizontally, this can be done very efficiently and is therefore negligible. As the results will show, our solution is still very fast and competitive.

3.5.5 Postprocessing

After having found a sequence of nodes in the graph using the above presented algorithm, these nodes do not form an optimal path since it still contains more nodes than necessary – one for each traversed portal. Now, it is necessary to abandon the graph and return to the original map and apply a postprocessing of the resulting node sequence right within the map. Additionally, an adaptation of the path to a three-dimensional environment is possible by incorporating the slope of the terrain when moving on the path.

Path Optimization

The goal of this postprocessing stage is to find a sequence of waypoints that constitutes a path with a minimal number of waypoints. In order to achieve that, a *cone-of-sight algorithm* is proposed which is based on visibility of points and portals.

Again, our algorithm uses the advantage of the horizontal portals by Seidel's algorithm. Without loss of generality, it is assumed to move vertically upwards and the algorithm is presented with the example given in Figure 3.31i). The start node is at point A and the path has a sequence of portals to pass. Two different portals on fourth position are depicted as dashed lines to show how the algorithm works in different cases.

The starting point and the first portal form together a cone of sight as depicted in Figure 3.31ii). The algorithm keeps track of three points L , R , and the cone's starting point, in this case A . L , respectively R , denote the borders of the cone. When looking at the next portal in Figure 3.31iii), its left end lies inside the cone. Therefore, the left border is narrowed by placing L to this point. Since the right end of this portal is outside our cone, it will not restrict the current search. Looking at the third portal in Figure 3.31iv) further restricts our cone, this time from the right side. R is moved inwards and lies now on the right end of the third portal. These steps are continued until reaching a portal that completely lies outside the cone.

Portal 4a in Figure 3.31iv) is outside the cone on the left. Therefore a waypoint has to be placed on the location of L and the algorithm is restarted at this point. The new situation is shown in Figure 3.31v). This back-tracking step explains why the algorithm has to keep track of L and R . Alternative 4b shows that the portal lies on the right side of the cone. Therefore, R has to be set as the first waypoint and the algorithm continues from the third portal as depicted in Figure 3.31vi).

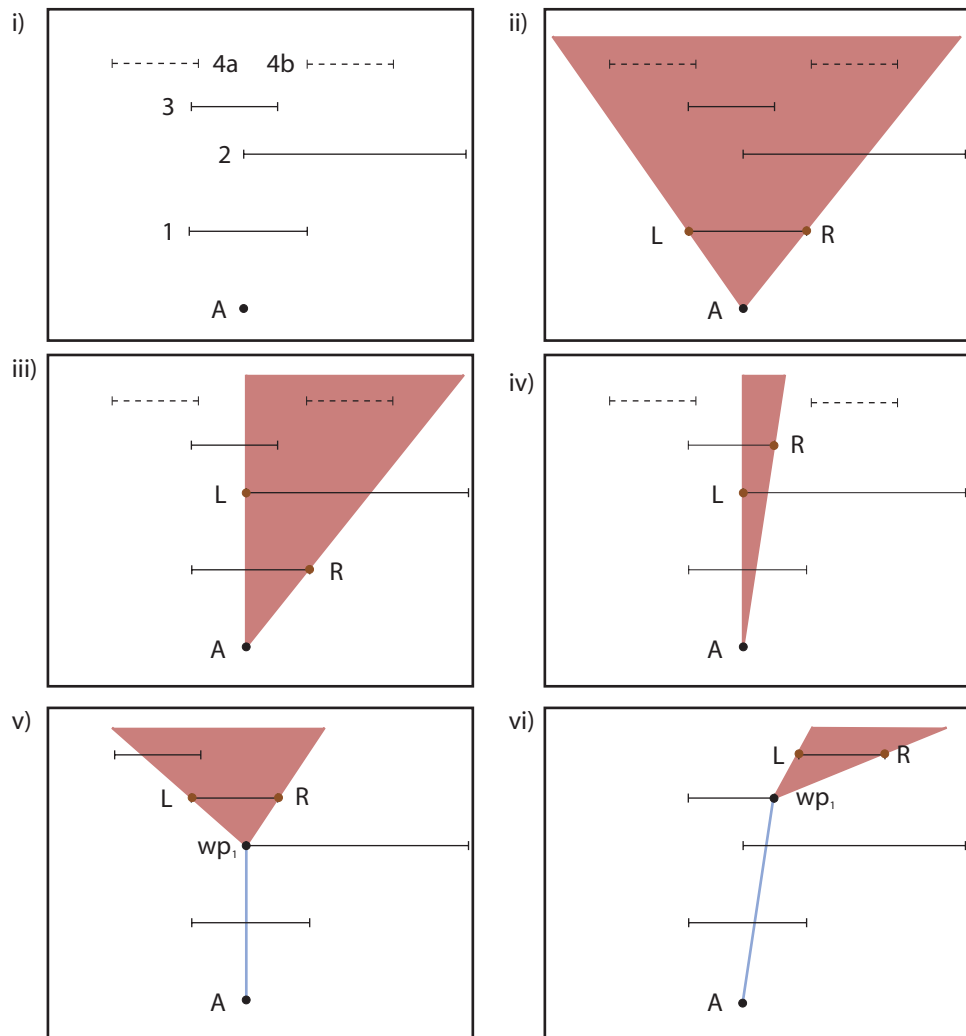


FIGURE 3.31 Path Optimization: The Cone-of-Sight algorithm.

This algorithm guarantees that the shortest path for a given portal sequence is found. Thus, the straightness criterion for paths introduced in Section 3.5.1 is met. As a counterexample, the Rubber Banding Algorithm presented in [Dav00] does not achieve this. The here presented algorithm benefits from the fact that portals are always horizontal which allows again for fast intersection calculations.

Movement in 3D

The system is now able to efficiently compute qualitative paths but still constitutes a purely 2D navigation facility. This section presents an approach how to extend the system in order to take height information into account. This post-processing step is not necessary but results in aesthetically pleasing paths.

Instead of having agents that strictly follow the path in a straight line from one waypoint to another, they can be allowed to diverge to some degree. An approach is proposed, where the path is adapted to the slope of the terrain. In order to avoid steep slopes, the agent turns away from its ideal course using the current position p

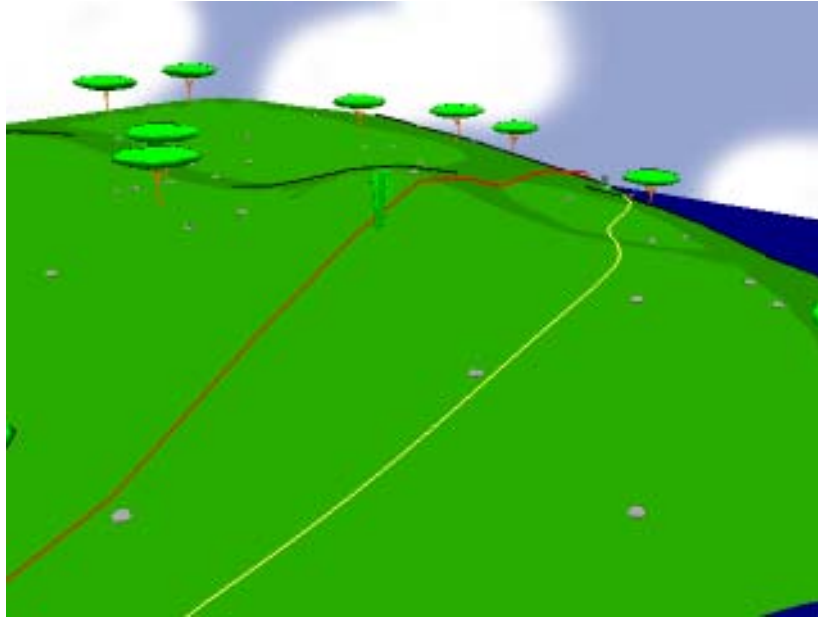


FIGURE 3.32 Path adapted to the slope of the terrain. The red line depicts the path given by the path-planning subsystem and the green line the actual movement of the agent adapted to the slope.

of the agent, the next waypoint w_i , and an arbitrary function $f: \mathfrak{R} \rightarrow [0, \angle max]$, where $\angle max$ is the maximal deviation angle. This results in

$$\delta(\mathbf{p}) = f(\|\mathbf{w}_i - \mathbf{p}\|) \|\nabla(\mathbf{p})\| \cos(\varphi), \quad (3.10)$$

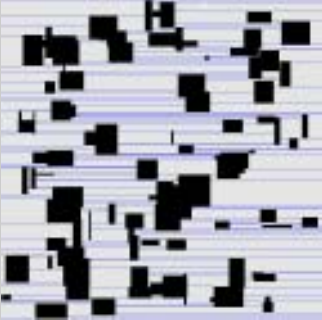
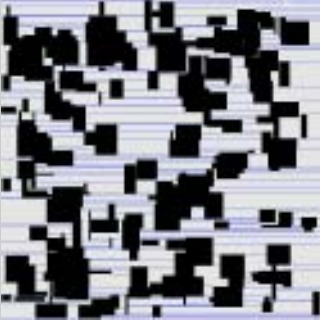
where

$$\varphi = \angle(\mathbf{w}_i - \mathbf{p}, \nabla(\mathbf{p})). \quad (3.11)$$

$\delta(\mathbf{p})$ in Equation 3.10 denotes the deflection from the angle pointing to the next waypoint w_i . This formula is only applied when $\varphi > 0$, thus, when the agent is ascending. The first term $f(\|\mathbf{w}_i - \mathbf{p}\|)$ ensures, that the deviation decreases as the waypoint is approached. In order to assure that the goal point w_i is really reached, the function f should become 0 when the distance $\mathbf{w}_i - \mathbf{p}$ decreases. The second term $\|\nabla(\mathbf{p})\|$ accounts for the steepness – the greater the length of the gradient, the larger the deviation from the straight path will be. The last term $\cos(\varphi)$ weighs the direction of the gradient against the orientation of the movement. The deviation is maximal when the next waypoint lies in the direction of the maximal steepness. If the slope is parallel to the moving direction ($\varphi = 90^\circ$) the deviation is zero. Figure 3.32 shows an example where an agent follows a path on an uneven terrain. The resulting trace of its course in green seems to be far more natural than the red path generated by the path-planning unit. Nevertheless, the agent will end up in the same location as the path request has had.

Of course, such an adapted path could pass an obstacle region, because it deviates from the original path. But since every waypoint is reached exactly and waypoints usually lie at the border of an obstacle, this problem is negligible. Additionally, the

TABLE 3.1 The characteristics of the maps used for comparing our approach with others. The last three rows show the number of nodes of the graphs built on the maps. The last two rows are derived from the original map by setting the maximal portal width to 330 and 100 respectively.

	Loose Map	Dense Map
Map		
Size	1000x1000	1000x1000
# Obstacles	100	200
Center Nodes	281	395
Width 1/3 Nodes	323	408
Width 1/10 Nodes	567	568

user has the possibility to specify the maximal deviation angle to keep the agent near the original path. Furthermore, one could enlarge the obstacle regions in order to introduce a tolerance bound around each obstacle [You01].

3.5.6 Results in Path-Planning

In order to compare this approach with different traditional approaches a test suite has been set up that automatically generates paths. The measurements were taken by calculating paths with random start and goal locations on two different maps. The characteristics of these maps are outlined in Table 3.1. The presented approach has been compared with three implementations which have been presented before:

- **Center:** This approach uses a static graph built on the portal centers of the tessellation.
- **Width 1/3:** The maximal portal width of the above approach has been set to a third of the map width and the portals are connected with the maximal fanout. The resulting graph is more dense but approximates the node locations more accurately.
- **Width 1/10:** The maximal portal width has been set to a tenth of the map width with an according fanout.

All these implementations use the same underlying A^{*} mechanism with the same performance optimizations as the presented approach. These approaches also use the path optimization procedure presented in Section 3.5.5. Therefore, all these path planning systems are comparable to each other.

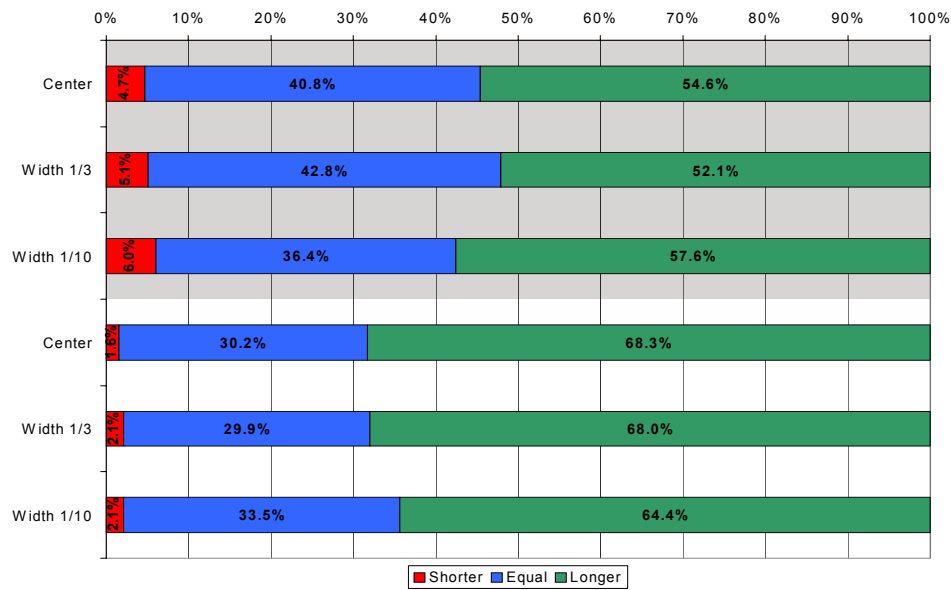


FIGURE 3.33 Comparing the path length of traditional approaches with the dynamic A* approach. The top three rows (gray) were generated on the dense map, while the bottom three rows (white) used the loose map described in Table 3.1. On each map the presented approach is compared to three different implementations that use static graphs. Red denotes cases where traditional approaches found a shorter path, blue for equal length of both approaches, and green when the dynamic A* approach found a shorter path.

Astonishingly, in more than 90% of the cases our approach finds paths of equal or shorter length as shown in Figure 3.33. The paths are absolutely shorter than the other approaches in over 50% of the tests. This result is even more distinct on the loose map with less obstacles (97%, respectively 60%). Using a maximal portal width only slightly improves the quality of the result as can be seen. On the loose map this technique is more effective since the portal width is more likely to be large.

Comparing the path generation time of the different approaches, Figure 3.34 shows that the presented approach is competitive. On the dense map (top rows), the algorithms with approximately the same number of nodes perform similar while the difference grows with less obstacles. The approach with the maximal portal width set to a tenth of the map width always performs slower since its graph has more nodes as shown in Table 3.1. For our approach, absolute time values are 0.19 ms on average on the loose map and 0.28 ms on the dense map on a 1 GHz Pentium III computer with 512 MB RAM.

Based on these results and the above description of the algorithm, we can state that the presented algorithm is fast, robust, and can be used for arbitrary static terrains with polygonal obstacles. The paths are on average shorter than with traditional approaches and even the computation time is slightly faster. The shorter paths are due to a better cost approximation that result from moveable graph nodes and the two strategies used to determine their exact positions. Due to a post-processing step which removes unnecessary nodes, the paths resulting from our path-planning sub systems are made up of a minimal number of waypoints. The visual appearance has been enhanced by incorporating the slope of the terrain while following a path.

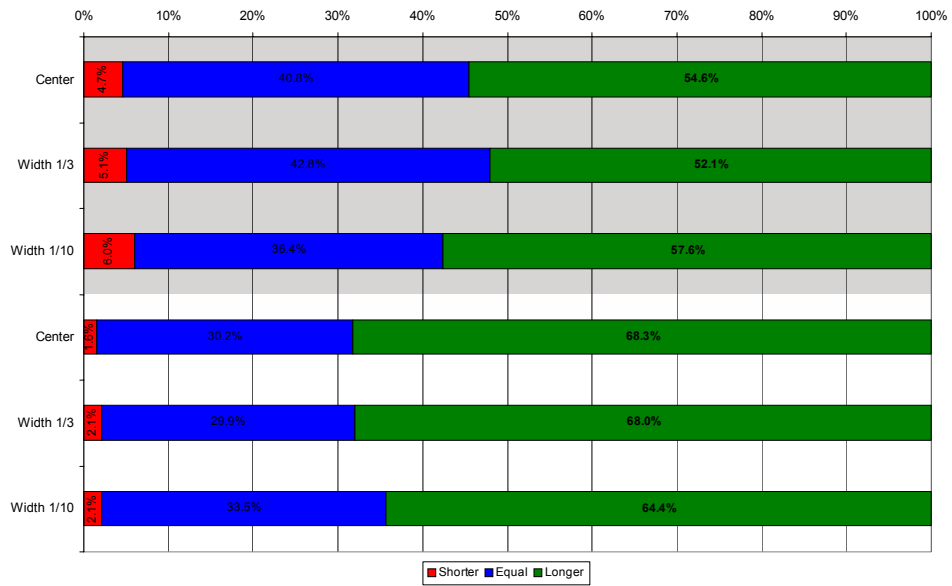


FIGURE 3.34 Comparing the run time of traditional approaches with the dynamic A* approach. The order of the rows is the same as in Figure 3.33. Green denotes cases where the dynamic A* approach was faster.

3.6 RESULTS

The as yet presented system is able to simulate the behavior described in Section 3.1.1. In this section, some examples of this model and the methods presented will conclude this chapter.

All examples were generated using a simple simulation environment implemented at ETH Zürich – the GAIA engine – on a Pentium IV 2.6 GHz system with 2.0 GB of memory and a ATI Radeon 8500 graphics board. The scenery is defined by an heightfield where a value below zero denotes water. The landscapes are created using a simple terrain-generator based on subdivision and can be edited in a separate program in order to create specific terrains. Additional obstacles, such as trees and bushes, can be placed wherever needed or wanted using a simple mechanism. Figure 3.35 shows two different sceneries.

Furthermore, the GAIA engine allows to render the scene in different modes – a realistic rendering or two different comic rendering modes as shown in Figure 3.36. On the left, the realistic rendering is shown while the right image presents one of the comic rendering modes. As mentioned in Section 3.1.1, the intention of this thesis is not to provide a novel behavior model and the correctness of the presented behavior with respect to nature is not that important, thus, we use the comic mode to enforce the abstraction from a natural impression.

First, the structured group example with the family is shown in Figure 3.37. Both images show the same scene where the left one is the rendered view and the right one a color-coded representation. For each different type of agent, a different color is used, where red depicts the father, blue stands for the mother and yellow boxes represent the children. Note, that the right image reveals a white box which

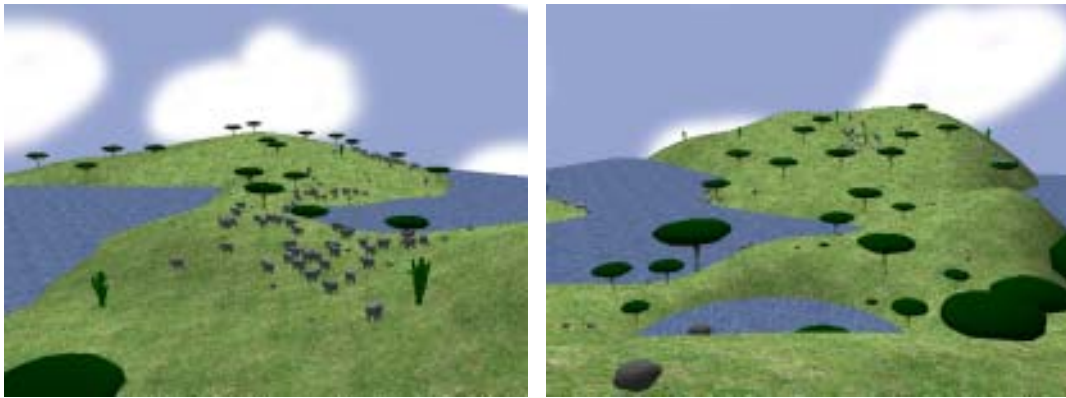


FIGURE 3.35 Two different sceneries. The terrain in the GAIA engine is based on a height-field and allows for easy replacement of terrains.

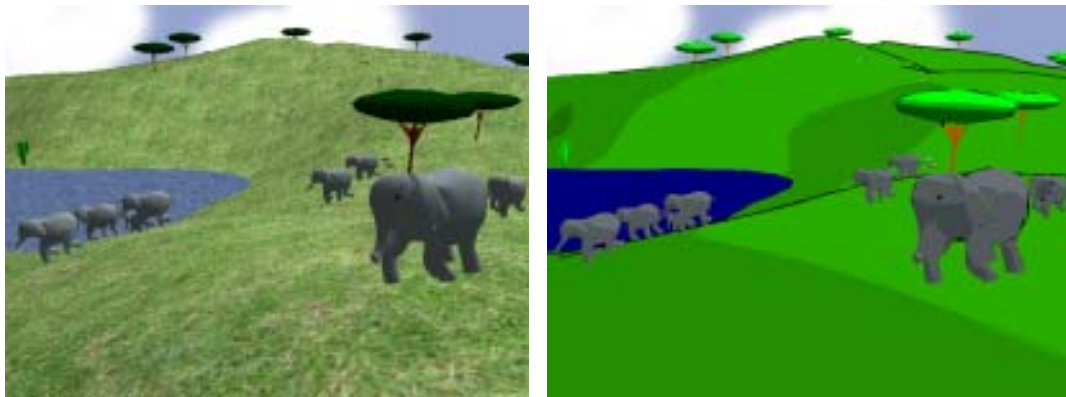


FIGURE 3.36 Two different rendering modes. The left image shows the realistic rendering mode and the right one the same scene in the comic mode.

is not visible on the left image. It is the group agent representation which is usually not rendered but keeps the family together.

Second, a modulo group example is depicted in Figure 3.38. In this example, all agents of the group are first separated into two categories – for example male/female – but additionally, every third member has become a child. The left image shows the group rendered normally. The viewer can only distinguish adult and young ones by their size. On the right, some of the underlying information is shown as color-coded cubes. The adult male elephants are shown in red and the female ones in green. The children can be divided into male/female as well, where blue boxes represent the male ones and purple ones replace female children.

The examples shown in Figure 3.39 depict different hierarchical group structures. The red/green lines depict the hierarchical structure, where the red end is towards the inferior and the green end towards the superior agent. On the left, the hierarchy corresponds to $a=5$ and $r=2$, meaning that two out of five agents in each

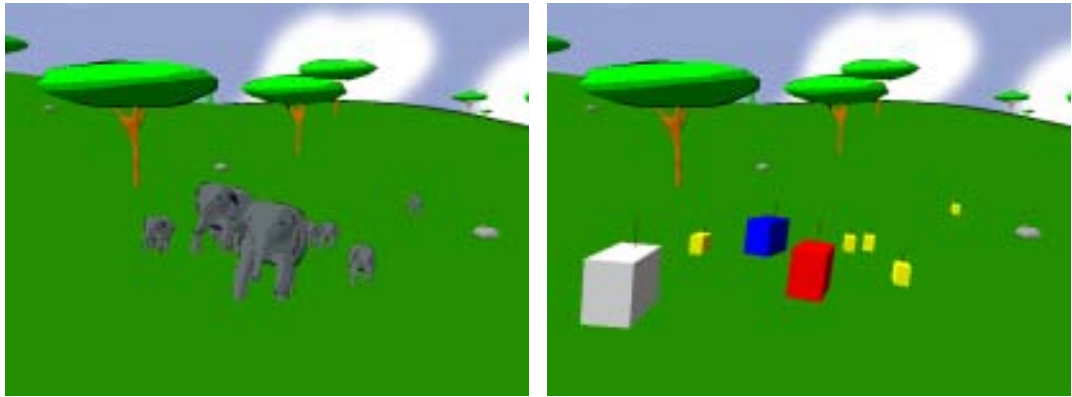


FIGURE 3.37 The structured group example. On the left, a family of elephants based on the example in Figure 3.14 is shown. On the right, a color-coded representation presents the structure. The red box is the father, the blue depicts the mother and the yellow boxes represent the children. The white box does not appear on the left image since it is the group agent representation which is usually not rendered but keeps the family together.

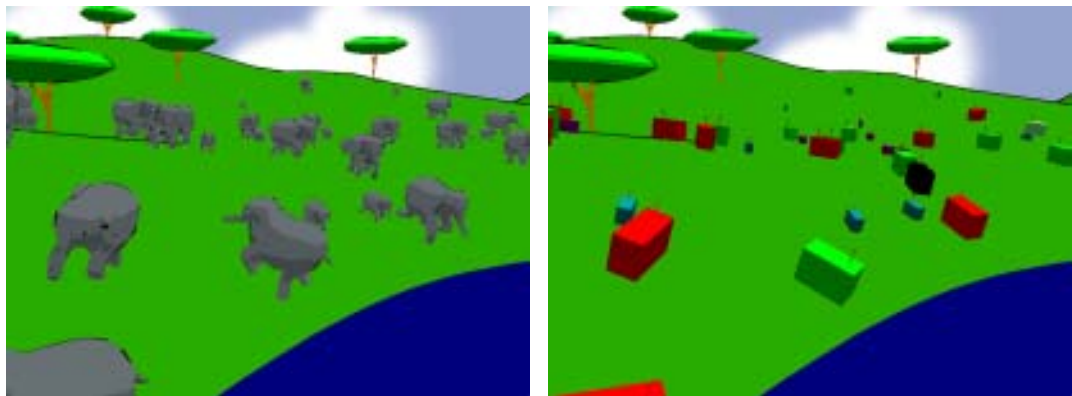


FIGURE 3.38 The modulo group example. On the left, a group with 50 members is shown where some can be identified as children and others as adult elephants. The right image reveals the underlying structure of the group which additionally distinguishes between two types of elephants, for example male and female ones. Green boxes are adult males, red ones are adult females, blue ones are male children, and, finally, female children are denoted in purple. Again, the group representation is not shown on the left image but as a black box in the right part.

subgroup are superior to another subgroup as described in Figure 3.18. The group starts on the right with the five rightmost elephants denote the first level. On the right, another hierarchical group is shown, this time $a=4$ and $r=3$. Both groups have the size of 20 members and implement a leader-follower behavior which keeps the group together.

We also implemented Reynolds flocking algorithm such that an agent aligns with its neighbors according to their velocity and orientation if there are any neigh-

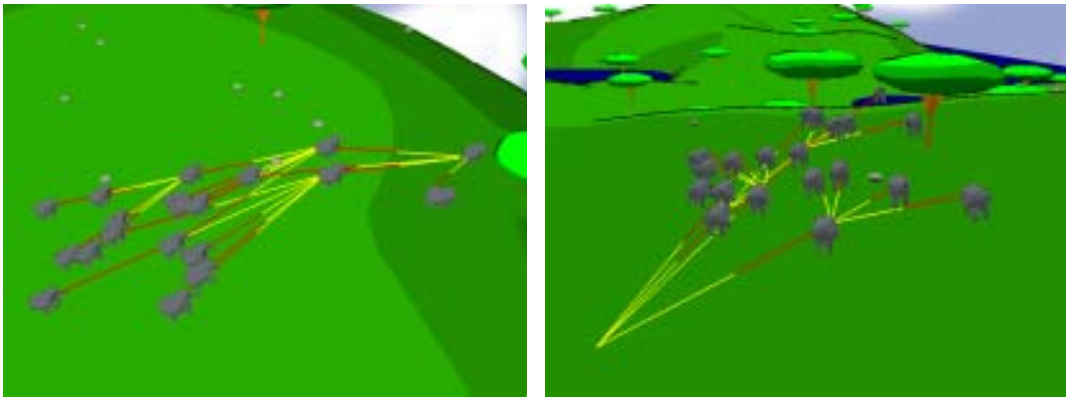


FIGURE 3.39 The hierarchical group examples. On the left, a hierarchical group of 20 members with $a=5$ and $r=2$ as described in Figure 3.18 is shown. On the right, another group of 20 members is shown, this time with $a=4$ and $r=3$. The relationship in the hierarchy is denoted by the red/yellow lines where the red part is at the inferior and the yellow part at the superior agent. Note, that the front-most elephant on the left is not the group agent but one of the agents in the first hierarchy level.

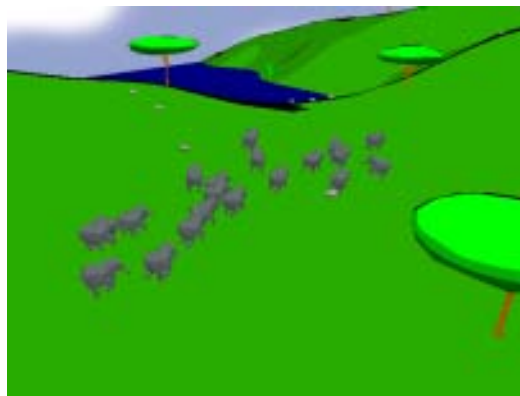


FIGURE 3.40 Reynold's herding algorithm.

bors as shown in Figure 3.40. If an agent comes near the water, it will avoid the water rather than remaining in the herd. However, Reynolds algorithm is not suitable for large crowds since it requires to compute the local neighbors for each individual which is hardly feasible with hundreds of agents. Nevertheless, with small groups of characters, Reynolds approach is feasible and works well when having found appropriate parameters for the weighting of the different forces.

The last example in this section shows a full scene with over 1500 agents, acting individually or being engaged in hierarchical groups. Two screenshots of the scenario are shown in Figure 3.41 This scene has been used to measure the frame-rate of the over-all system inclusive rendering with a full simulation. Full simulation means that all agents are activated in every time-step such that all can act on the most current simulation state. The plot in Figure 3.42 shows the frame-rate of the

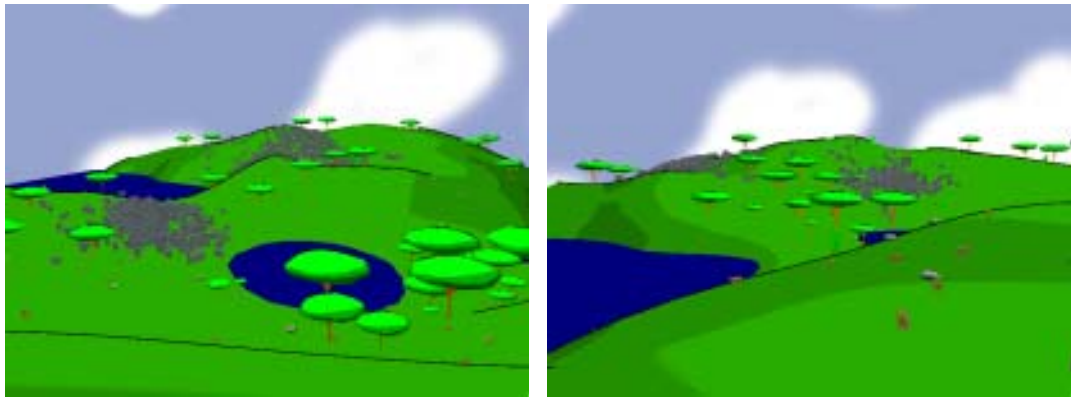


FIGURE 3.41 A full scene example with over 1500 agents. The agents are engaged in very large hierarchical groups (elephants) or individually (lions).

simulation depending on the number of agents. Up to 1000 agents can be simulated fully without a great impact on the frame-rate. Note, that the frame-rate is much higher when only small parts of the scene are rendered. The views from which these values were generated provide a great overview of the scene and, thus, the rendering cost is not neglectable. Our system also provides acceptable frame-rates for up to 3000 agents and more. However, the system is not optimized with respect to performance and we expect that significant improvement could be achieved when reengineering the framework.

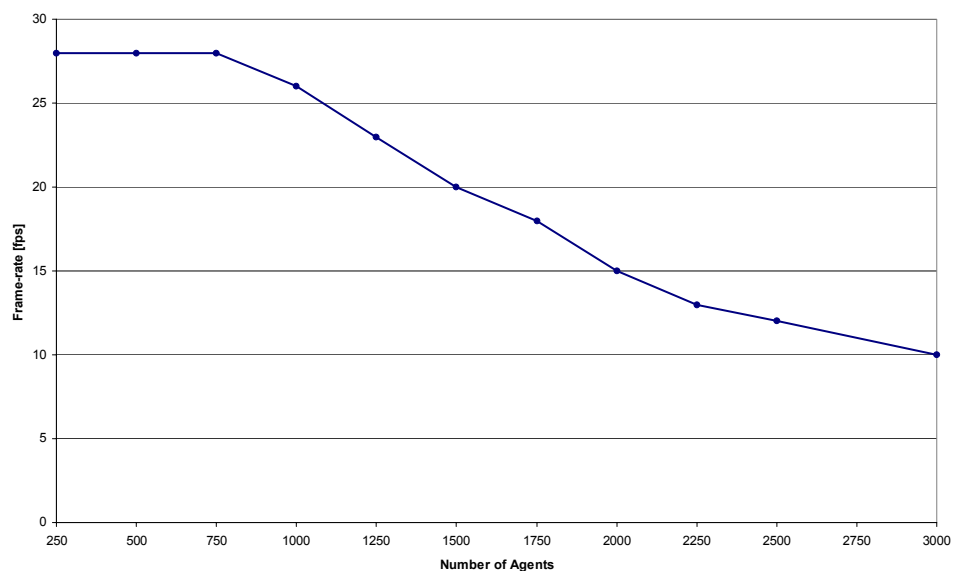


FIGURE 3.42 The average frame-rate depending on the number of agents. Up to 1000 agents can be fully simulated without restricting the frame-rate under a rate of 25 frames per second. The frame-rate drops below 10 frames per second when simulating more than 3000 agents.

When looking at the different stages of the behavior model in different scenarios, we obtain the chart shown in Figure 3.43. The chart shows the absolute times needed to determine the reactive behavior of agents in different scenarios. As can be seen, the single agent needs with around 0.06 milliseconds almost twice the time to make a decision and execute it as the average agent in the Full Scene scenario where only about 0.03 milliseconds are needed. This depends on the complexity of the agents which is fairly low in the full scene scenario but high in the single agent scenario.

The relative time needed to make one decision cycle is shown in Figure 3.44 where the colors refer to the same tasks as in Figure 3.43 and the underlying measurements are the same. Obviously all scenarios show approximately the same distribution of the time. The perception task needs around 25% of the over-all time where the situation recognition and action selection use 25–30%. The rest, around 45–55% of the time is spent in the action execution mechanism which can be explained by the complex data structure and routines to determine the execution of an action. For example, the path-finding task is part of the action execution because it is done when a corresponding action starts. Therefore, the action execution mechanism uses the largest part of the over-all time.

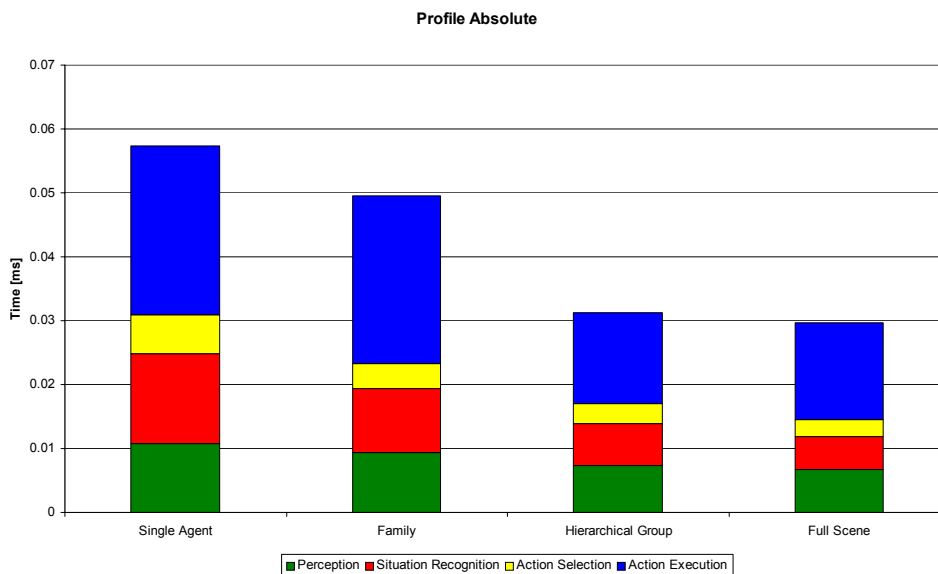


FIGURE 3.43 Absolute timings for the different stages of the reactive behavior model. The underlying measurements were taken for different scenarios such as a single agent, a family, a hierarchical group and the full scene. The chart shows the time needed for each of the four stages to determine the reactive behavior.

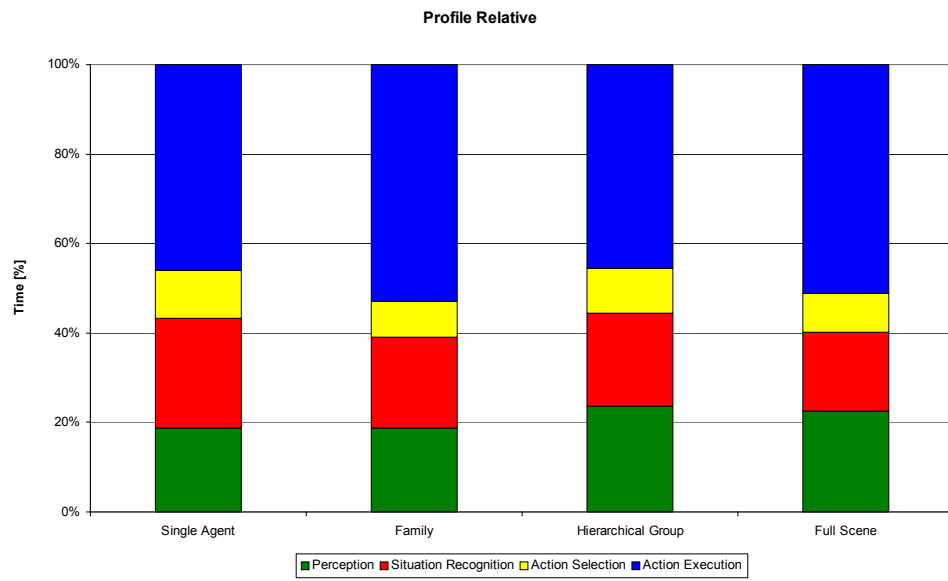


FIGURE 3.44 Relative timings for the different stages for the same scenarios. The chart shows the same results as Figure 3.43 with the relative time needed for each stage.

PROACTIVE AGENTS

In the last chapter, our approach for reactive agents has been presented including the generation of heterogeneous groups and hierarchies. But reactive agents can only provide a limited set of behaviors and it is impossible to achieve an adaptive behavior without creating rules for every possible situation that can occur. Proactive agents, in the contrast, are able to present more believable behavior by exploring the future and comparing different possibilities with respect to a certain goal. Additionally, they can take other agents' decisions into account.

This chapter will first introduce proactive agents before directly going into inference mechanisms that are needed to act proactively. Such inference mechanisms are in the need of graph search algorithms to find the best possible solution to execute. Such algorithms are explored in detail for a single agent before putting everything together with respect to concurrent planning in the dynamic real-time environment which puts several restrictions on these algorithms. Then, one section is devoted to the integration of proactive agents into our environment by extending the known reactive agents from the last chapter to proactive ones. The chapter concludes with the results achieved in our environment by integrating proactive agents.

4.1 PROACTIVE BEHAVIOR

As has been shown in the previous chapter, the behavior of reactive agents relies only on a set of predefined rules. These rules determine the presented behavior and are described as *condition-action rules* [RN96]. Since such rules are very fast, the computational expenses for a rational agent are rather low. But if it is intended to have an agent which presents some kind of “intelligent” behavior, these rules are very limited. Of course, it would be possible to create a rule for every possible situation, but the according knowledge-base would get very large as does the computational effort to find the right rule for the current situation. For example, when an agent

approaches a road junction, it should decide whether to go to the right or to the left. This decision should depend on the desired destination and is very hard to be taken a priori. Thus, the need of some sort of *goal* information that describes where the agent would like to go arises. Then, the agent could determine on the fly to which side it should turn off. Obviously, the presence of a goal information implies that the agent has also the possibility to explore the future by concatenating possible actions and by considering their influence on the environment. This process is called *searching* or *planning* and will be described within this chapter.

In our environment, several new scenarios get feasible when integrating proactive agents. A very simple one is searching for food or other items. An agent that perceives its environment can register if an object of interest is within its field of view. Then, it can decide to take that item and dynamically adapt its actual plan to get to that item and take it. This scenario includes only the agent itself and no others. But there are scenarios where more than one single agent is incorporated. Such a scenario is local path planning in a dynamic environment. As has been shown in the previous chapter, the path-planning sub-system can deal with static environments but will not consider the local dynamically changing world. A proactive agent could try to plan the next few steps that lead to the static waypoints of the given path by including the neighbors and their movements in its explorations. Another scenario is a sheep-dog that watches a herd of sheeps. When one sheep escapes the herd the dog has to return it to the herd. Using only a reactive agent approach would not guarantee to solve this problem. The dog could probably scare the sheep away by approaching from the wrong direction – especially when the herd is moving, too. Other examples include hunting other agents or guiding a group of agents towards a particular location. All these scenarios include more than one agent which makes them difficult to be modeled with only reactive rules.

Note that this sort of decision-making is fundamentally different from the reactive agents condition-action rules since it takes into account the question “What will happen in the future if I do this and that?” The reactive agent’s rules do not explicitly contain this information since it is predefined by the designer. Although the reactive agent is much more efficient, the goal-based agent is more flexible since it adapts its behavior with respect to the goal and can therefore easily be changed. For the reactive agent, many rules have to be changed. Additionally, it allows to define multiple goals that can be selected according to the current situation. Therefore, the proactive agents are expected to act with a more sophisticated behavior than the reactive ones. For example, the agents can adapt their movements to dynamic changes of the environment which was not possible with the agents presented in the previous section. Otherwise, an agent can decide online to collect food and change its plan accordingly.

4.1.1 Definitions

The general definition of a goal-based agent has been given in Section 2.2.3. Here, the proactive agent as a sub-type or extension of the goal-based agent from the previous chapter is presented.

Russel and Norvig describe the *proactive agent*¹ as an entity that decides what to do by finding sequences of actions that lead to desirable states [RN96]. The process of looking for such a sequence is called *search*. A search algorithm takes a *problem* as input and returns a *solution* in the form of an action sequence.

They define a *problem* as a collection of information that the agent will use to decide what to do. The basic elements of a problem are states and actions:

- ▶ The *initial state* that the agent knows itself to be in.
- ▶ The set of possible actions available. The term *operator* is used to denote the description of an action in terms of which state will be reached after applying the operator in a particular state.

Both together, they define the *state space* of the problem: The set of all states reachable from the initial state by applying any sequence of actions. A *path* is any sequence of actions in this state space that leads from one state to another.

Furthermore, the problem has other elements:

- ▶ The *goal test* can be applied to a state to determine whether this state is a goal state or not. This can also be a function that returns a measurement indicating the progress towards the goal.
- ▶ And a *path cost function* that assigns a cost to a path which is the sum of the costs of the individual actions along the path. This function can be used to compare two paths in order to use the one with fewer costs.

Such a problem is the input to the search algorithm which returns a *solution* as output. A solution is a sequence of actions that form a path that leads from the initial state to a state that fulfills the goal test. A *partial solution* is possible when the problem's goal test is not binary but continuous, thus, the algorithm can measure the "distance" to the goal. With such a continuous goal test, it is possible to differentiate between different states and select the one that is more likely to lead towards the goal. Then, the partial solution is a sequence of actions that form a path beginning in the initial state, not ending in a goal state, and whose last node has a better goal test measurement than the initial state and any other in the path.

General thoughts about the inference mechanism are discussed in Section 4.2. Furthermore, one can imagine that the planner can have different strategies when exploring the future. Many different strategies exist and we will now have a deeper look into these and discussing their applicability in our environment within Section 4.3.

Exploring the future can take as long as the agent is allowed to do so. Therefore, an agent could take all the time and prevent others from acting or planning in the real-time environment. But, in our case, the agents should plan concurrently and must break their planning after a certain amount of time. We will present a solution where the agent has the possibility to interrupt the planning process without restarting every time from scratch. This problem is addressed in Section 4.4.

Another problem arises immediately after allowing multiple agents planning concurrently and their planning process lasts longer than one simulation step. In this

1. [RN96] refers to it as *problem-solving agent*.

case, the planning process will be outdated after some time since the environment changes dynamically. We will discuss that in Section 4.5.

4.2 INFERENCE MECHANISM

The basic idea is to use the existing reactive agent approach from the previous chapter as a base and extend it to support proactive behavior. But it should still be possible to create purely reactive agents that can be simulated rather fast. A two layer architecture that allows for both reactive and proactive behavior together has been chosen therefore. A schematic overview of this architecture is depicted in Figure 4.1. Basically, the proactive layer determines the normal behavior by delib-

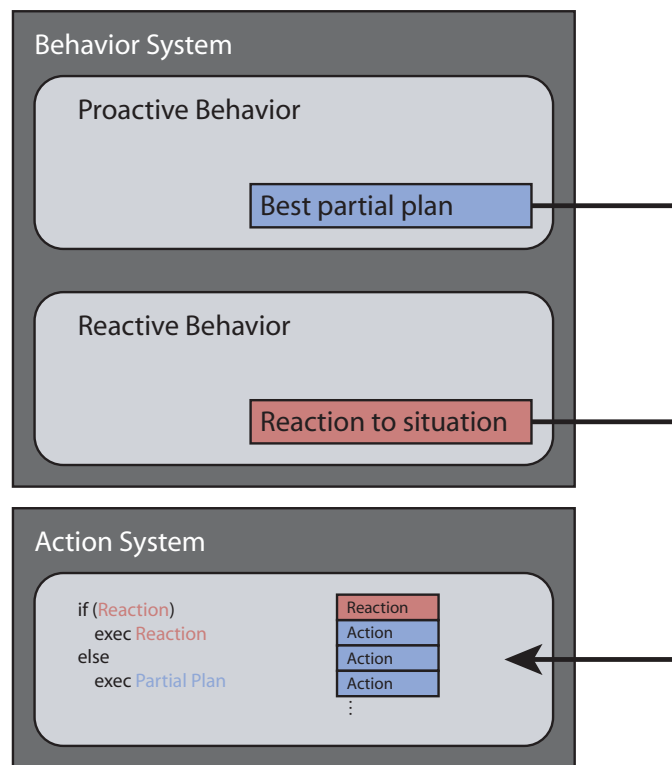


FIGURE 4.1 The two-layer architecture for proactive agents. It has a fundamental reactive system which determines the behavior by simple condition-action rules. The second layer is the deliberative layer which generates goal-based behavior. Normally, the agent acts according to the proactive behavior but sometimes, the reactive subsystem might break in and interrupt it by inserting a reaction.

erating about the best actions to take in order to approach the agents goal. The reactive layer gets active only when some sort of reaction is necessary, for example avoiding an obstacle. Such an extension can be easily integrated into the blackboard architecture described in Section 3.1.3. Actions which are delivered by the proactive sub-system are marked as *actions* and are therefore inserted at the end of the

action queue (see Section 3.4.3). On the other hand, the reactive sub-system generates only *reactions* which are inserted on top of the queue and are therefore executed immediately.

Therefore, an enhancement of the agents architecture in two different ways is necessary. First, the knowledge about the current goal has to be added to the knowledge-base. Second, the sense-decide-act cycle has to be adapted into a sense-infer-decide-act cycle by adding a deliberative sub-system to the blackboard-architecture.

These steps and explanations about the taken decisions are described within the next sections. While the integration of goals is a rather minor step, the design of a suitable planning mechanism is not straightforward at all. At the end of this chapter, we will describe the necessary adaptations of the reactive agent model presented in the last chapter.

4.2.1 Goals

As stated above, the need for a *goal representation* arises. Also, when dealing with multiple possible goals, we either need an inference mechanism that supports multiple goals at the same time or we need a goal selection mechanism that decides which goal should be achieved in order to follow only one goal at the same time. The first approach is very complex and could be a research project of its own. Nevertheless, a single goal representation can be based on a complex goal test function can consider multiple individual goals in parallel, for example searching for food while following a path. Therefore, we decided to restrict the inference mechanism to one goal at the same time and add a simple goal selection mechanism – which is described in the next section.

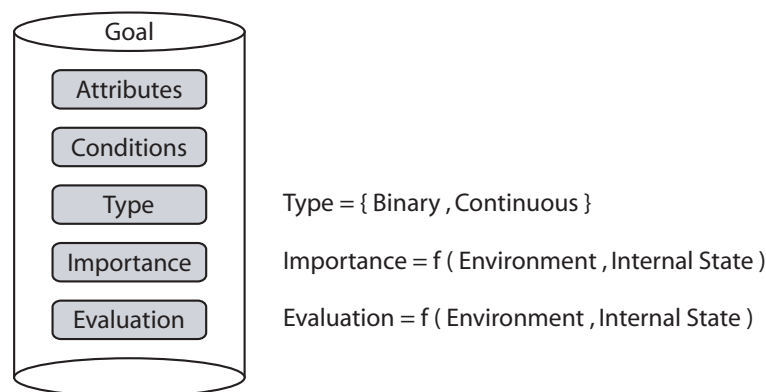


FIGURE 4.2

The goal component.

It consists of attributes, conditions, a type, and an importance and evaluation function. Using attributes, the goal can be adapted and customized. The conditions and the importance function determine the goal's applicability. A binary goal can only distinguish between "goal reached" and "goal not reached yet", therefore, the evaluation function is binary, too. Thus, a continuous goal's evaluation function provides a measurement of the approximative distance to the goal.

In our case, the goal representation is a simple component of the knowledge-base and extends the list presented in Section 3.3.3. Of course, a goal is an attribute container as all other components. Also, it is an condition container to check whether the goal can be activated or not.

Then, a goal should provide a method that returns a measurement of an agent's state with respect to that goal. Of course, the value should be low for states dissimilar to the final goal state and it should have a maximum at each state that is represented by the goal. This function can depend on some attributes which allow for flexible and generic formulations of goals. There might be some sort of goals that can only be evaluated in a binary way – such as having something or not without knowing its present location. Therefore, we distinguish between *continuous goals* and *binary goals*. Obviously, binary goals are less mighty than continuous goals since they do not allow for a stepwise refinement towards the goal. If a binary goal is not within the planning distance the agent will not succeed in finding it. On the other side, a continuous goal will lead the agent towards the goal in small steps.

These are the primary components of each goal. Further methods and members will be explained within the next sections. At the end, in Section 4.6, we will again come back to the goal component and summarize the properties of goals.

4.2.2 Goal Selection

The need for a goal selection mechanism has been explained in the last section. When an agent has more than one goal, it should select only one of these and activate it. This mechanism is achieved in a way similar to the action selection mechanism in Section 3.4.2. Each goal provides a method which returns a value describing the importance of this goal depending on internal and external states. The goal with the highest score is then selected and activated. We call this mechanism *motivational process*, since the motivation determines our actual goals.

Of course, the goal selection mechanism must not change the active goal too much since the agent needs some time to start planning and finding a suitable sequence of actions. If the agent switches very fast between different goals, the resulting behavior might be unintuitive. Therefore, the motivational process adds a certain value d to the importance of the currently active goal without exceeding the maximal value. If another goal returns a higher value the difference between the according importances will be $2d$ afterwards. Therefore, it is unlikely that these goals will be exchanged again immediately afterwards. But since the goals determine their importance by themselves and it is based on internal as well as external states, we cannot guarantee a forth and back between two goals. But this mechanism prevents from goal-flickering if these values are more continuous than discrete.

4.2.3 Goal Evaluation

After having activated a goal, the agent should try to find some sequence of actions that will lead it towards the goal. Such a planning mechanism is basically a search over a graph of possible states in the future. As stated in Section 4.1 a proactive agent needs an initial state, a set of actions, a goal information, and a path cost function. In order to find a solution to this problem it also needs a *search strategy*. Russel and Norvig describe a general search algorithm as shown in Figure 4.3 [RN96]. Obviously, this algorithm will not generate partial plans since it runs until a solution is

```

function General-Search(problem, strategy)
  returns solution or failure
  init search tree with initial state of problem
  loop
    if no candidates for expansion then return failure
    choose leaf node for expansion according to strategy
    if node contains goal state then return solution
    else expand node and add resulting nodes to search tree
  end

```

FIGURE 4.3 A general search algorithm.

found or no more nodes can be expanded. A general search algorithm that deals also with partial plans would have to remember the currently best solution and its progress value. Each time a new node gets expanded and it is no goal node, the value of the current node has to be compared to the actually best value using the goal test function. If the value of the current node is larger, the current node is the end node of the currently best plan.

The agent starts at the current state and applies some of the available actions in order to get a possible state in the future. Of course, such actions should not have the same timely behavior as the executable actions in the knowledge-base since the planner should immediately know the outcome of an action without waiting for execution. For example, when the planner applies a action which moves the agent forward for a certain amount of time, the position after having executed this action can be calculated directly. Therefore, each action in the knowledge-base has to provide at least one *planner action* which represents this action and can be applied by the planner.

4.3 SEARCH ALGORITHMS

In the first section we stated that the inference mechanism can use different planning strategies in order to find the goal state. This section is devoted to these strategies. We will start with an overview before looking at each algorithm more precisely. At the end, we will compare these algorithms and discuss their usefulness and applicability in our environment.

The task of searching can be illustrated by a *search tree* whose root node is the current state and the edges represent the different actions that can be taken. Thus, the nodes represent possible states of the environment. A *search algorithm* starts with the root node and repeatedly selects a node to expand. *Expansion* is the generation of successor nodes by applying possible actions to the actually selected node. The selection of the node and the order of the actions with which the node is expanded is determined by the search strategy. At this point, it is important to distinguish between the *space of possible states* and the *search tree*. The space of possible states is a limited set of states that can be reached while there is often an unlimited number of paths in this space. Therefore, a fully expanded search tree can have up to an infinite number of nodes.

The choice of an adequate search algorithm for a specific application depends on four criterions [RN96]:

1. *Completeness*. If there is a solution, there is a guarantee that it will be found.
2. *Temporal Complexity*. The duration of the search until a solution is found.
3. *Memory Complexity*. The amount of memory needed for searching the tree.
4. *Optimality*. If there is more than one solution, the algorithm will find the best one.

Furthermore, search algorithms can be separated into two different categories [RN96]:

- *Uninformed or blind search algorithms* have no knowledge about the actions and their outcome. They only know the set of possible actions and have to explore many possibilities before finding a good solution. The algorithm can only distinguish between a goal state and a non-goal state.
- *Informed or heuristic search algorithms* on the other hand have some knowledge about the actions and their outcome and can therefore search in a specific direction and do not have to consider sequences that do not lead towards the goal state.

At this point, we will not restrict our application to one of these categories since both kind of actions are possible. Of course, the first category seems to be more flexible and generic since it does not expect the actions to provide some information. But on the other hand, informed search algorithms are much faster than uninformed ones and since speed is a major issue in our real-time environment, we should also have a look at these.

Within these two categories, there exist many different algorithms with different properties. The next two sections will present an overview of such algorithms. But these algorithms cover only single-person problems. Afterwards, we will have a short look into two-person problems where two parties try to optimize their outcome. At the end of this section, we will discuss the applicability of the presented algorithms and present the major problems that are expected to emerge in our environment.

4.3.1 Uninformed Search Algorithms

As stated above, uninformed search algorithms have no knowledge about the actions, their outcome, and the number of steps until the goal state will be reached. They can only separate states that meet the goal conditions from such that do not.

This section presents an overview of the major uninformed search algorithms: Breadth-first search, depth-first search, limited depth search, iterative deepening search, and bidirectional search.

Breadth-first search.

This very simple strategy starts by expanding the root node. Afterwards, all expanded nodes on the first level are expanded. Then, the next level and so on. Generally, all nodes of level d will be expanded before the nodes on level $d+1$. The breadth-first search guarantees to find a solution if one exists. If there are multiple solutions, it will find the shortest one. Regarding the criterions from the last section, the breadth-first search is complete and optimal. Concerning the complexities, this strategy has a time complexity of $O(b^d)$ with d being the length of the solution

and b the branching factor, i.e. the number of actions that are possible from each state. b^d , because the number of expansions for a solution of length d is

$$\text{Expansions}_{\text{breadth-first}} = 1 + b + b^2 + b^3 + \dots + b^d = \sum_{i=0}^d b^i. \quad (4.1)$$

Since all nodes have to be kept in memory, the memory complexity is of the same order. This huge memory requirement is the reason why this strategy is of limited use to our application.

Uniform cost search.

While the breadth-first search finds the solution with the least number of actions this might not be the solution with the lowest cost according to a path cost function. When the search minimizes the cost of the solution by using the path cost function $g(n)$ of a node n , we talk about a uniform cost search. This strategy always expands the node with the lowest cost. If $g(n) = \text{depth}(n)$ then it is the same as breadth-first search. The uniform cost search guarantees to find the cheapest solution. Although it is optimal and complete, it is still rather inefficient compared to other search algorithms.

Depth-first search.

Instead of expanding the node at the lowest level as the breadth-first search, the depth-first search always expands a node at the highest level. Only when a node is not expandable any more and it is no node corresponding to the goal requirements, the strategy goes back and continues using a node at a lower level. The advantage of this search algorithm are the low memory requirements. The algorithm just needs to keep track of one path from the root node to the current leaf node together with the remaining child nodes of already expanded nodes on the path. For a search node with a branching factor b and a maximal depth of m the memory complexity is therefore in $O(bm)$. The time complexity is obviously in $O(b^m)$. For search problems with a large number of solutions, the depth-first search can be much faster than the breadth-first search since the probability to find a solution after having visited a small part of the search space is rather high. But the depth-first search has two main disadvantages: First, it is not optimal since there is no guarantee to find the best solution. Second, it is not complete if the search tree has an infinite height. Then, the algorithm can select a subtree which contains no solution at all. Therefore, the depth-first search should only be considered when searching a depth-limited tree.

Limited depth search.

The limited depth search circumvents the problems of the depth-first search by limiting the maximal depth of the search, i.e. the search tree will be expanded only until a certain depth is reached. If the limit is chosen large enough, this approach is complete but still not optimal. The maximal number of expansions of a limited depth search with depth d and a branching factor b is

$$\text{Expansions}_{\text{depth-limited}} = 1 + b + b^2 + b^3 + \dots + b^d = \sum_{i=0}^d b^i \quad (4.2)$$

which is basically the same as the breadth-first search.

The complexity of the limited depth search are similar to the depth-first search: The time complexity is in $O(b^l)$ and memory complexity in $O(bl)$ with l being the depth limit. The selection of an appropriate limit is very challenging since the maximal depth to a goal node should be known a priori to guarantee the completeness. And this property is not given for uninformed search algorithms.

Iterative deepening search.

This strategy utilizes the advantages of the limited depth search and circumvents the problem of finding an appropriate depth limit by incrementally trying all depth limits starting at 1. This strategy is a combination of the advantages of breadth-first and depth-first search. It is optimal and complete as the breadth-first search but has the low memory requirements of the depth-first search. The order of the expansion of nodes is similar to the breadth-first search except that some nodes get expanded more than once. Therefore, it seems that the iterative deepening search is rather wasteful but for most problems, this overhead is relatively small since the number of multiply expanded nodes compared to the total number of expansions gets smaller with an increasing depth limit. For the iterative deepening search, the nodes at the highest level get expanded once, at the second highest level twice and so on until the root which is expanded $d+1$ times. This sums up to

$$\begin{aligned}
 \text{Expansions}_{\text{iterative}} &= (d+1) \cdot 1 + (d) \cdot b + (d-1) \cdot b^2 \\
 &\quad \dots + 3b^{d-2} + 2b^{d-1} + 1b^d \\
 &= \sum_{i=0}^d (d+1-i) \cdot b^i
 \end{aligned} \tag{4.3}$$

For example, when $b=10$ and $d=5$, the number of expansions is 111'111 in the case of a depth limited search. Using the iterative deepening search, the total number of expansions is 123'456. The resulting overhead is in this case approximately 11%. The overhead for different other settings has been compared, too [Hau03b, Hau03a]. With increasing b the overhead gets smaller while d has no specific influence on the overhead. Even though a large branching factor implies a low overhead, the iterative deepening search with a branching factor of 2 takes about twice as long as the breadth-first search. The time complexity remains in $O(b^d)$ and the memory complexity in $O(bd)$. Generally, the iterative deepening search is used for searches with a very large state space and when the depth of the solution is not known a priori.

Bidirectional search.

The basic idea of the bidirectional search is to start both from the start state and the goal state concurrently. The search stops if both searches meet somewhere in the "middle". Assuming that the solution has depth d , the bidirectional search will find with a timely complexity in $O(2b^{d/2})=O(b^{d/2})$ since both the forward as the backward search have to search only until $d/2$ which seems very interesting. But the implementation of a bidirectional has some drawbacks. First, the actions that expand the nodes have to be reversible in order to achieve a backward oriented

search. Second, there probably exists more than one goal state, thus, the algorithm should be able to select the best one in order to return an optimal result. Third, it should be possible to test efficiently whether a newly expanded node has already been expanded in the other search tree. In order to guarantee an optimal search, both searches have to keep all expanded nodes in memory which results in a memory complexity of $O(b^{d/2})$.

Comparison.

Table 4.1 compares the six uninformed search strategies with respect to time and memory requirements, optimality, and completeness. b is the branching factor, d the depth of the solution, m the maximal depth of the search tree, and l the depth limit. Clearly, all these uninformed search strategies have different advantages and disadvantages that need to be weighted depending on the requirements of the given problem. For example, the depth-limited search is only complete, if l is larger than d or the bi-directional search is only optimal if both sides remember all already traversed paths. Although all strategies make sense under particular circumstances, not all algorithms are suited for our real-time planning approach, as will become clear in Section 4.4.

TABLE 4.1 Comparison of the uninformed search strategies.
 b is the branching factor; d the depth of the solution; m is the maximal depth of the search tree; l is the depth limit

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bi-Directional
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal	yes	yes	no	no	yes	(yes)
Complete	yes	yes	no	yes, if $l \geq d$	yes	yes

4.3.2 Informed Search Algorithms

Compared to the uninformed search strategies, the informed ones use problem-specific knowledge about the search in order to make the search more efficient. Usually, they need an *evaluation function* which estimates for each node the probability of success when expanding this node. Obviously, this evaluation function can be hard to find and is seldom very accurate. In order to provide a generic planning module in our system, informed search strategies cannot be used except when the goal provides this function. Nevertheless, this section provides a short overview of the major informed search algorithms.

Best-first search.

The most simple informed search algorithm is the best-first search. It always expands the node whose evaluation function value is maximal. Assuming that the evaluation function is perfect, this strategy would be rather a direct walk to the goal

than a search for the goal state. As stated above, the evaluation function is not perfect but rather an estimation of the probability of success. Therefore, the best-first search can be easily lead into a wrong direction. It is an “informed depth-first search” and is therefore whether optimal nor complete.

Greedy search.

If the evaluation function is a *heuristic function* $h(n)$ that estimates the smallest possible cost from the actual node n to the goal node, then this strategy is called greedy search. It always expands the node with the lowest cost estimation. As the best-first search, the greedy search is rather efficient but still not optimal or complete since the heuristic function can lead the search into a wrong direction.

A* search.

This search strategy has already been explained in detail in Section 3.5.2 of the previous chapter. The evaluation function of the A* algorithm is the sum $f(n)=g(n)+h(n)$ of the cost function $g(n)$ and the heuristic $h(n)$. Therefore, A* minimizes the total path cost and always expands the node with the smallest cost for the whole solution. The A* search can be optimal if the heuristic never overestimates the cost to the goal. In this case it is even complete and very efficient.

Iterative deepening A* search (IDA*).

In the last section, we saw that iterative deepening can be used to reduce the memory requirements. Using the same trick on A* results in the iterative deepening A* algorithm. In this algorithm, each iteration is a depth-first search which is modified to use a cost limit over $f(n)$ rather than a depth limit. Thus, all nodes inside a certain cost-limit will be expanded during one iteration. IDA* is optimal and complete as the A* search algorithm. But it only requires memory proportional to the longest path that it explores. In most cases, $O(bd)$ is a good estimate for the memory complexity.

Simplified Memory-Bounded A* (SMA*).

Another algorithm that is memory bound as IDA* is SMA*. It circumvents one major drawback of IDA* by using all the available memory to remember already visited states. This makes the search more efficient since some nodes do not have to be expanded multiple times. It is also optimal and complete but it is also more efficient than IDA*.

Iterative improvement algorithms.

The class of iterative improvement algorithms is completely different from the ones we discussed before. They do not start at the initial state but with a complete configuration. Then, they make modifications to this configuration in order to find a better one. *Hill-Climbing*, *Gradient Descent*, and *Simulated Annealing* are the most popular alternatives within these algorithms. The task to find a complete configuration initially does not suit the requirements of our approach. Therefore, we will not discuss this class of algorithms here, but refer to [RN96] as reference.

4.3.3 Discussion of Search Algorithms for One-person Problems

Generally, we'd like to use optimal and complete algorithms since the presented behavior of the agents in our environment should be as good as possible. We have seen that there are different algorithms that meet these requirements. As we have

stated, the informed search algorithms are usually more efficient than the uninformed ones. But the need for an evaluation function is a hard constraint on the search, since this function is different for every problem. Since we can not guarantee that such an evaluation function is available, we will concentrate on the uninformed search algorithms. But as we have seen, the general search algorithm in Figure 4.3 is the same for all presented search strategies. Therefore, we will design our approach to use the general algorithm in order to provide several different search strategies in our environment such that it could be easily extended with informed search algorithms.

4.3.4 Two-person Problems

The above presented algorithms have something in common: They only consider one agent that acts in some environment. But what happens if another agent is proactive and both act against each other? Such problems are known in the AI community as *two-person problems* or more colloquial as *games* – chess for example. The presence of an opponent makes the search more complicated since he introduces uncertainty to the decisions of the other one. But this is not the uncertainty as when throwing a dice since both opponents are expected to act at the best possible rate.

A two-person problem consists of a similar set of properties as the one-player problem in Section 4.1.1 [RN96]:

- ▶ The *initial state*,
- ▶ a set of *operators*,
- ▶ a *terminal test* which determines if the problem has been resolved, and
- ▶ a *utility function* which gives a numeric value for the outcome that can also be negative.

Each person is allowed to apply one operator before giving the opponent the same possibility. Now, we will present strategies that can be applied when dealing with two-person problems. We will not cover strategies that deal with an element of chance.

Minimax Algorithm.

The most general approach is the *minimax algorithm*. The strategy of the player is to find a sequence of operators including the opponent's moves that will lead to a winning terminal state with maximal utility. He has to keep in mind that the opponent has the same strategy and will do everything to optimize his own achievement. We consider a very simple two-person problem as depicted in Figure 4.4 where each person has three possible actions and the problem ends after each person has executed one of these. We generate the tree by starting in the initial state and applying all possible actions for each person which results in nine leaf nodes depicting final states. These leaf nodes are then labeled according to the utility function which gives us the value the final state has for the first person which tries to maximize. Assuming that the second person tries to minimize the outcome it will always select the move that results in the lowest value in the leaf nodes. Therefore, we label each node on the second persons level with the minimum of all leaf nodes attached to it. For example, the left sub-tree results in values of 3, 12, or 8. The second player will chose action A_{11} in order to minimize the result, therefore, the first node on the second level is labeled with the value 3. On the other hand, the first player maxi-

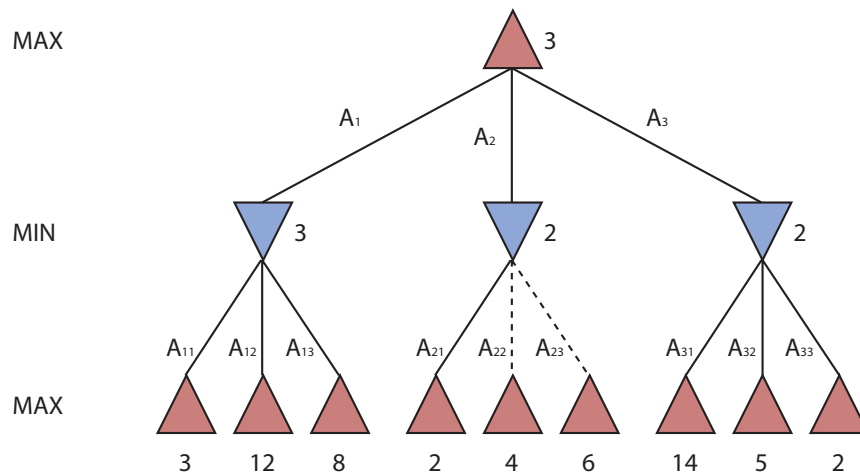


FIGURE 4.4 A tree generated by the minimax algorithm. The Δ nodes are moves by MAX and the ∇ nodes moves by MIN. The leaf nodes are labeled with their respective value for MAX according to the game rules. The labels at the inner nodes are computed by the minimax algorithm. As can be seen, MAX's best move is to take A_1 with MIN's best response being A_{11} which results in a value of 3 for MAX. All other moves of MAX would have resulted in a lower outcome. When applying $\alpha\beta$ pruning, A_{22} and A_{23} will not be considered.

mizes the outcome of its moves. Therefore, he will select the move that will result in the highest possible utility. Therefore, we choose the largest value of the second level as label for the node on the first level which is again 3. Therefore, the first player has to take action A_1 as the first one. If he would take another action, the second player could lead the outcome to a value of 2 which is less than the 3 expected. If the opponent does not play in an optimal way, the result will be even better. The whole algorithm can be easily implemented recursively.

The minimax algorithm has a time complexity of $O(b^m)$ with m being the maximal depth of the tree and b the branching factor, e.g. the number of operators. The space requirements are only linear in m and b . However, the time cost is totally impractical – for chess for example, b is about 35 and m around 100 which results in 35^{100} nodes to visit.

Alpha-Beta Pruning.

The process of eliminating a branch of the search tree from consideration without having examined it is called *pruning*. This strategy uses a particular technique known as *alpha-beta pruning*. It generates the same result as the minimax algorithm but prunes away branches that cannot possibly influence the final decision.

Reconsider the problem shown in Figure 4.4. When traversing the tree in a recursive order, the root node will have the value 3 after having traversed the left subtree. When going down the middle subtree, we reach the node after action A_{21} which has a utility of 2. Therefore, we know, that the whole subtree would yield at most 2 for the root node value. But since we already know that we can achieve a value of 3, we can simply skip visiting the remaining nodes. The right subtree, how-

ever, will be traversed as a whole since this decision can only be made after having expanded the action A_{33} .

In order to do this efficiently, the algorithm keeps track of two values: α denotes the value of the best choice the first person can achieve so far and β is the value of the best choice for the second person, i.e. the lowest possible value. Then alpha-beta pruning will update these values and prune subtrees as soon as it is clear that its values are worse than α or better than β .

Discussion.

When the player as well as the opponent are planning ahead, it is essential to consider these algorithms such that the result can be maximized for the player even when the opponent responds perfectly to minimize it. These algorithms have been developed for two-person games such as chess or checkers where both players make their moves alternately. In contrast, the underlying environment of this thesis allows the characters to make their moves simultaneous. Therefore, the minimax algorithm can not be applied directly. Each character would have to additionally consider the opponents planning process while planning its own moves resulting in an even worse computational effort which is hardly feasible in a real-time environment.

However, at the end of this chapter some scenarios will be presented with more than one character being involved. Therein, the opponents are assumed to act reactively to the planning agent's behavior without looking ahead. Thus, it is easy to simulate the second characters behavior which results in an appealing behavior.

4.3.5 Applicability

Not every algorithm presented in this section can be applied directly in our environment. First, as explained, the two-person problems on which the minimax algorithm relies are not comparable to the situation in our real-time simulation of characters where all act simultaneously. Second, the informed search algorithms require an evaluation function to determine which actions are expected to provide the best gain with respect to the outcome, as explained in Section 4.3.3. Such an evaluation function is feasible for local path planning for example. When the planning mechanism has to provide maximal flexibility and should also be able to deal with unexpected cases, one has to consider the uninformed search algorithms first. These algorithms allow for unbiased planning that does not rely on a predefined evaluation function which eventually prevents the character from exploring states that are not directly coupled with the current goal. Consider, for example, the situation where a character moves toward a particular position but fails to collect some food nearby by detouring. When the evaluation function does not include the possibility to collect food, the agent will not consider the food and pass by even though the food might increase its overall fitness value.

Therefore, the main focus of this chapter will lie on uninformed search algorithms that can be applied to any problem. Nevertheless, such algorithms are not optimal with respect to performance since many unpromising search states will be explored that lead to no successful end state.

However, depending on the scenario and the current goal, the choice of an appropriate search algorithm is expected to provide better performance and a more

natural resulting behavior. There are problems where an optimal result is necessary and the depth-first or depth-limited search will fail to create a proper solution. The opposite way around, some problems may require to find a solution as fast as possible without asking for the fastest or optimal sequence to the goal. Additionally, there are two major problems which will be pointed out in the next section and which limit the possible algorithms due to the environmental properties of this thesis. And, as the next chapter will show, not all algorithms are suited for real-time planning. This will further restrict the number of possible algorithms.

4.3.6 Major Problems

We have seen that we have a large repository of search strategies to select one. But not only the choice of the most suitable algorithm is essential for a good result. There are two major problems that have not been addressed so far. The first deals with repeated states that make the search inefficient, the other addresses the lack of an appropriate undo-function available for every action.

Avoiding Repeated States. We have to avoid repeated states during the search since we can waste a large amount of time by inspecting particular states more than once. If two different paths reach the same state – for example the sequences *forward, turn-right, forward, turn left* results in the same location and orientation as *turn-right, forward, turn-left, forward* in Figure 4.5 – then we should have the possibility to determine that in an efficient way.

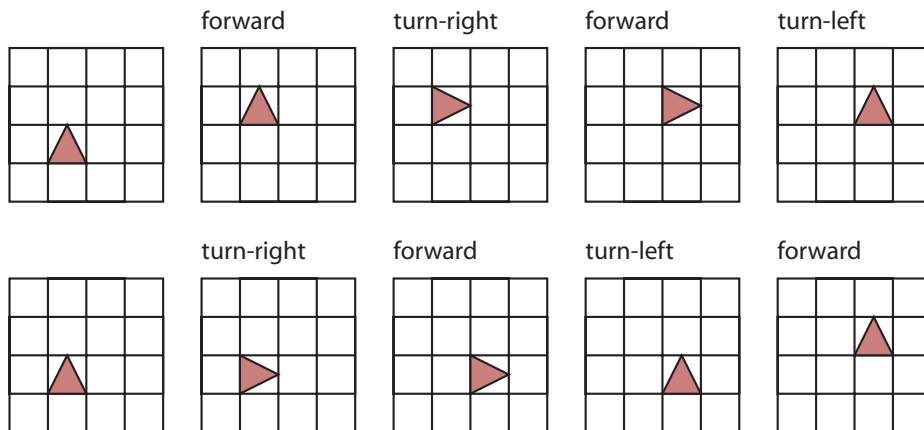


FIGURE 4.5 A simple example for repeated states.

Beginning with the same position and orientation, the sequence *forward, turn-right-forward, turn-left* generates the same state as the sequence *turn-right, forward, turn-left, forward*. The search algorithm should be able to detect such repeated states in order to prevent the planning mechanism from spending time on already visited states.

According to Russel and Norvig [RN96] there exist three different ways to deal with repeated states, in increasing order of effectiveness and computational overhead:

- ▶ Do not return to the state the search is just coming from. This implies that the set of operators does not include the reverse of the last applied operator.
- ▶ Do not create paths that contain cycles. The expansion has to prevent the generation of nodes that is the same as any in the node's ancestors.
- ▶ Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory which results in a memory complexity of potentially $O(b^d)$ but rather $O(s)$ where s is the number of states in the entire state space.

The first method is very easy to implement but not very effective. Not returning to precedent state would even not solve the example from above. The second method implies to always check the actual path against the newly expanded node and is more promising than the first one. The third option is the most effective but at the same time very complex. [RN96] suggest to use a hash-table that stores all generated nodes. This makes the check for duplicates reasonably efficient. But still, the trade-off between the cost of memory and checking and the cost of extra search remains. It depends on the problem and it's "loopiness".

In our environment, this problem seems especially hard to handle. Instead of discrete states we have a continuous environment. If we would like to use a hash-table in our application the need for a appropriate hash-function arises. This hash-function should generate a unique value for each different state. But what features determine the state of the agent? We should at least consider the position. But, for example, is the orientation necessarily part of the state or not? And what about collected items? It is obviously not the same state when an agent returns to a particular position after having gathered food. And what about internal states such as hunger or others? It seems that there is no straight answer to these questions.

We decided to only consider the position of an agent in a fixed grid with a reasonable edge length. This will prevent the search from wasting time by visiting certain *locations* twice or more. It remains the problem that any agent must not visit the same location twice during a search even if it has done something in between that has changed its state, e.g. eating something or taking an object. But as we will see, the planning process has to be restarted regularly with a relatively high frequency of a few seconds and therefore, we assume that the remaining features that have an influence on the state can be neglected.

Undo-Problem. As we have seen, most memory-efficient search algorithms basically keep the path to the current node in memory. If the algorithm reaches a node that cannot be expanded anymore, it has to go back recursively in order to select another node. But what happens if an operator cannot provide such a functionality to undo the last step? For example, the action could have an influence on the environment, e.g. taking an object or influence the movement of a neighbor.

We could regenerate the last visited state by reapplying all actions in the path from the initial state. However, this is very inefficient when the search algorithm has to backtrack often. Another option would be to store at each node the state of the world before expanding that node. Then, we could simply go back to the last node and take its state as the current state without the need for an undo-function provided by the action. Again, this problem is related to the repeated states problem. The question remains which features actually constitute the state of the agent.

We would have to keep track of the state of each agent in the world and even the world with its objects.

We decided to use the second approach but again, we restrict the set of features that determine the state to only a few. In this case, we consider the position, orientation, and velocity of an agent as its state. Additionally, we restrict the set of involved agents to only a few. Most scenarios only involve one or two agents, rarely more. We will only store the states of these while the other agents are regarded as having no influence at all. In Section 4.6, the according implementation will be explained in detail when presenting our proactive agent model.

4.4 CONCURRENT REAL-TIME PLANNING

As we have seen in the last section, search algorithms can be very time-consuming and if the goal is unreachable, they might run forever. But in our environment, only a few milliseconds are available at each simulation step. During these few milliseconds, not only one agent should be activated. In the contrary – as much agents as possible should get the possibility to explore their future.

Therefore, the need for concurrent planning with restricted time arises. We will address this problem within this section. First, the class of anytime algorithm will be introduced and defined. This class of algorithms offers the possibility to stop and be reactivated later without losing the last known state. Furthermore, anytime algorithms provide a continuously improved solution the more time they get. This is a very promising approach to our problem but it restricts the set of possible search algorithms that we have presented in the last section. We will discuss this problem and present according planning algorithms that are suitable for our system.

4.4.1 Anytime Algorithms

According to Grass' definition an algorithm is an *Anytime Algorithm* if it fulfills the following properties [Gra96]:

- It has a *mass of quality*: Instead of a binary notation of correctness, the anytime algorithm returns a result together with a mass of quality which denotes the usefulness of the intermediate result. This means it delivers a value that indicates how far the current results is from the result after a complete run of the algorithm.
- *Predictability*: Anytime algorithms should predict the quality of the returned solution if a particular computation time and some information about the used data is given.
- *Interruptability and continuity*: An anytime algorithm has to be interruptible and the partial solutions available at such a moment are returned. Furthermore, the algorithm should be able to continue its work in order to increase the quality of the intermediate result.
- *Monotonic behavior*: Anytime algorithms keep or increase the quality of the intermediate result the more time they have available. This implies that the quality of the result has to increase monotonically.

Such algorithms are widely used in real-time environments with few resources and where any result is better than none. As we can see, these algorithms have some very nice properties that can help us to achieve the concurrent planning ability for our system. Especially the interruptability, continuity, and monotonical behavior are properties that are desirable for our solution, whereas the predictability is of less interest but nevertheless an interesting property.

In our case, interruptability can be achieved using two different ways. First, we could use an external process that starts and stops the according algorithms. Second, the algorithm could control itself and stop when the available time has exhausted. The first approach would imply the usage of threads within our system – one for the supervisor process and one for each anytime algorithm. This seems not very promising since the management of threads is not for free. Therefore, we decided to adapt the above presented definition to our needs and use self-interruptability as a harder constraint than above. Nevertheless, this will not restrict the choice of an algorithm.

4.4.2 Anytime Planning

The choice to use anytime algorithms for our search problems results in *anytime planning algorithms*. But not every search strategy can be adapted to be used as an anytime planning algorithm. In this section, we will consider the applicability of different search algorithms as anytime algorithms.

Because of the interruptability of anytime algorithms, these have to keep the current state in memory during an interruption. Therefore, the memory-efficient search strategies are more interesting to be implemented as anytime algorithms. As we have seen in Table 4.1, the depth-first search, the limited depth-first search, and the iterative deepening search are very memory-efficient. Since the latter is also optimal and complete, we will concentrate especially on this algorithm. All other one-person search strategies have an exponential complexity with respect to memory consumption. When using one of these, we would have to keep the whole currently expanded search tree in memory for each agent which sums up very fast when dealing with hundreds of agents that act concurrently.

Considering two-person problems, we see that both the minimax and the alpha-beta pruning algorithm are suitable since both work recursively and have also a linear memory complexity. Therefore, these are suitable as anytime algorithms, too.

First, we change the general search algorithm from Figure 4.3 into an anytime algorithm as shown in Figure 4.6. Only a few modifications are necessary. First, the algorithm gets an additional argument: The time available for planning. It determines the run-time of the algorithm. Then, since the function will be called more than once, the initialization should only be done the first time. The simple search loop in the original search algorithm now checks if the available time has exhausted. If not, the algorithm will expand one node before checking the time again. Additionally, the algorithm should remember the currently best intermediate solution even though it might be only partial and returns it after the time is over.

As we can see, these changes do not have a great influence on the search algorithm per se. But we have to consider that an anytime algorithm has to keep its cur-

```

function General-Anytime-Search(problem, strategy, time)
  returns solution or failure
  if not planning already
    init search tree with initial state of problem
  while not finished and time available
    if there are no candidates for expansion the return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return corresponding
    solution
    else expand the node and add the resulting nodes to the
    search tree
    remember currently best solution
  end
  return currently best solution

```

FIGURE 4.6 A general anytime search algorithm.

rent state in memory. Therefore, not all search strategies we have seen are convertible into efficient anytime planning algorithms.

4.5 DYNAMIC ENVIRONMENT

The concurrent planning of multiple agents leads to another problem that arises in our dynamic environment. The anytime planning algorithm will distribute the calculations over a certain time. Therefore, the initial state that is used as the root node of the resulting search tree is already out of date after the first interruption. Then, the search is based on an initial state that does not correspond to the actual state any more.

We could decide to restart planning with the actual state every time the planner is activated. But then, the usage of an anytime planning algorithm would make no sense anymore since the major advantage of anytime algorithms is the interruptability and continuity. Also, if we invest each time only a small amount of time, the planning algorithm could only execute a few steps which will not lead to a good intermediate solution – not to mention that a goal state is not likely to be found within such few time. Therefore, the usage of an anytime planning algorithm is necessary and implicates that we have to plan on a static environment or we have to find a way how the dynamic changes can be integrated without necessarily replanning each time the world has changed.

Therefore, we have to look for possibilities on how to handle the dynamic changes in the environment during the search. Several problems have to be discussed:

- *How can we plan on a static environment when the corresponding world is dynamic?*
 The anytime planning algorithm uses an initial state that remains fixed during the search process. Changing the initial state would imply to regenerate all possible successor states which is basically replanning.
- *What should happen if the environment changes during a search?*
 Small changes or changes not related to the search could be omitted. Other information such as the position of other involved agents have to be considered as critical information.

► *How can we handle partial plans?*

If the planning process is interrupted as intended, we will get the currently best partial solution in return. This solution might not be the best and it is in the majority of cases not complete either. But it has to be a sequence of actions that leads towards a goal state.

► *What happens if the search can not find a goal solution?*

Since the world can change it is possible that there is no solution that originates from the initial state going to a goal state at each moment. Or the goal function might lead the search into a local maxima from which the agent can not find a better state within its search horizon.

Within this section, we will discuss these questions and present some solutions to handle those cases.

Regular Replanning. A simple solution to the first question is regular replanning. Assuming that the environment changes in a continuous manner, we have to force the planning algorithm to restart planning regularly. Immediately after the planning process has been started, the state of the real world and in the planner are the same. After the first interruption, these states differ a little bit since the world has evolved since starting the process but the states inside the planning process have no knowledge about the changes in the world. With every additional interruption, this difference grows and after a certain time, the information in the planner is so much outdated that it makes no sense to plan on this information anymore. Then, the planning process is restarted even though it might not have found a final goal state. Thus, we have to handle partial plans that are most probably not complete. Of course, the difference grows even faster the more agents are involved. Therefore, we divide the global maximal planning time by the number of involved agents in order to get a measurement for deciding upon a reinitialization.

Planning Proxies. But we have still the problem that we cannot plan directly in the dynamic environment. We could obviously generate a copy of the whole world at the moment of the initial state and let the planner act on this copy as long as the planning lasts. But this is a complete waste of memory. The objects that are purely static are not needed to be copied since they will not change. And since only a few agents are usually involved in the planning process, there is no need for copying the remaining agents, too. Therefore, we decided to use proxies that represent the agents in the static planning environment. The next section will discuss more details about these proxies. Here, we just have to know that for each involved agent a proxy is generated which is used in the planning simulation. Each proxy agent knows about its original agent and vice versa. Therefore, we can check the difference between the actual state in the world and in the planning process and use the quasi static proxies instead of the dynamic agents in the world. This approach can also be extended to objects, too, where each involved object is represented by a proxy object. This offers a nice solution to the second question.

Handle Changes. The second question is challenging. First of all, how can we determine which agents or objects have changed since starting the search. More important, we should know how these changes affect the planning process in order to decide how to handle this information.

The above introduced concept using proxies provides an easy way to determine changes in the world. For each agent, we know its initial state and the current state

in the world. If the difference is too large, we can force the planning process to restart with the current states.

Involving objects in the planning process yields another problem. If we decide during the planning process to use an object but this object is taken away exactly in this moment, the planning process could not know about that. It would return a solution that includes the interaction with an object that is no longer available. Our solution to this problem is to generate a proxy for this object. The real object itself knows that there is a proxy connected to it. If the object finds out that its state has changed, e.g. the position has moved, the door has been closed, etc., it will inform all connected proxies about it. Then, the proxy can inform its corresponding planning algorithm which can decide whether to restart or back up to the point where the object was not involved yet.

Handle Partial Plans. When using an anytime planning algorithm, we usually get a partial solution each time we interrupt the algorithm. And if a regular replanning is initiated, the algorithm usually has not found a complete solution yet. Therefore, we have to find a way how to deal with partial solutions. Of course, the partial solution should have a certain minimal length before considering to execute it. If this limit is reached, the partial plan can be executed. When executing a partial plan, some issues have to be considered:

- *The agent might have already executed a part of a former plan.*

Executing the plan each time from the initial state makes no sense. The agent can have already executed the first action of the partial plan returned by the last interruption. But there is no guarantee that the current partial plan has the same first action.

Therefore, we have to check that when generating the current partial plan in the anytime planning algorithm. The algorithm knows the partial plan it has generated the last time. If this plan is active, the algorithm knows that the agent has already started to execute the plan. The algorithm goes through the actions that constitute the plan and searches for the action currently being executed. Then, it knows that all actions before this particular action were already executed and when generating a new partial plan, it will skip the same number of actions at the begin assuming that the first steps of both partial plans are the same.

In most cases, this approach is sufficient enough to provide good results. But, of course, it is not perfect at all. The new partial plan could have a completely different and important action after the initial state than the old one. After skipping this action during the generation of the new plan this action will not be executed at all. However, due to the regular reinitialization, this action will become part of another partial plan if it is still feasible.

- *The agent has executed the partial plan but has no actions to continue.*

This can be considered as the opposite of the first issue. The agent is executing its current partial plan and the planner does not generate a new one. This might come from the fact that the larger the partial plan the longer it takes to add an additional action because of the exponential complexity of the search. Therefore, we adapt the regular replanning rule from above to account not only for the exhausted time but also the state of execution of the current partial plan. If the agent reaches the last action of the plan, the planner is forced to reset and

restart planning again from scratch. If we would restart only after the whole partial plan has been executed the agent might have to wait for the new partial plan to be generated. When the restart happens before the last action is executed, the planner has some time to generate a new plan which then replaces the old one, in most cases before the agent has finished the current action.

- ▶ Additionally, we can not guarantee that this partial solution will finally lead to the goal. Therefore, it is possible that a partial plan can lead the agent in the direction of a local maxima of the goal test function. Due to the regular replanning, we do not know if the problem actually has a solution. We will tackle this problem by failure methods which are called when the search is stuck. These methods provide the possibility to rethink the current situation and take some action to handle it.

Handle unresolvable situations. Using partial plans might not lead every time to a goal state. Since the partial plan is not complete and it is not guaranteed that even the direction of the partial plan is towards a goal state using partial plans can lead the agent into local maxima of the goal's test function. There is no solution that prevents the algorithm from doing so if we have no information about where to find the final state if the algorithm is not complete. But even for complete algorithms, we cannot search over all possible states since the planning process has to be restarted regularly. Therefore, the search cannot reach states over a particular limit in the depth of the search. Another problem, for example, is a sliding door that opens and closes automatically. If all solutions to a problem have to pass this door and the door is closed in the initial state, obviously no solution can be found at all.

Since the current system uses the proactive behavior mainly for local path-planning that includes other actions, e.g. collecting objects or eating food, the main problem for local maxima is therefore the coordination of the movement. And for this, we can use the global path-planning system (see Section 3.5) to generate a static path from the actual position to the goal location. This is done relatively fast compared to an exhaustive search using a generic uninformed search algorithm. We can use the static plan to even improve our solution. If the global path-planning returns a path then it is obviously possible to reach the goal location from the current position. And since the global path consists of waypoints that are connectable by straight lines we can use these as intermediate goals for our local path-planning that takes place within the planner. Thus, if we implement a goal that tries to reach a certain location, it first has to check whether a global path exists and then use the waypoints of this path as intermediate goals. This can be easily done without a great effort.

Nevertheless, we can not guarantee that this approach will not lead into a local maxima or another unresolvable situation as the sliding door. Therefore, each goal description has to provide a method that is called on failure. Failure means that the search strategy has not found a state with a higher value than the current within its limited planning range. Then, the goal can decide what to do and how to handle such a situation. If there is no obvious solution, the goal should deactivate itself and allow another goal to take over.

4.5.1 Planning Proxies

Generating a search tree is usually based on a static environment which does not change during the search. When using anytime algorithms that are interruptible, we can not plan directly on the real dynamic environment because it evolves and, thus, changes during the interruption. As stated above, we do not have to copy the whole world since most parts of it are static anyway and many agents and objects are not involved in the planning process. Therefore, we plan directly on the same static environment but use proxies that represent the involved agents and objects. A proxy is a simplified knowledge-base object representation which basically stores the position and other information needed by the planning algorithm as shown in Figure 4.7.

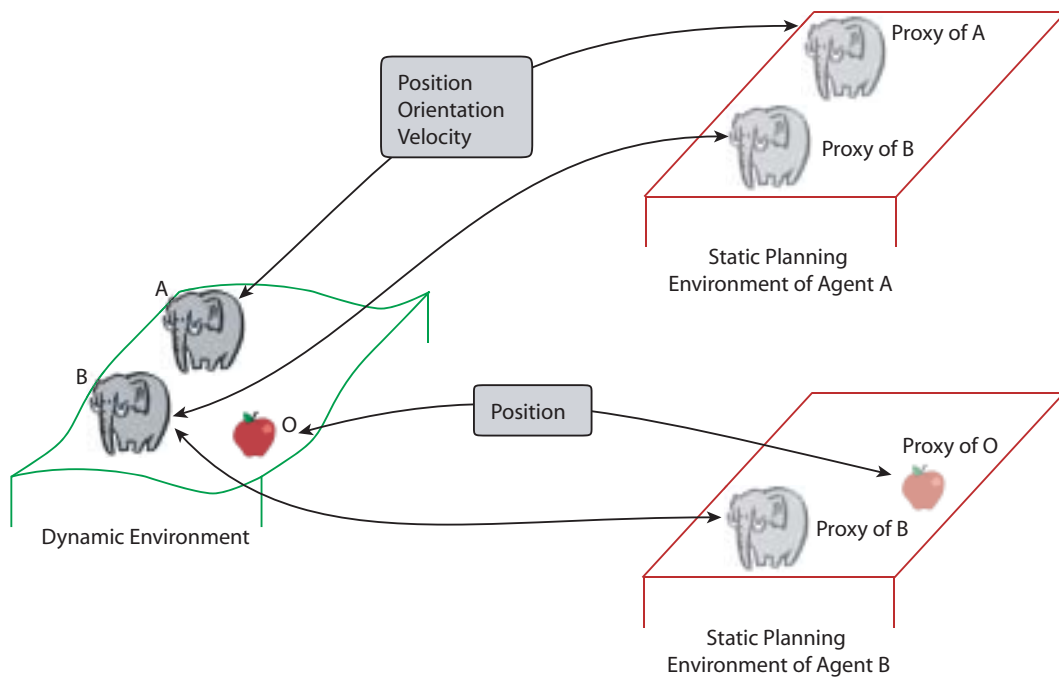


FIGURE 4.7 Proxies represent the real agent in the static planning environment.

The most basic proxy is the *object proxy*. It represents a non-moving object and stores the actual position of the object in the planning process. The proxy object offers a method to update the information which basically retrieves the actual position of the corresponding object from the simulated world.

Next, the *agent proxy* represents the planning agent and extends the object proxy by storing additionally the velocity and orientation. The agent proxy is furthermore extended to the *agent simulation proxy* which offers also the possibility to simulate some very simple reactive behavior. This allows the planning process to incorporate a model of the behavior of the involved agents. For example, the scenario with the sheep-dog can be considered as a two-person problem. Then we would have to plan the behavior of both involved agents. But we can also assume that the sheep has a very simple reactive behavior which does not include a planning strategy. Then, the

agent simulation proxy of the sheep would move just according to the position of the dog which allows the dog to find a reasonable solution. Of course, this works only if the search tree has only a small depth. Since we introduce an error by using a simplified behavior model, this error gets larger the deeper the search tree is. But for our setup with the regular replanning initiations this works very well as the results will show.

The usage of proxies offers some advantages to our planning mechanism. First, we can choose the time for updates individually. Usually, the proxies are updated when the search starts or a replanning occurs. We could also update the information also during the search when knowing that the object has not been involved yet. In between two updates, the search is purely static. And, when using the iterative deepening algorithm, we can update the proxies each time the depth limit has been reached and the search basically restarts. Second, as mentioned above, we can easily handle changes in the environment using dependencies. Each real world object knows if there exists a proxy representing this particular object. If the objects state changes in the dynamic environment, it informs all proxies that the state has changed. Then, the according planning process can decide whether to update the proxy – if the object has not been involved yet – or restart the search from scratch after the update.

So, each planning process handles at least one proxy, namely the planning agent itself. Depending on the goal, there might be some other agents or objects involved which will be represented by agent simulation proxies. For example, in the sheep-dog scenario, the evading sheep will be represented by another agent, too. Therefore, each goal has to provide a list of involved agents from which the proxies are set up according to the specification. During the search, all these proxies will be simulated using their simple reactive simulation model.

4.6 THE PROACTIVE AGENT MODEL

As stated at the begin of Section 4.2, the proactive agent's behavior will make use of the reactive mechanism, too. Therefore, we want to extend the reactive agent model presented in Section 3.1 et seq. with several necessary extensions:

- ▶ *Enable spatio-temporal planning*

We have to plan in the spatio-temporal space since we have to know where and when an agent does something. The time which is used by an action to be executed is considered as the primary dimension. Then, the position and orientation of the objects can be updated according to the timestep.

- ▶ *Introduce continuous planer actions and a converter to executable actions*

As stated in Section 4.2.3, the actions that are used in the planning process cannot be the same as the executable ones in the dynamic environment since they are designed to have a duration. During the planning process, this timely behavior should be incorporated directly into the action. Furthermore, the actions in the planner do not necessarily have a one-to-one relationship to their according executable action. Therefore, a simple converter mechanism is needed.

- ▶ *Add novel components to the agent model*
 At the moment, the knowledge-base only supports components for the reactive agent model. We have to design a planner, goal, and search strategy component that meets the requirements defined in past sections.
- ▶ *Enable time-consuming sub-systems*
 Opposite to the already available sub-systems of the reactive agent model, such as the sensory system, the reactive system, or the action system, the planning system can consume as much time as possible. Therefore, the agent itself (or some higher instance) has to decide how much time the planer has available for inference and pass this value to the planner such that it interrupts the anytime planning algorithm accordingly.

Within the next few sections, we will discuss these topics and complete the final model for proactive agents.

4.6.1 Spatio-Temporal Planning

In our environment, an agent has to plan in the spatio-temporal space because the actions depend on their duration and the execution can move or turn the agent within the environment. To meet the spatial requirements, the agent proxies store the position, orientation, and velocity of the agent. The actions that generate spatial movements will be presented in the next section. But the planning mechanisms has to additionally adapt to temporal planning and provide appropriate functionality.

Each executable action has an intrinsic duration or a user-defined one. If the action's duration is given a priori, the planner has to increase the internal time accordingly. But what happens if the action's duration is variable? Then, the planner can specify the temporal extension itself, maybe depending on the goal's specification of a base time-step. We can expect that the more the planning is in the future, the more inaccurate the predicted states will be. Therefore, it makes no sense to plan with very small steps in the far future as when planning with very large time steps for the upcoming action. The duration of such an action should actually depend not only on the goal but also on the current depth of the search tree state.

We propose the following rule to determine the duration to take for variable actions:

$$duration(d) = duration_{base} \cdot b^d \quad (4.4)$$

where d denotes the actual depth, $duration_{base}$ the duration provided by the definition of the goal, and b the exponential base that is used to account for the increasing uncertainty. The value for the base should be chosen within $[1.0, 2.0]$. Values around 2.0 already lead to proportionally very large timesteps and make the search too inaccurate. The base value can be used to address the problem concerning the local maxima in Section 4.5. If a search shows no progress the planner could decide to increase the base in order to reach states within a larger spatio-temporal horizon.

The duration of each action is now defined, be it either by the action itself or the duration specified in Equation 4.4. Next, we have to design appropriate planner actions that can deal with a predefined duration.

4.6.2 Planner Actions

Each executable action should provide at least one action that can be used in the planner to simulate the action. These actions differ only slightly from the actions introduced in Section 3.3.3.

First, the actions have to provide a function to test whether the action is executable or not for a specific amount of time. The time value originates from the equation above and indicates the desired duration in case that the action has no predefined one. This function is called every time before the action is actually expanding a state. If this function returns a positive value, the action is actually executed and returns its real duration.

Second, the planner actions have to implement an undo-method. This method should not provide functionality to restore the original position, orientation or velocity of the agent since this is done automatically by the proxies. But the action might have some influence on other aspects such as the amount of hunger or the possessed items and so on. Then, this method has to restore the original state of the proxy accordingly.

Third, each planner action has to generate the according executable action for the action system on demand. Thus, the planner action has to provide an action that achieves the same as the planner action has done during planning. The search strategy component acts as a wrapper from planner actions to action system compatible ones. It provides both the sequence of planner actions and the sequence of according executable actions.

A special case of a planner action is the goto-action that is used for moving the agent in the environment. Normally, the goto-action depends on the destination but since the destination is not known a priori we cannot search all possible destinations within some range in the continuous environment. Therefore, we have to provide a set of different planner actions that have an effect as shown in Figure 4.8 and discretize the goto-action. Assuming that the agent has already found the direction of interest, we do not have to search in the backward direction. Therefore, only

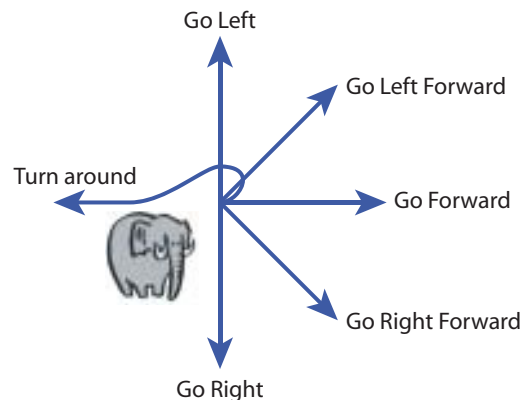


FIGURE 4.8 The planner actions representing the goto-action.

the turn-around planner action provides the possibility to change the orientation for more than 90 degrees. Of course, the intermediate actions left-forward and right-forward are not necessary to span a plane such that possible destinations can be found, they just help to find smoother solutions. One could add further actions in between the ones shown in Figure 4.8. But this would increase the branching factor which has a great influence on the complexity of the search tree and should be kept low as has been shown in Section 4.3.

4.6.3 The knowledge-base components

In this section, we will present an overview of the basic components that are necessary to create a proactive agent. Some components have already been partly introduced, e.g. the goal in Section 4.2.1. Nevertheless, some important properties have not been known at this point. Therefore, a summary of the search strategy, goal, and planner component from a modelling viewpoint is given here. The strategy is basically the implementation of the different search algorithms. The goal provides the goal test function and the planner is a generic search algorithm that uses the strategy and goal component to drive the search.

Anytime Strategies. As manifested, the search strategy determines the next possible node to expand using some of the presented algorithms in Section 4.3. We define the inner part of the while-loop in Figure 4.6 as the main component of the strategy which therefore has to store the list of expanded and expandable nodes.

Before using the strategy, it has to be initialized. For example, the values for $duration_{base}$ and $base$ are retrieved from the goal's specification. Additionally, some strategies might need further properties such as the maximal depth or similar options. For the regular replanning event, the strategy also provides a method that resets the strategy using the same goal as provided during initialization. During the search, a generic planner like in Figure 4.6 repeatedly calls the strategy to execute one step until the time available has exhausted. This is the method where all strategies differ from each other. After each step, the strategy provides the current state which is compared to the currently best one using the goal's test function and probably replaces it.

As stated in Section 4.3.6, not every planner action can provide an undo-method. Therefore, we decided to use a stack of history entries that store the position, orientation, and velocity of each involved agent such that the primary states of the proxy can be restored easily. This stack is used in every strategy and provides two methods: One to insert the current state of all proxies and one to reset the states of the proxies to the values that have been stored before as shown in Figure 4.9. Other states than the mentioned ones have to be maintained by the according planner action's undo method.

Here, we will present the anytime iterative deepening algorithm as an example. An overview of the algorithm is shown in Figure 4.9. This figure shows the part within the while loop of Figure 4.6.

The algorithm starts by determining the next possible action which is tested for possible execution. After the current states of all agents have been added to the history the action is executed and the algorithm remembers the current action on this level, adds the expanded action to the actual path and increases the actual depth. If

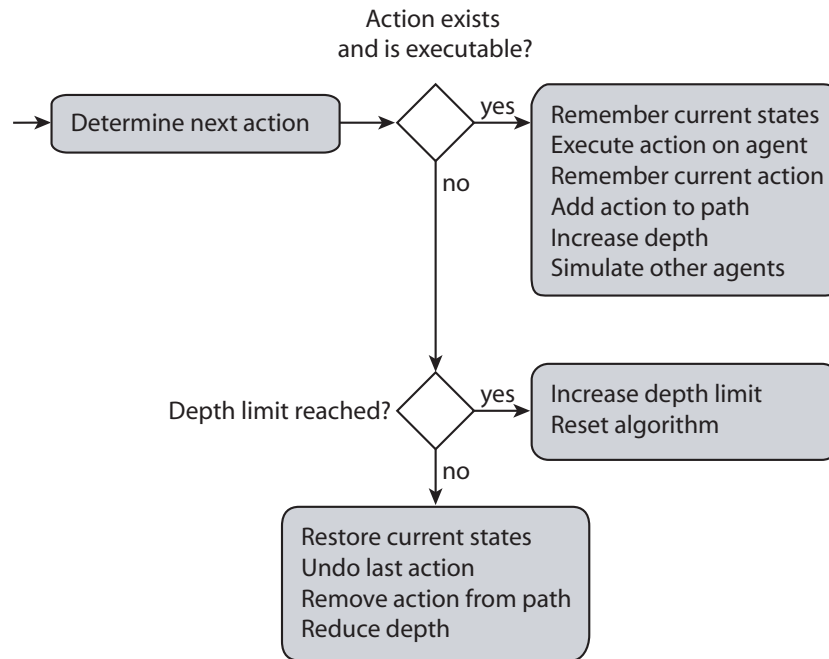


FIGURE 4.9 The anytime iterative deepening algorithm core loop. This algorithm is executed as often as possible before the time available has exhausted.

there are some other agents involved, they will be simulated using a simplified reactive model which will be described in the next section. If the action could not be executed because the maximal depth has been reached, the algorithm increases the depth limit and resets itself. If the depth limit has not been reached, it restores the states in the current node from the history and removes the last action from the actual path. When no executable action could be found the algorithm goes back one step and continues on that level.

So, every time this algorithm has found a new executable action it will return the partial plan leading to the state after this action. The generic planner then uses the goal test function to determine the value of this intermediate solution. If the value is larger than the previously best plan then the best plan is updated. The algorithm can be called as often as possible. Each time, the depth limit is reached it will be increased and the algorithm starts from scratch. The search will stop when the goal is reached.

Goal. The basic parts of a goal have been discussed in Section 4.2.1: The attribute- and condition-container property, the binary or continuous test method, and the motivation test. In this part, we will summarize the further properties that we have developed within the last sections. An overview of this component is given in Figure 4.10.

We have seen that the choice of which algorithm to take strongly depends on the actual goal that is pursued. Therefore, a goal has also the search strategy associated which is summarized afterwards. Additionally, the goal may depend on some other agents or objects that are involved in the planning process, e.g. the evading sheep or

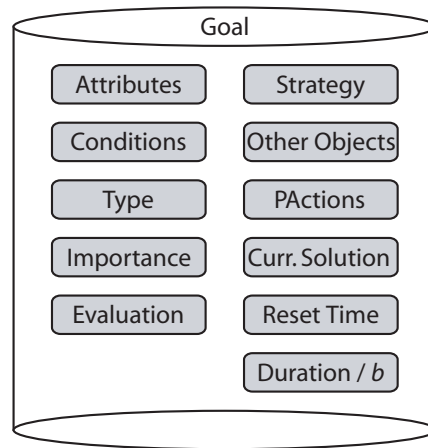


FIGURE 4.10 The extended goal component. Compared to Figure 4.2, the right column has been added depicting the additional components of the final goal component. The search strategy, the other involved objects, the planner actions, the currently best solution, the reset time and the base duration with the exponential base b .

the attacking enemy. The set of possible planner actions that defines the branching factor in the search process is stored in the goal, too. This list should be restricted to only the necessary actions in order to reduce the complexity of the search. By default, the goal simply collects the provided planner actions of all actions that are associated to a situation component of the agents reactive system. Of course, the goal can also specify some additional planner actions that are not within the set of reactive actions.

During the planning process, the planner has to provide the best partial plan. This depends on the goal, therefore, it also stores a copy of the currently best solution already transformed into executable actions. It is valid as long as the replanning is not initiated anyway due to expiration of the solution. In order to check this time, the goal needs a replanning frequency or a reset time.

Also, the goal needs a replanning frequency $duration_{base}$ and the value for $base$ to calculate the duration for each steps. The calculation takes place in the algorithm, but the values are bound to the goal since these values have a strong relationship to the goal.

Last but not least, we stated in Section 4.3.6 that the search can lead into unresolvable situations. Therefore, the goal has to provide a method that will be called in such a case. This method should try to resolve the problem, by default by resetting the goal once and in case of a second failure before any other reset by deactivating itself. But there might be some purpose-built goal that handles failures with a special treatment or a change in the test function and bypass the default case.

As stated in Section 4.3.6, we have to avoid repeated states during the search. We decided to use a hash-map that stores the visited positions during the search. Since the positions can be reduced to two dimensions, the hash-function has been implemented as

$$hash(p) = ext \cdot \left\lfloor \frac{p_x}{grid} \right\rfloor + \left\lfloor \frac{p_y}{grid} \right\rfloor \tag{4.5}$$

where p denotes the actual position in 2D, ext is the extension of the environment in y-direction, and $grid$ is the size of the grid that is used to discretize the positions. This maps every possible position within the environment to a hash-value which is used to mark the position as visited. Inbetween two resets, this particular position cannot be visited again even if the agents state has changed in between.

Anytime Planner. Each proactive agent must have a generic anytime planner component as shown in Figure 4.11 which provides the agent’s inference mechanism. Basically, the planner receives a goal that provides the goal test function and an anytime strategy the determines the order the search tree nodes are visited.

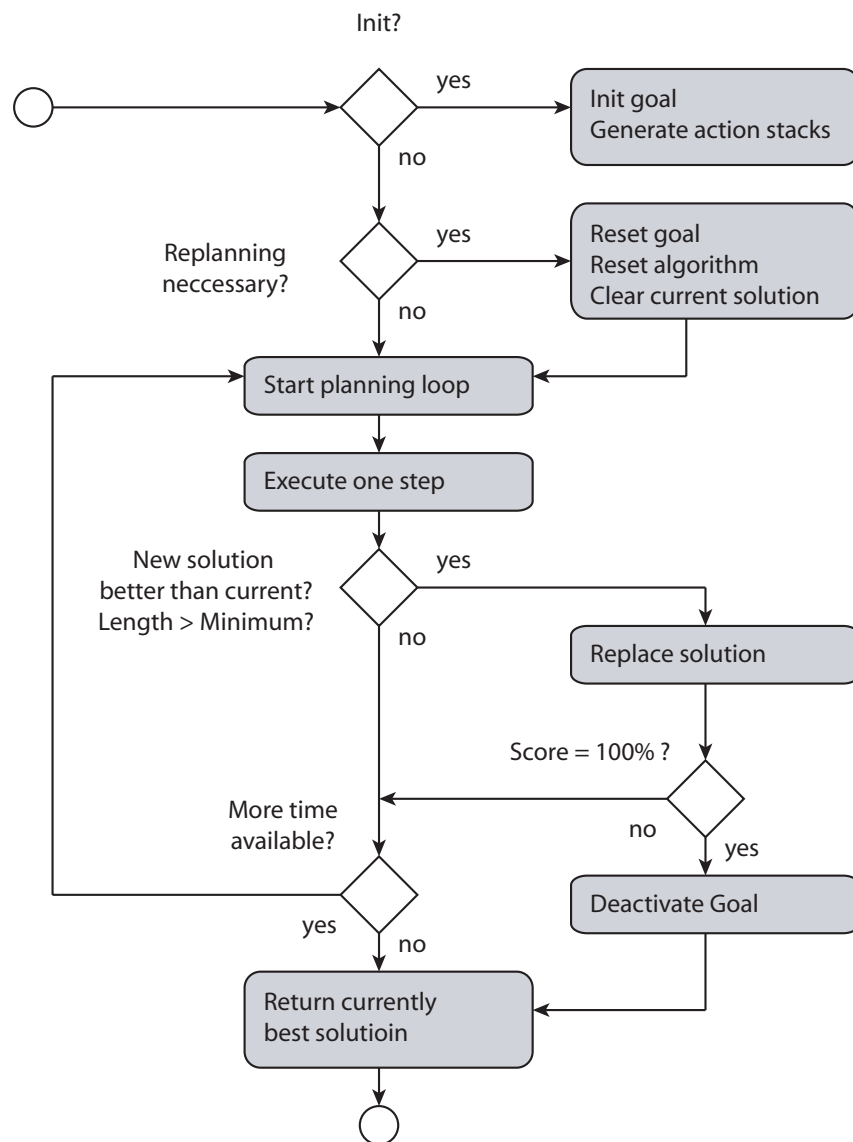


FIGURE 4.11 The planner component workflow.

The generic planner initializes itself and the goal which collects all necessary planner actions. For each planner action a stack of preallocated copies of this planner action is generated. The usage of preallocated action speeds up the search because the allocation and deallocation during runtime is crucial to the performance. These stacks have a predefined initial length. If the search process needs more than the previously allocated ones, the stack increases the number of planner actions. Therefore, during the first few iterations, the stacks will generate a suitable number of planner actions that seldom have to be increased.

Before starting a particular search, the planner resets the planning process if the regular replanning time has exhausted or the last action of the currently best partial plan has been reached. If replanning is necessary, the planner resets the goal and the strategy and clears the current solution. Afterwards, the search loop from Figure 4.6 starts until the time available has been spent.

As declared, the strategy provides a method that executes one step of the search algorithm. The anytime planner is responsible that this method is called as often as possible during the available time frame. After each step, the generic planner retrieves the score of the actual search state from the goal and compares it to the score of the currently best partial solution. If the score is better and the current solution has exceeded the minimal path length, the planner gets the sequence of executable actions from the search strategy. If the score equals 100%, the goal is considered to be reached and will be inactivated in order to let the motivational process select another one.

But this generic anytime planner differs highly from all other components that constitute a proactive agent. The basic components inherited from the reactive agent all have a deterministic comportment while the anytime planner can run as long as needed until finding a goal state. Therefore, another adaptation to the existing reactive agent model has to be considered.

4.6.4 Blackboard Extension: Time-consuming Subsystems

The anytime planner is the only component of a proactive agent which can take as long as time is available. Therefore, it needs a maximal time to run that specifies when the planner should interrupt itself and return its currently best solution. When thinking about time-consuming sub-systems, other possibilities for such components arise: For example, a learning process that tries to optimize the behavior of the agent by changing some of its attributes. Such a learning system could obviously take as much time as available, too. Another such system could try to predict futural events to show surprise or disappointment in order to enhance the visual appearance.

Since our framework should support extensible agents whose behavior can be changed by adding novel extensions, the blackboard mechanism presented in Section 3.1.3 has to be adapted and the interface to the time-consuming sub-systems is crucial. Assuming, the agent knows how much time it has available, it has to find out how much time remains beside the basic reactive behavior which is necessary on each step. The remaining time can be spent on the available time-consuming sub-systems. At the moment, our agents only support one type of such components – the planning system. Therefore, we can spend all the remaining time into planning. If the agents are extended by other time-consuming components we

would have to think about a more sophisticated scheme that distributes the time available. There are various approaches possible, such as giving each component the same amount of time, using priorities that determine how the time is split up, or using a round-robin scheme that regularly selects one component to be activated. But such discussions are out of the scope of this work.

The design of the reactive agents as presented in Section 3.1 has been chosen carefully to allow for further extensions such as proactive behavior. The blackboard mechanism described in Section 3.1.3 is the base for the proactive agent model. It can be easily extended with additional units and slots which provide the mechanisms needed for proactive agents. As described in Section 4.2, the behavior mechanism is now in principle a two layered scheme where the reactive and proactive behavior are both activated during the agent's cycle. As shown in Figure 4.12, the approach shown in Figure 3.2 has been changed only slightly. Additionally, the goal selection and planning mechanism had been added as units that access the blackboard which has been extended by two new slots: The current goal is selected by the goal selection mechanism and provides the planning process the knowledge necessary to find an appropriate action sequence. This sequence is stored in the current partial plan slot that is accessed by the action execution system as described in Section 4.2.

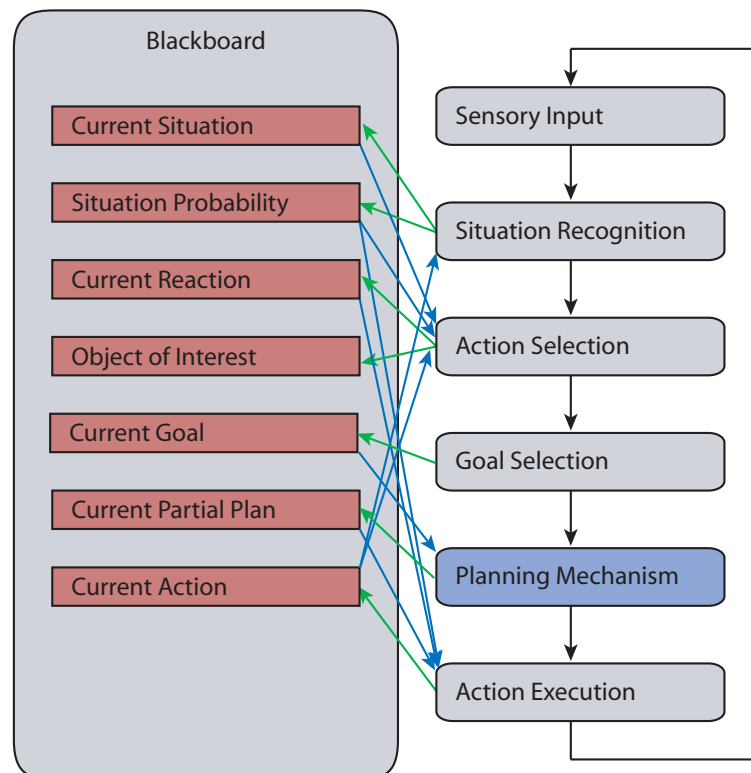


FIGURE 4.12 The blackboard mechanism extended for proactive agents. Compared to Figure 3.2, the Motivation Selection and Planning Mechanism have been added to the units and the Current Goal and Current Partial Plan to the slots in the blackboard. The Planning Mechanism unit (blue) is a time-consuming unit whereas all other units operating time is deterministic.

4.7 RESULTS

As in the last chapter, we conclude this chapter with some results about the mentioned mechanisms. In this chapter, the proactive agent model has been presented and therefore, we will have a closer look at such agents. As a proof of concept, we will show some screenshots of scenarios that present proactive behavior. The presented behavior is not very elaborated, however, the basic mechanisms can be shown.

The first example is a single elephant using its planning system to find a path in the dynamic environment. The goal is to reach a certain location, for example a waypoint of a static path, by considering dynamic changes and minor obstacles in the environment which are not considered by the static path-planning system presented in Section 3.5. In Figure 4.13, a sequence of screenshots is depicted where the camera follows such an elephant. The currently active plan is shown as a yellow line. As can be imagined, the plan gets updated regularly and the elephant adapts its way according to the plan. In average, the elephant has a plan length of around three or four steps ahead.



FIGURE 4.13 Sequence of an agent planning.
The agent tries to reach its goal with the planning system.

The next example shows the same scenario but this time, multiple agents are planning concurrently. In the scene depicted in Figure 4.14, fifty agents act proactively and generate individual plans. Because of the anytime algorithm, the agents do not interfere and can refine their current plan gradually.

Now, we add additional items to the scene that can be collected by the agents. They will now walk around and when an internal variable representing the hunger of the animal exceeds a limit, an apple in the neighborhood is collected and the variable's value decreased. The internal variable is increased in every step such that this event occurs regularly. The sequence of screenshots in Figure 4.15 depict the scene. First, the elephant walks through the scene in a rather straight direction. It gets hungry and perceives food depicted by an apple. Immediately, it starts planning a path dynamically to the apple and eats it. Afterwards, it continues to stroll around.

If there were another apple lying nearby, the elephant could act differently if the value of hunger would be still high enough to look for food. Else, it ignores the apple and continue its way.

A more sophisticated scenario is the sheep-dog which brings back a sheep that left the herd. Since the sheep fears the dog, it will walk away from the dog. This

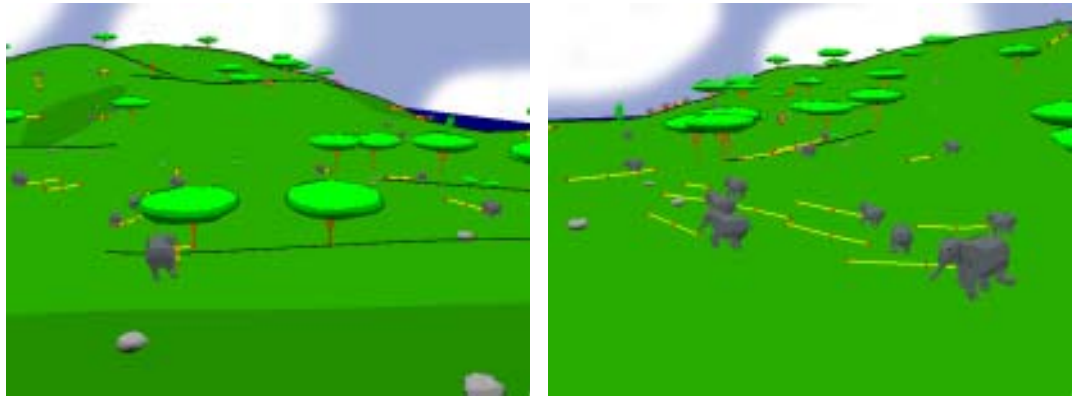


FIGURE 4.14 Multiple agents planning concurrently. Each elephant plans as depicted in Figure 4.13. The whole scene consists of 50 agents which act concurrently in a proactive manner. Due to the anytime planning algorithm, all have approximately the same plan length.

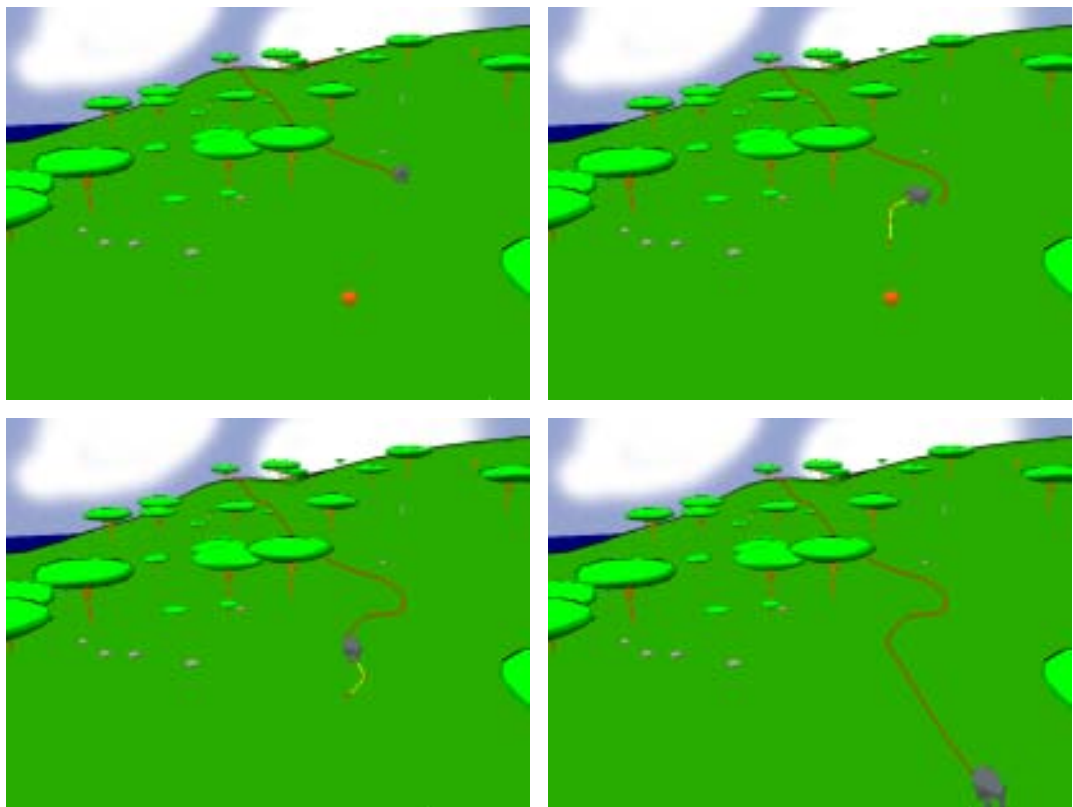


FIGURE 4.15 An elephant collecting food. Top left: The elephant strolls around the scene and gets hungry. Top right: It perceives food (red apple) and starts planning a path towards the apple. Bottom left: It has found the apple and eats it. Bottom right: Afterwards it continues its previous behavior – strolling around.

makes this scenario hard to be implemented using purely reactive behavior. The dog has to approach the sheep from the backside in order to push it back to the herd. Especially when the dog is in between the sheep and the herd, this is hardly feasible using reactive rules but very easy with our proactive model. It is just necessary to define an appropriate evaluation function depending on the position of the herd, dog, and sheep and the dog adapts itself automatically to the situation. The screenshots in Figure 4.16 present some according situations. The yellow lines visualize the current plan of the rear elephant and the blue lines the expected reaction to this plan. However, the actual movement of the other elephant will differ from this expectation and the planning agent has to adapt its plan accordingly. On the top row, the goal location is in the back of both elephants. Therefore, the rear elephant has to make the front one turn around. In order to achieve this goal, it has to circle around the front one. On the bottom row, the goal location is in the same direction

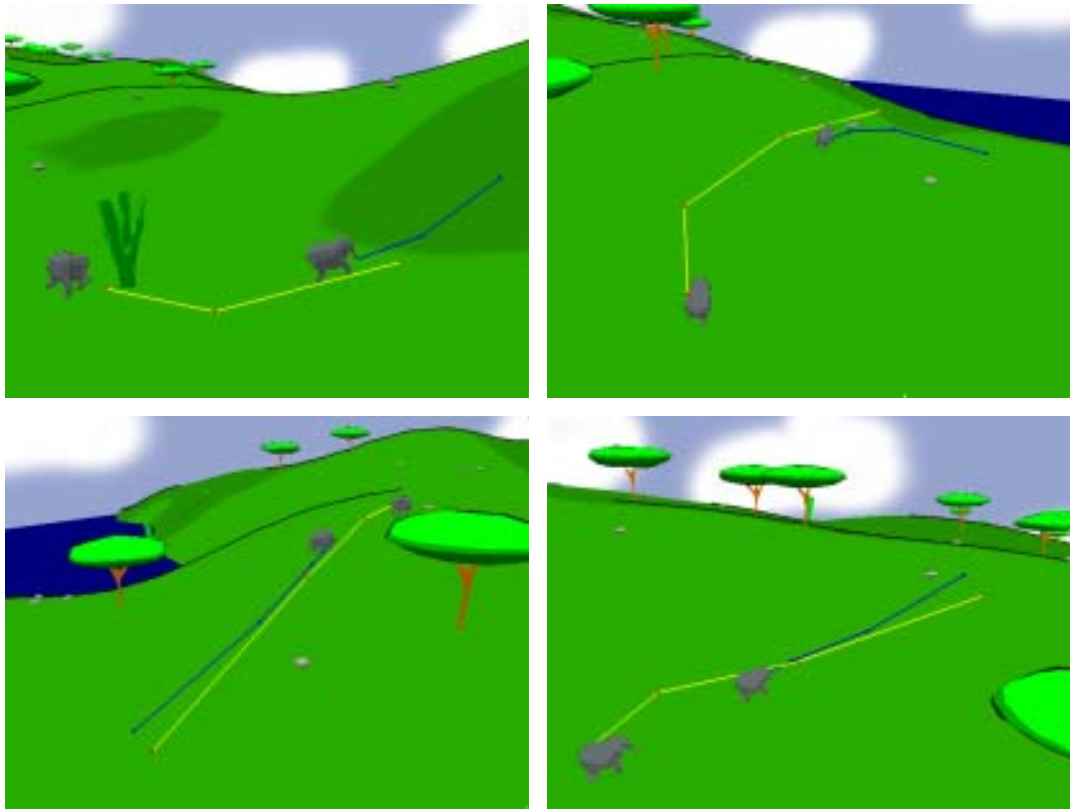


FIGURE 4.16 Sheep-dog behavior. The rear elephant tries to bring the front elephant to a specific location. The front elephant always flees from the rear one such that it is necessary to walk around the front one. The yellow lines depict the rear elephant's actual plan and the blue lines the according response expected by the rear one. Note, that the blue lines will not match the actual path of the front elephant.
 Top: The goal is in the back of the scenery and therefore, the direction should be inverse. Therefore, the chasing elephant has to circle around the front one in order to make it turn around.
 Bottom: The goal is in the same direction and therefore, the rear elephant just has to push the other one towards the goal location.

as the elephants are moving to. Therefore, the rear one just needs to remain behind the other one and adjust the direction only slightly.

Of course, this scenario needs no adaptation when dealing with a moving goal location such as a herd. Because the planner automatically updates the destination accordingly the plan will adapt to the new situation instantly.

In order to get a significant profiling of the planning architecture, we decided to allow the agents to do full planning with different maximal depths. Full planning means that every agent is allowed to search the whole search space in every step. The according results are shown in Figure 4.17 where the vertical axis denoting the time in milliseconds is logarithmic. We compared two different setups: The blue scenario uses agents that have six different planer actions while the red scenario consists of agents that have twelve planer actions. This number corresponds to the fanout of the search tree. Then, we allowed the agents to plan up to different maximal depths of the search tree in order to restrict the time needed. With three to five steps planned ahead, we clearly see that the time needed to create such plans gets tremendous when the complexity is large. For example, an agent with 12 planer actions and a maximal depth of five would need about 850 milliseconds to traverse the whole search space. These results emphasize the need for anytime algorithms which break this long calulations into small chunks.

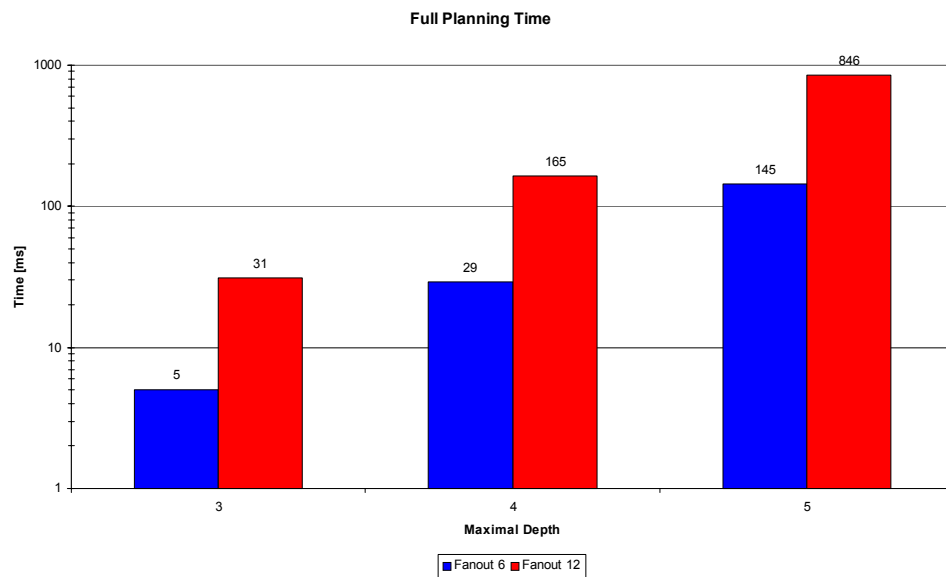


FIGURE 4.17 The average time to achieve a full plan for different depths and fanouts. The chart shows the time in milliseconds which is needed to traverse the whole search space for problems with different fanouts, i.e. the number of available planer actions, and maximal depths of the search tree. Note that the vertical scale is logarithmic.

When turning on the anytime functionality, there is no full planning possible anymore, since the reset time restricts the maximal time to generate a plan before the need for a reset arises. Since we can not guarantee that the maximal depth is reached, we have to measure the average depth of a search until it needs to be

restarted. Thus, the reset time has an influence on the average planning depth of anytime planning algorithms. But also an increasing number of agents in these scenarios should reduce the average depth. We set up a scenario with a variable number of agents and different reset times. We allowed the agents to pursue a goal and each time, the planning algorithm was restarted, we measured the depth of the previous solution. As explained in this chapter, this can happen either when the reset time has been reached or when the agent has reached the last action of the actual plan. The according results are shown in Figure 4.18. The three lines denote the different reset times of 10, 5, and 1 second, respectively. The horizontal axis denotes the number of agents and the vertical one shows the average depth of all these agents. The average depth is reduced as expected with an increasing number of agents and with a shorter reset time. For example, a single agent planning with a reset time of 5 seconds (green line) achieves an average depth of approximately 4 steps while one in a group of 100 agents realizes a value of only 3.2. This chart should be considered carefully, since these values also depend on the rendering time which depends also on the number of agents and has an influence on the average activation frequency.

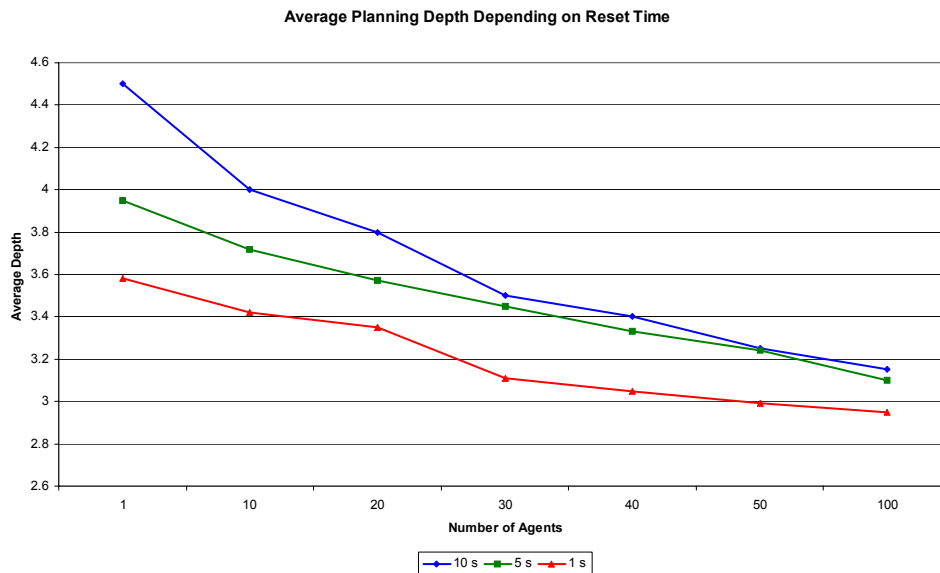


FIGURE 4.18 The average planning depth with anytime algorithms in different setups. The three lines denote three different reset times of 10, 5, and 1 second, respectively. The number of agents in the scenario is denoted on the horizontal axis while the average planning depth is shown on the vertical axis.

HIERARCHIES AND LEVEL-OF-DETAIL

During the last chapters, we have developed a model for reactive as well as proactive agents in a real-time environment. We have seen that reactive agents can be simulated rather fast while proactive agents can use as much time as available. This makes great demands on the agent engine that simulates the agents with respect to time-management.

This chapter will present methods that have been developed during the research that lead to this thesis. The methods aim at breaking down the complexity and speeding up the simulation while maintaining a proper visual impression for the viewer. First, we will present the ideas such that the reader has a short overview of this chapter and the used concepts. Then, after a section on related work, we will rehash the hierarchical group structures that can be used to break down the complexity of some problems. Afterwards, we will introduce a *level-of-detail (LOD)* approach that determines a priority for each agent depending on the camera position. These priorities are used within a *scheduling algorithm* that distributes the time available to the agents. Last but not least, we will introduce a method that uses again the hierarchies within groups to pass the control over to superior agents when the individuals have a low priority. The chapter will conclude with results that show the applicability of our methods.

5.1 INTRODUCTION

As known from computer graphics, there exist various methods to speed up the processing of the graphics output. For example, multi-resolution approaches for meshes and view-frustum culling are only a few among them. The main goals of such methods are to maintain the visual impression or quality while reducing com-

putational costs wherever possible. We will discuss such and other approaches for our behavior simulation in the next sections.

In our environment, time is the most crucial factor. We have to find some way to reduce the complexity of our approach. Obviously, we can utilize the hierarchical structures within recursively defined groups of agents as presented in Section 3.3.5. By defining dependencies within the generated hierarchical tree the agents decisions can rely on the behavior of their superior agents. Such an approach has been used to speed up moving a group of agents to a specific location. Furthermore, the herding behavior of such groups can be speed up accordingly. We will present a hierarchical solution to path-planning and to Reynolds flocking algorithm in Section 5.3.

As we have seen, proactive agents are very time-consuming and have to be restricted in order to allow concurrent behavior. We therefore have to provide a time-limit to each agent when activating it such that the agent can return after at most this limit. Thus, we need some scheduling method that determines when and for how long an agent can be activated. A simple round-robin scheduler simply distributes the time equally over all agents which seems fair but not optimal. This decision should depend on the visibility and the distance to the viewer. Hence, we introduce an approach for level-of-detail on the behavioral and cognitive level. Each agent gets a priority assigned which indicates the importance of that agent within the current view of the world. As opposed to view-frustum culling, we cannot totally neglect invisible agents since the world is expected to evolve continuously. Our level-of-detail method will be presented in Section 5.4 before accounting for the according scheduling algorithm in Section 5.5.

The last two chapters have introduced the reactive and proactive agent model. While the former is fast but not sophisticated, the latter is time-consuming but more advanced. Hence, a group of proactive agents is in great demand of time. It makes only sense to allow each agent to act proactively on a individual basis if it is visible and near the camera. If the whole group is far away, the individuals will not be recognized as such anymore. Therefore, we could make use of the hierarchical organization and pass the control over individuals up to the superior agents until one agent controls the whole group. This approach is presented and discussed in Section 5.6.

5.2 RELATED WORK

LOD concepts such as view-dependent terrain simplification algorithms, multi-resolution modeling, and geometry simplification are widely used and well known in computer graphics [GGS95, Hop97, Paj98, HG94]. Most approaches are restricted to geometric modeling issues, including some approaches dealing with LOD for animation [CH97, SF99]. Only few work has been done on LOD in the kinematic and physical layer. Cheney et al. introduce proxies as computational inexpensive replacements of invisible dynamic objects which are used to efficiently simulate large crowds of agents moving along paths where distant agents are replaced by probabilistic approximations [CAF01, ACF01]. Brogan and Hodgins automatically builds a simplified model of the characters movement abilities which is used to speed up simulation [BH02].

On the behavioral layer, hierarchical sensors, actions and contexts that allow more complex behaviors and also group engagement were discussed by Atkin et al. [AKW+01]. Group behavior has also been thoroughly investigated in [MT97, UT01]. Musse and Thalmann present a hierarchical model for simulating virtual human crowds [MT01]. These models all rely on the reactive agent concept, whereas Funge introduces a cognitive modeling language which easily generates sophisticated behavior of individuals through a knowledge representation that allows for reasoning and planning in addition to reactive behavior [FTT99]. Canamero presents an approach for motivational behavior [Can97], Aylett et al. motivations and continuous planning together [ACP00], and Grosz et al. discuss planning within groups of agents [GHK99]. As discussed in Chapter 2, Bruderlin et al. and Isla et al. exploit hierarchical approaches within an agent [BFEM97, IBDB01], while Atkin et al. present a system which makes use of command hierarchies within groups of agents [AKW+01]. O'Hara proposes a system that automatically generates hierarchies of stable subgroups for Reynolds flocking algorithm [O'H02].

With respect to LOD on the behavioral level, O'Sullivan et al. present a framework which allows for LODs within geometry, motion, and on the cognitive level even for groups [OCV+02]. Their approach uses role-passing [MDCO02] to adapt a character's possible behavior depending on its LOD. An approach by Musse et al. introduces three different levels of autonomy for a virtual character: guided, programmed, and autonomous [MKT99]. However, they only compare these levels against each other and do not infer on switching from one level to another automatically.

5.3 HIERARCHIES

As stated above, the hierarchies within recursively defined groups of agents (see Section 3.3.5) can be used to break down the complexity of various problems. But hierarchies appear not only within groups of agents but also in other domains that have been discussed in the last sections – for example hierarchical sensors, situations, and also planning. In this section, the emphasis is placed on the hierarchies within groups of agents. We show how such a group structure can be used to speed up the task of path-planning for a whole group of agents that wants to move to a particular destination. Another example is a hierarchical flocking algorithm that generates similar behavior as Reynolds' famous boids [Rey87].

When using such hierarchies, each agent has to know its superior and inferior agents. Additionally, the agents of the same level that have the same parent can be worth to know sometimes as we will see. Last but not least, the agent has to know the head of the group itself. This setup is depicted in Figure 5.1. The according references are set up during instantiation of the hierarchical group.

5.3.1 Hierarchical Path-Planning

When a whole group of agents has to move to a particular destination it would be wise to not allow every member to request a path to the destination using the path-planning system introduced in Section 3.5. As we have seen, each path-request has some computational cost which is not negligible if hundreds of agents would request a path.

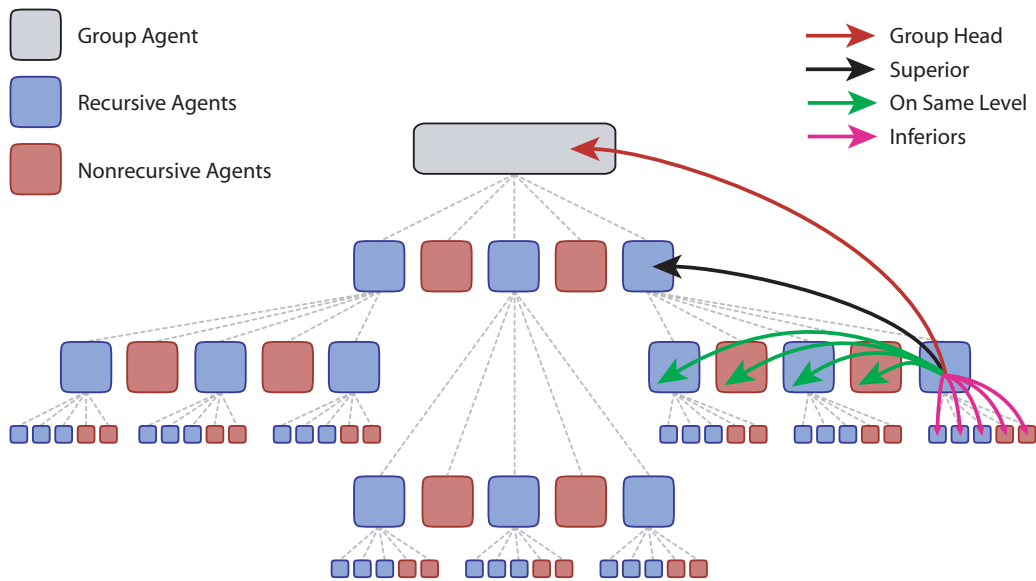


FIGURE 5.1 References to other agents within a recursively defined group of agents.

Obviously, the whole group moves from approximately the same location to the same destination, therefore, the resulting paths would be almost the same in most cases. If each agent is glued to its superior agent, only the group head has to plan a path and follow it. Then, each agent would consecutively follow its superior agent resulting in the whole group following the group agent.

When using this approach some problems arise. First, we have to find a way to glue agents together and, second, we have to deal with exceptions that can occur in an open environment.

To enable such group behavior, we make use of the attribute which stores a reference to the leading instance, i.e. the superior agent. By using this reference, we can easily access the knowledge of this instance and make the decisions dependant on this information. When having access to the position of the superior agent, it is easy to define a *leader-follower situation* that keeps an agent near its leader. This situation is implemented in a generic way such that it is easy to add this situation also to agents that do not belong to a hierarchical group. The leader-follower situation has four main attributes and two possible actions which are shown in Figure 5.2.

The *leader* is a reference to another agent, in our case the superior agent in the hierarchical organization. Δ denotes a small vector which is added to the leader's position such that different agents do not radially move directly towards the agent but towards its neighborhood. Finally, there are two important distance measurements: First, the *allowed distance* defines a circle (grey area) around the leader in which nothing happens at all. Second, when the agent is within the *path distance* but farther away than the allowed distance (red area), the agent walks directly towards the position of the leader including Δ . Because of the reference to the leader, the agent will automatically adapt to the leader's movement – if necessary.

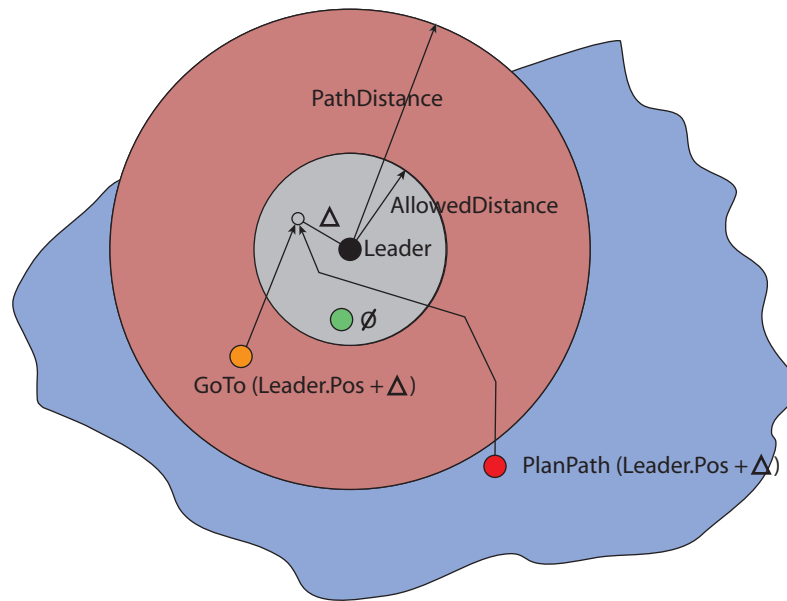


FIGURE 5.2 The leader-follower situation that is used for hierarchical path-planning.

However, it is still possible that the agent loses contact to its leader due to an obstacle. Unfortunately, this situation is unavoidable since this situation does not include the environments information. If the agent is farther away than the path distance (blue area), it will plan a path to the same position. Although this path will not adapt to the leader's movements, it will bring the agent on the shortest path back to the leader avoiding any deadlock. If the leader has moved in the meantime, the agent again applies this situation in order to catch up again. If both the follower and the leader have the same velocity, it might take arbitrary long to catch up to the leader. Therefore, our solution does not only calculate the path to the leader's position but also makes the leader slow down a little.

This situation allows members of a group to follow their group instance or their superior agent. Thus, the whole group will keep automatically together by using a simple reactive behavior. And it drastically reduces path-planning costs within the group to a minimum since only the group agent has to regularly generate a path to the destination. The others automatically follow the leader and initiate a path-planning only when the leader is too far away.

5.3.2 Hierarchical Flocking

Reynolds flocking algorithm [Rey87] relies on the spatial neighbors of each individual. We extended this approach by including not the spatial neighbors but the hierarchically related agents from within the group. Since each member can adapt its behavior based on the properties of the known hierarchical related agents as depicted in Figure 5.1, it is possible to generate new forms of herds, flocks, and schools.

Reynolds algorithm relies on three basic behaviors: Collision avoidance, velocity matching, and flock centering. Each of these three behaviors generates a force which are all weighted and added resulting in a force that affects the boids move-

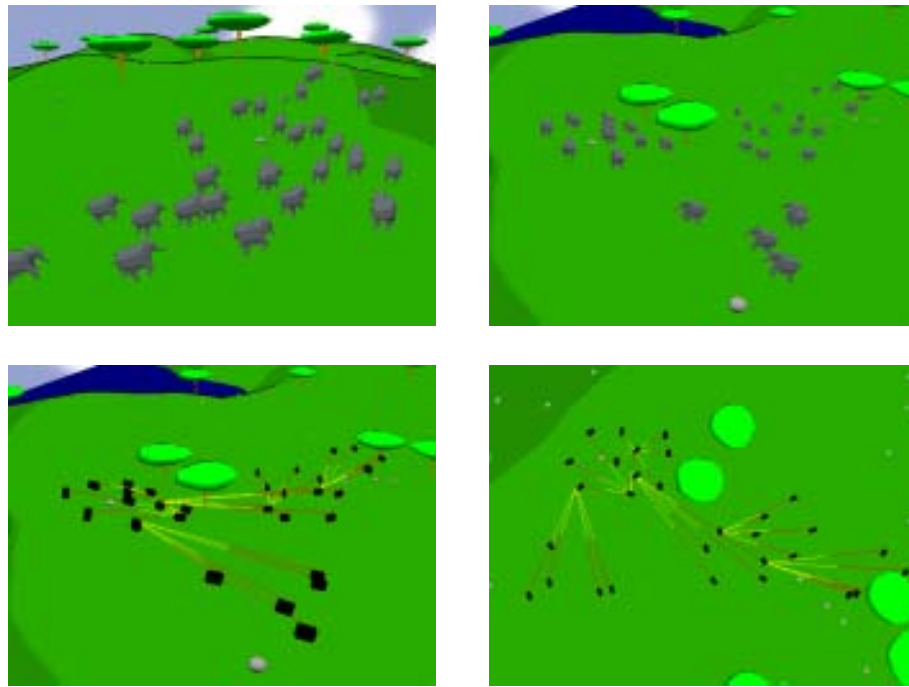


FIGURE 5.3 The hierarchical herding approach based on Reynold’s algorithm.
 Top left: A hierarchical herd in movement.
 Top right: The subgroups of each hierarchy-level separate from each other when changing the weights.
 Bottom left: A technical view of the scene in ii). The lines depict the hierarchical order where the green side points to the superior and the red side to the inferior agent.
 Bottom right: Top view of the same scene. Agents of the same hierarchical sub-group align and keep together while the inter-hierarchy relationship is less weighted.

ment. In order to calculate these forces, the local neighbors of each boid have to be determined and the average position, orientation, and velocity are computed. The determination of the local neighbors is a complex task [Rih04] and is closely related to the underlying data-structure representing the distribution of the agents. But the already available hierarchies within groups of agents can be used to substitute the dynamic neighbors by the hierarchical ones. Based on Figure 5.1, we generate the forces of Reynolds approach with respect to three groups of agents. First, the parent-force \mathbf{F}_p is related to the parent or superior agent. Second, the hierarchy-force \mathbf{F}_h is calculated with all agents on the same hierarchy level, and, third, the child-force \mathbf{F}_c with respect to the children of the current agent. Then, we can simply weight and add these forces and compute a resulting force which influences the desired orientation and velocity of each group member:

$$\mathbf{F}_{tot} = w_p \cdot \mathbf{F}_p + w_h \cdot \mathbf{F}_h + w_c \cdot \mathbf{F}_c. \quad (5.1)$$

When comparing this approach with the one of Reynolds, several varieties are expected to show up. First, the behavior of the individual characters will be slightly different since not the real neighbors are considered but only the hierarchical neighbors. However, this allows for different independent subgroups and can be leveled out by adjusting the weights as Figure 5.3i) shows. Second, a major advantage of this

approach is that the algorithm is independent on the number of agents simulated simultaneously. While the original approach relies on the varying number of local neighbors whose determination is time-consuming and can depend on the density of agents around the current one, this approach is independent on the distribution of agents and works without considering the local neighbors. Therefore, its runtime is almost constant as shown in Figure 5.4. Another advantage is the ability to build herds of families that keep together as a herd but form independent sub-herds of families as shown in Figure 5.3ii-iv). This can also be seen as a disadvantage because sub-herds do not align when crossing each other. However, compared to the original algorithm by Reynolds, the results of our approach look very similar but have reduced computational expenses. A deeper analysis of this approach as well as implementation issues are discussed in [Kno04].

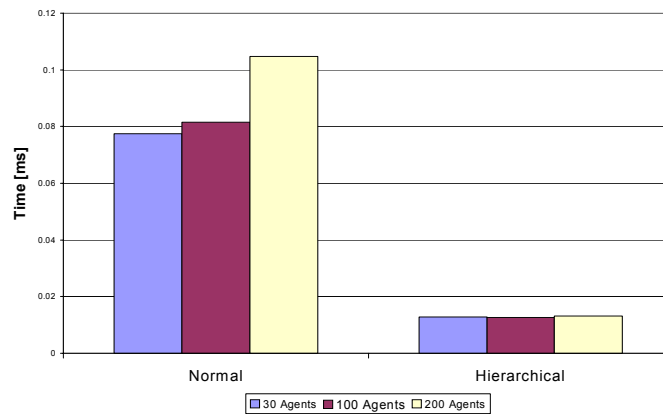


FIGURE 5.4

Analysis of the hierarchical herding algorithm.

The values depicted are the time needed to determine one agent's behavior. On the left, the values for the original approach of Reynolds are depicted when applied in our environment. On the right, the values are shown for our hierarchical approach. The large difference results from the neighborhood query which is necessary in Reynolds' approach. However, it is obvious that our approach does not depend on the number of agents while the traditional approach does.

5.4 LEVEL-OF-DETAIL

In 1999, Funge presented his work on cognitive modeling [FTT99] in which he extended the computer graphics modeling hierarchy by two layers as shown in Figure 5.5: The yellow behavioral and the cognitive layer in red. As discussed in the related work section of this chapter, LOD concepts on the lowest layers are and have been widely researched. Also, for the kinematic and physical layer exist some approaches that introduce multi-resolution techniques to speed up the simulation. On the behavioral and cognitive layer, there exist only few multi-resolution approaches. In this section, we present a multi-resolution approach for the behavioral and cognitive level which allows to use the same LOD over all layers of the modeling hierarchy. The underlying idea of the method is described first before going into the details.

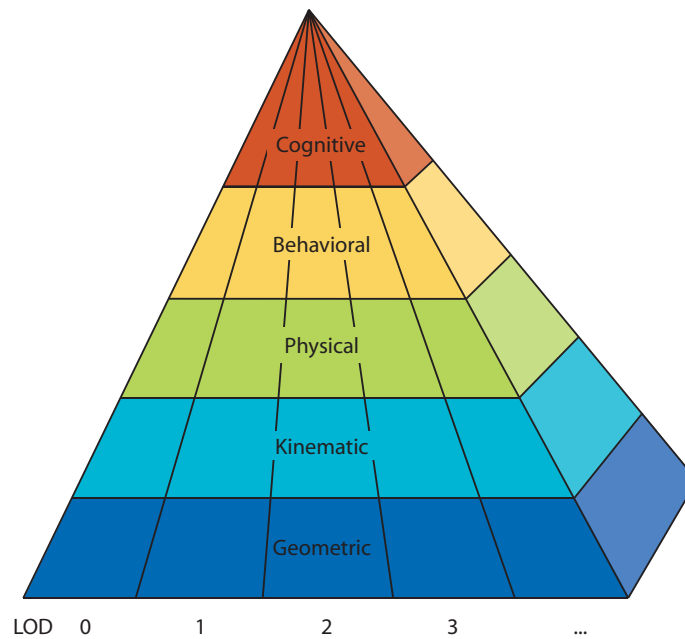


FIGURE 5.5 The extended computer graphics modeling hierarchy with level-of-detail. The hierarchy was first presented by Funge in [FTT99]. We extend it by introducing a level-of-detail on the top layer which can be used also on the bottom layers, too.

5.4.1 Idea

When looking at multi-resolution approaches on the lowest layer of the modeling hierarchy, there are in principle two methods to reduce the complexity. First, objects, that are not visible are culled and not rendered at all. Second, objects far away are reduced to a simple geometry that consists of a small percentage of the triangles of the original model. On the kinematic and physical layer, similar approaches exist: Invisible objects are simulated by using a simple and fast schemes while visible objects are simulated with more accuracy. As before, the simulation complexity can also be reduced to a simpler geometry for objects far away which allows for faster calculations. The main difference between geometry and kinematic/physical approaches is that invisible objects have to be simulated on the upper layers but can be neglected in the lowest layer.

Obviously, for the top two layers, it is also impossible to totally neglect invisible objects, i.e. agents. These agents do not stop living outside the visible area. If we would totally neglect invisible agents, these would stop moving and when returning to a previously visited location, the world would not have changed at all. In our environment, each agent is permanently active and acts as steadily as possible. However, invisible agents can be activated much less frequent than visible ones. Hence, the same approximations as on the kinematic/physical layer can be adapted to the top layers:

- ▶ Agents that are near the viewer are more important than objects far away.
- ▶ Agents that are invisible are even less important than objects far away but must not be neglected anyway.

But we have also to account for the possibility that an invisible agent can soon enter the visible area. It should not start acting more sophisticated right after entering the visible area but rather when approaching it in order to be “ready” when entering the scene. Thus, the simulation maintains the visual quality while reducing computational costs.

5.4.2 Setup

First, we have to determine the LOD for each agent. As stated in the last section, the LOD depends on the visibility and the distance to the camera. Therefore, our approach segments the environment into three disjoint areas as shown in Figure 5.6:

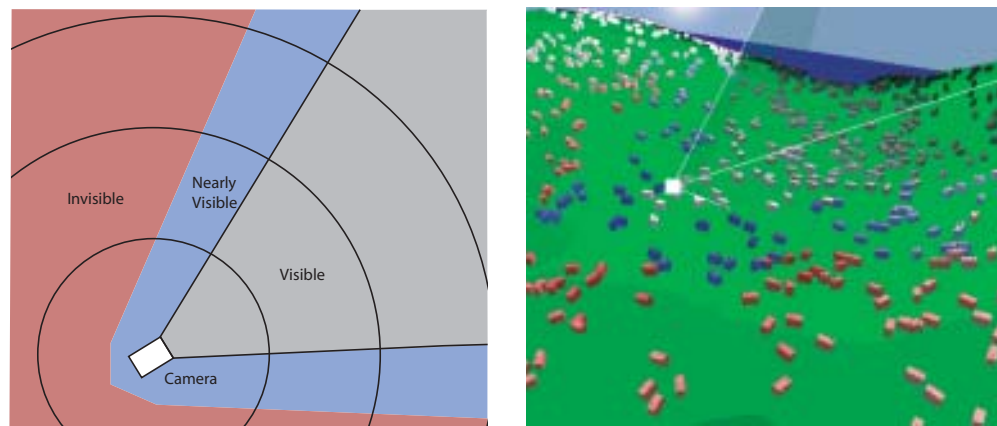


FIGURE 5.6

The setup scheme for the level-of-detail.

The whole environment is segmented into three main areas: The visible (gray), nearly visible (blue) and invisible area (red). Furthermore, these areas are further segmented based on the distance to the viewer. On the right, a screenshot of the application is shown with the same color-scheme. The camera (white) is shown with its view frustum.

- ▶ *Visible area* (gray): This is the part of the environment which is currently rendered.
- ▶ *Nearly visible area* (blue): This is the area adjacent to the visible area.
- ▶ *Invisible area* (red): The rest of the environment.

Furthermore, each area is additionally segmented based on the distance to the viewer. Since the primary segmentation of the environment is done anyway during rendering this classification does not introduce any additional cost to the simulation. The objects are usually managed in a quadtree data structure. Before rendering, the quadtree is traversed to determine the visibility for the view frustum culling of each cell as follows. First, all agents are marked as invisible. In a first iteration, all quadtree cells are determined of which at least one edge is inside the view frustum. During the traversal, the agents in these cells are all marked as nearly visible. Additionally, each object and agent within these cells is checked individually which reduces the set of potentially visible to really visible ones.

According to Figure 5.6, each of these zones is further divided into several sub-zones depending on the distance to the camera. In each zone, all objects get a value assigned where lower values specify more important areas. We start by setting the values for the visible agents according to

$$lod_{vis}(d) = \left\lfloor d \cdot \frac{lod_{max}}{d_{max}} \right\rfloor, \quad (5.2)$$

where lod_{max} denotes the maximal number of levels, d_{max} the maximal distance, and d the distance to the camera. Then, the values for the nearly visible agents are set to

$$lod_{nvis}(d) = \max(lod_{vis}(d) + c_{nvis}, lod_{max}), \quad (5.3)$$

where c_{nvis} is a positive and constant value. Therefore, nearly visible agents are a little less important than visible ones. Invisible agents get a LOD according to

$$lod_{invis}(d) = \max(lod_{nvis}(d) + c_{invis}, lod_{max}), \quad (5.4)$$

where c_{invis} has the same meaning like c_{nvis} . Due to the clamping, all values remain in $[0, lod_{max}]$. Thus, after processing all the agents each one has a LOD assigned with lower values denoting more and higher values less importance. One could imagine to additionally multiply the values of the preceding step with a scaling factor. This would stretch the scale radially and further reduce the importance of agents not in sight. We have tried this but have not noticed any visual improvement.

Since the LOD depends only on the camera's position and orientation and the agent's position, the setup should be performed at least when the camera's position or orientation has changed much. Since the agent's position can change, too, the setup has to be applied regularly. This depends on the maximal velocity of the agents. If the scene consists of fast moving agents, the LOD setup should be activated more often.

Thus, we have created integer values that characterize each agent's importance. But only having this value has no immediate effect on the behavior. Now, we have to find a way how to activate the agents according to their importance and how to distribute the available time in a similar way.

5.5 AGENT SCHEDULING

After the importance of each agent has been set using the LOD scheme presented in the previous section, the need for a scheduling algorithm arises. This algorithm should select the agents which need to be activated and it should determine the amount of time each agent gets.

Scheduling is primary known from process scheduling within an operation system. We will therefore first describe some of the common approaches for process scheduling and discuss their applicability in our environment. Afterwards, we will discuss the differences between process scheduling and scheduling in our environ-

ment and introduce an agent scheduling algorithm that meets our needs including some results.

5.5.1 Process Scheduling

If an operating system supports more than one concurrent process on a system¹ it needs to decide which process should get activated next. This task is usually done by a so called *scheduler*.

There are two different kind of scheduling algorithms:

- ▶ *Preemptive scheduling*. The process runs as long as the scheduler allows it. The scheduler determines a time quantum after which the active process is replaced by another. Here, the scheduler has full control over the assignment of processes to the CPU.
- ▶ *Non-preemptive scheduling*. The process runs as long as it needs. Another process is activated either when the active process decides to hand over the CPU to the next one or when it terminates.

In order to compare different scheduling algorithms, the following set of criteria is used:

- ▶ *Response time*. This is the time between the process wants to use the CPU until it receives it.
- ▶ *Throughput*. The number of processes which terminate during one time unit. This presumes that the processes terminate sometimes.
- ▶ *Turnaround time*. This describes the duration of a terminating process between generation of the process until termination.
- ▶ *Efficiency*. The percentage of CPU time which is actually used by processes. This criterion has become less important as a result of the increasing performance of current hardware.
- ▶ *Wait time*. The total amount of time during a process likes to use the CPU but is not allowed to do so.

Obviously, some of these criteria are conflicting. For example, a scheduling algorithm with a short response time will have a low efficiency because it has to switch processes rather often which takes some processing itself.

There are also other criteria which are not as good quantifiable as the above mentioned. Nevertheless, they are not less important to scheduling:

- ▶ *Fairness*. A scheduling algorithm is considered fair if it distributes the total CPU time according to each process' priority in a nondiscriminatory fashion. This implies that even low-priority processes get a certain amount of time, but high-priority processes get more according to their priority.
- ▶ *Starvation*. This is the situation when one process gets excluded from using the CPU. This is usually a problem when a low-priority process is locked out by other high-priority processes. A fair algorithm eliminates the occurrence of starvation.

1. Assuming that it is a system with only one CPU.

We will now present some of the most common process scheduling algorithms and compare them using the criteria introduced above. These algorithms are first-come first-serve, round robin, shortest job first, and priority queue.

First-come first-serve (FCFS). This is the most simple scheme possible. The process requesting the CPU first will be activated first while later processes have to wait. As according data-structure the FIFO queue is used commonly. FCFS is a non-preemptive method. Therefore, this algorithm is not applicable to multi-user systems since one user could actually block other users from executing processes. But FCFS has the advantages to be very easy to implement, to have a small overhead and to be very fair. The main disadvantage has been mentioned already: It is possible that one process can block all other during a long period. For example, all I/O processes can be blocked by a computational intensive process. In average, the FCFS has a rather low waiting time.

This algorithm is not very useful in our environment, since all processes or agents are active anyway. The criterion which determines the order of the execution is therefore invalid and not usable.

Round Robin. This is one of the oldest and most common scheduling algorithms [Tan92]. It is also one of the fairest ones. Basically, all processes are queued in a cyclic list which are activated in this order. Usually, a time quantum determines the run-time of each process. Therefore, round robin is preemptive and expects the processes to be interruptible. The value of such a time quantum, however, is not straight-forward because there is a trade-off between a large overhead due to process management and short response times. A short time quantum implies a low efficiency while a long time quantum increases the waiting time drastically. As the FCFS method, round robin is also easy to implement. Additionally, it needs only one attribute, the time quantum. The major disadvantage is that a process' priority is not respected because of the rather simple data-structure.

This algorithm treats every process the same way which is fair on the one hand but not desirable on the other hand. In a environment with a very large number of agents, even the invisible agents would get the same time as the visible ones. Therefore, only a small fraction of the agents are activated during one cycle which results in a low frequency and therefore in erroneous behavior. Although it reduces the cost of overhead to a minimum which results in a maximal efficiency, we are looking rather for an adaptive method that takes into account the level-of-detail.

Shortest job first. This method tries to minimize the average waiting time for all processes. In order to do so, the process is chosen for execution which will return the CPU as soon as possible. The duration while a process can use the CPU without interruption is called CPU burst. Therefore, a preemptive shortest job first algorithm will chose the process with the shortest next CPU burst. In case of a non-preemptive algorithm the process with the shortest run-time will be chosen instead. In this case, the processes are called *jobs*. Usually, jobs are not interactive and have a predictable run-time. For interactive processes, the overall run-time is normally not predictable as is the next CPU burst. Therefore, the algorithm usually relies on the statistic of the previous runs of each process in order to determine the possible expectable CPU burst. This method can reduce the average waiting time of each process to a minimum but only if the prediction of the expected CPU burst is as precise as possible.

This strategy uses the expected run-time as criterion. But as stated above, this task is rather difficult and in our environment even more since the time to compute proactive behavior can take up to infinite time. Furthermore, the average waiting time which is minimized with this method is not the most important variable in our environment. For example, invisible agents can be simulated with a rather low frequency without a great negative influence on the visible behavior.

Priority queue. This method has a large number of variations, but all use the same underlying principle. The basic priority queue is based on the idea that there are usually some processes with a higher importance than others or that there are some processes which have a critical response time. For example, in a text processor, the screen and cursor have to get updated regularly and as often as possible. But the grammatical online check which runs in the background is of secondary importance and does not need a small response time. Therefore, the priority queue algorithm assigns a *priority* to each process. These processes are then inserted into queues based on their priority as shown in Figure 5.7.

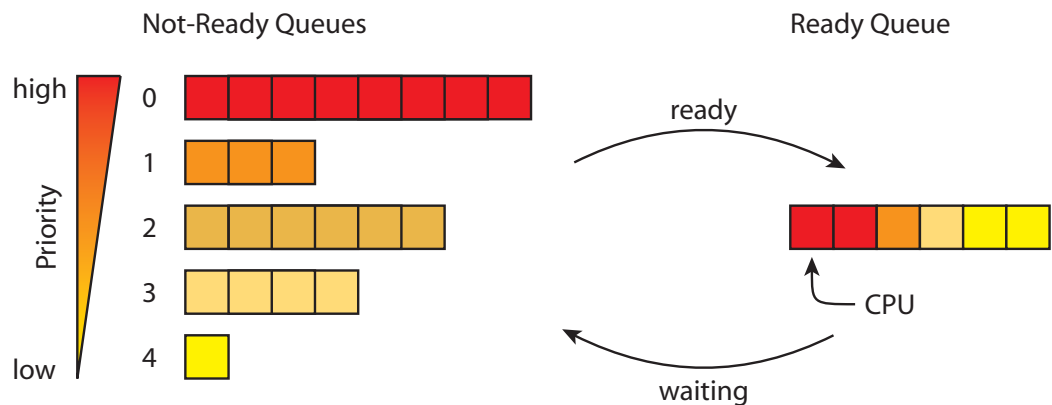


FIGURE 5.7 Schematic view of the priority queue algorithm.

In the Not-Ready Queues are jobs waiting for resources or input. They are placed in an ordered fashion in the Ready Queue, when needing the CPU. Then, the CPU traverses the Ready Queue and executes the jobs.

Each queue maintains a list of processes which are currently not requesting the CPU. If a particular process needs to be activated it is placed into the *ready queue* which is ordered by decreasing priority. Then, the CPU is assigned to the process in the ready queue with the highest priority, i.e. the front-most one. In case of a non-preemptive version, the algorithm will select the process with the next lower priority after the current has released the CPU. If a process is waiting for a resource it will be deleted immediately from the ready queue and placed back into its priority queue before the scheduler executes the process with the highest priority.

As major drawback, this algorithm can generate starvation: A few high-priority processes can detain many low-priority processes. This situation can be resolved by using the *aging principle*: Each handled process' priority is reduced after execution while the waiting process' priority is increased regularly. Therefore, all the low-priority processes will sometime get a priority which allows for execution. Of course,

the aging has to be applied such that the original priority is considered in order to allow high-priority processes get proportionally more time than others. Another disadvantage of this method is the rather large management overhead with several different queues and possibly changing priorities.

The major advantage of the priority queue scheduling algorithm is the possibility to classify processes by their importance or priority. Other implementations actually allow for a dynamically changing priority.

The priority queue scheme seems to be the most promising method presented here. It uses priorities as criterion to decide about the next process to activate which has a strong relationship to the level-of-detail in our environment. Agents near the camera have a low LOD value while distant or invisible ones have a high LOD value which is inversely proportional to the priority they should have. But the priority queue has also a disadvantage: Because the LOD value is updated regularly, the priority queue algorithm has to update its internal data-structures immediately after a LOD refresh. Ideally, this renewal should occur permanently but as stated in Section 5.4, the LOD values do not change that often and a full refresh is necessary only with a rather low frequency of about 1 Hz.

5.5.2 Agent-Scheduling

These scheduling algorithms with their different characteristics are not equally useful in our environment for different reasons. First, process scheduling is not exactly the same as agent scheduling. Second, the usability in our simulation environment restricts the choice for a scheduling algorithm further more as discussed in the last section. Third, opposite to processes, our agents have two completely different phases: the short reactive and the arbitrary long proactive phase. Before choosing a scheduling algorithm, we have to deal with these restrictions.

Obviously, our environment is designed to use a kind mixture between preemptive and non-preemptive scheduling. The system should decide about how much time the agent gets but the agent has to decide itself when to release. Of course, the difference between the time an agent gets and the time it actually uses should be as small as possible. This seems to not restrict the choice very much.

The fact that the proactive agents have two different phases is much more important. While some agents might have only the reactive system activated, some can be proactive. Thus, some agents have always the same run-time while others can spend as much time as available. And, the reactive phase is not interruptible at all. Either it is activated and fully executed or it should no be activated at all. All steps from sensing over situation-awareness to the action selection mechanism are necessary to provide a reaction at all. If this process would be interrupted in between, the agent could not use the intermediate information during the next run because it would be outdated. And therefore, the afforded work would be useless.

The reactive and proactive phase bare two completely different requirements to the scheduling algorithm. The reactive phase must be activated as often as possible, thus, has a high frequency. So, the visual quality for the viewer can be maintained. If the reactive behavior is neglected and the frequency drops, the resulting behavior would be erroneous with agents crossing water or moving over the borders of the

environment. Therefore, the most important criterion for the reactive phase is the response time.

The proactive phase, however, does not require such a high frequency, because an existing partial solution remains executable for a few seconds at least. Hence, a renewal or extension is not absolutely necessary. But on the other hand, the time available for planning should be reasonable high in order to allow for a feasible solution. As we have seen in the last chapter, the quality of a solution found by the planning mechanism is proportional to the time spent for searching, especially when using anytime planning algorithms as presented in Section 4.4.2.

TABLE 5.1 Comparison of the reactive phase and the proactive mode.

Mode	Frequency	Time needs	Run-Time Deviation	Necessary?
Reactive	High	Very few	Low	Yes
Proactive	Depending on the goal and environment	As much as possible	High	No

The requirements of both modes are, as shown in Table 5.1, a short response time and a long run-time, which are in conflict to each other. Improving one side will reduce the quality on the other side. Therefore, the best solution is a compromise between both these requirements. As stated in Table 5.1, the average deviation of the run-time for the reactive behavior is very low. For all agents in our environment is expected to be approximately the same. Only interruptions of the operating system can increase the run-time noticeable.

Therefore, we look for a scheduling solution that incorporates both the requirements of the reactive and the proactive mode. The general priority scheduling algorithm presented in Section 5.5.1 is not usable in this form.

5.5.3 Time Accounts

Because the simulation environment is executed in a single thread, the control over the execution is passed exclusively to the agents. Each agent has to take care not to pass over the limit given by the scheduler. If the simulated behavior is only reactive, this limit is not important, since the reactive simulation should not be interrupted anyway. In the case of proactive behavior, this fact is important to remember. Since the execution of the anytime planning algorithms has some granularity itself, it is expected that a proactive agent will use slightly more time than allowed to.

As we have seen, anytime algorithms have the property to increase the quality of the result with increasing time available to run. And, it might be better to plan a rather long time in one piece than planning more frequently but with shorter duration due to memory caching strategies. This leads to the idea that each agent should be able to decide itself whether to plan frequently with short duration or less frequently but for a longer period by accumulating the time available for planning. We believe that this idea is very useful because the efficiency of the planning mechanism

depends mostly on the properties of the goal. This could depend on some dynamic objects and its according plan should therefore be updated as often as possible. Or it could be a more static problem with a large branching factor and should obviously get updated less frequently but needs more time to create a useful solution.

In order to allow an agent to accumulate its time, we introduce a *time account* for each agent which is handled by the scheduler. After initialization, each account is set to zero. If an agent is activated and provided with a certain amount of time but does not use all of it, the remaining difference will be added to the account. If the agent is activated the next time, it receives the time for this slot – approximately the same as in the first run – plus the account’s value which is readjusted again after the agent has finished. If the agent uses more time than provided then the difference will be subtracted from the account. With that, the agent’s account can have a negative value, too. Agents with a negative account will not be allowed to act proactively and can therefore only activate their reactive behavior. Thus, it will need a rather short time to complete its task and can possibly refill its account over the next few activations.

If the scheduler provides each agent at least a little more time than the reactive behavior needs, then the account would be refilled step by step until the next proactive phase. Such a realistic minimal time per agent cannot be predefined per se because it is strongly depending on the capacity of the used hardware and has to be determined online. Therefore, the scheduling algorithm will not allow any agent to act proactively during the first few cycles over all agents. During this phase, the scheduler allows each agent to run only its reactive behavior routines and measures the time needed for each agent. With this information, the scheduler calculates the average over all reactive phases and increases this value by ten percent. This value is of course agent-dependent and, thus, calculated independently for each agent.

In order to prevent agents from accumulating too much time, the account is limited with a lower and upper bound. With that, an agent cannot spend more than, for example, a millisecond and prevents the scheduler from starvation. Obviously, the time accounts have the tendency to get larger when the agent is not planning. Thus, all purely reactive agents will have a positive time account which has no influence on the mechanism. Agents that do not plan will accumulate, too, and if they decide to become proactive again, their time-account will allow to generate a good initial plan immediately.

5.5.4 Priority Queue Agent Scheduler Algorithm

This section will describe the scheduling algorithm whose properties have been acquired during the last sections. Our algorithm is based on the priority queue scheduling algorithm and uses time accounts for the agents.

For each registered agent, the scheduler maintains a set of different information such as a reference to the agent, the agent’s actual priority or level-of-detail, and the time account of that agent. After having registered all agents in the scheduler, they are placed into their according priority queue. Of course, after each recalculation of the LOD values, the scheduler has to be set up again.

For each run, the scheduler receives the overall available time for the agent engine. This time should be split up into time quanta according to the agents level-

of-detail. First, the algorithm determines a quantum for each priority queue depending on the priority of the queue and the number of agents contained:

$$T_i = T \cdot q_i, \quad (5.5)$$

where T is the total time available for this simulation step. q_i is the quantum of each queue determined by

$$q_i = \frac{N-i}{\bar{n}}, \quad (5.6)$$

where N denotes the number of priority queues, i.e. the upper bound of the LOD. Here, we have to remember that agents with a low LOD have high priority and vice versa. Therefore, the algorithm subtracts the actual priority from the number of queues. \bar{n} is the weighted sum of all agents over all queues according to

$$\bar{n} = \sum_{i=1}^N (N-i) \cdot s_i, \quad (5.7)$$

with s_i being the size of the i -th priority queue. The weighted sum is necessary to make the time quantum depending on both the size of the queue and its priority.

The determined time quanta for each queue q_i will be a value in $[0,1]$ which sum up to 1. Therefore, the sum of all T_i will be T . After having determined the time available for each queue and agent the scheduler starts activating the agents by starting at the queue with the highest priority.

Assuming an agent has an average reactive run-time of t_{react} (including the ten percent added), we can determine the amount \bar{t}_j needed for each agent:

$$\bar{t}_j = \begin{cases} t_{react} & \text{if } \frac{T_i}{s_i} < t_{react} \\ \frac{T_i}{s_i} & \text{else} \end{cases}. \quad (5.8)$$

Equation 5.8 allows each agent to run at least its reactive behavior without decreasing its time account. If the agent has a high priority it will receive more time than t_{react} and is probably capable to act proactively.

In Section 5.5.3, we introduced time accounts for each agent to accumulate time not needed over time. Using these, an agent gets as total amount of time t_j according to

$$t_j = \begin{cases} \bar{t}_j + t_{account(i)} & \text{if } t_{account(i)} > 0 \\ t_{react} & \text{else} \end{cases}. \quad (5.9)$$

Of course, each queue can run only for T_i but the sum of all t_j can be larger because of the minimal run-time t_{react} for each agent. Therefore, the algorithm is

using a round robin scheme inside the queue by remembering the last agent activated and starting at the next at the ensuing run. Additionally, we apply the round robin scheme also to the selection of the current queue such that all queues are activated on a regular basis. The resulting algorithm is depicted in Figure 5.8.

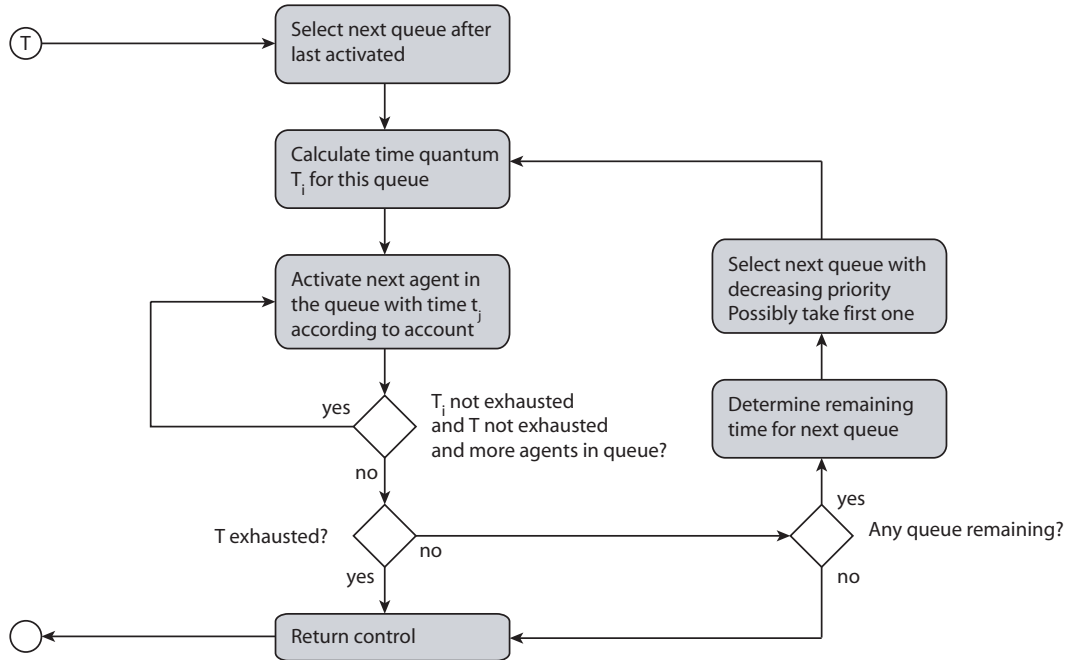


FIGURE 5.8 The priority queue agent scheduling algorithm.

This solution resolves most problems but not all. Consider the situation where the last queue contains only one agent but the total time available always exhausts before the algorithm reaches this queue. Then, this agent will not be activated at all. A possible solution to this problem is to decrease each q_i by a few percent resulting in a sum smaller than 1. Then, the scheduler would have some buffer that is used to run queues that would not have been activated else. Of course, in the ideal case this is a waste of time. But also the operating system can have an influence on the behavior of the scheduling algorithm. The process can be interrupted at any time which extends the time for the actually executed agent. But the scheduler does not know about the interruption and assumes that the agent has used a lot more time than it had available. The according time account is then decreased resulting in a reactive behavior over the next few runs of this particular agent.

But also the opposite situation is possible. If most of the agents with a positive time-account decide not to plan during this run but to accumulate the time they will not use their time t_j . Then the according list is traversed faster than expected with some unused time left. Ideally, this time should be provided to the other queues to have more time available for lower priority agents where possibly not all agents can be activated. Of course, this only happens if all the agents in the queue have been activated during this run. Then, our algorithm determines the time remaining with respect to T_i . Then, it is provided to the next queue and added to

T_{i+1} . So, the next queue gets more time than expected because the queue with the higher priority has not used its whole time quantum.

Two special cases have to be considered to guarantee fairness and prevent from starvation. First, it is possible that all agents are placed into the same priority queue. Second, all agents are equally distributed over all queues. The first case is not unrealistic at all: If the camera is high above the landscape looking down, all agents are visible and have approximately the same distance to the camera. Then, the whole time available is assigned to this particular queue. Within this queue, the round robin scheduling algorithm is used resulting in a round robin scheduler with time accounts. Therefore, all agents have an equal priority and the round robin scheme guarantees fairness and no starvation occurs. The second case is the ideal case and the opposite of the first scenario. Therefore, all expected cases will be somewhere in between the first and the second case. Since we use a round robin scheme inside the queue and over the queues, too, we expect that the over-all scheduling will be fair, too.

5.5.5 Results

In general, the total amount of time available for the simulation is independent from the used scheduler – a scheduler can at most distribute this time reasonable to the agents. Additionally, the administrative effort of a complicated scheduler will decrease the efficiency of the whole system. Therefore, we expect that the overall performance of the priority agent scheduling algorithm will not exceed a round robin scheduler's performance. However, the time available should be distributed in a way such that the visual impression is improved, although less time is available from an objective point of view.

First, we compare the average activation frequency of the agents of the round robin scheduler compared to our approach. The results are visualized in Figure 5.9. The activation frequency is the inverse of the response time which is especially important for the reactive behavior. As can be seen, agents in the foreground which are in one of the first priority queues have a high activation frequency and therefore a low response time. Agents in the background which are in a queue with a lower priority have, as expected, a lower activation frequency. The time available using the round robin scheduler is the same and results in an equal frequency for all agents denoted by the horizontal line. As intended, the priority queue algorithm assigns proportionally more to agents that are near the camera than for agents that are invisible or far away.

An agent moving from a low-priority to a high-priority queue will receive a higher activation frequency. The priority queue algorithm allows for smooth transitions over all queues such that a change of queue will not affect the visual impression by changing the behavior drastically.

The time needed for re-assigning the agents to their queues and calculating the time quanta is given in Figure 5.10

In order to validate the time accounts of our scheduling approach, we set up a scenario that allows the agents to act proactively. Due to a understandable result, this scenario has a reduced number of only four priority queues. The scenario consists of twelve proactive agents from which always three were assigned to one prior-

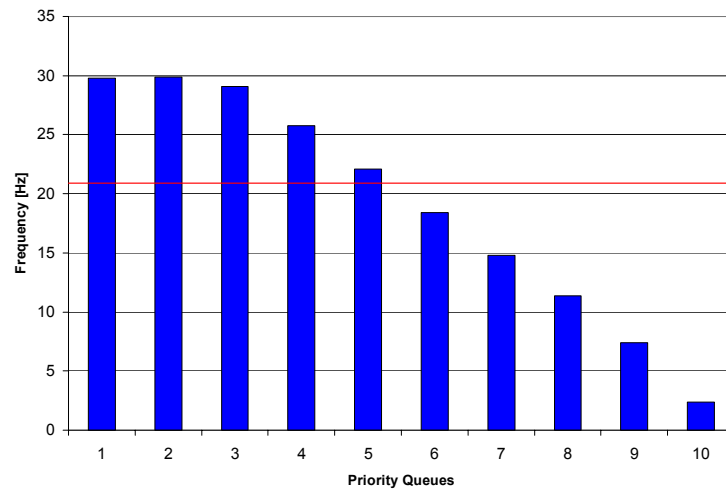


FIGURE 5.9 A comparison of the average activation frequency. The round robin scheduler’s frequency (red) is equal for all agents, while the priority queue agent scheduler’s frequency (blue) depends on the priority.

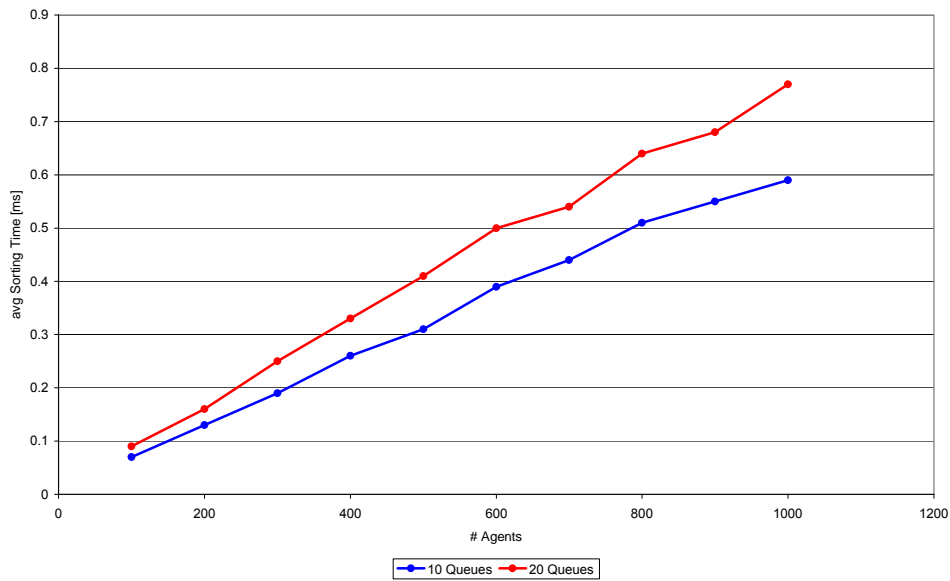


FIGURE 5.10 The time needed to resort the queues and calculating the time quanta. This depends on the number of agents and queues. The blue line denotes the case with 10 and the red one with 20 queues.

ity queue without the possibility to change that setting dynamically. The maximal time account in this setup has been set to five milliseconds. Each agent is allowed to pursue one goal until it is reached. Afterwards, the agent remains inactive. The average time account for each priority queue during execution is depicted in Figure 5.11. Note, that not all goals have the same temporal length to achieve the

solution. Therefore, the duration until completion of the planning task is individual and not directly depending on the agent's priority.

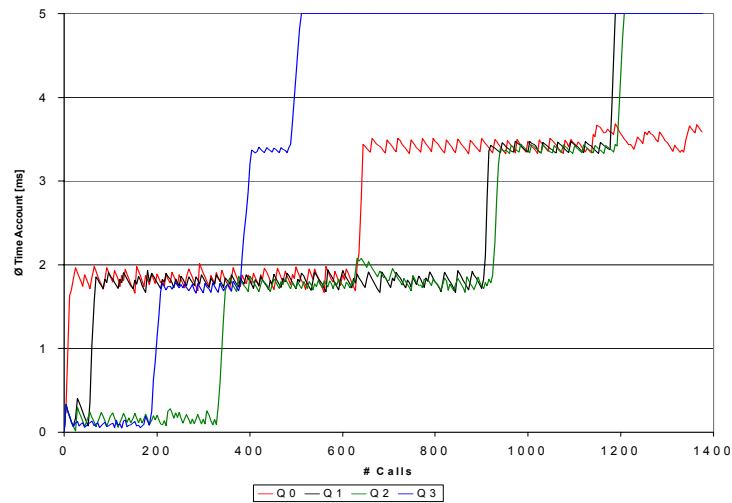


FIGURE 5.11 The average time account for each priority queue in a special setup.

As can be seen, the average time account of the agents first remains a little over zero which means that all three agents in this queue use the time available for planning a solution to their goal. The high-frequency oscillation is a result of the accumulation of time in the account because each agent is only allowed to plan if the time available for the agent is over 0.1 milliseconds. When looking for example at Q3, at call #195, the curve jumps to a value of approximately 1.76 ms. This is due to the fact that one of the three agents has reached its goal and remains inactive – resulting in a time account increasing to the maximum of five milliseconds. At call #400, the curve increases again because the second agent in this queue has reached its goal. Now, the average of all three agents remains at 3.35 ms which is approximately $(5 + 5 + 0.1)/3$. At call #488, the last agent reaches its goal and now all three agents are inactive and have a time account of five milliseconds.

5.6 HIERARCHICAL CONTROL

Another approach to speed up the simulation also uses the hierarchical organization within a group of agents. But opposite to the scenarios above which use the hierarchy to add some kind of special behavior to the agent, the following approach is not supervised by the agent itself but rather by the simulation system that activates the individual agents. The method presented here influences the type of behavior of each agent.

As discussed in the related work section, Musse et al. distinguish three types of behavior: guided, programmed, and autonomous [MKT99]. Our system already supports two of them: The programmed type can be compared to our reactive agent model since its behavior is deterministic. The autonomous type is basically the same as our proactive agents because the agent can infer on what to do. The guided agent

does not decide anything since all its actions are determined by an external entity. In our environment, the guided agent type is not foreseen yet but a similar approach can be applied.

We can imagine that within a group of proactive agents far away, the individual behavior of an agent gets less important for the observer than the behavior of the group as a whole. We use this fact as a starting point for a further reduction of complexity in our environment. The basic scenario consists of a group of proactive agents. If this group is near the camera, we would like all agents to act individually, possibly with some respect to the group. If the distance to the group increases, some of these agents could give their proactive control to another agent in the group and just stay near this agent. From the distance, this would not decrease the visual impression since both agents are not fully distinguishable. The more the whole group gets afar, the less individual behavior is necessary. The behavior of individual agents can be reduced to only reactive behavior and a few or even only one agent can guide the group proactively. The possibility of generating hierarchical groups within the presented system allows for an elegant way to achieve this goal.

This section explores this approach and presents first the algorithm which is used to pass the control to a superior agent and vice versa. Afterwards, results concerning this approach are presented.

5.6.1 Algorithm

Two different cases have to be considered: First, we have to determine the moment when an agent has to release the control to the superior agent and, second, the same procedure in the opposite direction.

An agent should pass the control over itself to a superior agent if the time available for planning gets too short and does not satisfy the requirements of the planning module anymore. Then, the additional time available for planning can be passed to the superior agent which is assumed to have few time available, too. Thus, the superior agent gets an increased time account and has more time available for planning which should be sufficient to generate appropriate plans. Of course, this agent could be able to release the control to its own superior agent, too.

This decision does not only depend on the time available for planning but also on the LOD value. In a heavily crowded environment, we expect that each agent has only very few additional time and this would immediately lead to a collapse to a few agents controlling others which only follow their controllers. But for agents near the camera, we would like to see all agents act independently even if they only provide reactive behavior. Therefore, the number of controlled agents depends on the actual LOD value and the additional time available.

The opposite direction is to release the control over other agents in two cases. First, the agent could control more agents than its LOD allows. Second, the agent can have so much additional time such that its controlled agents could possibly act proactively again. The first case can be resolved by releasing the control over some agents until falling below the limit of the current LOD. In order to allow for a fair controlling, the agents are released in a first-in-first-out order. The second case is handled by comparing the actual additional time available to a controlling agent to a certain threshold. If the threshold is exceeded, a fix number of agents will be

released such that they will use their additional time themselves in order to plan towards a goal. If the additional time of the controlling agent is still over the threshold during the next run, the same number of agents will be released again until the additional time falls below the threshold.

The scheduling algorithm presented in Section 5.5.4 has to be extended. After the calculation of the total time t_j for an agent, the scheduler checks regularly if some change of control or of time-accounts is necessary:

- ▶ If the current agent is acting individually, its time account is positive and smaller than a threshold, and if it is not controlling the maximal number of agents for its LOD already, then the agent gives the control over itself and all controlled agents to its superior agent and gets controlled, too.
- ▶ If the current agent is not being controlled, controls more agents than its LOD allows, and its time account is larger than a threshold or its LOD is too small, then the agent releases some of its controlled agents and remains acting individually.
- ▶ If the current agent is not acting individually, it passes its whole time account to the controlling agent and sets its own time account to zero.

The emerging effect of these rules is that groups far away are controlled by only one agent which plans for the whole group. Hence, the group members will just act reactive and follow the controlling agent's plans. When the group comes towards the camera, the control is first split up such that several subgroups are generated where each subgroup is controlled by one proactive agent. These subgroups will be split up again the lower their LOD gets. Just in front of the camera, no controlled agent is visible since the LOD does not allow to control other agents.

Figure 5.12 shows a situation where the hierarchical control is applied. Three large and hierarchical herds are placed at different distances from the camera. In these images, the agents have been replaced by color-coded boxes. Yellow boxes represent individually acting agents, blue boxes substitute agents that act autonomously but have control over others, and the red boxes depict agents controlled by another agent. The gray lines denote the controller of the red marked agents. Figure 5.12i) shows the rendering from the camera perspective. Obviously, most agents in front of the camera are yellow and, thus, act autonomously. Some appear in blue because they control some agents out of the view frustum of the camera, as Figure 5.12ii) reveals. The camera is depicted as a white spot with its view-frustum. In this image, one can see that in the distant herd, several agents are already controlled by others while some still act autonomously. Figure 5.12iii) shows a different scene from a similar viewpoint. Consistently, almost all agents in the frontmost herd act individually while the distant groups reduce their complexity by allowing the hierarchical control algorithm to gain influence. In Figure 5.12iv), the opposite direction is shown with the camera at the horizon. At this distance, the herd is controlled by only a few agents – almost all boxes are red and only a few are either blue or still yellow.

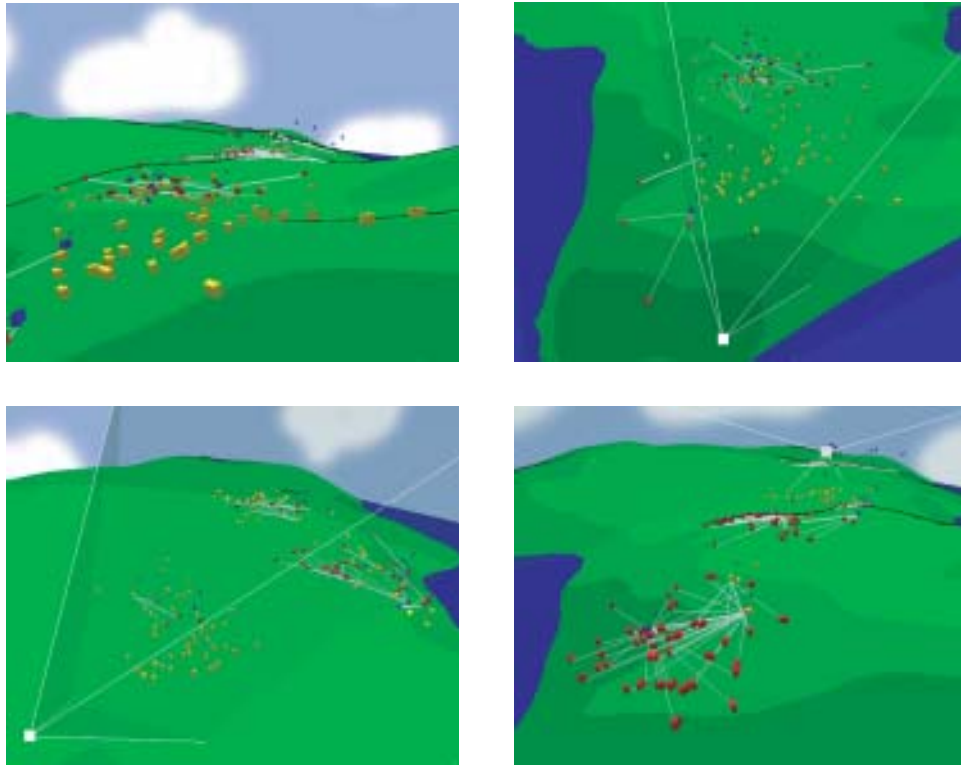


FIGURE 5.12 The hierarchical control algorithm.
 Top left: View from the camera perspective. The yellow boxes denote individually acting agents, the blue ones are controlling other agents, and the red ones are being controlled by others.
 Top right: The same scene from an overhead perspective. The white spot is the current camera with its view frustum. The agents near the camera are all autonomous while there are controlled ones in the back.
 Bottom left: Another scene from a similar viewpoint. In the distant herds more agents are being controlled than in nearby ones.
 Bottom right: A view from the opposite direction. At a distance that far from the camera, only few agents act individually while most agents are being controlled.

5.7 SUMMARY

Finally, we summarize this chapter by presenting an overview of the acquired system. The overall process of our solution is depicted in Figure 5.13.

The first step of our solution is the determination of the level-of-detail for each agent on a regular basis (a). This classification is done using the agent's distance to the camera and its visibility. Afterwards, each agent has an integer value assigned which corresponds to the priority of this agent. Next, the total amount of time for the actual simulation step is provided to the agent engine by the game engine (b).

This maximal duration is then split up in the scheduler (c). The agents are activated on a regular basis with a frequency corresponding to their priority. Each agent receives an amount of time which is proportional to its LOD value. This duration has a lower threshold such that purely reactive behavior is still possible. High-pri-

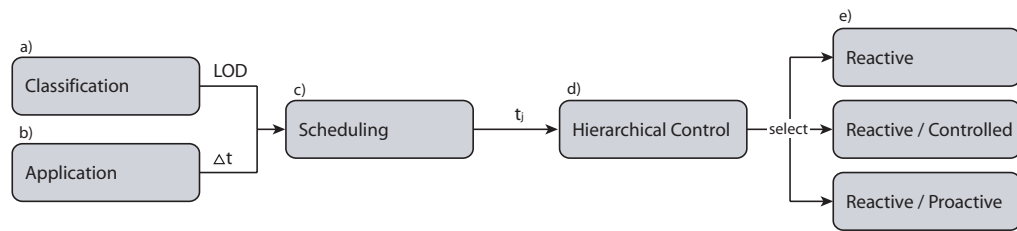


FIGURE 5.13 Overview of the solution.

The scheduler (c) determines upon the LOD classification (a) and the total available time (b) the time for each agent. Then, the hierarchical control mechanism (d) selects the appropriate behavior from three possibilities: Only reactive, controlled reactive, or proactive (e).

riority agents might get more time than the minimum which possibly allows the agent to plan into the future.

A group of agents can be used to further reduce the complexity of the approach. Based on the currently available time and the LOD of an agent, the hierarchical control unit decides whether the agent can act autonomously or needs to pass the control over it to a hierarchically superior agent (d). Distant agents which have only few time available for planning will then pass their additional time to the superior agent and follow it using only reactive behavior rules. In opposition, an agent controlling other inferior agents which gets near the camera or has much additional time available will release the inferiors which will start acting autonomous immediately. This process is more efficient on hierarchical groups than on flat groups and without consequences for purely individual agents.

Thus, the set of possible behavior types is threefold (e) as shown in Table 5.2. The most basic behavior is the purely reactive type which has been introduced in Chapter 3. It is rather individual as long as the situations and actions do not depend on another agent's bearing. Because the behavior is predefined with a finite set of simple rules it is very fast to evaluate. But this also restricts the possible attitude of the character to the designer's programmed knowledge. The second behavior is the controlled reactive type which only appears in groups. This is basically the same as the reactive behavior but the agent additionally keeps tight to its superior controlling agent which can act proactive. Therefore, the controlled reactive behavior is not as individual anymore as the pure reactive but since the controlling agent acts proactive, the resulting behavior of the whole group remains reasonable. The most individual behavior type is the proactive one which has been introduced in Chapter 4. The proactive behavior is fundamentally different from the previous two types. It allows the agent to follow a goal in the dynamic environment. Therefore, the presented behavior adapts to external changes immediately and also in advance by anticipating the future. The drawback of such a sophisticated behavior routine is the increased time consumption. Therefore, this behavior type is reduced to the controlled reactive behavior if time gets short and the agent has a low priority.

As Table 5.2 shows, these three behavior types offer a variety of different requirements and properties. The possible behavior ranges from low to high intelligence and autonomy. The presented algorithm allows for smooth transitions

TABLE 5.2 The three different types of presented behavior. The reactive behavior is the base of the controlled reactive behavior where the agent has to follow a superior agent. The proactive behavior includes also the reactive behavior but additionally allows for pursuing a goal.

Behavior Type	Reactive	Reactive / Controlled	Proactive
Level of Intelligence	Medium	Low	High
Level of Individuality	Fully individual	Partly individual, depending on a superior agent which is acting proactive	Fully individual
Level of Autonomy	Medium	Medium-Low	High
Dependencies	Only if defined in the behavior rules	The behavior of the controlling agent	Defined by the goal
Time	Very few	Very few	Takes as much as available
Frequency	High	High	Variable Depending on the goal and environment
Complexity	Variable	Medium	High

between these behavior types depending on the level of detail of each agent and the time available to the overall simulation. Thus, it is possible to generate a full simulation of all agents at low frame-rates or to maintain interactive frame-rates with reducing the level of sophistication of invisible or distant agents.

In order to compare the achieved frame-rates, we set up a scenario containing 1700 agents with purely reactive behavior. Each agent behavior simulation takes approximately 0.02 milliseconds, therefore, the total amount of time needed to simulate all agents would take approximately 34 milliseconds. Thus, with full simulation, the system could not achieve interactive frame-rates.

Table 5.3 shows the according results. The top three rows are cases where all or almost all agents can be simulated during on cycle. Therefore, the resulting behavior is perfect with respect to the applied reactive rules. For 5 to 20 milliseconds, the visible behavior still looks accurate, however, the behavior of invisible agents gets little erroneous. When only one millisecond is available, the activation frequency are too low and also the visible behavior gets erroneous, for example agents that walk into the water.

Table 5.4 shows the qualitative results of the second test. In this scenario, we compared the proactive behavior in different setups. A scene with about 500 distributed proactive agents is simulated. When setting the allowed time for behavior simulation to 5 milliseconds, we cannot see any proactive behavior when looking at the full scene. However, the more the number of visible agents is reduced, the more of them start acting proactively since they receive more time than before. When the fraction of visible agents is reduced even more to around five percent,

TABLE 5.3 Frame-rate achieved by restricting the total time available. With a total of 1700 reactive agents and each taking approximately 0.02 milliseconds to make its decision.

T [ms]	Frame-rate [fps]
50	15
40	17
30	19
20	25
10	34
5	43
1	50

TABLE 5.4 Comparing the resulting behavior in different setups. The frame-rate (FR) is independent of the number of proactive agents. v: visible, nv: nearly visible, iv: invisible

T [ms]	FR [fps]	(v/nv/iv) [%]	Observed Behavior
5	56	40/20/40	No proactive behavior. all reactive
5	56	10/20/70	Some agents in the foreground act proactively, but most agents remain reactive only
5	56	5/5/90	Most visible agents act proactively, when turning the camera, some agents already plan, others start to do so
10	43	40/20/40	When overlooking the scene, most agents in the foreground act proactively while agents in the middle ground and background remain reactive
10	43	10/20/70	All visible agents act proactively, even those entering the scene and when moving the camera
15	33	40/20/40	All agents in the forefront are proactive, most in the middle ground, too, while the ones in the background are reactive only.

also the nearly visible agents start planning which can be observed when turning the camera and the visible agents are already executing a plan. When increasing the available time to ten milliseconds allows for more visible agents with the same behavior but reduces the frame-rate accordingly. For example, when approximately half of the agents are visible, the ones in the foreground act proactively while the rest remains reactive. With only ten percent visible, all these do planning even those entering the view when turning the camera.

In a final exploration, we measured different variables over a certain time period. We kept the focus on four categories of information: The time spent for different tasks, the behavior type of the agents, their visibility, and the LOD distribution.

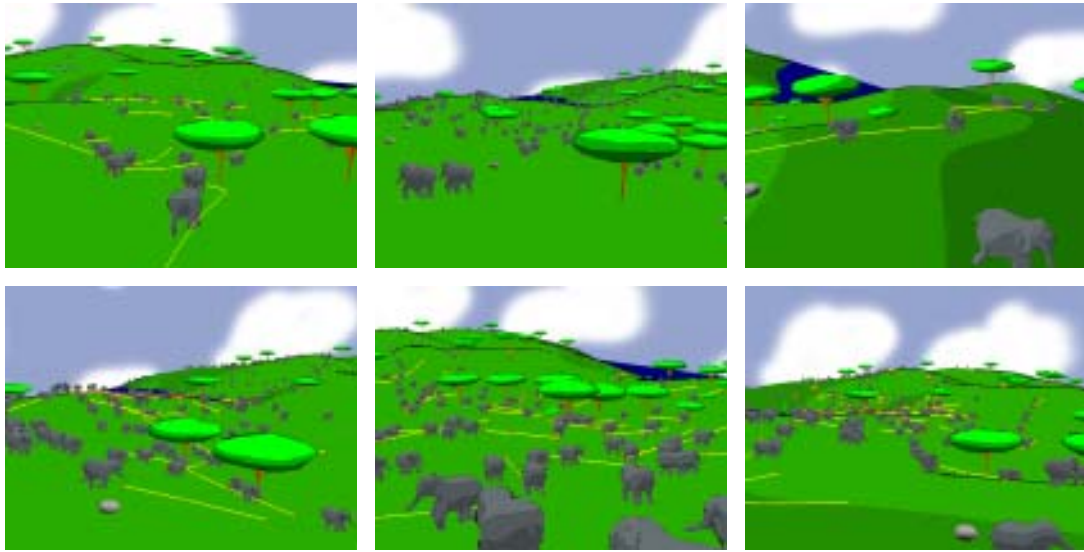


FIGURE 5.14 Screenshots of scenes similar to Table 5.4.
 Top left: 100 proactive agents at 5 ms.
 Top middle: 400 proactive agents at 5 ms, most agents visible
 Top right: Same as middle one but only a few are really visible
 Bottom left: 400 agents at 10 ms, most are visible
 Bottom middle: Same as left, but fewer agents are visible
 Bottom right: 400 agents at 15 ms

Corresponding charts for two different scenarios are shown in Figure 5.15 and Figure 5.16. Both scenarios consist of approximately 200 agents. The underlying scenario of Figure 5.15 has many proactive agents and some distributed reactive agents of which some are engaged in hierarchical groups. The scenario of Figure 5.16 has fewer proactive agents and many reactive agents of which about half are part of hierarchical groups and the other half is not.

In these figures, one chart is provided for each category of information. First, the percentage of time spent for different tasks is plotted where the following tasks are shown: LOD setup, scheduler setup, perception, situation recognition, reactive behavior, action system, motivation detection, and proactive behavior. As can be seen, the management overhead for the LOD setup and the scheduler is on average below 10% of the time spent for all tasks – and most of this time is spent with the LOD setup. In the first scenario with more proactive agents, about 30–50% is spent with planning.

The second chart is devoted to the behavior type which distinguishes between controlled, reactive only, and proactive. Obviously, most behavior is purely reactive which has been expected due to the fact that the reactive behavior is activated every time but the proactive behavior only with according time-accounts. The percentage of controlled agents is fairly low which is due to the small number of hierarchical agents. This could be enforced more with another setup in the control change algorithm.

The third chart shows the average visibility of agents over time which is partially linked to the last chart which shows the distribution over the LOD levels. Note,

that the agents can be invisible but, nevertheless, have a medium LOD value. On the opposite, distant visible agents can have a high LOD value. As can be seen for the first scenario, most of the agents are not visible with minor exceptions. Therefore, around 50% of the agents are placed in the LOD level 9 and get a minimum of time and activation. In the second scenario, we can see that most of the agents were visible towards the end of the measurement which was caused by the camera overlooking the whole scene. Thus, only few time can be spent on proactive behavior since many agents have to be activated regularly.

These results show that our approach works well with the presented scenarios and is able to distribute the time in such a way that the needs of the simulation are met. The overhead of the LOD approach is manageable and this mechanism provides good results for the visual impression. Depending on the simulated scenario, the system adapts directly and distributes the available resources in a considerate manner.

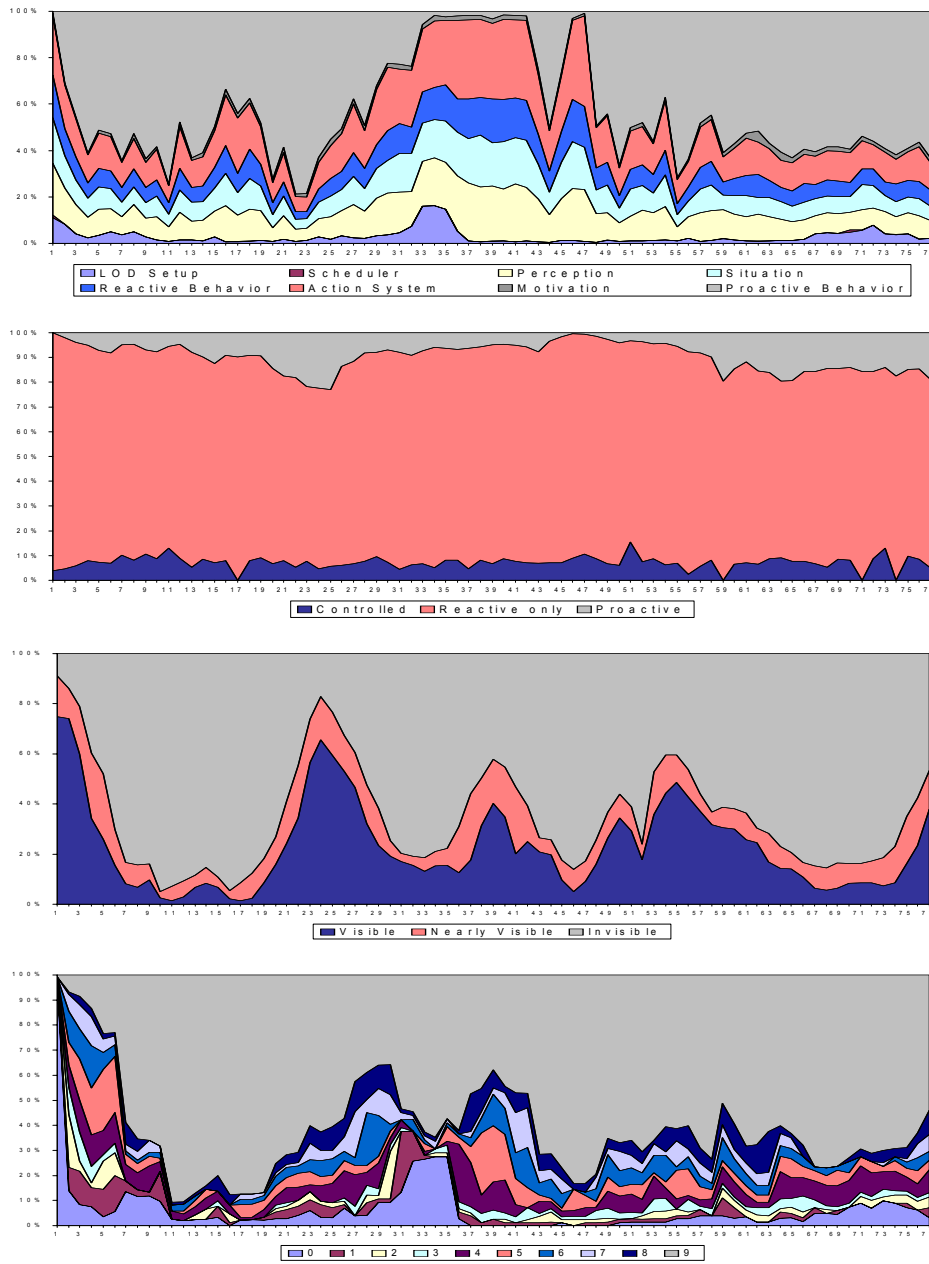


FIGURE 5.15 The behavior of the whole soution over time for a particular scenario. The first chart shows the percentage of time spent on different tasks. The second chart shows the behavior type of the agents. The third chart shows the visibility of the agents over time. The last chart depicts the LOD distribution of the agents.

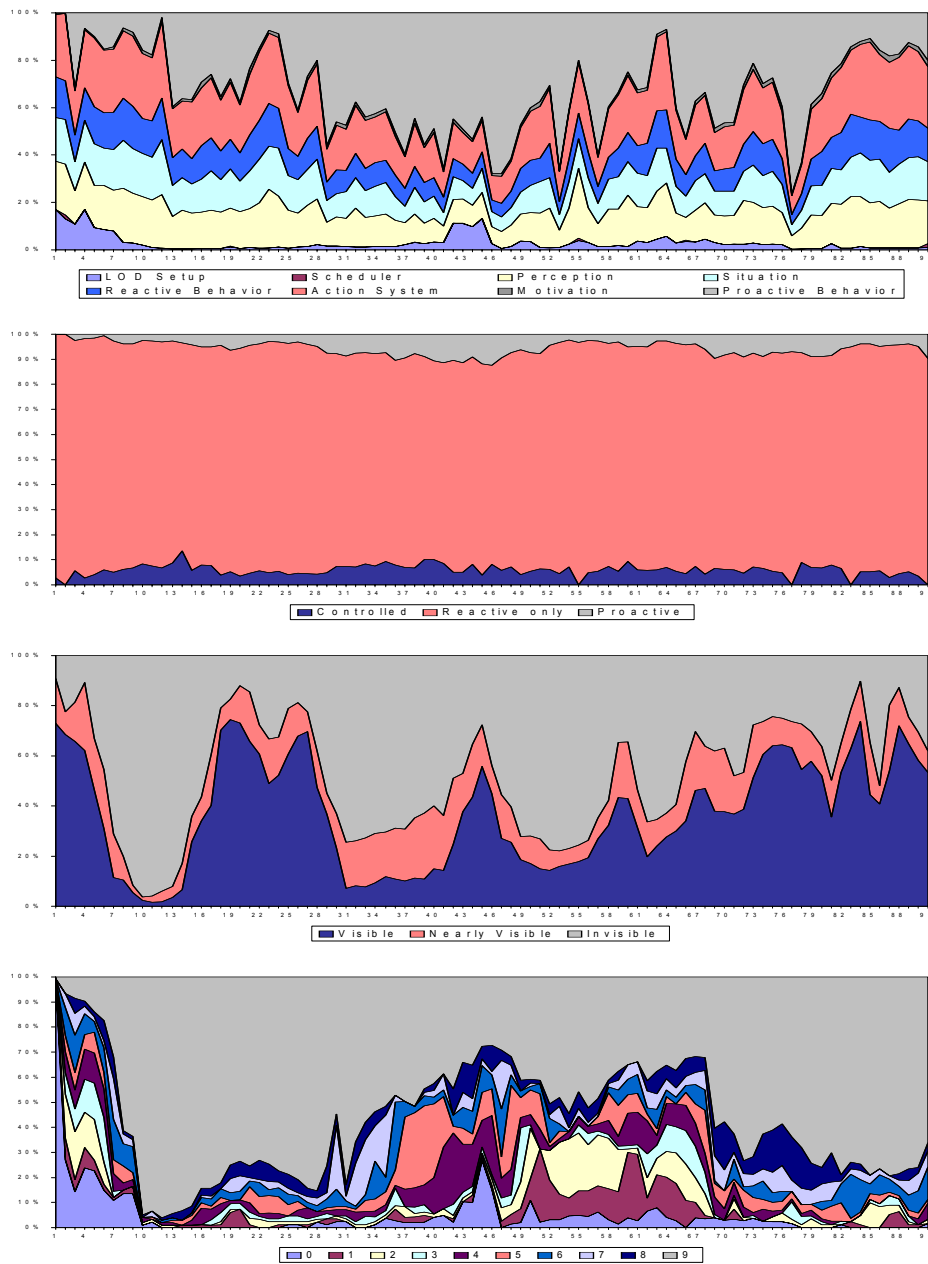


FIGURE 5.16 The behavior of the whole soutuion over time for another scenario. The first chart shows the percentage of time spent on different tasks. The second chart shows the behavior type of the agents. The third chart shows the visibility of the agents over time. The last chart depicts the LOD distribution of the agents.

SYSTEM OVERVIEW

This chapter will present an overview of the whole system which has been presented during the previous chapters. We will go into details only where necessary and start with a global system description and the general architecture. Afterwards, the most important components are discussed before an overview of the interfaces concludes this chapter.

6.1 INTRODUCTION

This chapter presents the agent engine which allows for the simulation of a character's behavior in a dynamic real-time environment as described in this thesis. This agent engine is intended to support a main application such as a game engine which provides an environment and takes care of the rendering issues, user interaction and the physical simulation as shown in Figure 6.1. The simulation environment is expected to provide a real-time simulation and interactive frame-rates.

Usually, a game engine provides a main loop during which the following steps are executed repeatedly: User input check, networking, simulation update, and rendering of the current state. The third step, the simulation update, is strongly coupled with our agent engine. Therein, a game engine usually activates the physical simulation and moves the objects and characters according to the time passed since the last update. Before moving any character, the character should have the possibility to change its direction, speed or other properties which is done in the agent engine.

The coupling of the simulation environment and the agent engine takes place just before the simulation update step. It consists of three different operation types. First, the characters need to be activated regularly such that they can adapt their presented behavior accordingly. Second, the characters need information from the environment which is handled by a sensor interface. Third, the character needs to

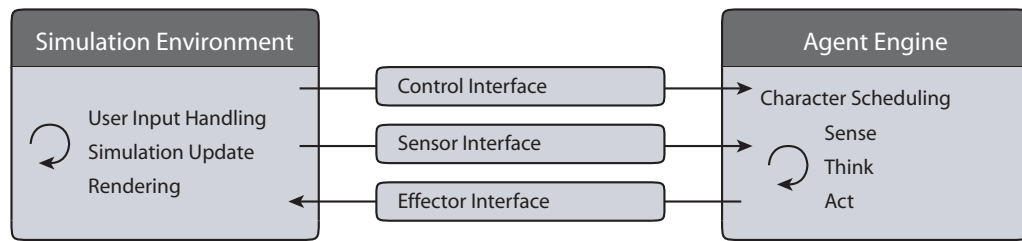


FIGURE 6.1 The tasks and interaction of the simulation environment and the agent engine. The simulation environment handles the usual game engine tasks. During the simulation update, the agent engine is activated and the scheduled agents start their individual cycle during which the sensor interface receives information from the environment and the effector interface reports the actions taken back to the simulation.

submit its decisions back to the game engine which moves the character accordingly and displays some appropriate animations. Therefore, the interface between the simulation environment and the agent engine is threefold. The *control interface* receives the time available for the behavior simulation and basically activates the scheduler. The scheduler then starts the agents according to its strategy. As we know from Chapter 2, each agent's simulation procedure consists of three steps: Sense, think, and act. During the sense step, the *sensor interface's* call-back functions are used to update the agent engine's own representation with information provided by the simulation environment. With the updated information available, the agent deliberates over its possibilities either in a reactive or proactive manner, depending on the scheduler's decision and the level of autonomy. Then, some action is executed in the action system which uses the effector interface to update some variables in the simulation environment.

In the following paragraphs, we will start by discussing the simulation environment, the interfaces between the simulation environment and the agent engine, and the components of the agent engine that make use of these interfaces.

6.2 SIMULATION

The presented behavior engine is useless without a simulation environment that takes care of the movement of the individuals and the rendering of the entire scene.

Since the target platform is a computer game, the environment should support a real-time simulation where one second of simulation time corresponds to one second of real time. Also, interactive frame-rates should be feasible which primary depends on the performance of the simulation environment. Since the behavior engine is flexible in this aspect, one can determine the remaining time for an acceptable frame-rate and allow the behavior engine to spend it. Therefore, no time will be wasted and an almost constant frame-rate can be expected, and, as the results show, are achieved.

Thus, the behavior simulation is very flexible with this mechanism. When allowing the engine to use as much time as needed, all agents can be fully simulated.

Using that approach, one can generate the behavior off-line and replay the outcome of the simulation without using the behavior engine on-line. We did not make use of that approach but generated all examples and scenarios in the on-line and real-time mode.

In order to become fully operative, the simulation environment that uses the behavior engine has to fulfill several requirements as addressed at the beginning of this section. It has to provide a simulation update mechanism, a rendering function, as well as a user input mechanism. The simulation update measures the time spent since the last call and advances the characters and objects according to physical laws. In our environment, this is a simple forward integration based on the current velocity and orientation. This mechanism can also handle collision handling such that the behavior engine has no need to concern itself with that matter.

6.3 INTERFACES

As we have seen and depicted in Figure 6.1, there are three different interfaces between the simulation environment and the agent engine. In this section, we will describe these and discuss related issues.

6.3.1 Control Interface

The control interface is the main connectivity between the simulation environment and the agent engine. The main methods provide the following functionality:

- ▶ *Loading the agents* according to a definition in an agent description file as described in Section 3.3.2. This method is usually called before starting the whole simulation process. It generates all agents according to the definition and initializes the attributes.
- ▶ *Setting the total run-time* which influences the overall performance. This is a very mighty method since several possible scenarios are possible. One can increase the currently available run-time up to several seconds which will slow down the simulation but concurrently allows all agents to be activated during one simulation cycle. Or on the other extreme, one can decrease the value to a few milliseconds such that only a portion of all agents will be activated but the simulation environment can keep its simulation frequency almost constant allowing for a constant frame-rate independent of the total number of agents simulated.
- ▶ *Activating the agent engine's scheduler* which then activates as much agents as possible with respect to the total run-time. As long as the total run-time has not exhausted, the scheduler will continue to activate agents. The scheduler ensures that every agent will be activated at most once – even though some time might be left at the end when returning the control back to the simulation environment.
- ▶ *Rendering of additional information* and allowing other components to render, too. This is important when debug information are needed during a real-time simulation. But not only the agent engine provides such debug information but also some other components such as the scheduler or each agent itself.

- Handling *user commands* which are not handled by the game engine and are passed to the agent engine which probably distributes these further to other components, such as agents. This includes any global command to change settings in the agent engine or commands directly to an agent which are presented in appendix A.

6.3.2 Sensor Interface

The sensor interface is a set of callback functions that have to be implemented in the simulation environment, i.e. the game engine. These functions should provide information that is perceived by the agents as explained in Section 3.4.1. There exist two different types of sensor callback functions: There are *global* sensory information and *agent-specific* information.

The global information concerns the environment itself, the camera's position and orientation. For the environment, information about the ground such as the height or the gradient at a particular position is essential. Also, the dimension of the ground and the contours of the lakes are needed, for example, in the navigation system used for path-planning. Very important for planning is a method which checks whether a particular position is valid or not.

The agent-specific information can be different for each agent. In our rather simple environment, the basic information needed by an agent can be reduced to its position, orientation and velocity. Additionally, the current level-of-detail and the current neighbors are determined by the simulation environment, and are therefore also part of the sensor interface.

The determination of the neighboring objects and agents in the dynamic environment is complex and therefore computationally expensive. We compared different data-structures with respect to the speed of updates and queries and the memory requirements as shown in Table 6.1. [Rih04]

TABLE 6.1 Comparison of different data-structures used for local neighborhood queries. The insert/update column describes the effort to insert or move an agent in the environment. The query column shows the costs for determining the local neighbors of a particular agent and the last column describes the memory requirements of the data-structure.

	Insert/Update	Query	Memory
Grid	fast and constant	fast	inefficient
Quadtree	medium	fast	adaptive
R-Tree	inefficient	fast	adaptive
k-d Tree	inefficient	fast	adaptive

As can be seen, all data-structures provide a comparable cost to determine the local neighbors and have only very small influence on the overall efficiency. Therefore, the insert/update cost have to be compared in order to select the best alternative. This operation will take place during every simulation cycle after the

environment has moved the agents according to their velocity and orientation. Thus, its efficiency is even more important than the query's efficiency since the query is executed only when an agent becomes active. When comparing the four data-structures, the grid and quadtree seem to be the most promising approaches but with different properties. After the evaluation above, we decided to use the quadtree for static objects due to its adaptive memory requirements, and the grid for dynamic objects due to its fast and constant update-operations.

A more thorough inspection of the query efficiency of both these approaches has been done, too. There are two different types of neighborhood queries:

- ▶ *Neighbors in sight (NIS)*. This query determines all neighbors within a certain distance from the querying agent without any restriction of the number of agents.
- ▶ *k-nearest neighbors (kNN)*. This query returns the k nearest neighbors without any restriction of the distance.

Both these queries can be useful in particular situations. NIS is usually needed to simulate the vision system of an agent with a restricted field of view. kNN is necessary when only a restricted number of agents are needed. Both these queries have different properties and the results are out of scope of this thesis but can be found in [Rih04].

6.3.3 Effector Interface

Opposite to the sensor interface, the effector interface is used to set values for the simulation or to propagate the decisions of the agents back to the simulation environment. The effector interface is an abstract class that has to be implemented by the simulation environment. Since the effector interface is agent-specific, each agent representative in the environment has to provide the according functionality.

Using that interface, each agent in the agent engine has access to its counterpart in the simulation environment. After initialization, each agent provides its initial position, orientation, velocity, and name to the environment. The maximal velocity, debug color, and the name of the model which is used to display the agent are passed to the environment, too, but only for debug puposes.

During the simulation, each agent can change its direction and its speed by setting the desired values. The simulation environment is responsible to generate natural looking turns such that the orientation does not change too much during one time step. Also, the actual velocity should be changed only smoothly such that abrupt changes of the velocity do not occur.

6.4 CORE COMPONENTS

This section presents an overview of the most important components of the agent engine. When zooming into the agent engine in Figure 6.1, it can be decomposed into different sub-parts as shown in Figure 6.2:

- ▶ The agent engine (AE) itself is a singleton, thus, only one instance of the AE exists. It is the main component and implements the controller interface as stated above. The AE stores a lists of all agents, one for the abstract agents

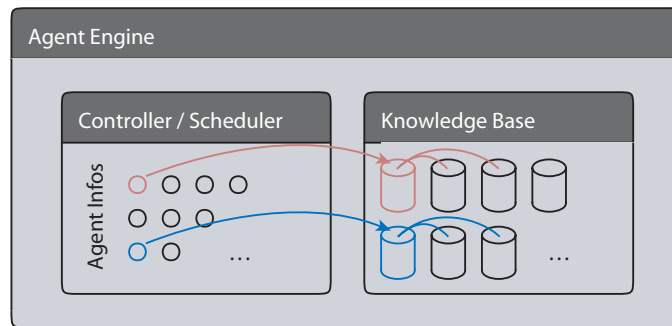


FIGURE 6.2 The main components of the agent engine. The agent engine provides the main interface and maintains all other components, basically a controller/scheduler and the knowledge base. Each agent has a representation in the controller as well as a related component in the knowledge base denoted by the arrow. The knowledge base components are collections of multiple other components.

needed during the construction of agent instances, one for the agent groups, and one for all agent instances that are currently active. Furthermore, it provides access to the controller and the knowledge base.

- The *controller/scheduler* is the main entry point used by the control interface and handles scheduling and the hierarchical control mechanism. Depending on the AE's definition, either the round robin or priority based scheduling algorithm is active. During each activation, the controller receives the total available time for simulating the behavior. During the first few cycles, no proactive behavior is allowed at all in order to determine the minimal time each agent needs according to Section 5.5.3.

Each active agent is registered at the scheduler after its initialization which generates an *agent info* object that provides the information necessary for the scheduler. The agent info instance stores whether the agent is autonomous, controlled, or controlling other agents. In the last case, the inferior agents are placed in a special list such that the controller knows which agents to release. Additionally, the actual time-account and its upper and lower threshold are stored in each agent info object

- Each of these agent info objects also has an instance of the *agent* object which provides the basic interface between the agent info and the knowledge base. It is used to load the agents definition, creating the according knowledge base components, and to activate an agent such that its behavior simulation starts. During the cycle of each agent instance, the same methods are executed, independent on the type of the agent. Only the associated knowledge-base components have an influence on the simulation and, thus, on the behavior.

The agent is basically a blackboard which has several components that need some input and create some output needed by other blackboard components. These components are either time-consuming or not. When an agent is not allowed to act proactively, only the time-independent components are activated.

A reactive agent consists of the components *Sensor* (receive sensor data from the

simulation environment), *Situation* (determine the actual situation), *Reactive* (determine the action according to the actual situation), and *ActionSystem* (execute the actions). A proactive agent additionally needs the components *Motivation* (determine the actual goal) and *Planning* (generate a sequence of actions according to the goal) before the *ActionSystem* is activated.

- ▶ The *knowledge base* is the container for all individual agent-specific knowledge. Each of the agents represented in the scheduler has one associated container in the knowledge base which is related to several other components as presented in Section 3.3.3 and Section 4.6.3. It is possible that multiple components refer to a single component.

CONCLUSIONS & OUTLOOK

In this chapter, we summarize the methods and results presented in the preceding chapters of this thesis. Finally, we will point out directions for future work in the last paragraph.

7.1 SUMMARY

In this thesis, a framework has been presented that allows for a real-time behavior simulation of a large number of characters in a dynamic environment. The generated behavior is both reactive and proactive, thus, allows a character to plan ahead in order to achieve a goal while maintaining a correct state with reactive behavior following unforeseen exceptions. The framework can be used in different simulation environments and provides a quality of service with respect to the response time such that the environment is allowed to achieve constant frame-rates practically independent of the number and complexity of agents.

Reactive and Proactive Behavior. The characters behavior relies on the concept of agents and is built out of simple basic components. First, the sensory system uses an interface to the environment to provide actual information to the agent. Secondly, the reactive system is activated to determine a situation which matches best to the current environmental and internal state. If there is such a situation, it provides a reaction to resolve it. Optionally, the agent might be allowed to plan ahead in order to achieve a goal. This is done in a third step and provides a partial plan in the direction of the goal state. The last step is the execution of the current action which is usually the partial plan but can be overridden by the reaction provided. Thus, the agent can act proactively but his reactive system prevents him from entering a dangerous or unexpected situation.

Behavior Composition. The framework permits to compose a sophisticated character out of simple basic behavior patterns which can be weighted additionally in

order to prefer a particular basic behavior. This mechanism allows a behavior designer to first implement reusable basic components and then combining and extending these in order to provide more sophisticated characters. The weighting of combined components makes it possible to prefer a particular basic behavior to another one.

Group Generation. These characters can be grouped together using different forms of groups that simplify social behavior. In structured groups, the members can be individually specified, in heterogeneous groups, modulo-based rules are used to specify regular parts of the group, and in hierarchical groups, the members are defined recursively such that an internal hierarchy of group members is achieved. Especially, the hierarchical groups present the potential for a reduction of the computational effort as has been shown for two different grouping behavior models.

Concurrent Planning in a Dynamic Environment. Planning proactively in a dynamic real-time environment poses different requirements to the planning system. First, the system has to be interruptible such that other agents are allowed to be activated, too. Therefore, we used anytime search algorithms which fulfill this requirement. Secondly, several mechanisms are necessary such that the agent will always have an actual plan available. We have presented a solution which allows the agents to pursue a plan while generating a new one with only negligible interruptions. Thirdly, we have discussed and presented search strategies that are applicable in our environment. Fourthly, the dynamic environment poses different demands on the planning system such that changes in the environment can be reflected in the planning unit.

Level-of-Detail for Behavior. The so far described framework has a complexity that does not achieve a visually appealing simulation with a large number of agents. Therefore, a level-of-detail approach has been introduced which divides the environment into areas of different importance where the area in front of the camera is most important. An associated scheduling algorithm distributes the few milliseconds available for the simulation to the agents according to their importance. Thus, the visible behavior is improved without neglecting the invisible agents. Furthermore, agents engaged in a hierarchical group with low importance can reduce the over-all complexity of the group by passing the control to a proactive and superior agent while remaining purely reactive. When the group's importance rises again, the mechanism will release the control back to the individuals.

Extensible Framework. Our framework is extensible with respect to additional behavior such as learning or other human capabilities presented in Section 2.4. Furthermore, many components are designed such that the addition of other and novel algorithms is relatively simple.

We have shown with different examples and some quantitative and qualitative measurements that this approach allows for the generation of sophisticated behavior in a real-time environment. The simulation provides interactive frame-rates almost independent of the number of agents.

7.2 OUTLOOK & FUTURE WORK

The foregoing framework is not yet complete with respect to a truly sophisticated behavior simulation. Also, the research on proactive agents in real-time environments is still relatively unexplored but could be a key component of tomorrow's games. Therefore, we expect that research in this field will continue and some results will get into commercial applications. In this section, we will point out possible future directions related to the work presented in this thesis.

Other Hierarchies. The afore mentioned hierarchies in groups and actions are certainly not all possibilities, where hierarchical approaches can have a significant impact on possible variations and computational effort. Some work has already been presented with respect to hierarchical sensors [IBDB01], goals or situations.

Utility-based Approach. The current proactive behavior relies on the selection of the currently best plan to a certain situation. If there are multiple possibilities to chose from, the utility-based approach could compare these and select the one which matches best to the current needs and desires of the agent.

Proactive Behavior. The presented system supports proactive behavior for a single agent based on rather general algorithms. We expect that the incorporation of other search strategies, such as informed ones, could improve the performance of the application and the quality of the generated plans. Also, two-person problems have been discussed but have not yet been implemented due to the computational demands. However, many decisions rely on the behavior of an other agent and as long as this behavior is just simulated in a reactive manner, the resulting behavior is erroneous, especially, when the other one is acting proactively, too.

Additional Units. Currently, the agent's behavior model consists of both reactive and proactive behavior. The system design allows to enhance the model by adding novel units to the blackboard which can be either time-consuming or not. We envision that a learning unit could improve the behavior of the agents over time by monitoring the state and decisions of the agent. Upon this information, the learning unit could alter internal variables or parameters in order to improve the over-all behavior. Such a learning unit would impose the need for an evaluation function which determines the degree of success given the current state and actions taken. An anticipation unit is another possible extension to improve the behavior. As stated in [KB00], the behavior of animals is largely based on expectations. With anticipation, an animal can display astonishment if an unexpected situation occurs. Of course, there are many further units possible, such as emotions or creativity.

Behavior Editor. The current system requires the designer to specify the components and agents directly in a XML file as described in the text. An editor with a graphical user interface could improve the usability of the framework considerably. The description as well as the combination of the base components including the generation of agent instances would be easier when the result could be displayed directly.

A P P E N D I X



SOFTWARE COMPONENTS

The following appendix presents a short overview of the software components used and developed in this research project.

This research project has been developed using the Microsoft Visual Development Studio 7.0 in C++. The solution is divided into several projects, each responsible for a particular mechanism as follows:

- *czBasic*, *czKnowledgeBase*, *czActionSystem*, *czAgent*, and *czAgentEngine* are the core projects of the behavior simulation engine. *czBasic* provides some fundamental classes such as the interfaces, the timer, and the parameters. *czKnowledgeBase* handles the storage and data-structures for all knowledge base components. The action system (*czActionSystem*) is used by all agents of which various implementations are available in *czAgent*. At the end, in *czAgentEngine* all other projects are assembled. This project provides the main interface to access the functionality of the behavior simulation.
- *GaiaEngine* and *TestSimulation* are two different simulation environments. *TestSimulation* is the old version which renders the characters only as billboards without considering their orientation. The newer simulation environment, the *GaiaEngine*, provides LOD rendering, animated characters, extended environment definition possibilities, and two different rendering modes – realistic and comic rendering.
- *qtCreaZoo* is a run-time control interface that directly affects the rendering and behavior of the simulation environment during run-time. It is implemented using Qt [QT 05]. A screenshot is shown in Figure 7.1.
- *Cal3D* is a software library for animating characters developed by Bruno Heidelberger [Hei01]. It is used to animate the characters used in the Gaia Engine.

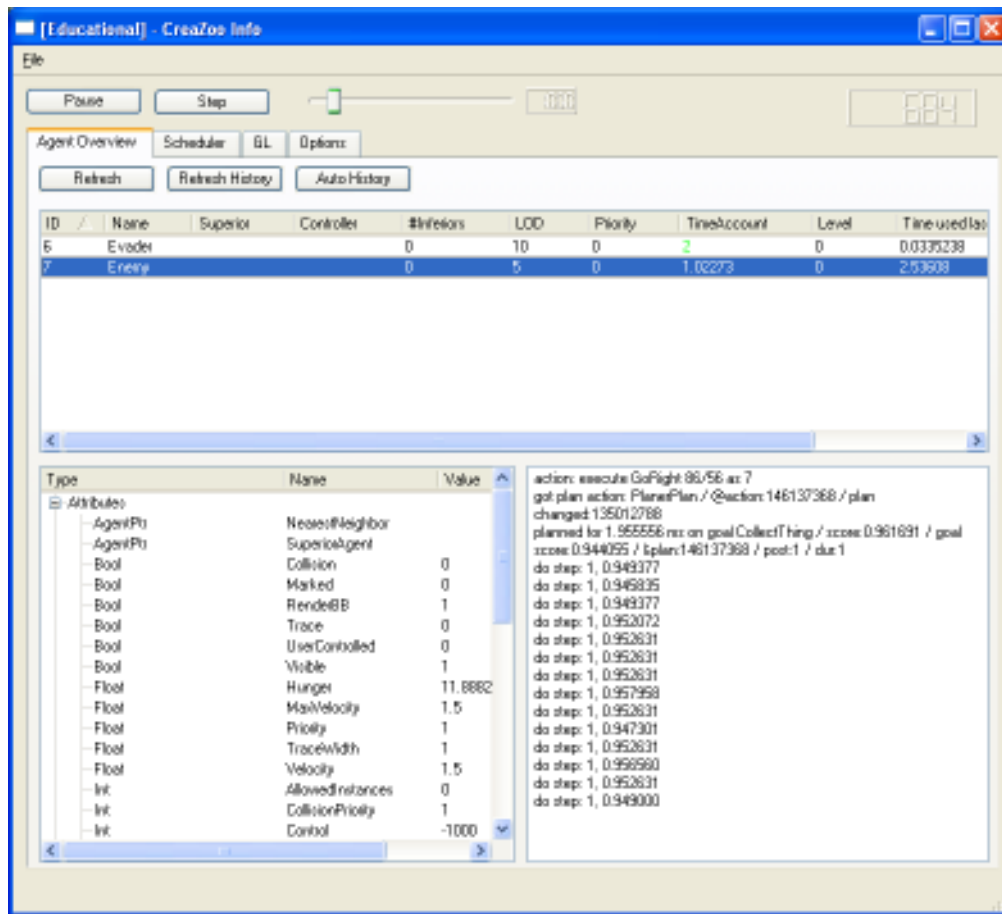


FIGURE 7.1 A screenshot of the run-time control interface. On top, the active agents are displayed, on the bottom left, some of the knowledge of the selected agent is shown and on the right its current history.

- *ContourExtraction*, *Clipping*, and *Tessellation* are projects used in the path-planning engine. The contour extraction package is used to find the contour lines of lakes which are handled as obstacles in the path-planning architecture. These contours and the other obstacles such as trees are first clipped, then tessellated before the dynamic A* algorithm can be applied.

A P P E N D I X

B

USER COMMANDS

During the simulation, the user can interact with the simulation in the GAIA engine by pressing distinct keys or by entering user commands on the console.

B.1 KEY COMMANDS

General

p	Pause behavior simulation and movement
h	Print help
ESC	Exit
TAB	Toggle Console on/off

Camera movement

w	Move camera forward
s	Move camera backward
a	Move camera to the left
d	Move camera to the right
q	Move camera upwards (not always available)
e	Move camera downwards (not always available)
g	Stick camera to an agent (2 different modes)
[or]	During stuck camera: Switch agent
y	Toggle between three different cameras
m	Change camera state: Free over terrain / always same height / free

Rendering

c	Switch between realistic and comic rendering
f	Toggle fog
b	Switch between colored cubes and 3D meshes

Debug

k	Switch to debug rendering
n	Display the names of the agents
l	Display the LOD for each agent
v	Display velocity of each agent
i	Print scheduler information
j	Print camera information
t	Print statistics of GAIA engine
backspace	Switch to debug rendering of GAIA engine

B.2 CONSOLE COMMANDS

cls/clear	Clear command line
help Gaia	Display GAIA console commands
camera x y z lx ly lz	Set camera position and orientation (1-values: look at)
cmd ...	Command to the behavior simulation

B.2.1 Behavior Simulation Console Commands

cmd name r ...

Recursive command. If *name* is in a hierarchy or group, the command after the *r* will be passed to the inferior agents, too. Works with all commands in this section.

cmd name goto x y

Makes *name* following a static path of the path-planning system

cmd name go x y

Makes *name* going to the destination by using its planning capabilities.

cmd name bring other x y

Makes *name* following *other* and bring it to the desired destination using the proactive behavior.

cmd name clear

Reset agent name:

Stopping current action, removing last user command

cmd name stop

Stop agent name:

Setting velocity to zero, removing last user command

cmd name hon [n], cmd name hoff

Activate (`hon`) or deactivate (`hoff`) the agent's history. The history can be displayed in the run-time control interface or on the command line. `n` denotes the number of entries in the history.

cmd name hprint

Print agent history on command line.

cmd name print

Print agent information on command line. Only active in Debug mode.

cmd ctrl name, cmd rel name

Change control of a specific agent. `ctrl` will make the agent pass the control to the superior agent and `rel` will make an agent release all control over other agents.



REFERENCES

- [ACF01] Okan Arikan, Stephen Chenney, and D. A. Forsyth. Efficient multi-agent path planning. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 151–162. Springer-Verlag New York, Inc., 2001.
- [ACP00] R. Aylett, A.M. Coddington, and G.J. Petley. Agent-based continuous planing. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)*, 2000.
- [AKW+01] Marc S. Atkin, Gary W. King, David L. Westbrook, Brent Heeringa, and Paul R. Cohen. Hierarchical agent control: a framework for defining agent behavior. In *Proceedings of the fifth international conference on Autonomous agents*, pages 425–432. ACM Press, 2001.
- [AV01] G. Armano and E. Vargiu. An adaptive approach for planning in dynamic environments. In *Proceedings of the International Conference on Artificial Intelligence (ICAI 2001)*, 2001.
- [AWC99] M. S. Atkin, D. L. Westbrook, and P. R. Cohen. Capture the flag: Military simulation meets computer games. In *Papers from the AAI 1999 Spring Symposium on Artificial Intelligence and Computer Games*, pages 1–5. AAAI Press, 1999.
- [BB01] Ana L. C. Bazzan and Rafael H. Bordini. A framework for the simulation of agents with emotions. In *Proceedings of the fifth international conference on Autonomous agents*, pages 292–299. ACM Press, 2001.
- [BBB+98] Norman Badler, R. Bindiganavale, J. Bourne, M. Palmer, J. Shi, and W. Schuler. A parameterized action representation for virtual human agents. In *Workshop on Embodied Conversational Characters, Lake Tahoe, CA*, 1998.

- [BD94] M. Boddy and T. Dean. Decision-theoretic deliberation scheduling for problem solving in time-constrained environments, 1994.
- [BDH+01] Jan Broersen, Mehdi Dastani, Joris Hulstijn, Zisheng Huang, and Leendert der van Torre. The boid architecture: conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the fifth international conference on Autonomous agents*, pages 9–16. ACM Press, 2001.
- [BDI+02] Bruce Blumberg, Marc Downie, Yuri Ivanov, Matt Berlin, Michael Patrick Johnson, and Bill Tomlinson. Integrated learning for interactive synthetic characters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 417–426. ACM Press, 2002.
- [BFEM97] A. Bruderlin, S. Fels, S. Esser, and K. Mase. Hierarchical agent interface for animation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 27–31, 1997.
- [BG95] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 47–54. ACM Press, 1995.
- [BH95] K. Balakrishnan and V. Honavar. Evolutionary Design of Neural Architectures: A Preliminary Taxonomy and Guide to Literature. Technical report, Department of Computer Science, Iowa State University, Ames, Iowa, 1995.
- [BH02] David C. Brogan and Jessica K. Hodgins. Simulation level of detail for multi-agent control. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 199–206. ACM Press, 2002.
- [BID+01] R. Burke, D. Isla, M. Downie, Y. Ivanov, and B. Blumberg. Creaturesmarts: The art and architecture of a virtual brain. In *Proceedings of the Game Developers Conference*, pages 147–166, 2001.
- [BLA02] O. Burchan Bayazit, Jyh-Ming Lien, and Nancy M. Amato. Roadmap-based flocking for complex environments. In *Proceedings of the 2002 Pacific Graphics*, 2002.
- [BLM99] Luis Blando, Karl Lieberherr, and Mira Mezini. Modeling behavior with personalities. In *International Conference on Knowledge and Software Engineering*, 1999.
- [BM97] Bruce Mitchell Blumberg and Pattie Maes. *Old tricks, new dogs: ethology and interactive creatures*. Ph.d. thesis, Massachusetts Institute of Technology, 1997.
- [BNS91] A. Bar-Noy and B. Schieber. The canadian traveler problem. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 261–270, 1991.
- [Bod93] M. Boddy. Temporal reasoning for planning and reasoning. *SIGART Bulletin*, 4:17–20, 1993.
- [Bra87] Michael Bratman. *Intentions, Plans, and Practical Reason*. MIT Press, Cambridge, Massachusetts, 1987.
- [Bur96] Christopher J. C. Burges. Simplified support vector decision rules. In *International Conference on Machine Learning*, pages 71–77, 1996.

- [Bur98] A. Burt. Modelling motivational behaviour in intelligent agents in virtual worlds, 1998.
- [CAF01] S. Chenney, O. Arikan, and D.A. Forsyth. Proxy simulations for efficient dynamics. In *Proceedings of Eurographics 2001 (Short Presentations)*, 2001.
- [Can88] J. Canny. *The Complexity of Robot Motion Planning*. MIT Press, 1988.
- [Can97] D. Canamero. Modeling motivations and emotions as a basis for intelligent behavior. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents, New York, NY*, pages 148–155. ACM Press, 1997.
- [CBC01] L. Chen, K. Bechkoum, and G. Clapworthy. A logical approach to high-level agent control. In *Proceedings of the fifth international conference on Autonomous agents*, pages 1–8. ACM Press, 2001.
- [CH97] Deborah A. Carlson and Jessica K. Hodgins. Simulation levels of detail for real-time animation. In Wayne A. Davis, Marilyn Mantei, and R. Victor Klassen, editors, *Graphics Interface '97*, pages 1–8. Canadian Human-Computer Communications Society, 1997.
- [CJ92] Robert J. Collins and David R. Jefferson. Antfarm: Towards simulated evolution. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 579–601. Addison-Wesley, Redwood City, CA, 1992.
- [CJH02] Eric Chown, Randolph M. Jones, and Amy E. Henninger. An architecture for emotional decision-making agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 352–353. ACM Press, 2002.
- [CMT01] Angela Caicedo, Jean-Sébastien Monzani, and Daniel Thalmann. Toward like-like agents: Intergrating tasks, verbal communication and behavioural engines. *VR Journal*, Springer, 2001. To appear.
- [CTT03] Toni Conde, William Tambellini, and Daniel Thalmann. Behavioral animation of autonomous virtual agents helped by reinforcement learning. In *Lecture Notes in Computer Science, Vol. 272*, Springer Verlag, Berlin, pages 175–180, 2003.
- [CW01] Andr L. V. Coelho and Daniel Weingaertner. Evolving coordination strategies in simulated robot soccer. In *Proceedings of the fifth international conference on Autonomous agents*, pages 147–148. ACM Press, 2001.
- [Dau00] Kerstin Dautenhahn. Socially intelligent agents and the primate social brain - towards a science of social minds. In *Proceedings of the AAI Fall Symposium "Socially Intelligent Agents"*. AAI Press, 2000.
- [Dav00] Ian Lane Davis. Warp speed: Path planning for star trek: Armada. In *AAAI Spring Symposium Technical Report (2000 AAI Spring Symposium)*, 2000.
- [DB88] T. Dean and M. Boddy. Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399, 1988.
- [DD+96] Jon Doyle, Thomas Dean, et al. Strategic directions in artificial intelligence. *ACM Computing Surveys*, 28(4):653–670, 1996.

- [DeL00] Mark DeLoura, editor. *Game Programming Gems*. Charles River Media, Hingham, MA, 2000.
- [DeL01] Mark DeLoura, editor. *Game Programming Gems 2*. Charles River Media, Hingham, MA, 2001.
- [FN71] Richard Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [Fra99] I. Frank. Explanation count. In *Papers from the AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games*, pages 77–80. AAAI Press, 1999.
- [FTT99] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 29–38. ACM Press/Addison-Wesley Publishing Co., 1999.
- [Fun98] John David Funge. *Making them behave: cognitive models for computer animation*. PhD thesis, University of Toronto, 1998.
- [Fun00] John Funge. Cognitive modeling for games and animation. *Communications of the ACM*, 43(7):40–48, 2000.
- [FvdPT01] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 251–260. ACM Press, 2001.
- [GCM97] Stephen Grand, Dave Cliff, and Anil Malhotra. Creatures: artificial life autonomous software agents for home entertainment. In *Proceedings of the first international conference on Autonomous agents*, pages 22–29. ACM Press, 1997.
- [GG95] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In *Proceedings of the IEEE Visualization '95*, pages 135–142. IEEE Computer Society Press, 1995.
- [GHK99] Barbara Grosz, Luke Hunsberger, and Sarit Kraus. Planning and acting together. *The AI Magazine*, 20(4):23–34, 1999.
- [GL00] C. Geiger and M. Latzel. Prototyping of complex plan based behavior for 3d actors. In *Proceedings of the fourth international conference on Autonomous agents*, pages 451–458. ACM Press, 2000.
- [GL01] Piotr J. Gmytrasiewicz and Christine L. Lisetti. Emotions and personality in agent design and modeling. In *Proceedings of the 8th International Conference on User Modeling 2001*, pages 237–239. Springer-Verlag, 2001.
- [GL02] Piotr J. Gmytrasiewicz and Christine L. Lisetti. Emotions and personality in agent design. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 360–361. ACM Press, 2002.
- [Gra96] Joshua Grass. Reasoning about computational resource allocation. *Crossroads*, 3(1):16–20, 1996.

- [GT95] Radek Grzeszczuk and Demetri Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 63–70. ACM Press, 1995.
- [GTH98] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: fast neural network emulation and control of physics-based models. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 9–20. ACM Press, 1998.
- [Hau03a] Christoph Hauser. Anytime Planung in creaZoo. Masters thesis, Computer Science Departement, ETH Zürich, 2003.
- [Hau03b] Christoph Hauser. Entwicklung eines Echtzeitplaners. Semester thesis, Computer Science Departement, ETH Zürich, 2003.
- [Haw00] Nick Hawes. Real-time goal-oriented behaviour for computer game agents. In *Proceedings of Game-On2000, 1st International Conference on Intelligent Games and Simulation*, pages 71–75, 2000.
- [Haw01] Nick Hawes. Anytime planning for agent behavior. In *Proceedings of PLANSIG 2001*, pages 1–14, 2001.
- [Haw03] Nick Hawes. *Anytime Deliberation for Computer Game Agents*. PhD thesis, University of Birmingham, 2003.
- [Hei01] Bruno Heidelberger. Cal3d - character animation library. <http://cal3d.sourceforge.net/>, 2001.
- [HG94] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc.
- [HKR93] T.C. Hu, A.B. Kahng, and G. Robins. Optimal robust path planning in general environments. *IEEE Transactions on Robotics and Automation*, 9:775–784, 1993.
- [Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.
- [HP98] M. C. Horsch and D. Poole. An anytime algorithm for decision making under uncertainty, 1998.
- [HR85] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.
- [HR97] Frederick Hayes-Roth. Artificial intelligence - what works and what doesn't? *AI Magazine*, 18(2):99–113, 1997.
- [IB02] Damian Isla and Bruce Blumberg. New challenges for character-based ai for games. In *AAAI Spring Symposium on AI and Interactive Entertainment*, 2002.
- [IBDB01] D. Isla, R. Burke, M. Downie, and B. Blumberg. A layered brain architecture for synthetic characters. In *IJCAI, August 2001*, 2001.

- [KB00] Christopher Kline and Bruce Blumberg. Observation-based expectation generation and response for believable reactive agents. In *Proceedings of the fourth international conference on Autonomous agents*, pages 46–47. ACM Press, 2000.
- [KKKL94] Yoshihito Koga, Koichi Kondo, James Kuffner, and Jean-Claude Latombe. Planning motions with intentions. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 395–408. ACM Press, 1994.
- [KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
- [KLMR95] Lydia E. Kavraki, Jean-Claude Latombe, Rajeev Motwani, and Prabhakar Raghavan. Randomized query processing in robot path planning. In *ACM Symposium on Theory of Computing*, pages 353–362, 1995.
- [KMT02] Sumedha Kshirsagar and Nadia Magnenat-Thalmann. Virtual humans personified. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 356–357. ACM Press, 2002.
- [Kno04] Daniel Knoblauch. Gruppenverhalten in Echtzeit. Semester thesis, Computer Science Department, ETH Zürich, 2004.
- [KS86] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
- [KSLO96] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.
- [Kuf98] James J. Kuffner. Goal-directed navigation for animated characters using real-time path planning and control. In *CAPTECH*, pages 171–186, 1998.
- [Kuf99] James J. Kuffner. *Autonomous Agents for Real-Time Animation*. Ph.d. thesis, Department of Computer Science of Stanford University, 1999.
- [KZ92] C. Ronald Kube and Hong Zhang. Collective robotic intelligence. In *1992 International Conference on Simulation of Adaptive Behaviour*, pages 460–468, 1992.
- [Lai01] John E. Laird. It knows what you’re going to do: adding anticipation to a quakebot. In *Proceedings of the fifth international conference on Autonomous agents*, pages 385–392. ACM Press, 2001.
- [Lat69] J. C. Latombe. *Robot Motion Planning*. MIT Press, 1969.
- [LBD+90] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In D. Touretzky, editor, *Neural Information Processing Systems*, volume 2. Morgan Kaufman, 1990.
- [Leg99] Chris Leger. Phd thesis: Automated synthesis and optimization of robot configurations: An evolutionary approach, 1999.
- [LL01] John Laird and Michael Van Lent. Human-Level AI’s Killer Application: Interactive Computer Games. *AI Magazine*, 22(2):15–26, 2001.

- [LLL+94] Yves Lespérance, Hector J. Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. A logical approach to high level robot programming – a progress report. In B. Kuipers, editor, *Working notes of the 1994 AAAI fall symposium on Control of the Physical World by Intelligent Systems*, New Orleans, LA, November 1994.
- [LR94] Fangzhen Lin and Raymond Reiter. Forget it! In R. Greiner and D. Subramanian, editors, *Working Notes, AAAI Fall Symposium on Relevance*, pages 154–159, Menlo Park, California, 1994. American Association for Artificial Intelligence.
- [Mal97] H. Mallot. Behavior-oriented approaches to cognition: Theoretical perspectives, 1997.
- [MDBP95] P. Maes, T. Darrell, B. Blumberg, and A. Pentland. The alive system: full-body interaction with autonomous agents. In *Proceedings of the Computer Animation*, page 11. IEEE Computer Society, 1995.
- [MDCO02] B. MacNamee, S. Dobbyn, P. Cunningham, and C. O’Sullivan. Men behaving appropriately: Integrating the role passing technique into the aloha system. In *Proceedings of the AISB02 symposium: Animating Expressive Characters for Social Interactions (short paper)*, pages 59–62, 2002.
- [MF94] Melanie Mitchell and Stephanie Forrest. Genetic algorithms and artificial life. *Artificial Life*, 1(3):267–289, 1994.
- [MG03] Stacy Marsella and Jonathan Gratch. Modeling coping behavior in virtual humans: don’t worry, be happy. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 313–320. ACM Press, 2003.
- [Mil88] Gavin S. P. Miller. The motion dynamics of snakes and worms. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 169–173. ACM Press, 1988.
- [MKT99] S.R. Musse, M. Kallmann, and D. Thalmann. Level of autonomy for virtual human agents. In *Proceedings of the ECAL’99 (5th European Conference on Artificial Life)*, Lausanne, Switzerland, pages 345–349, 1999.
- [MM98] A. Martinoli and F. Mondada. Probabilistic modelling of a bio-inspired collective experiment with real robots, 1998.
- [MMG01] Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *Proceedings of the fifth international conference on Autonomous agents*, pages 246–253. ACM Press, 2001.
- [MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.
- [MT97] Soraia Raupp Musse and Daniel Thalmann. A model of human crowd behavior. In *Proceedings of the Workshop of Computer Animation and Simulation of Eurographics’97*, 1997.

- [MT01] Soraia R. Musse and Daniel Thalmann. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001.
- [MW05] MW. Merriam webster online dictionary. <http://www.m-w.com/>, 2005.
- [MYIM01] Yoichiro Matsuno, Tatsuya Ymazaki, Shin Ishii, and Jun Matsuno. A multi-agent reinforcement learning method for a partially-observable competitive game. In *Proceedings of the fifth international conference on Autonomous agents*, pages 39–40. ACM Press, 2001.
- [Nar00] Alexander Nareyek. Intelligent agents for computer games. In *Computers and Games, Second International Conference, CG 2000*, pages 414–422, 2000.
- [NG03] Christoph Niederberger and Markus H. Gross. Hierarchical and heterogenous reactive agents for real-time applications. In P. Brunet and D. Fellner, editors, *Proceedings of the Eurographics 2003, Computer Graphics Forum, Conference Issue*, pages 323–331, 2003.
- [NG04] Christoph Niederberger and Markus H. Gross. Generic path planning for real-time applications. In Bob Werner, editor, *Proceedings of the Computer Graphics International Conference 2004*, pages 299–306. IEEE Press, 2004.
- [Nie01] Christoph Niederberger. Punktbasierte Rekonstruktion von Objekten. Masters thesis, Computer Science Departement, ETH Zürich, 2001.
- [OCV+02] C. O’Sullivan, J. Cassell, H. Vilhjalmsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang. Levels of detail for crowds and groups. *Computer Graphics Journal*, 21(4):733–741, 2002.
- [OFG97] Edgar Osuna, Robert Freund, and Federico Girosi. Support vector machines: Training and applications. Technical Report AIM-1602, MIT AI Lab and Center for Biological and Computational Learning (Dept. of Brain and Cognitive Sciences), 1997.
- [O’H02] Noel O’Hara. Hierarchical impostors for the flocking algorithm in 3d. *Computer Graphics Journal*, 21(4):723–731, 2002.
- [O’R94] Joseph O’Rourke. *Computational geometry in C*. Cambridge University Press, 1994.
- [OY85] Colm O’Dunlaing and C. Yap. A retraction method for planning the motion of a disk. In *J. Algorithms*, pages 187–192, 1985.
- [Paj98] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings of the conference on Visualization ’98*, pages 19–26, 1998.
- [Pom89] Dean Pomerleau. Alvin: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems 1*. Morgan Kaufmann, 1989.
- [QT 05] QT Online Reference. *Qt - Reference and Specification*. Homepage: <http://www.trolltech.org/>, 2005.

- [Rab00] Steve Rabin. A* speed optimization. *Game Programming Gems*, pages 272–287, 2000.
- [Rad03] Dejan Radovic. Real-time Navigation System. Masters thesis, Computer Science Departement, ETH Zürich, 2003.
- [Rao96] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.
- [RCB98] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics & Applications*, 18(5), – 1998.
- [Rei96] W. Scott Neal Reilly. *Believable Social and Emotional Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.
- [Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34. ACM Press, 1987.
- [RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.
- [RH91] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 349–358. ACM Press, 1991.
- [Rih04] Jan Rihak. Local Neighborhood Information in dynamischen Umgebungen. Semester thesis, Computer Science Departement, ETH Zürich, 2004.
- [Rip96] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, 1996.
- [RL95] Ashwin Ram and David Leake. Goal-driven learning, 1995.
- [RN96] S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.
- [Ros97] J. Rosenblatt. Behavior-based planning for intelligent autonomous vehicles. In *Proceedings of AV Symposium on Intelligent Autonomous Vehicles, Madrid, Spain, 1997*.
- [Sau02] Robert Saunders. *Curious Design Agents and Artificial Creativity*. Ph.d. thesis, Department of Architectural and Design Science, Faculty of Architecture, University of Sydney, 2002.
- [Sei90] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Technical report, "Institute for Computer Science", Department of Mathematics, Freie Universität Berlin, 1990.

- [SF99] D. Schmalstieg and A. Fuhrmann. Coarse viewdependent levels of detail for hierarchical and deformable models, 1999.
- [Sim91] Karl Sims. Artificial evolution for computer graphics. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328. ACM Press, 1991.
- [Sim94] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM Press, 1994.
- [SLL02] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 19–26. ACM Press, 2002.
- [Slo99] Aaron Sloman. Architectural requirements for human-like agents both natural and artificial. (what sorts of machines can love?). *Human Cognition and Social Agent Technology*, 1999.
- [Son92] E. D. Sontag. Neural nets as systems models and controllers. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems, Yale University*, pages 73–79, 1992.
- [SSH87] Jacob T. Schwartz, Micha Sharir, and John E. Hopcroft. *Planning, Geometry and Complexity of Robot Motion*. Ablex Publishing Corp., 1987.
- [Sto00] Bryan Stout. The basics of a* for path planning. *Game Programming Gems*, pages 254–263, 2000.
- [SW01] Murray Shanahan and Mark Witkowski. High-level robot control through logic. In *Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages*, pages 104–121. Springer-Verlag, 2001.
- [TA02] Paul Tozour and Ion Storm Austin. Building a near-optimal navigation mesh. *AI Game Programming Wisdom*, pages 171–185, 2002.
- [Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.
- [TAW02] Kagan Tumer, Adrian K. Agogino, and David H. Wolpert. Learning sequences of actions in collectives of autonomous agents. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 378–385. ACM Press, 2002.
- [TB] B. Tomlinson and B. Blumberg. Social behavior, emotion and learning in a pack of virtual wolves.
- [Tha02] Daniel Thalmann. Simulating a human society: The challenges. In *Proceedings of Computer Graphics International 2002, Vince John, Earnshaw Rae (Ed.)*, 2002.
- [Tom96] Marco Tomassini. Evolutionary algorithms. In Eduardo Sanchez and Marco Tomassini, editors, *Towards Evolvable Hardware; The Evolutionary Engineering Approach*, pages 19–47, Berlin, 1996. Springer.

- [TPH02] John Thangarajah, Lin Padgham, and James Harland. Representation and reasoning for goals in bdi agents. In *Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 259–265. Australian Computer Society, Inc., 2002.
- [Tre02] Dante Treglia, editor. *Game Programming Gems 3*. Charles River Media, Hingham, MA, 2002.
- [TRG96] Demetri Terzopoulos, Tamer Rabie, and Radek Grzeszczuk. Perception and learning in artificial animals. In *Proceedings of the Fifth International Conference on the Synthesis and the Simulation of Living Systems*, pages 346–353, 1996.
- [TT94] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.
- [UT01] Branislav Ulicny and Daniel Thalmann. Crowd simulation for interactive virtual environments and vr training systems. In *Proceedings of the Eurographics Workshop of Computer Animation and Simulation'01*, pages 163–170. Springer-Verlag, 2001.
- [Vel] J. Vel'asquez. When robots weep: Emotional memories and decision-making.
- [vLL99] Michael van Lent and John Laird. Developing an artificial intelligence engine, 1999.
- [Weh98] T. Wehrle. Motivations behind modeling emotional agents: Whose emotion does your robot have. In *Proceedings of the SAB'98 Workshop. University of Zurich, Switzerland*, 1998.
- [Wei66] Joseph Weizenbaum. Eliza: a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, 1966.
- [Whi93] Darrell Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Department of Computer Science, Colorado State University, 1993.
- [Wil00] Ian Wilson. The artificial emotion engine, 2000.
- [WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
- [WM98] David E. Wilkins and Karen L. Myers. A multiagent planning architecture. In *Artificial Intelligence Planning Systems*, pages 154–163, 1998.
- [YB94] B. Yamauchi and R. Beer. Sequential behaviour and learning in evolved dynamical neural networks, 1994.
- [YBS00] Song-Yee Yoon, Bruce M. Blumberg, and Gerald E. Schneider. Motivation driven learning for interactive synthetic characters. In *Proceedings of the fourth international conference on Autonomous agents*, pages 365–372. ACM Press, 2000.
- [You01] Thomas Young. Expanded geometry for points-of-visibility pathfinding. *Game Programming Gems 2*, pages 317–323, 2001.
- [ZCB96] N. Zaera, D. Cliff, and J. Bruten. (not) evolving collective behaviours in synthetic fish. In *From Animals to Animats 4: Proceedings of the Fourth International Con-*

ference on Simulation of Adaptive Behavior (SAB96), pages 635–644. MIT Press
Bradford Books, 1996.

- [ZR95] S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms, 1995.

A P P E N D I X



CURRICULUM VITAE

Christoph Beat Niederberger

April 21st, 1975	Born in Basel, Switzerland, Citizen of Dallenwil, NW, Switzerland
1994	Matura Typus C (Swiss high school examination) Gymnasium Bäumlhof, Basel, Switzerland
2001	Diploma in Computer Science as Dipl. Informatik-Ing. ETH, Swiss Federal Institute of Technology, ETH, Zürich, Switzerland
2001-2005	Research and teaching assistant, Ph.D. student, Computer Graphics Laboratory, headed by Prof. M. Gross, Institute of Scientific Computing, Swiss Federal Institute of Technology, ETH, Zürich, Switzerland

