Hierarchical and Heterogenous Reactive Agents for Real-Time Applications

C. Niederberger, M. Gross

Departement of Computer Science
ETH Zürich
CH-8092 Zürich
Switzerland

Abstract

We present a generic concept for autonomous agents with reactive behavior based on situation recognition in real-time environments. Our approach facilitates behavior development through specialization of existing behavior types or weighted multiple inheritance in order to create new types. Additionally, the system allows for the simultaneous generation of hierarchical and semi-individual group organizations using specification and recursive or modulo-based patterns. Our framework is designed to support the creation of large numbers of secondary characters with individual and group behavior in simulation environments such as game engines. The engine allows for the specification of a maximal time-per-run in order to guarantee a minimal and constant frame-rate. We demonstrate the usefulness of our approach by various examples with up to hundreds of individuals.

Categories and Subject Descriptors (according to ACM CCS): I.2.11 [Distributed Artificial Intelligence]: Multiagent systems, I.6.7 [Simulation Support Systems]: Environments

1. Introduction

Secondary characters in games often lack personality because they all have the same underlying behavior patterns. We propose a system that simplifies the creation of new individuals by allowing multiple inheritance of basic behaviors in order to build complex individuals based on simple ones, as shown in Figure 1 on the top. This allows for the easy definition of specialized characters. However, since many characters should manifest subtle differences in order to display their own personality, we also weight the inheritance to magnify or scale down the probability of recognizing certain situations. Therefore, we compose different types of agents rather than building special ones for every type. In order to generate individual knowledge, we can use randomized attributes to further personalize individuals of the same type.

Furthermore, we allow for the construction of collective behavior by introducing attributes which contain a reference to another agent. This enables actions to communicate and collaborate between single agent instances, e.g. by informing other members of a group about an event or by querying the position of a leader and thereby following that position. This is shown in the middle of Figure 1. Building groups with our system can easily be done by defining abstract group patterns that are later instantiated with a specified count of agents, depicted in Figure 1 at the bottom. We permit specialization within the abstract group by

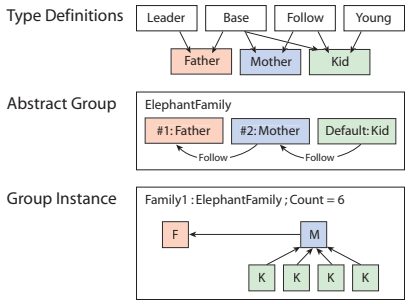

Figure 1: *Generating new types, grouping together and instantiation of the group including the hierarchy.*

using simple rules, e.g. "*individual #1 is the father*", modulo-based rules, e.g. "*every second individual is male, every other female*", and recursive definitions, e.g. "*a group consists of a leader and five followers, each becoming also a leader of a similar group*". During instantiation, our system generates the specified number of instances by using the predefined types for each agent. The above mentioned reference attributes are also automatically updated from the abstract definition to the real group thereby allowing the usage of generic group behavior within an unknown organized group.

Our system is especially designed for adding a large number of secondary characters with individual and group-based reactive behavior to game engines. To support constant or minimal frame-rates independent of the number of simulated agents, the engine is capable of specifying the maximum amount of time per run. The engine only simulates as many agents as possible using a round-robin scheduling algorithm and then returns the control.

2. Related Work

Since the seminal paper by Reynolds in 1987 [17] the number of publications on the use of behavior modeling to generate computer animations has increased substantially. Almost all approaches have used the concept of autonomous agents [18] where each instance has its own perception, behavior, and effectors. Commonly, these approaches have tried to generate artificial life that emerges from a small predefined set of behavior rules. Reactive agents with or without internal states [17,23] are the primary approach for real-time environments, because they are implemented easily and quickly. We expect games such as "SimCity" [19] and "The Sims" [21] to use similar approaches as presented for the behavior of their individuals.

Such individual behavior, including movement and locomotion, has been researched in many different fields of study, mainly on animals. The motion dynamics of snakes and worms by Miller [13] were followed by Terzopoulos' movements of fish [9], Tu and Terzopoulos' physics, locomotion, and perception of fish [23], and finally Terzopoulos' perception and learning of fish [22]. Bio-inspired modeling has been employed for the commercial computer game Creatures [7] to create simple pets which can be played with. Cognitive Modeling by Funge [6] 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. More recent publications have dealt with learning of movements [10,22,20] or learning of behavior [26,12,3], emotions [5,25,8], motivations [5], beliefs, obligations, intentions, and desires [4] or goal-oriented behavior in real-time environments [11]. As a limitation, these behaviors are hard-wired and especially designed for a specific individual. Perlin et al. [16] developed a general framework for scripting interactive actors. They introduced a layered behavior model for creating complex behaviors.. Blando et al. [2] 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. Hierarchical sensors, actions and contexts were introduced by Atkin et al. [1] that allow more complex behaviors and also group engagement. Group behavior has also been thoroughly investigated in [14, 24]. Musse and Thalmann [15] presented a hierarchical model for simulating virtual human crowds.

Our approach presents weighted multiple inheritance and allows for the definition of convenient complex and heterogenous group behavior. Section 3 gives an overview of the concept, the core components are described in Section 4. Section 5 presents the method for generating agents.

3. System Overview

Our system is comprised of three main components as depicted in Figure 2. First, the game engine acts as the main controller of the framework. Second, each agent controlled by the agent engine has a well defined sense-think-act cycle which is the same for every agent. The difference between two agents is generated from the associated knowledge base components and is therefore agent-centric. Therefore, the common knowledge base is the third main component of the system. It stores the complete knowledge of every agent and, since it can contain shared objects, provides access for other agents thereby enabling collaborative behavior. The knowledge base components are agent representations, sensors, situations including actions and conditions, and attributes. The sensors are allowed to query the game engine to gather information through a sensor interface, while actions can use the effector interface to change values such as orientation or the actual state, e.g. walking or running.

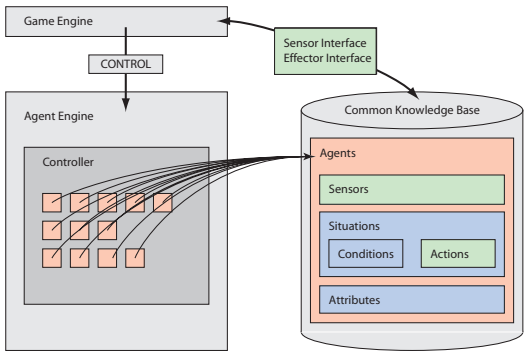

Figure 2: *The core components of the agent system.*

The engine is intended to support a game simulation without lowering the frame rate below a given value. We therefore allow the system to run for only a few milliseconds during the simulation cycle of the world and then return the control. Thus, we primarily expect to stay over a minimal frame rate independent of the underlying hardware resources and to have short simulation cycles independent of the number of agents simulated by the agent engine. An increase in the quality of behavior is achieved when faster hardware is used and the frame rate remains the same.

4. Core Components

In this section, we introduce the core modules of the system. We start with the agent engine itself before explaining the main components of a single agent. Finally, we describe the knowledge base and its components.

4.1. Agent Engine

The agent engine is the main controller for all agents. For each run, the engine receives a time frame during which it is allowed to simulate any number of agents depending on their time consumption. The controller maintains a list of all agents and decides which agents should be activated during a run. Since it can run only a short time during each cycle, we have to assume that not every agent can be activated on each run. Currently, we use a round-robin controller that traverses all agents regularly. In the future, a more advanced scheme might take into account that agents out of sight or at a large distance do not have to receive as much computation time as the agents which are closer to the user or camera.

4.2. Agents

Every agent is responsible for one simulation instance. It runs the same cycle independent of the associated knowledge for every instance.

On each call, the agent first senses the environment by traversing its associated sensors to gather agent-specific information, for example the nearest neighbor or the position of another agent. Every sensor gathers its information and updates the values in the knowledge base of the agent to make it available. The sensors have to be predefined and can be parametrized through attributes.

Subsequently, the *situation awareness module* is used to determine the current situation of the agent by traversing all associated situations. A situation is a state that can be determined by using internal and external knowledge. A situation has a probability that indicates how much it needs to get activated. It therefore checks all associated conditions and only if all conditions return *true*, it determines the probability of this situation. Therefore, each situation has to provide a method for determining its probability using the current knowledge of the agent. The situation that returns the highest value is selected and queried about which action to carry out. Each situation can have several associated actions. The best matching action is then selected to be carried out and therefore inserted into the action queue.

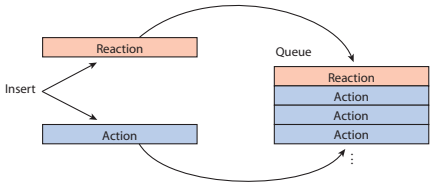

Figure 3: *Inserting an action.*

The action queue offers two different methods for inserting actions, one for reactions and one for normal actions. This is displayed in Figure 3. Since a reaction should be executed as fast as possible, we place it on the top of the queue. Other actions, for example path-finding actions, are placed at the end of the queue in order to maintain a sequential order. The execution system fetches the top item of the queue for the next action.

Every action provides methods for initialization, execution and termination. Additionally, it can have a set of preconditions which are tested before it is activated. An action ends when all postconditions have been reached and/or the duration has been exceeded. Afterwards, it is removed from the action queue.

4.3. Common Knowledge Base

The common knowledge base is the central component of our system. It stores the knowledge of all agents in one large container. The knowledge base provides several basic types such as KBAgents, sensors, situations, actions, conditions, and attributes. Every type has a factory for generating new instances of the desired type. Additionally, we created a container for each type which can store any number of instances of this type. Thus, we made each type an *attribute container* and can therefore add any number of attributes to every type.

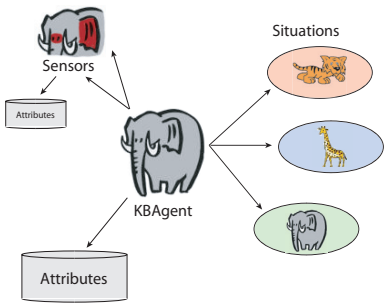

Figure 4: *An example of an agent. It has sensors to perceive and situations to recognize.*

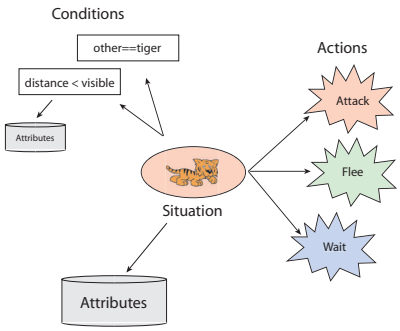

Figure 5: *An example of a situation for a simple agent. It has access to conditions and actions.*

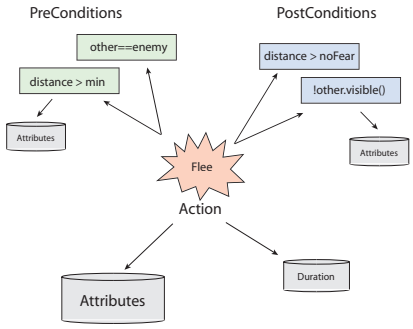

Figure 6: *An action needs preconditions, postconditions, or a duration.*

Currently, we support attributes which can store either an integer, float, boolean, vector, or string values. Every attribute excluding the latter, also supports random values. The user can specify minimal and maximal values or a probability in the case of the boolean type. Each attribute has an evaluation type which determines if the attribute is evaluated only once, during the initialization, or every time the value is requested. The user can also re-evaluate the random attributes on demand even if the attribute should be evaluated only once.

Every agent within the engine is associated with one KBAgent instance which works as a container for the complete knowledge of this instance, shown in Figure 4. A KBAgent consists of sensors for perceiving the environment, some attributes, and certain situations that can be recognized. A situation is recognized, when its associated conditions evaluate to hold. As Figure 5 illustrates, a situation has multiple *actions* and also some attributes. When one of these actions is selected according to the situation and knowledge of the agent it is inserted into the action queue. Before an action is executed, its preconditions are tested. It ends, when either the duration has been exceeded or the postconditions have been fulfilled. Figure 6 exemplifies an action. Currently, the framework does support neither hierarchical actions nor hierarchical situations. With very little effort, the action system could be enhanced to support both hierarchical actions and situations.

In order to enable sensing the environment and executing actions, the knowledge base has a sensor interface which provides hooks for the global simulation. On one hand, this environment has to provide methods for gathering the required information, for instance the position, orientation, velocity, or neighborhood information. On the other hand, the effector interface is used to set values such as orientation and velocity. A simulation in which the position is physically correct according to velocity and orientation is expected, therefore, the position cannot be set by the agent.

5. Agent Generation

This section presents the agent generation process, a preprocessing stage, in which all instances are created and linked together. It is based on descriptions in an agent definition language. Like a construction set for children with which complex buildings can be built out of simple bricks, we create more sophisticated behavior by combining simple basic components.

5.1. Definition of Agent Instances

Our agent definition language uses XML and enables the user to compose new types of behavior for instances and generate groups of instances. New behavior is not scripted in this language, rather with the aforementioned concept of attributes. With this method, the user can easily use attributes to parametrize all knowledge linked to an agent. That assumes a set of generic behavior patterns that can be individualized by attributes.

An *agent instance* is a single agent and can be either real or abstract. Real agents will actually get linked to an agent in the engine as a simulation object, while abstract agents just represent templates for other agent instances and are not intended to be activated at any time. An instance can inherit its knowledge from one or multiple parents or it can create a new type of agent by specifying the knowledge base components which must be used to create it. Since our intention is to create many agents with different behavior, the inheritance is also weighted to adjust the behavior of the new instance by manipulating the evaluation values of situations. During inheritance, the return value is multiplied by this weight. Thus, we can easily define abstract basic behavior agents and compose new agents by thoroughly weighting the basic agents into a new one. Figure 7 illustrates this process. Therefore, situations should have a continuous return function which is not clamped to values in [0..1]. This is automatically done by the situation awareness module during run-time.

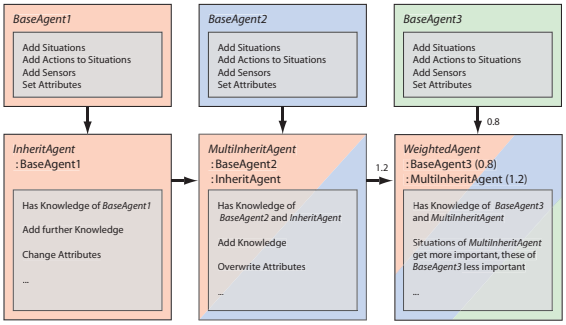

Figure 7: *Composing agents using building blocks of simple basic behavior.*

Agent groups are introduced in order to build collaborative behavior. Agent groups represent more than one instance where the group instance is the parent of all further group members. As a single instance, an agent group can also be abstract and provide a powerful template. Abstract groups only have to provide templates for a future instantiation of the group and its members. Non-abstract groups have to generate the explicitly specified number of children.

First, we look at simple groups. A group is defined similar to an agent instance, however, the instances of the group can be additionally specified. We can specify additional attributes, situations, sensors, etc. And, we can also enhance the instances by inheriting knowledge from other agents, as described in the above section on single instances. Additionally, we can do this for any single instance individually.

For example, we can specify a family by adding a father with male and adult behavior to specify his knowledge and behavior, adding an instance which inherits female and adult behavior as the wife. In an abstract group we could also specify that every additional instance should inherit child behavior. When, at a later stage, we realize this abstract pattern with an instance count of five, the engine generates two parents with three children as depicted in Figure 8. Since the group itself is also an agent, we can add family behavior, for example, grouping at lunch time. The group could then inform all instances to gather at the family house.

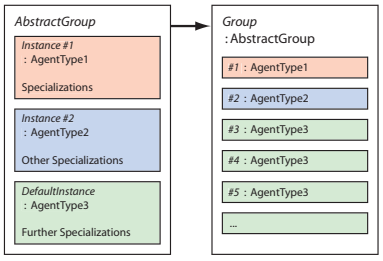

Figure 8: *A simple group with specializations and its instantiation.*

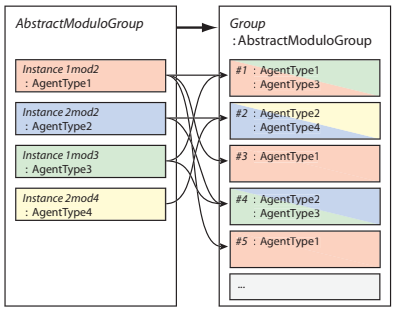

Figure 9: *A modulo group and its instantiation.*

To enhance the group building process, we introduce *modulo rules* which can be used to easily specify regular patterns in large groups. With these rules, we can specify for example that every second member of a group should be male, while every other should be female, or every tenth should be able to do a certain task. Since all members of a group are consecutively numbered, we can easily apply such rules during creation. For every modulo specification, we create a new agent with this specific behavior. During the generation of the real instances, we always check if there is a modulo group matching the current counter and copy the modulo groups knowledge into the current instance as shown in Figure 9.

A key element of organized groups is *recursive definitions*, which are only allowed within abstract groups. If one or more instances of an abstract group inherits the knowledge of the group itself, it is considered to be *recursive*. During generation, this instance inherits the knowledge of the group to which it belongs and is able to generate instances by itself as a sub-group. For example, if an abstract group defines two instances out of three in order to inherit the group, a concrete instantiation of this group with 21 members would generate a group according to Figure 10.

5.2. Single Instance Generation

In order to generate a new instance of a single agent, the following steps are always carried out, including abstract agents:

```
Agent CreateAgent(definition)
{
   // create a new agent with a unique ID
   Agent a = Engine.CreateNewAgent();

   // use the default parent to construct a base KBAgent
   if (definition contains default parent)
      a.KBAgent = DefaultParent.clone();
   else
      a.KBAgent = DefaultAgent.clone();

   // add the knowledge of other parents with their weight
   for each (parent in definition)
      a.KBAgent.add(parent, parent.weight);
```

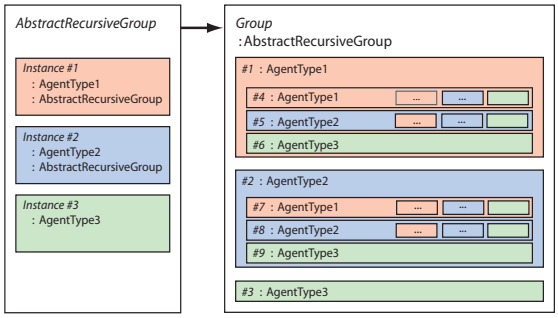

Figure 10: *A recursive group and its instantiation.*

```
    // load and init attributes, sensors, situations, actions, and conditions
    for all (knowledge in a.KBAgent) {
        knowledge.load();
        knowledge.init();
    }

    // add KBAgent to knowledge base
    KB.Add(a.KBAgent);

    // add the agent to the agent engine
    Engine.addAgent(a);

    return a;
}
```

These steps are repeated for every agent instance. Abstract agents are removed from the engine after the preprocessing steps.

5.3. Group Instance Generation

Abstract and real groups are generated differently. Abstract groups only have to prepare the neccessary abstract instances to generate any real instance of the group and thereafter of the group members. This is independent of the number of instances a concrete instantiation will have. The following pseudo code shows the preparation of such instances:

```
PrepareInstances(group-definition)
{
    // create the group representative and parent of all members
    AgentGroup group = CreateAgent(group-definition);

    // create specialized member templates
    for each (specification in group-definition)
        AbstractAgent spec = CreateAgent(specification);
        KB.Add(spec);

    // create modulo rule based templates
    for each (modulo rule in group-definition)
        AbstractAgent modulo = CreateAgent(modulo rule);
        group.AddModuloRule(modulo rule);
        KB.Add(modulo);

    // create default template
    AbstractAgent default = 0;
    if (group-definition contains a default specification)
        default = CreateAgent(default specification);
    else
        default = CreateAgent(group-definition);

    KB.Add(default);
    KB.Add(group);
}
```

This procedure generates an abstract instance for every possible base type of this group. This includes the group itself, all specifications, modulo rules, and the mandatory default instance. These are added to the knowledge base to make them available for a later instantiation of the group.

The steps to create a non-abstract group are very similar since such groups can also specify further knowledge. The number of instances has to be explicitly denoted. First, a group representation instance is generated. Then, every member is generated by loading the globally specified settings and overwriting these with the settings specified for all instances. If the parent has modulo rules, the appropriate rules for this specific instance are used to overwrite the settings. In the case that the current instance is specified explicitly, these settings are applied in a final step. Each member stores a reference to the group representation as parent. For recursive groups, the ID of the parent is calculated based on the number of recursive agents in the group template. Therefore, the hierarchy defined through a recursive pattern is maintained. In each case, every instance has a parent and can access or modify its knowledge directly.

5.4. Group Behavior

To simplify group behavior, we introduce an attribute which stores a reference to an agent or an agent group. We can use this method, for example, to denote a leading instance. By using this reference, we can easily access the knowledge of this instance and make the decisions dependant on this information. The name of the target has to be specified in the definition and is updated during instantiation from the according abstract group. It is replaced by the name of the group member which has been instatiated from the originally specified target. If members of a recursive group have such references to other members of the recursive group, they will be automatically set to the according instance in the hierarchy tree.

It is easy to implement, for example, a leader-follower situation which uses a reference to a leading instance, an allowed distance, and a path distance as attributes. This situation is illustrated in Figure 11 including the actions and attributes. If the allowed distance is exceeded, the agent will move directly towards the leader or to a point within the neighborhood of the leader (Leader.Pos + Δ). Even if the leader moves around, the agent will automatically adapt its direction. However, it is possible that an instance loses contact to its leader due to an obstacle, unfortunately this situation is unavoidable. If the distance exceeds the path distance, the agent will use the path planning unit to find a correct path and follow it. Although this path will not adapt to the positional changes of the leader, it is necessary to find a way out of such deadlock situations. Additionally, the instance can inform the leading instance about its situation and slow it down to catch up. Thus, members of a group can

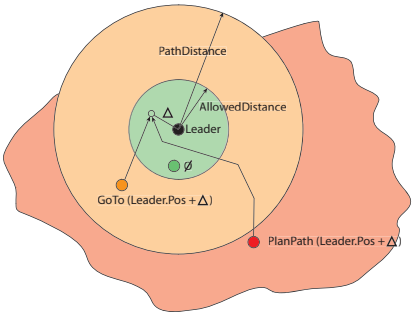

Figure 11: *The leader-follower situation with attributes and actions.*

follow the group instance and and, therefore, automatically stay together. This reduces path planning costs within the group to a minimum since only the group has to regularly plan a path. The others will automatically follow their leader and only need to plan a path when their leader is too far away.

6. Results

We demonstrate the usefulness of our approach by various examples which exemplify complex behavior that has been created by using simple base definitions. We assign a time-per-run of 5 ms to each simulation step. All our experiments were computed on a Pentium IV 2.8 GHz PC with 2 GB main memory and a GeForce4 graphics accelerator. As can be seen in the accompanying video, the simulation features frame-rates above 100 Hz for several hundred agents. The test scene is a hilly terrain with a few lakes that serve as obstacles, as depicted in Figure 12. All agents in our examples inherit a base behavior that prohibits them from walking across lakes or from leaving the terrain. Most of the examples were coded in less than a hundred lines of XML definition.

Figure 13 shows a group with specialized members created using 3 modulo rules. The rules divide the group into two parts, color coded with red and blue. Furthermore, the color of every third member is overwritten with white.

We made use of specialization to implement the generic family presented in Figure 1. Family members include a father, a mother and a number of children, as illustrated in Figure 14. Although we employed the same models to render the adult animals, their behavior is different. This is highlighted using color codes in Figure 15.

A larger herd is shown in Figure 16. In this example, we created a multilevel hierarchy by recursively defining the group including leader-follower behavior. The color codes in Figure 17 reveal the type of agent and its behavior. Red agents are leaders at different levels, while the blue ones are followers of a particular leader. Due to the multilevel hierarchy the red leaders on a lower level are as well followers of leaders which belong to higher levels. The three frontmost leaders are followers of the group instance which is not visible. The thick black line indicates a preplanned path of the group instance, whereas the thin one represents its actual trace. They do not match exactly because we adapted the orientation, hence, the path was adapted to the slope of the terrain. In this case, only one path planning action has to be computed in order to guide the entire group to its goal. In a flat, unorganized group without hierarchy each individual would have to plan its own path. Since path planning is computationally expensive, our hierarchical approach is much more efficient. Path computation in our example costs about 1 ms. If every group member were to individually plan a path, the overall computational costs would amount to 70 ms. This takes 14 full engine cycles for path planning given the 5 ms time limitation, as compared to 1 ms which we have achieved with our method.

Figures 18 and 19 show a large scenario with approximately 700 agents featuring all the different types of behavior presented thus far. Although only two thirds of the agents can be simulated during the assigned 5 ms time-per-run the frame-rate lies well above 100 Hz. Such real-time performance would not be feasible using a flat, unstructured simulation.

The first part of the video presents a real-time animation of the basic behavior of independent animals moving aimlessly around. They all avoid lakes and the border region of the terrain. Their behavior is defined using two simple modulo rules. All agents inherit the base type, where half of them has a higher weight for the water avoidance situation. Additionally, the velocity attribute is different for the two subgroups. Further animations show the scenarios presented in Figures 14, 16, and 18. In the last example, the number of agents exceeds the controller's time limit. As a result, every agent cannot be simulated during a single run. Nonetheless, we can not detect any substantial decrease in the quality of the resulting behavior.

7. Conclusions and Future Work

We have developed a framework for autonomous agents with reactive behavior. It supports behavior definition for a wide range of different behaviors through weighted multiple inheritance and specification. The system provides mechanisms for group behavior and group definitions. The collective behavior is enabled by automatically updating references to other agents, for example to a leader. Groups can be defined by specifying individual instances separately or by creating default instances. Modulo-based patterns allow for the enhancement of behaviors in parts of the group. Additionally, recursive definitions can be used to hierarchically organize groups. Through the specification of a maximal time-per-run, our framework guarantees minimal and constant frame-rates for up to several hundred agents. We believe that the presented concept is especially useful

for interactive simulations involving multiple agents, such as games.

In the future, we plan to investigate more sophisticated scheduling algorithms. In addition, time-dependent attributes such as interval-based epistemic fluents [6] would allow for the enhancement of the presented behaviors.

References

1. M. S. Atkin, G. W. King, D. L. Westbrook, B. Heeringa, and P. 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.
2. L. Blando, K. Lieberherr, and M. Mezini. "Modeling behavior with personalities." In *International Conference on Knowledge and Software Engineering*, 1999.
3. B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M. P. Johnson, and B. 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.
4. J. Broersen, M. Dastani, J. Hulstijn, Z. Huang, and L. 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.
5. D. Canamero. "Modeling motivations and emotions as a basis for intelligent behavior." In W. L. Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents, New York, NY*, pages 148–155. ACM Press, 1997.
6. J. Funge, X. Tu, and D. 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.
7. S. Grand, D. Cliff, and A. 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.
8. J. Gratch and S. Marsella. "Tears and fears: modeling emotions and emotional behaviors in synthetic agents." In *Proceedings of the fifth international conference on Autonomous agents*, pages 278–285. ACM Press, 2001.
9. R. Grzeszczuk and D. 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.
10. R. Grzeszczuk, D. Terzopoulos, and G. 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.
11. N. Hawes. "Real-time goal-oriented behvaiour for computer game agents," 2000.
12. C. Isbell, C. Shelton, M. Kearns, S. Singh, and P. Stone. "A social reinforcement learning agent," 2001.
13. G. 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.
14. S. R. Musse and D. Thalmann. "A model of human crowd behavior." In *Proceedings of the Workshop of Computer Animation and Simulation of Eurographics'97*, 1997.
15. S. R. Musse and D. Thalmann. "Hierarchical model for real time simulation of virtual human crowds." *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001.
16. K. Perlin and A. Goldberg. "Improv: A system for scripting interactive actors in virtual worlds." *Computer Graphics*, 30(Annual Conference Series):205–216, 1996.
17. C. 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.
18. S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. PrenticeHall, 1996.
19. SimCity. *Electronic Arts Inc. (EA)*. http://www.simcity.com.
20. K. Sims. "Evolving virtual creatures." In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM Press, 1994.
21. T. Sims. *Electronic Arts Inc. (EA)*. http://thesims.ea.com.
22. D. Terzopoulos, T. Rabie, and R. 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.
23. X. Tu and D. Terzopoulos. "Artificial fishes: Physics, locomotion, perception, behavior." *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.
24. B. Ulicny and D. 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.
25. I. Wilson. "The artificial emotion engine." *2000 Spring Spymposuim on AI and Interactive Entertainment*, 2000. (available only online)
26. S.-Y. Yoon, B. M. Blumberg, and G. 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.

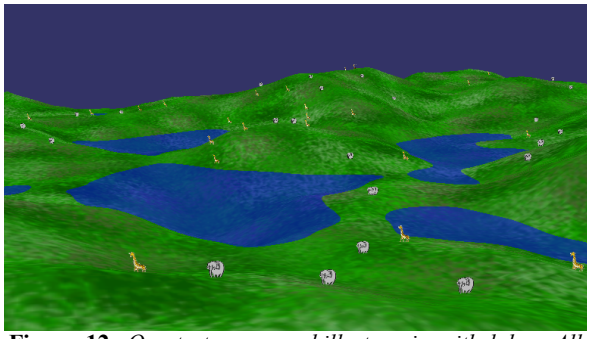

Figure 12: *Our test scene: a hilly terrain with lakes. All agents inherit lake-avoidance as a basic behavior.*

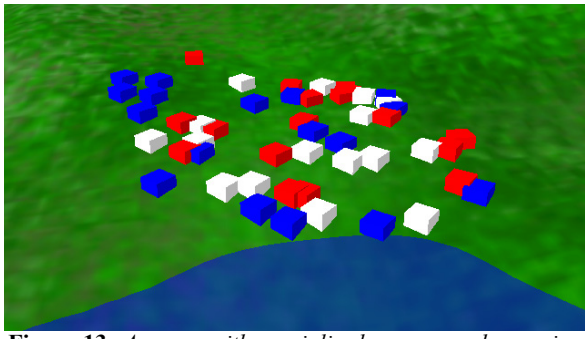

Figure 13: *A group with specialized group members using modulo rules. The colors illustrate the various types of agents.*

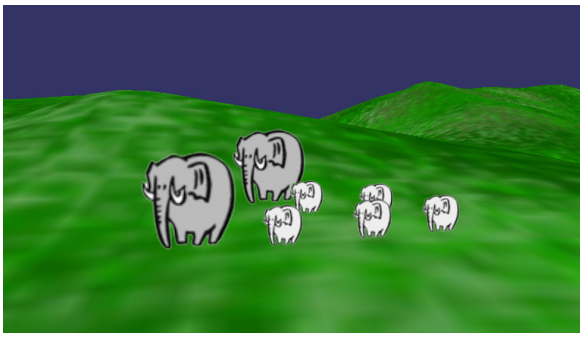

Figure 14: *A close-up of a family with two parents and five children. Their behavior forces them to herd.*

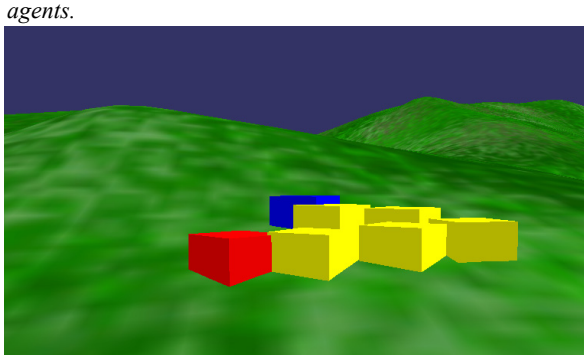

Figure 15: *Family in color codes. The father (leader) is red, the mother blue, and the children are yellow.*

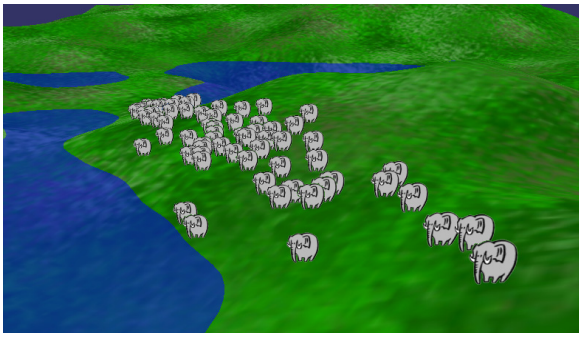

Figure 16: *A recursively defined group with hierarchical leader-follower behavior and 70 members.*

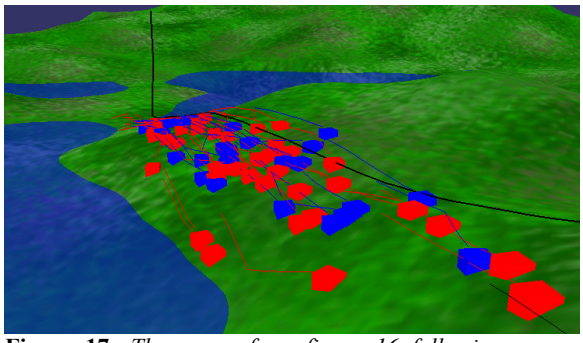

Figure 17: *The group from figure 16 following a pre-planned path (black). Leaders are red, followers blue.*

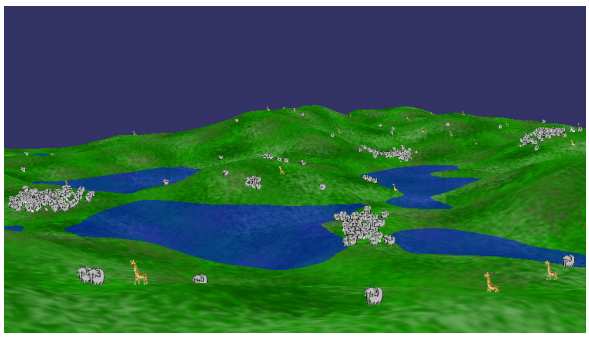

Figure 18: *A full scene with approximately 700 agents with complex behavior.*

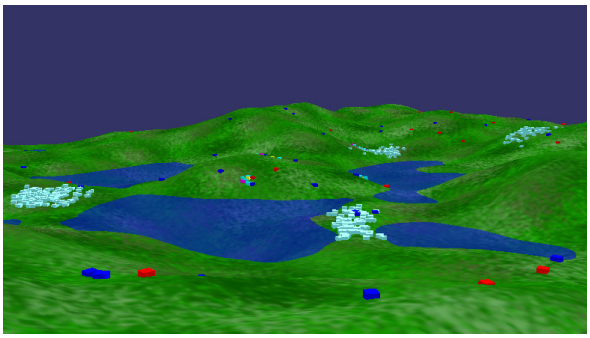

Figure 19: *Same scene as figure 18. The groups and individuals are color-coded to illustrate the diversity.*