Spark — A Generic Simulator for Physical Multi-agent Simulations

Oliver Obst and Markus Rollmann
Universität Koblenz-Landau, AI Research Group,
Universitätsstr. 1, D-56070 Koblenz
{fruit,rollmark}@uni-koblenz.de

March 21, 2005

We describe a new multi-agent simulation system, called Spark, for physical agents in three-dimensional environments. Our goal in creating Spark was to provide a great amount of flexibility for creating new types of agents and simulations. To achieve this, we implemented a flexible application framework and exhausted the idea of replaceable components in the resulting system. In comparison to specialized simulators, users can effortlessly create new simulations by using a scene description language. Spark is a powerful and flexible tool to state different multi-agent research questions. It was already used as official simulator for the first three-dimensional RoboCup Simulation League competition. We present the concepts we used to achieve the flexibility in our system and show how we seamlessly integrated the different subsystems into one user-friendly framework.

1 Introduction

Simulated environments are a commonly used method for researching artificial intelligence methods in physical multi-agent systems. Simulations are especially useful for two different types of problems: (1) to experiment with different sensors, actuators or morphologies of agents and (2) to study team behavior with a set of given agents. Additionally, the connection between both types of problems is an interesting research problem.

To address each of these problem types without simulators, the actual hardware would have been to be built and set up in several experiments. Doing so with a number of real robots is often an expensive and also a difficult task because of the amount of parameters generally involved. For many approaches, like for instance in machine learning, experiments have to be repeated a great number of times.

In this paper we describe a multi-agent simulation system, called Spark, for physical agents in three-dimensional environments. The purpose of Spark is to provide a framework for building various simulations: it is a generic tool rather than a specific simulation. The purpose of a simulation build with Spark is determined by the designer who is using Spark. Simulations can be used to address all of problem types mentioned in the first paragraph. It was our goal to provide a great amount of flexibility for simulation designers, so that it is possible to choose how much attention should be paid to each of these problems. We show how we achieved this flexibility by exhausting the idea of replaceable components in the underlying framework.

The flexibility of Spark as a tool comes together with a user-friendly way to create simulations by using a scene description language and pluggable components. For users of the system creating agents for a given simulation it is interesting that they do not need to know internals of the system because agents are decoupled from the simulator. To achieve reliable and reproducible results, we integrated prior work from both physical and multi-agent simulation research into Spark.

This paper is organized as follows: Section 2 describes the related work and similar approaches. Section 3 contains an overview over the system and its most important components. In Section 4, we explain the basic architecture of Spark and how we achieved the flexibility of the system. The way agents and simulator interact along with the components responsible for this is described in Section 5, and Section 6 contains a desciption of the physical side of the simulation. Section 7 presents the scene description language, a language wich can be used to describe the setup of simulations in Spark, before we conclude in Section 8.

2 Related Work

A large number of simulators has been developed in both multi-agent and robotics research. From the multi-agent perspective, the primary interest is usually to study team behavior. In this domain, RoboCup Simulation League [1] is a prominent benchmark, where teams of soccer playing robots try to win against each other. In Soccer Simulation League, two teams of 11 autonomous agents compete in a two dimensional, discrete-time simulation. From the first official RoboCup competition[1] up to today the simulator [2] has continuously been enhanced, but its limitation to a two dimensional world remained.

Beyond this, a physical multi-agent simulator can be useful for example to research the interdependency between single agent abilities and team behavior, for instance when a group of robots has to move in areas with obstacles which can be avoided or removed cooperatively. A sample application for Spark different from robotic soccer is for instance research in genetic algorithms for controlling robots, where a large number of individuals is used to find a solution for a given problem.

On the RoboCup 2003 Symposium, we proposed a new approach to a three-dimensional physically realistic soccer simulator [3]. This system was a prototype of the simulator we

[1]The first Robot World Cup Soccer Games and Conferences were held in conjunction with the International Joint Conference on Artificial Intelligence (IJCAI) in 1997.

describe here, however the specific features for reproducible and distributed simulations, simple construction of articulated bodies and the scene description language we describe here were missing. For RoboCup 2004, our system was successfully used for the first official competition in RoboCup Simulation League 3D (see also [4]). It is planned to continue RoboCup competitions with a simulation built using Spark.

An entire different tool to study the behavior of a large number of agents in two- or three-dimensional continuous virtual worlds is XRaptor [5; 6]. For XRaptor, an agent is either a point, a circular area or a spherical volume. A detailed physical simulation is not supported by XRaptor, though in principle possible. The agent processes are not entirely decoupled from the simulation loop, unlike in Spark. Consequently, XRaptor is primarily useful for reactive agent types.

For roboticists, the primary purpose of a simulation system is often to set up reproducible experiments and provide prototyping environments for mobile robots. Some of the existing simulators are tailored to specific robots platforms, most however address a number of robot types. The simulators below fall more or less into this category.

Webots [7] is a commercially available mobile robotics simulation software that is intended as a rapid prototyping environment for modeling, programming and simulating mobile robots. It includes robot libraries that allow the direct transfer of control programs to existing mobile robots platforms. Like Spark, it uses the ODE library for accurate physics simulation. It comes with tools for visualization and for editing properties of objects in the world. The focus of Webots is the accurate modeling of existing robot platforms. This affects the level of abstraction of the provided sensors and effectors. These are low level in order to match their real life counterparts. In this type of simulation a major part of the robot's job is the classification of sensor data for self-localization and obstacle avoidance. In comparison the focus of Spark is more towards general principles of multi-agent research as for example coordination or learning in multi-agent systems.

ÜberSim [8] is a simulator specifically designed as a robot development tool for the RoboCup small-size soccer league. It uses a fixed level of abstraction to model the perception and action interfaces for the simulated robots. It provides a set of predefined robot models and can be parameterized only at compile time. Like Spark, ÜberSim is an Open Source project and uses the ODE physics library.

M-ROSE [9] is a 2D simulator used for the rapid development of robot controllers. It features a three step approach for learning a desired controller behavior. First the individual motion profile of a robot is learned using a neural net. The learned profile is then the basis for a simulator specialized for this robot type, in which the controller learning tasks are performed. The trained controller is then transfered to the real robot to validate its performance. This simulator is specialized for the development of controllers for robots with realistic sensor inputs. The approach is quite different to ours in that it lacks features like for instance a full physical collision detection.

The ultimate simulation system addresses all of these questions, and the direction Spark is aiming for. Admittedly, it does so from the multi-agent side of the spectrum, because with RoboCup Soccer Simulation League as one implemented application this is where its origin lies. With our underlying physics system and the way sensors and effectors are realized, simulations built with Spark are not constrained to high-level

abstractions of multi-agent systems.

3 System Overview

In this part of the paper we present an overview over the simulation system and the very basic parts of Spark.

One of the first goals in developing a new simulator was to create a flexible application framework, which facilitates exchanging single modules and extending the simulator. Both aspects are important to create different simulations for instance by exchanging or adding protocols, entire robot models or single parts of simulated robots, rules of the simulation, or log file formats. The framework we created is called *Zeitgeist*. It is the central component of the system as shown in Figure 1, and provides access to all services of the simulator.

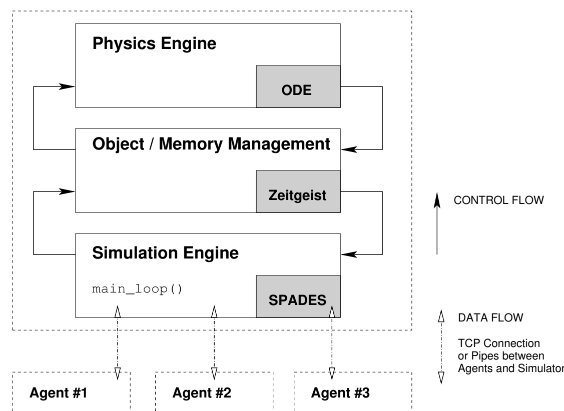

Figure 1: Simulator Overview

Zeitgeist is implemented in a strictly object-oriented fashion using `C++`. Almost all classes we implemented are derived from class `Leaf`. Instances of these classes can be stored in a tree during run time, the basic data structure we are using to handle objects. There are basically three different kinds of objects managed by Zeitgeist: 1. objects representing the simulated situation, the current *scene*, 2. objects encapsulating the central functionality of the simulator, so called *servers* and 3. factories for creating new objects based on strings. This feature is important for flexibility and extensibility, because it allows to change the simulator setup to be configured by script languages or by text based configuration files and the use of plugins without recompilation of the entire simulator.

For changes with no recompilation of the simulator code, it is possible to attach custom scripts to all objects handled by the Zeitgeist object management system. The script language we are using for this is Ruby, an object-oriented scripting language. Using Ruby, we also implemented a Lisp-like language for describing simulation scenarios.

A physics engine as second main component of Spark is responsible for updating locations and velocities of simulated objects according to a physical model. Instead of

implementing a physics engine by ourselves, we are using ODE [14], the Open Dynamics Engine. ODE supports simulation of simple rigid bodies and complex objects, i.e. simple bodies connected using joints.

The purpose of the Spark simulator is to provide a common platform for different simulations of physical agent systems and their environment. Goals of such simulations are for instance the development of controllers of real robots, but also research in multi-agent team behavior for agents in physical environments. The interaction between simulator and agents and the main control loop of the simulator is part of the simulation engine, the third main component of the system.

For scientific simulations it is important to carry out reproducible experiments. Using a larger number of agents, and also using several machines to host agents involved in the simulation interferes with the reproducibility of experiments because factors like network traffic, latencies, and available CPU power may vary from experiment to experiment. The most simple and also the fastest way to deal with these properties of a distributed simulation is to ignore them, and in fact we implemented a simulation engine that does exactly this. In order to be able to run reproducible simulations, the simple simulation engine can be replaced by a publically available middleware for agent-based simulation which is called SPADES [13]. SPADES keeps track of the "thinking" time of agents, takes care of the event handling between agents and simulator and guarantees reproducible results for distributed simulations.

4 The Zeitgeist Application Framework

One of the first implementation steps in building Spark was to create a flexible application framework, called Zeitgeist. Zeitgeist was invented[2] as application framework for the simulator, but has also been used successfully to create other applications such as software agents and monitors for the simulation. This section contains a description of the Zeitgeist application framework. The following subsection contains an overview over the different objects and components managed by Zeitgeist. In Section 4.2, we describe details of the class object system responsible for the necessary flexibility of Zeitgeist and some of its central components. The last part of this section identifies the underlying pattern of the framework.

4.1 Zeitgeist Basics

Zeitgeist is a framework for handling data objects and functional components of a system in a uniform way. All objects managed by Zeitgeist are derived from a class `Leaf`. Managed objects are stored in a tree-like data structure, and can be accessed by a string representing a path in the tree. To this end, each object stores its node name along with references to its parent and its child nodes. Based on these means we have a flexible mechanism to locate and reference objects at runtime: Given a path expression, similar to that used in a file system, Zeitgeist is able to retrieve the corresponding object

[2] original implementation by Marco Kögler, see also [3]

instance. Figure 2 is a class diagram of the basic Zeitgeist classes.

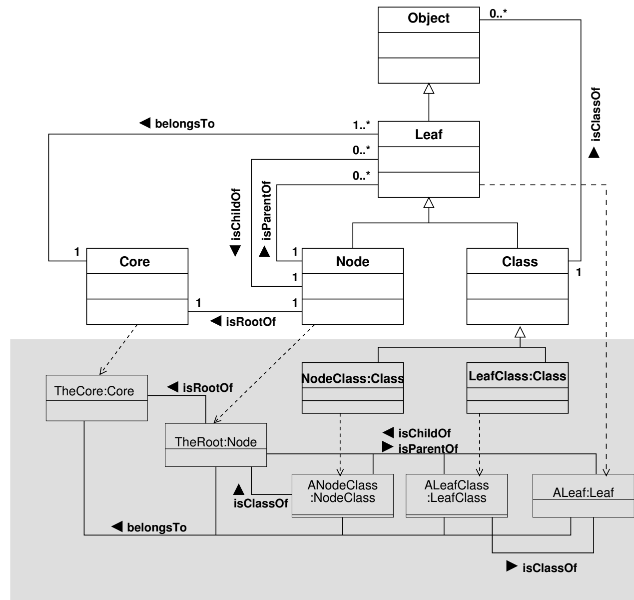

Figure 2: Class diagram with the classes responsible for the object hierarchy. The shaded part contains sample classes and instances.

All objects related to the physical simulation are stored below a special node in the tree, the *scene* node. We refer to the subtree starting with the scene node as *scene graph*. In the scene graph, specific groupings of objects express their responsibility for each other. This enables the objects to automatically care for the proper interaction for instance between the geometry and the physics of a given entity. When moving nodes in the scene graph, subnodes maintain their relative position to their parent node. This feature is useful to arrange groups of objects.

Regarding managed objects as resources, Zeitgeist implements a concept called pathname space mapping. Pathname space mapping appeared in the QNX operating system and has been used to realize the QNX resource manager concept [12]. Resources are addressed by a path through the hierarchy given as string. With this concept, it is not only possible to store and access data of simulated entities in the tree, but also functional components of a system. Central functional components of Spark are called *servers* in our terminology. Servers are simply objects installed somewhere in the object hierarchy; they expose their functionality at locations which are known to other functional components. They access the server object at the known location and call the respective methods. Figure 4 contains a sample setup of Zeitgeist with a few selected nodes as occurring during a simulation run. The scene graph contains a representation of a ball and a car consisting of several components. Some of the servers providing the main functionality of the simulator can be seen on the top half of the figure. They are responsible for providing access to the simulated scene, for logging messages, for sending output to connected monitors, the agent management, and providing access to files respectively.

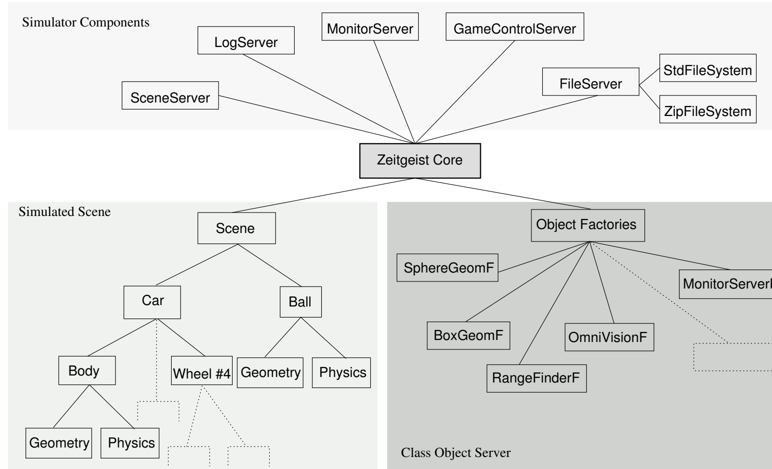

Figure 3: Zeitgeist Object and Memory Management

Object factories are the third kind of objects stored in the tree; they are key to extensibility of our framework. In Figure 4, they can be identified in the lower right corner. This part of the system can be seen as special kind of server, where the objective is to create new objects based on strings during run time. It is called the Zeitgeist class object system and subject of the next section.

4.2 The Zeitgeist Class Object System

Together with support for loadable modules and an interface to scripting languages, object factories are responsible for the ability to extend the system without recompilation. The combination of the object factories in conjunction with pathname space mapping can be used to create objects of classes that are unknown at compile time of the simulation system. This feature is useful because additional functionality can be added to the system by adding plugins. It is possible to get instances of custom classes realized as plugins via configuration scripts. These instances are then installed as servers. Zeitgeist makes further use of the pathname space mapping concept when the implementation of services is delegated from servers to helper classes. In the object hierarchy, these helper classes are installed immediately below the server node. This leaves the server object as a lean mediator to several exchangeable sub-services with one common interface.

Sample uses of these concepts are the monitor server and the file server. The monitor server is responsible for sending data to a program visualizing the simulated scene, implementing a specific protocol. During simulator setup, an initialization script creates the monitor server instance of the desired type at a predefined location. During run time, the simulator sends its data to the monitor server at this location. To change the monitor protocol, a new type of monitor server has to be implemented along with a factory creating monitor server objects of the new type[3]. The new monitor server

[3]The Zeitgeist framework supports creating factories from ordinary classes with simple macro calls.

factory is installed as plugin, and the initialization script can be changed to create the new monitor server instance.

The file server is an example for delegating tasks to helper classes. The file server is a service that provides access to various mounted file systems. File systems are realized by objects implementing the file system interface used to access different file stores, like the standard file hierarchy of the operating system or like a file archive contained in a zip file. The file server implementation provides a single interface to transparently access different file system objects. During simulator run time, it is possible to create the file server and required file systems by using the file server factory and file system factories. The created file server is linked into the object hierarchy at a known location and the created file systems are installed directly below the file server. The great flexibility of the Spark system stems from the fact that all services have been implemented in this fashion. Adding this kind of flexibility does not add much overhead to the system: the lookup of the objects in our framework usually happens during initialization time and is cached by ordinary pointers.

4.3 Reflective Factory with Object Hierarchy

The basic pattern employed to implement the Zeitgeist Class Object System is the reflective factory pattern [10], also known as the Class Object pattern. It allows the factory based instantiation of objects at runtime, given the class name as a string. Products of the factory are instantiated classes. The instances maintain references to the factory that created them; this distinguishes the reflective factory pattern from the abstract factory pattern [11]. It enables every object to access meta data stored in the associated class object at runtime.

We exploit the knowledge about these relationships to store class names and information about supported interfaces in class objects. At runtime, we can query objects for their type and for supported interfaces. Script languages can access all Zeitgeist objects and call their methods, so that the information which methods are available is essential for scripting. In object oriented programming languages other than `C++`, such as Objective C, Smalltalk or Ruby, this kind of meta data is natively available. We have chosen `C++` as primary implementation language anyway, because it provided the most freedom in integrating external libraries. For instance, the agent middleware system we are using offers only a `C++` interface. The drawback of using `C++` here is that the meta information has to be added manually, which makes the whole pattern, like the original reflective factory pattern, special to `C++`. In Zeitgeist, this can be done using macro calls we provided for this purpose.

In the shaded part of Figure 2,

5 Central Simulator Components and Control Flow

In the previous section we have shown how simulated entities and simulator components are managed in Zeitgeist. In this section, we explain the necessary components to actually

run a simulation. For an entire simulation, the simulator, agents, and monitors to watch simulations are all different processes that have to work together.

The part of the system that contains the run loop and does the event management is the simulation engine. It cares for the timing, and controls the communication between the simulator and external processes. On this level of communication, agents and simulator use a meta protocol to register and unregister agents and to encapsulate the actions and sensations of actions in the simulation.

Because of these dedicated tasks is shown as a separate component in Figure 1. On the other hand, it is realized in the same spirit as other services described in the previous section. This means that also the simulation engine is a replaceable component. Because of this flexibility, there are different ways how the simulator components can be used.

We realized two different kinds of simulation engines, which users of the Spark simulation system can choose for their simulations: a straightforward implementation that realizes agent actions in the order in which they arrive at the simulator, and an implementation that cares for maximum reproducibility of distributed simulations. With the straightforward implementation, simulations and agents can be realized easily. We think that this simulation engine will be useful in application domains where the efficiency of the simulation is important because a large number of agent configurations and control parameters have to be evaluated, as in genetic evolution or machine learning.

The other simulation engine was implemented using SPADES [13], a middleware system for agent-based distributed simulations. This system provides an abstraction that allows world model and agent designers to ignore machine load on different machines, networking issues and reasoning about distributed event realization. We attached great importance to the separation between SPADES specific code and other Spark components so that is was easy to replace SPADES with our own simulation engine.

5.1 SPADES-based Simulations

SPADES operates on simulation events that are sequentially realized. Agents simply receive sensations and send actions. For a simulation designer, two kinds of latencies are of interest: firstly, the latency inherent in the communication between agents and simulator, and secondly the modeled latency (dead time) of real sensors and effectors. SPADES is able to address both kinds of latencies. It hides away the network latency using simulation time stamps, so that this kind of latency is non-existent from the agents point of view. It further allows for explicitly modeling the dead times of sensors and effectors, addressing the second kind of latency.

The system models agents as computational entities that receive sensation events from the simulation and return actions to be executed after some computation. Apart from the requirement that an agent can read and write to UNIX pipes, its internal architecture is not constrained in any way. In particular it is not required that agents are written in a special programming language or linked against a specific library. Agents are not executed as part of the simulator loop. This means that actions of agents do not have to be synchronized with the simulator. Therefore no single joint operation of agent and simulator is required at any particular time.

From the SPADES point of view, a simulation is structured into several groups of components: These are a simulation engine, a world model, one or more communication servers, agents participating in the simulation and possibly some connected monitors. Communication servers are the interface between agents and Spark, they run as part of the simulator and additionally on each machine on which agents run. The remote communication servers connect via TCP to the simulation engine and manage the communication between agents and simulator. Agents connect to the communication server on their host machine through a Unix pipe, where they track the CPU usage of the agents to calculate the agents' thinking latency. Using the SPADES communication servers facilitates distributing agents on several machines without implementing any network related code. On the other hand, SPADES provides an integrated communication server that is part of the simulation engine process, useful for single machine setups to avoid the TCP overhead.

The world model holds the state of the simulated world. In Spark it is mainly contained in the scene graph and managed by the so called *scene server*. The simulation server triggers updates of the state of the world by using the scene server. It advances the state up to the time of the next event as requested by the simulation engine. The simulation server is also responsible to realize pending events. The most common source of events are actions of agents. The final task in each simulation step is to possibly generate sensations that are sent to participating agents, depending on the data rate of simulated sensors. Sensations are events that carry perception data about the current state of the world.

5.2 Event Processing

In the interaction with the world model, SPADES advances the world model several time quanta until the next pending event. In the interaction with the agents, SPADES is a discrete event simulator, following its model of agents.

Events are therefore the basis for a straight forward interaction between an agent and the simulation: An agent waits until it receives a sensation event from the communication server it is connected to. Based on this sensation it then generates a set of actions that it sends back to the communication server. The thinking cycle is finished as soon as the communication server receives a "`done thinking`" message from the agent.

After sending a sensation to an agent, the corresponding communication server tracks the machine time used until it receives a "`done thinking`" message. The total amount of machine time used in the think cycle is then translated into simulation time. By correlating the consumed machine time with the corresponding simulation time SPADES assures that the simulation is reproducible and unaffected by network delays or load variations among machines hosting the agents.

These factors only affect the overall simulation speed and not the generated sequence of events. SPADES offers multiple timer models that offer different compromises between precision and overhead. These are for example a `jiffies` based and a `perfctr`[4] based

[4]This is a Linux kernel driver for low-level performance-monitoring counters, and support for per-process counters, see `http://sourceforge.net/projects/perfctr/`.

timer.

To keep track of the simulation time an agent used during a thinking cycle, it can request think time messages from SPADES. This is a type of inform message that does not start a new thinking cycle.

SPADES exploits concurrency by overlapping of events. It guarantees however that the order of event realization will not violate causality. That means no causally related events are realized out of order, for example like a sensation and a subsequent act event of an agent. In many cases however, the sense, think and act components can be overlapping in time.

5.3 Game Control Server

Actions and sensation use a simulation specific protocol, they are passed from the simulation engine to the so called game control server. The game control server converts them using a parser plugin into an internal message format. Using a plugin for the parser has the advantage that it is possible to use simulation specific message formats, they are not constrained by the simulation engine. All perceptor and effector plugins within the simulator that act on behalf of an agent work on the internal representation. This effectively separates their implementation from varying protocol details between the simulator and connected agents and allows them to be reused with different agent types.

Our internal representation is a nested list of named predicates, each with an arbitrary number of typed parameters. The parser plugin we currently use supports an external language based on S-expressions [15]. By simply exchanging the parser plugin, it is possible to switch for instance to an XML-based language. The parser plugin can be implemented without regard to any other network detail. For custom simulations however, replacing the parser should generally not be necessary as S-expressions can be used to encode arbitrary (also binary) data.

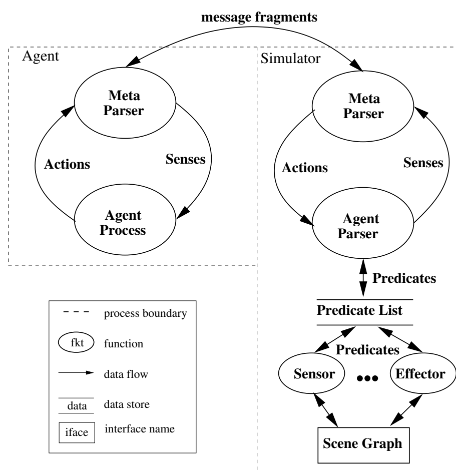

Figure 4: Data flow between agents and simulator

The agent management, implemented as part of the game control server, is respon-

sible to construct and maintain the internal agent representation. We call the internal representation of an agent in the simulator an agent proxy, as it carries out actions and collects sensor data on behalf of a connected remote agent. Agent proxies are a part of the internal scene graph; they are constructed for each agent when the agent connects to the simulator. A designated agent root node, called *agent aspect* identifies a subtree of the scene as an agent. The physical and visual representation of an agent however is not further differentiated from other objects in the simulation, they are just nodes below the agent aspect.

Agent proxies possess sensor and actuator nodes that reflect the capabilities of the represented agent type. It is also possible to implement agent actions that change the capabilities of agents during a simulation, for instance by requesting a new kind of sensor. On receipt of action messages from a connected agent the game control server parses it into a fixed internal representation, that resembles a nested list of parameterized predicates. The server dispatches the messages parts to the different agent actuators, that then act on behalf of the agent on the simulated world. Simulated sensors are implemented in a analogous way. The agent management periodically queries the perceptor nodes of the managed agent proxies to collect sensor data in the fixed internal representation. From this data the game control server generates sensor data for the remote agent in a the sensor data format of the particular agent.

The game control server is also responsible for the game management, that is the implementation of rules in a simulation, called *game control aspects* in our terminology. They implement aspects of a simulation that do not immediately follow from the physical simulation of the environment. Examples are performance measures of participating agents, like their score count in the soccer simulation. Game control aspects are implemented as a set of plugin that are registered to the game control server. The run loop triggers the update of the control aspects after each simulation step. Control aspects have complete control over simulations as they can access the scene graph. The game control server provides additional services to locate and access agent aspect nodes in the simulation.

The monitor management, responsible for the visualization of the simulation in cooperation with external monitors, is realized as a replaceable plugin in the same spirit as the control aspects. It delegates all of its tasks to a plugin that is installed below the monitor server node. External monitors can be connected from remote machines, and similar to the protocols used for agents, the network implementation focuses on modularity and a strict separation of protocol layers. This allows for the easy customization of the monitor protocol in the same spirit as the parser plugin of the game control server described above.

Each monitor protocol implementation is contained within a single class that implements the monitor interface. It is responsible to generate updates for and parse commands from a connected monitor. The monitor update protocol implementation itself does not know about further network details, for instance which transport and which meta protocol is used. The meta protocol is responsible to classify and assemble the different message fragments received via the transport protocol. Conversely it is also responsible to prepare messages to be dispatched over the network. One possible meta

protocol is to treat messages as strings that are prefixed with their type and length.

6 Physical Simulation

A physics engine is needed for the simulation of entities in our system. Instead of implementing our own physics subsystem, we integrated ODE [14], the Open Dynamics Engine. ODE is a free, high quality library for simulating articulated rigid body dynamics. ODE is a library with a plain C interface. Spark provides easy object oriented access to all ODE concepts, implemented on top of the Zeitgeist framework. All ODE concepts, rigid bodies, colliders and joints, are encapsulated by `C++` objects. Instances of these objects are installed into the scene graph. Specific groupings of objects express their responsibility for each other. This enables the objects to automatically care for the proper interaction. This concept is more natural for an object-oriented framework and hides the handle-based ODE interface.

6.1 Basic Concepts

Rigid bodies are the basic entity of the physical simulation. They have several constant properties like mass, their center of mass and mass distribution. Other properties change over time. These are their position and orientation in space and further linear and angular velocity.

Without any external influences a rigid body keeps its properties unchanged, resulting in a monotonous movement over time. ODE provides forces and torques as the two basic concepts used to act on rigid bodies. These two concepts model all interesting properties one expects from a physical simulation.

A good example for properties that are modeled using forces are shape and extent of a simulated object. These are not direct properties of rigid bodies and are irrelevant to their simulation unless two objects collide. In this case they should influence each other, which can be accurately described in terms of forces and torques that are applied on the two colliding bodies. ODE models shapes of a simulated objects with a so called collider. It represents a geometric object whose only purpose is to detect intersections with other colliders. A collider does usually not model the exact shape of the associated visible object but a computationally less expensive shape. ODE supports boxes, spheres, capped cylinders and planes as collision primitives. Technically it is also possible to detect collisions with arbitrary shapes and extents. Though not yet supported by Spark, we are currently about to implement this.

We represent simulated entities in the scene graph with a node containing a reference coordinate and orientation of the entity. The different properties of the entities, like the physics or the geometry mentioned above, are called *aspects* in Spark. Physic aspects represent physical properties of a body, the mass and a mass distribution. Geometry aspects are the colliders of an entity, they implement the shape of objects to handle collisions with other objects. There are further aspects with describing how nodes have to be rendered on the screen. Aspects are also represented as nodes in the scene graph and located in nodes below an entity. Aspects sometimes have to interact with each other, for

instance when two geometries collide, forces have to be applied to the physics nodes of each affected entity. Here, Spark simplifies the underlying handle-based approach as used by ODE and derives relationships between aspects by their location in the scene graph, so that simulation programmers do not have to care for managing the ODE handles and the appropriate interactions. Besides the automatic management of aspects, it is also possible to omit some of the aspects by simply not attaching the respective nodes to the entity. Static objects in a simulation, like for instance a simulated soccer field, possess a geometry, so that other objects can collide with them, but they do not possess a physics aspect because they should not be affected by gravity.

6.2 Joints and Articulated Bodies

When a collision is detected it must be resolved. The correct forces that prevent the objects to interpenetrate must be applied to the bodies. This is done with the help of contact joints that are generated in response to a detected collision. Joints are used to actively enforce a relationship between two connected bodies. Further supported joint types of ODE are ball and socket joint, hinge joint, two-hinge joint, slider joint and universal joint. These joints constrain the relative movement of the two connected bodies along one or more axes. Additionally joints can act as motors by enforcing the movement along the non-restricted axes. A set of bodies that are connected with joints form an articulated structure, used to simulate vehicles or legged creatures. Joints, like the geometry or the physics of a simple body, are represented by nodes of a special type. In the scene graph, single parts of a complex body share a common parent node. This also simplifies actually constructing a complex body: For example, ODE needs to know which bodies are connected by a joint. In our approach, a joint can simply be created and stored in the scene graph, and the attached bodies are given by a path expression. Relative path expressions further support the reuse of construction scripts and scene description languages that build upon path expressions, as we show in one of the next sections.

6.3 Agents as Objects in the Simulation

Agent programs are external processes for the simulator. The representation of the agents properties inside the simulator is almost equal to the representation of all other objects in the simulation. As mentioned above, agent proxies are part if the scene graph with a designated agent aspect node as root node. They can possess all aspects of regular objects. Additionally, agents possess perceptors and effectors, also represented by nodes in the scene graph below the agent aspect. Perceptors provide sensory input to the agent program associated with the representation of the agent in the simulator, and the agent program uses the effectors to act in its environment. Other objects in the simulation and the physics of the system can affect the situation of agents; this is reflected in the respective aspects by changing the positions or velocities.

6.4 Enhanced Usability of ODE Concepts

Spark also uses an object-oriented approach to handle the collisions occurring in a simulation. These are handled by collision handler classes, installed below colliders of simulated objects in a similar fashion as we implemented the native ODE concepts. This allows simulation dependent reactions when two objects collide. Examples are playing a sound if a body touches the ground or triggering special simulation events. The latter approach is used in the RoboCup soccer simulation to detect if a goal is scored: this is the case if the ball collides with the goal box collider of the opposite team.

The default reaction to a collision however is to resolve it. As described above, contact joints are used to prevent the bodies from interpenetrating each other. A contact joint takes several parameters that describe the contact surface: The resulting friction, if and how the two bodies slide along the contact surface and the "bouncyness" are some examples of parameters. Spark associates a surface description with each collider holding these parameters. When two objects collide, a resulting contact surface description is automatically calculated and applied by a contact joint handler.

7 Scene Description Language

Spark provides access to the managed scene graph in several ways. Besides the internal `C++` interface and external access via Ruby script language, Spark supports an extensible mechanism for scene description languages. This means scene setup can be both procedural or description-based. Scenes in scene description languages can be imported using one of any number of registered scene importer plugins. Scene importers are installed as servers in Zeitgeist, each supporting a different scene description language.

7.1 RubySceneGraph language

For Spark, we implemented one S-expression-based reference language, which is called *RubySceneGraph*. It maps the scene graph structure to Lisp-like S-expressions. An S-expressions is a list of elements, where each element is either an `atom` or another `list`. An atom is either a predefined keyword or a non-empty string literal that has no further syntactic structure. For an example, see Listing 1.

The RubySceneGraph interpreter recognizes a set of special atoms. The first atom in each subexpression determines its type. The set of keywords comprises four atoms that allow the interpreter to distinguishes five different expression types.

- The `RubySceneGraph` expression is the header expression of every scene graph file.
- The `node` expression declares a new scene graph node.
- The `importScene` expression is replaced with the content of a scene graph file.
- The `template` expression declares a set of parameters for a following scene fragment that can later be reused like a macro.

- Every other expression type is interpreted as a `method call`.

Apart from the different expression types listed above the language contains a replacement mechanism. Each atom literal starting with a dollar sign is interpreted as a template parameter and replaced with its actual value. We describe the semantic of the different expression types below together with some examples and a partial reference of available node types and methods.

7.2 File structure

The top level structure of a ruby scene file consists of two S-expressions. The first expression must be the header expression. It allows the parser to confirm the file type and to get information about the version of the used language. The header is followed by a single S-expression that contains the scene graph body. Any further expression is discarded. The body expression consists of an optional single template expression and a set of node expressions. The resulting structure is outlined in Listing 1. Note that lines starting with a semicolon are comment lines.

```
; the header expression
(RubySceneGraph 0 1)
(    ; the body of the file starts here

     ; declare this file as a template
     (template $lenX $lenY $lenZ $density $material)

     ; declare the top level scene graph node
     (node Box
         ; children of the top level node go here
         (node DragController
         )
     )
)
```

Listing 1: File Structure

7.3 Node Expression

Nodes in the scene graph are declared with the `(node <ClassName>)` expression. The *ClassName* argument stands for the name of a class registered to the Zeitgeist class factory system.

The semantics of a node expression is to instantiate a new scene graph object of the given class type. The importer uses the Zeitgeist class factory system to create the requested object. Nodes are installed as a child nodes of the enclosing node expression. If there is no enclosing node expression, the node is a top level node. The set of top level nodes are installed as children of the node importing the current file. This is either the root node of the scene graph, or a different node defined with the importScene expression. The nesting of node expressions directly defines the structure of the resulting scene graph with small syntactic overhead.

7.4 Scene Graph templates

The language further allows the reuse of scene graph parts in a macro like fashion. This enables the construction of a repository of predefined partial scenes, or complete agent descriptions. The macro concept is available through the `(importScene <filename> <parameter>*)` expression. This expression recursively calls the importer facilities of the system. It takes the nearest enclosing node expression as the relative root node to install the scene graph described within the given file.

Note that the given file must not necessarily be another RubySceneGraph file but any file type registered to the importer framework. This allows the nesting of scene graph parts expressed in different graph description languages. An example application of this feature is that parts of the resulting scene could be created by application programs for creating 3D models. By now, we do not exploit this feature yet.

The example in Listing 2 assumes a `box.rsg` file to construct a box with size and color according to the template expression given in Listing 1.

```
(RubySceneGraph 0 1)
(
    (node Transform              ; using templates
        (importScene box.rsg 1 3 0.8 10 matRed)
    )
    (node Transform              ; method calls
        (setLocalPos 10 20 5)
        (node Box (setExtents 1 1 1))
    )
)
```

Listing 2: Example for `importScene` and method calls.

7.5 Method Calls

A node created with the node expression above can further be parameterized with method calls in order to modify its default properties. Every expression that does not match one of the expression types described above is interpreted as a method call. It is read as an S-expression that starts with the name of the function, followed by an optional list of parameters: `(<method name> <parameter>*)`.

Each method call is evaluated in the context of the nearest enclosing node expression. The semantics of a method call is to invoke the Ruby script interface of the corresponding `C++` class, hence the name of this language.

This design decision allowed us to rapidly implement a complete scene description language during the development of the simulator. The language grows automatically as new methods are exported to Ruby.

An example usage of method calls is the setup of a transform node. Transform nodes are used to position and orient subnodes along a path in the scene graph relative to their respective parent node. Our `C++` class for transform nodes provides a method `SetLocalPos` to set the offset relative to its parent node. By exporting this method to Ruby, this method is also automatically part of our scene description language.

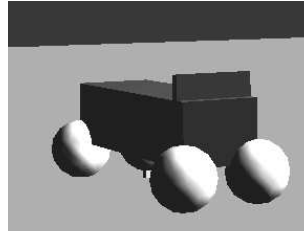

Figure 5: Constructed buggy from Listing 3

7.6 Buggy example

An example that demonstrates most of the language features is the construction of a simple buggy. It consists of a box connected to four spheres as its wheels. Each tire is connected to the buggy chassis using a two-hinge joint. This joint type behaves like two hinges connected in series.

The framework facilitates the straight forward installation of joints, as two connected bodies are referenced with path expressions relative to the joint node. The joint anchor is given in coordinates relative to the joint. The resulting buggy can be seen in Figure 5.

The buggy is further equipped with a motor that controls the left front wheel, together with a perceptor that reads back the current orientation of the wheel. Connecting this buggy scene to a controlling agent process using Spark gives a good example for the construction of agents featuring articulated bodies.

```
(RubySceneGraph 0 1)
((node Transform ; create the char chassis
       (setName chassis)
       (setLocalPos 0 0 0.5)
       (importScene box.rsg 1 3 0.8 10 matRed)
       (node Transform (setLocalPos 0 1.3 0.55)
             (node Box (setMaterial matBlue)
                       (setExtents 1 0.1 0.3)))))
 (node Transform  ; install the left back tire
       (setName leftBack)
       (setLocalPos -0.5 -1.5 0)
       (importScene sphere.rsg 0.4 2 matWhite)
       (node Transform (setLocalRotation 0 180 0)
             (node Hinge2Joint ; install the joint
                   (attach ../../sphereBody ../../../chassis/boxBody)
                   (setAnchor 0 0 0)
                   (setMaxMotorForce 1 4000) ; enable joint motor
                   (node Hinge2Perceptor)
                   (node Hinge2Effector)))))
 ; [...]
```

Listing 3: Buggy Construction Example (partial)

8 Results and Conclusions

In this paper we introduced Spark, a generic three-dimensional physical simulation system. Spark is built as an extensible set of plugins on top of Zeitgeist, an application framework that brings features of scripting languages to C++. As fundamental concept in Zeitgeist, we identified reflective factories used together with an object hierarchy as implementation pattern, which we believe to be useful for creating other applications as well. Spark features a scene graph language and network support, delivering a simulator that is ready for usage in research and education. Spark can be used to address problems of both multi-agent researchers like team behavior as well as research questions like the influence of changes in the morphology of single agents.

However, even if simulations can facilitate experiments in many cases, they are abstractions of other systems and usually cannot totally replace an implementation on the target system [16]. Consequently, a goal of simulation is not specialized solutions but the identification of general principles [6]. For this, creating reproducible experiments is of great value, which is supported through the integration of the SPADES middleware into Spark. An alternative simulation engine focuses on speed, giving up the exact reproducibility. Both engines come with full network support. Our system already showed its real world applicability as the official simulator of RoboCup Simulation League 2004.

We also started some work in developing wheeled and legged robot models so that we hope to be able to introduce a legged simulation league to RoboCup. The necessary primitives to do this are already implemented in our framework. In the current 3D soccer simulation, the agents' sensor data describe the complete object type and position of sensed objects. To address problems other than team behavior alone it is however possible to implement a realistic distance sensor or a camera. For future work we hope to be able to support description languages of other, more specialized simulators. There is already work in progress for simulating the Fujitsu HOAP-2 humanoid robot and to import Webots scene descriptions.

References

[1] Itsuki Noda, Hitoschi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer Server: a tool for research on multi-agent systems. volume 12, pages 233–250, 1998.

[2] Itsuki Noda. Soccer Server: A simulator of RoboCup. In *Proceedings of AI symposium '95*, pages 29–34. Japanese Society for Artificial Intelligence, 1995.

[3] Marco Kögler and Oliver Obst. Simulation league: The next generation. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 458 – 469. Springer, Berlin, Heidelberg, New York, 2004.

[4] Pedro Lima, Luís Custódio, Levent Akin, Adam Jacoff, Gerhard Kraezschmar, Beng Kiat Ng, Oliver Obst, Thomas Röfer, Yasutake Takahashi, and Changjiu Zhou.

Robocup 2004 competitions and symposium: A small kick for robots, a giant score for science. *AI Magazine*, 2005. To appear.

[5] Alexandra Mark, Daniel Polani, and Thomas Uthmann. A framework for sensor evolution in a population of braitenberg vehicle-like agents. In Christoph Adami, Richard K. Belew, Hiroaki Kitano, and Charles E. Taylor, editors, *Artificial Life VI, Proceedings of the Sixth International Conference on Artificial Life*, pages 428–432. MIT Press, 1998.

[6] Günter Bruns, Daniel Polani, and Thomas Uthmann. Eine virtuelle kontinuierliche Welt als Testbett für KI-Modelle. *Künstliche Intelligenz*, (1):60–62, 2001.

[7] Cyberbotics Ltd. *Webots User Guide*, April 2004.

[8] Brett Browning and Erick Tryzelaar. ÜberSim: a multi-robot simulator for robot soccer. In *Proceedings of AAMAS 2003*, pages 948–949, 2003.

[9] Sebastian Buck, Michael Beetz, and Thorsten Schmitt. M-ROSE: A multi robot simulation environment for learning cooperative behavior. In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa, editors, *Distributed Autonomous Robotic Systems 5*. Springer, 2002.

[10] Chris Hargrove. Reflective factory. `http://www.gamedev.net/reference/articles/article1415.asp`, December 2000.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] QNX Software Systems Ltd. *QNX Neutrino Realtime Operating System: System Architecture*, 2003.

[13] Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference*, volume 1, pages 817–825, 2003.

[14] Russell Smith. *Open Dynamics Engine (ODE) User Guide*, May 2004.

[15] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[16] Erann Gat. On the role of simulation in the study of autonomous mobile robots. In *Proceedings of the AAAI 1995 Spring Symposium*, pages 112–115, 1995.