

Spark — A Generic Simulator for Physical Multi-agent Simulations

Oliver Obst and Markus Rollmann

Universität Koblenz-Landau, AI Research Group, D-56070 Koblenz
{fruit,rollmark}@uni-koblenz.de

Abstract. In this paper 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 is 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. Spark is a generic tool for creating simulations that can be used to address all of the above mentioned problem types. It was our goal to provide a great amount of flexibility, so that for somebody creating new simulations 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.

For simulation designers, this flexibility 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 built Spark integrating prior work in both physical and multi-agent simulation.

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 [12] is a prominent benchmark. One of the landmark goals for RoboCup is that by mid-21st century, a team of fully autonomous humanoid robot soccer players shall win against the champion of the most recent World Cup. In Soccer Simulation League competitions, two teams of 11 autonomous agents, represented as circles, compete in a two dimensional, discrete-time simulation. From the first official RoboCup competition¹ up to now the simulator [11] has continuously been enhanced. However, its limitation to a two dimensional world remained. In order to accomplish the vision of the RoboCup Initiative, it is absolutely necessary to move the simulation into a three-dimensional world (cited from [8]). 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 would be a rescue scenario where different robots collect information about the status of collapsed buildings.

On the RoboCup 2003 Symposium, we proposed a new approach to a three-dimensional physically realistic soccer simulator [9]. This system was a prototype of the simulator we describe in this paper and not specially designed to simulate only soccer competitions, but a universal system for simulation of physical agents. However, the specific features for reproducible and distributed simulations, simple construction of articulated bodies and the scene description language we describe here were missing.

Despite this, in a road map discussion for Soccer Simulation League on RoboCup 2003, a huge majority of participants voted for adding the three-dimensional simulation to the competitions; this year a simulation built on top of Spark was used for the first three-dimensional RoboCup Simulation League competition. In contrast to the two-dimensional simulation, our implemented three-dimensional simulation possesses a higher complexity with respect to the possible team behavior while it maintains a good degree of abstraction with respect to the possible agent actions and sensations.

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.

An entire different tool to study the behavior of a large number of agents in two- or three-dimensional continuous virtual worlds is XRaptor [2]. 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 [4] 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 [1] 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 [3] 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 transferred 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 in fact this at least 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.

The remainder of this paper is organized as follows: The following section describes the application framework we created as base for the whole simulation system. In Sect. 4, we explain the functionality and integration of the core simulator engine. Section 5 shows how we integrated the underlying physics engine and provided a user-friendly way to access it by introducing the idea of connecting different simulation primitives via path expressions through a scene graph. Section 6 gives some details of the way network support has been added to the simulator, while Sect. 7 introduces a scene description language for setting up different kinds of simulations. Finally, Sect. 8 concludes the paper.

3 The Zeitgeist Application Framework

One of the first implementation steps was to create a flexible application framework, called Zeitgeist. Zeitgeist was invented² 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. The flexibility of Zeitgeist was one of the the key reasons why it was possible to refactor and build upon the prototype implementation instead of starting from scratch again. A variant of the reflective factory design pattern can be identified as key element for the flexibility of Zeitgeist. The reflective factory pattern was extended with methods supporting an object hierarchy of both created objects and factories in the same tree. We describe this pattern in the subsequent paragraphs. To our knowledge, a description of this element as design pattern cannot be found elsewhere, even though it is very likely that this pattern also occurs in other applications. We believe that it is also useful for other applications which have to provide a system of exchangeable and scriptable components.

3.1 Reflective Factory Pattern

The reflective factory pattern [7] is 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, i.e. instantiated classes, maintain references to the factory that created them.

The property that each instantiated object has the knowledge which factory created it distinguishes the reflective factory pattern from the abstract factory pattern [5]. It enables every object to access meta data stored in the associated class object at runtime. Zeitgeist exploits this to store class names and information about supported interfaces in the class objects, allowing for queries about the class type and supported interfaces at runtime. By using this information we made all objects in the simulator accessible to a scripting language. The availability of this kind of meta data is native to object oriented programming languages, such as Objective C, Smalltalk or Ruby, but not to C++. We have chosen C++ as primary implementation language and adding this information

² original implementation by Marco Kögler, see also [9]

“by hand” anyway, because it provided the most freedom in integrating external libraries. For instance the agent middleware system we are using (SPADES, see also below) offers only a C++ interface.

3.2 Reflective Factory with Object Hierarchy

In combination with the reflective factory pattern, *Zeitgeist* organizes factories and objects created by factories in a tree like structure, comparable to a virtual file system. 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 UNIX like file system, *Zeitgeist* is able to retrieve the corresponding object instance.

The object hierarchy is useful for implementation of a concept called pathname space mapping. Pathname space mapping appeared already in the QNX operating system and has been used to realize the QNX resource manager concept [13]. Resources are addressed by a path through the hierarchy given as string. The managed resources here are Spark services, 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 applications. Applications can get services at runtime by querying the known location.

Zeitgeist itself relies on the combination of the reflective factory in conjunction with the object hierarchy for the following reason: The factories themselves are installed at determined locations in the hierarchy. This 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 with no recompilation of the whole system, but just by adding plugins. From these custom classes realized as plugins, it is possible to get instances via configuration scripts and install the instances as servers again. *Zeitgeist* makes further use of the pathname space mapping concept when the implementation of services is delegated 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.

An example application is the file server in Spark, 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 Core Simulator

For an entire simulation, the simulator, agents, and monitors to watch simulations are all different processes that have to work together. The core of our simulator is the part of the system that contains the run loop and does the event management. It cares for the timing, and controls the communication between the simulator and external processes.

4.1 Run Loop

The core part of the system is realized in the same spirit as other services described in the previous section. Thus even the run loop of the simulator is replaceable. We realized two different kinds of run loops, 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. The other implementation was implemented using SPADES [14], 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.

4.2 System Overview

Both run loops described above rely on the same set of generic simulator services. These services comprise agent and simulation management, monitor management and the physics system.

The agent management, implemented as part of the so called game control server, is responsible 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. A designated agent root node, called *agent aspect* identifies a sub tree 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.

Agent proxies possess sensor and actuator nodes that reflect the capabilities of the represented agent type. 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 second responsibility of the game control server is 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 runloop triggers the update of the control aspects after each simulation step. Control aspects have complete control over simulation as they can access the scene graph. The game control server provides additional services to locate and access agent aspect nodes in the simulation. Further, the monitor management is realized in the same spirit as the control aspects as a replaceable plugin. It delegates all of its tasks to a plugin that is installed below the monitor server node. 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.

4.3 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.

4.4 System Structure

SPADES is one of the possible instantiations of the simulator run loop. 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. The simulation engine contains the main run loop of the SPADES system. It implements the event system and coordinates all network communication with connected monitors and communication servers. A communication server must be run on each machine on which agents run. It connects via TCP to the simulation engine and manages the communication with agents on the host machine through a Unix pipe as well as tracking their CPU usage to calculate their thinking latency.

The world model holds the state of the simulated world and advances it up to the time of the next event as requested by the simulation engine and is further responsible to realize events. The most common source of an event is an act event in order to carry out an agent action. Finally, the world model generates sensations that are sent to participating agents. These events carry perception data about the current state of the world.

4.5 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. After a sensation is sent 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.

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.

4.6 Spades Integration

In order to build a simulation, SPADES expects an implementation of a world model and custom event realizations for sense and act events. Both, the simulation engine and the custom world model become part of the same process.

In this way the Spark simulator implements the SPADES world model interface. We attached great importance to the separation between SPADES specific code and other Spark components so that SPADES can be replaced by other simulation engines easily. Currently the user can choose between SPADES, providing reproducibility with high accuracy and a custom simulation server focusing on raw speed. Here, we drop the reproducibility SPADES provides with the remaining concepts being similar. We implemented this engine because we think that it will be useful in application domains where a large number of agent

configurations and control parameters have to be evaluated, as for example in genetic evolution or machine learning.

5 Physical Simulation

Another equally important part is the physical simulation of the system. Instead of implementing an own physics subsystem, we integrated ODE [15], the Open Dynamics Engine. ODE is a free, high quality library for simulating articulated rigid body dynamics.

5.1 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.

5.2 Articulated Bodies and Joints

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. 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.

5.3 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. There are bodies (i.e. mass and a mass distribution) for the physical simulation, colliders to implement the shape of objects to handle collisions with other objects, ODE joints to connect single bodies and colliders to compound objects. Additionally, agents possess perceptors and effectors. 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. The way agent programs get their input is described in Sect. 6.

5.4 Enhanced Usability of ODE Concepts

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. In Spark, all created objects are stored in a tree of objects. By accessing well-defined locations of objects, it is possible to exchange different implementations of components with equal interfaces easily. The objects that are used to describe a scene are located below the scene server node, and are referred to as scene graph. In the scene graph, objects maintain a relative position to their parent node. This feature is useful to arrange groups of objects.

To relate arbitrary objects in the scene graph, for example to install joints between bodies, we use path expressions in the scene graph. This dramatically simplifies the construction of articulated bodies in comparison to the original handle-based ODE approach. As these expressions are relative to the joint node, they further support the reuse of construction scripts and scene description languages that build upon them, as we show in one of the next sections.

Spark also uses an object-oriented approach to handle the collisions occurring in a simulation. These are handled by collision handler classes grouped to 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 example 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.

6 Network Support

Spark supports both the separation of the simulation core from the connected monitoring applications and from agents participating remotely in the simulation. The network implementation focuses on modularity and reusability and a strict separation of protocol layers.

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.

A similar concept applies to the communication with agent processes. An independent meta protocol identifies messages received from an agent. A parser plugin is then responsible to convert these messages into an internal fixed representation. This 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 [10]. All perceptor and effector plugins within the simulator that act on behalf of an agent only 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. The external protocol is not constrained by the simulator. By simply exchanging the parser plugin, it is possible to switch for instance to an XML-based language. This parser can be implemented without regard to any other network detail. For custom simulations however, this should generally not be necessary as S-expressions can be used to encode arbitrary (also binary) data. For a diagram on the data flow between agents and simulator, we refer to Fig. 1.

7 Scene Description Language

Spark provides access to the managed scene graph in several ways. Besides the internal C++ interface and external access via script language, an extensible mechanism for scene description languages is implemented. This allows for both a procedural and a description-based scene setup.

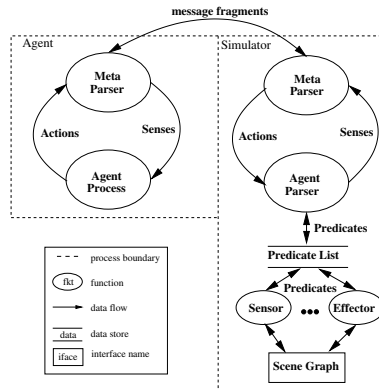


Fig. 1. Data flow between agents and simulator

A scene is imported using one of any number of registered scene importer plugins, each supporting a different scene description language. Currently one S-expression-based importer is implemented. The language we implemented as reference language, called *RubySceneGraph*, maps the scene graph structure to the nesting of Lisp-like expressions. A node in the scene graph is described with the `(node <ClassName>)` expression. The importer relies on the Zeitgeist class factory services to create an object of the requested type. A node expression can further be parameterized with function calls in order to access properties of a scene node. A function call expressed as S-expression is realized using the script function exported from corresponding the C++ class.

An example of the two concepts combined is the setup of a transform node. These node types are used to position and orient nodes along a path in the scene graph relative to their respective parent node. The transform node therefore provides a method `SetLocalPos` to set the offset to the parent node.

```
(RubySceneGraph 0 1)
((node Transform (setName myTransform)
  (setLocalPos 10 20 5)
  (node Box (setExtents 1 1 1))))
```

Listing 1. A minimal RubySceneGraph example

Listing 1 starts with the RubySceneGraph header giving the version number. It then creates a single transform node `myTransform`, and sets its offset relative to the parent node. This node is not explicitly given in the above example. In the hierarchy below the transform node a box node is constructed. This is a node that simply renders a box with the extents in the subsequent `setExtents` function call.

A second more elaborate example demonstrates two additional concepts available in this language. It allows the definition of scene graph templates, that take parameters to construct a set of similar scenes. The demo graph in Listing 1 is not complete as it omits two additional aspects of the box that are

needed for it to take part in the physical simulation. These are a collider and an associated rigid body. All three properties are usually aligned to each other, concerning their extents and assigned mass. This repetitive task can be expressed using a template, as shown in Listing 2.

```
(RubySceneGraph 0 1)
((template $lenX $lenY $lenZ $density $material)
 (node Box (setExtents $lenX $lenY $lenZ)
           (setMaterial $material))
 (node Body (setName boxBody)
            (setBox $density $lenX $lenY $lenZ))
 (node BoxCollider (setBoxLengths $lenX $lenY $lenZ)))
```

Listing 2. A RubySceneGraph template

The language further allows the reuse of scene graph parts, that are not necessarily templates, in a macro like fashion. This enables the construction of a repository of predefined partial scenes, or complete agent descriptions. The macro concept is not part of the language itself but implemented as a script function called `importScene`. It delegates its task to the generic scene graph importer, from where scenes are imported with one of the registered plugins. This allows the nesting of scene graph parts expressed in different graph description languages. An example application of this feature is that parts of a scene could be created by application programs to create 3D models. By now, we do not exploit this feature yet.

```
(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)

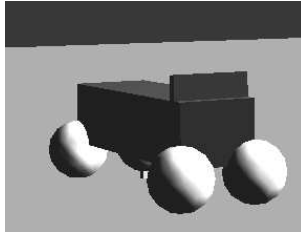


Fig. 2. Constructed buggy from Listing 3

Listing 3 is a partial example constructing a simple buggy that 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 Fig. 2. 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.

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 [6]. Consequently, a goal of simulation is not specialized solutions but the identification of general principles [2]. 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 shows its real world applicability as the official simulator of RoboCup Simulation League 2004. Because we are using simple types of agents in this first competition, interesting questions will be

if approaches previously successful in two-dimensional soccer are still applicable despite the higher complexity of the environment.

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 some interest from people doing research using real robots.

References

1. Brett Browning and Erick Tryzelaar. Übersim: a multi-robot simulator for robot soccer. In *Proceedings of AAMAS 2003*, pages 948–949, 2003.
2. Günter Bruns, Daniel Polani, and Thomas Uthmann. Eine virtuelle kontinuierliche Welt als Testbett für KI-Modelle. *Künstliche Intelligenz*, (1):60–62, 2001.
3. 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.
4. Cyberbotics Ltd. *Webots User Guide*, April 2004.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. 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.
7. Chris Hargrove. Reflective factory. <http://www.gamedev.net/reference/articles/article1415.asp>, December 2000.
8. Hiroaki Kitano and Minoru Asada. RoboCup humanoid challenge: That's one small step for a robot, one giant leap for mankind. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 419–424, 1998.
9. Marco Kögler and Oliver Obst. Simulation league: The next generation. In D. Polani, A. Bonarini, B. Browning, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, Lecture Notes in Artificial Intelligence. Springer, Berlin, Heidelberg, New York, 2004. To appear.
10. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.
11. Itsuki Noda. Soccer Server: A simulator of RoboCup. In *Proceedings of AI symposium '95*, pages 29–34. Japanese Society for Artificial Intelligence, 1995.
12. 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.
13. QNX Software Systems Ltd. *QNX Neutrino Realtime Operating System: System Architecture*, 2003.
14. 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.
15. Russell Smith. *Open Dynamics Engine (ODE) User Guide*, May 2004.