Simulation League:
The Next Generation

Marco Kögler, Oliver Obst

3/2003

Fachberichte
INFORMATIK

Simulation League: The Next Generation

Marco Kögler, Oliver Obst[*]

Universität Koblenz-Landau
Arbeitsgruppe Künstliche Intelligenz

{koegler,fruit}@uni-koblenz.de

February 20, 2003

We present a modular approach to model multi-agent simulations in 3D environments. Using this approach, we implemented a generic simulator which is totally decoupled from the actual simulation it performs. We believe that for Soccer Simulation League a transition to 3D states exiting new research problems and equally makes it more attractive to watch for spectators. We are proposing to use our framework as basis for a next generation Soccer Server.

1 Introduction

For eight years now Soccer Server [Noda, 1995] exists as a testbed for evaluating multiagent systems and has inspired a lot of researchers from Computer Science and Artificial Intelligence to compare their approaches. Since its beginning Simulation League is confined to two dimensions in order to reduce complexity. During the past years quite a number of advances have been made by the participating teams. As their performance increased, new features were added to the server to provide a more realistic simulation and a tougher challenge[1]. What started out as a simple model has evolved into a mature research platform and a more accurate representation of the original. Meanwhile, the server models different types of players with distinct abilities such as speed, size, and stamina. Players can also point into directions, turn "head" and body separately from each other and perform a kind of tackling. A coach can substitute players, analyze the match, and give advice and information to players in a standardized coach language [Chen *et al.*, 2001]. As rich as the new features appear to be, a number of caveats can be identified:

---

[1]Thanks to Tom Howard for his great and constant effort in maintaining the Soccer Server.

2D Simulation A fundamental problem with the current soccer server is that it simulates the game of soccer in a 2D world, making it seem more like a game of table hockey with soccer rules. Nevertheless, the game of soccer is a three-dimensional game. The ball and players can actually leave the ground. It is absolutely necessary to move the simulation into a 3D world in order to accomplish the mission statement of the RoboCup Federation (cited from [Kitano and Asada, 1998]): *By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA, against the winner of the most recent World Cup.* Small steps towards a 3D server have already been made [Howard *et al.*,]. We believe that the transition to a three-dimensional system states exiting new research problems, like for instance in the area of spatial resoning. Also applicability of approaches used in 2D for three-dimensional space has yet to be shown.

Physics The physics model used by the current RoboCup soccer server is very limited. Motion of objects is only described using acceleration, velocity and positional vectors. As objects do not have an associated mass, no real forces act upon the objects of the simulation. This also means that other effects, which usually result out of physics have to be tacked onto the simplistic model (e.g. effect of wind, or a ball slowing down). This only increases code complexity and the need for special-casing, resulting in a long-term maintenance problem. A robust physics system would handle all these effects transparently, as it is just driven by the forces which act on objects in the world. Essentially, what is now a code-driven system would become a data-driven system. This issue becomes even more important once the transition to a 3D world has been made. Also the combination of a three-dimensional representation plus a data-driven physics system situates robots in a world that follows the same basic principles as the world of real robots. It facilitates using robot models consisting of several components connected by joints and certain degrees of freedom.

It is just soccer! This is not really a problem for the soccer server, but nevertheless it is a limitation. If the above mentioned changes were made, the soccer server would allow player programs to perceive a 3D world and perform actions within this world. The real question which should be raised at that stage is: Why restrict this system to just the game of soccer? Or, more provocativley: What design decisions limit this system to just simulating soccer? A flexible approach would allow for various kinds of simulations including soccer or rescue so that the same platform could be used for tasks with different levels of complexities in research and education.

1.1 Resulting Goals

Now, that an overview of the subject matter at hand has been given, it should be clarified how all this relates to the system we created [Kögler, 2003]. The goal was to create a system which can provide simulations of 3D environments. We chose 3D environments, because it obviously

is a more general environment than a 2D environment. Also, the real world is three-dimensional. Thus, the choice to provide 3D simulations greatly simplifies the transition of real world environments to virtual simulator environments or vice versa.

Another important aspect is the integration of a robust physics system, which is responsible for producing realistic motion of the objects within the environment. This system should not be forced into the simulation, as that would limit the applicability. With "forced" we mean no general assumptions are made regarding the need for physics of the simulation. The system should remain unobtrusive and only provide services for simulations which request it.

As we are creating a simulator which can be used for multi-agent systems, we also have to think about how the agent lives and acts in the environment. Clear interfaces for the sensing of the environment have to be presented, as well as how actions can be performed.

Unfortunately, just providing a framework for simulations is not enough. We also have to provide a means to visualize the state of our 3D environment. It is hardly possible for a human spectator to comprehend spatial relationships of complex 3D environments without having a visual reference. Also, it is not only the outcome of the simulation which is of interest to the researcher, but also the way it was achieved.

Ultimately, we want the system to be applicable to not just a single type of simulation, but an entire class of simulations, namely those which can be represented within a 3D environment. Another challenge for this system will be that it should run in realtime on a modern desktop PC.

2 World Representation

One of the biggest challenges was designing the system used for the representation of the environment being simulated. We have already seen that the environment plays a significant role in multi-agent simulations, as it controls what and how agents can perceive the virtual world that they live in. The world representation has to be accessible to a variety of different subsystems of the simulation and visualization components and be able to handle all these accesses in realtime.

In this section we will take a closer look at what is necessary to represent a complete virtual 3D world. We begin by describing the concept of an environment and then examine how the objects living within that environment have to be represented. Following that, we discuss *scene graphs* and how this concept can be used to unify everything into a single, flexible and extensible data structure.

We want to create an environment in which agents can live in. Agents should not be the only objects, which exist in the environment, though. It also contains passive objects, such as chairs, tables, or soccer balls. Passive objects lack the agent property, whereas agents possess sensors and effectors to perceive and interact with the environment. Aside from this, passive objects also lack the ability to reflect their current state [Ferber, 1999]. Their state is only affected by the natural laws of the environment they live in and by agents performing actions on them (either direct or indirect). Using these ideas as a basis an environment can be defined as follows[2]:

---

[2]Definition inspired by Ferber's definition of multi-agent systems [Ferber, 1999].

Definition 1 *An environment E is a three-dimensional space and contains a set of objects O. Each object has a location within the environment and can be perceived and manipulated by agents. An environment contains a set of agents A, such that $A \subseteq O$.*

Definition 1 captures two important concepts. First, agents are always objects in the environment. This relationship allows agents to also be perceived and manipulated by other agents, yet it clearly illustrates that there is a semantical difference between the concept of agents and objects. Second, all objects are *situated*. This means that we are always able to tell where each object is located within the environment. This is extremely important, because without a location, the ability to perceive the environment would be ill-defined.

Definition 1 only places one real restriction on the environment, namely the three-dimensionality of the space it represents. Other than that the environment is solely defined through the objects which it contains.

3 Object Representation

An object living in our 3D environment has to meet the needs for many different subsystems of the simulation and visualization engine. For example, the visualization component has to have some information about how to display the object, whereas the physics component needs access to physical properties. The visualization component might require a complex 3D model, but a simple sphere which encloses the object suffices for the collision system. From now on, we will refer to these object properties as object *aspects*.

3.1 Object Aspects

In order to arrive at a scaleable and versatile system an object will be represented by modelling its aspects. Given the requirements of a 3D world and support for a physics simulation we arrive at the following categorization of aspects for an object:

Visual Aspect This aspect captures what the object looks like. It contains all the necessary information to display the object on the screen using the graphics application programmer interface (API) that the simulator program supports (in our case *OpenGL*).

Physics Aspect The physics aspect is used to provide an interface to the physics system. This aspect will collect different forces acting on the object, as well as its physical properties, such as mass and mass distribution (which affects how the object behaves when in motion).

Geometry Aspect The geometry aspect is used to define the solidity of the object, its shape and size as it is used by the subsystem which detects and resolves collisions. This is usually done with simple volumes, such as spheres, capsules and boxes. We need these simple collision proxies, because the visual aspect is usually much too complex for performing real-time collision detection. Obviously, the shape and size of the geometry aspect should
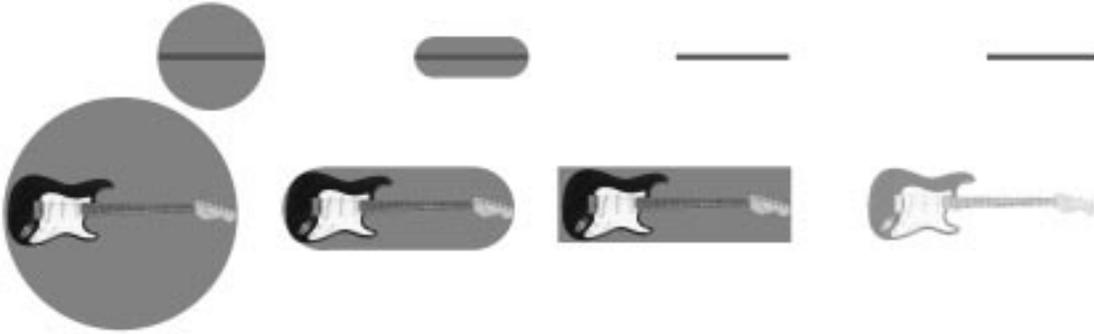
Figure 1: Collision proxies around object, large front-view and small top-view (from left to right): Sphere, capsule, box, and triangle mesh

be chosen in a way that at least somewhat resembles the visual aspect. Otherwise one could end up with objects looking like a box (visual aspect), but behaving like a sphere (geometry aspect). When it is not possible to find a suitable collision proxy, a triangle mesh is used. Fig. 1 shows different collision proxies approximating a 3D object.

All these aspects are linked by a single property of the object, its location in the 3D world. The location determines where the visual aspect will be displayed with respect to a virtual camera. The physics aspect modifies the location of the object, as changes in location are the result of motion and motion is controlled by physics. Two objects are colliding when their respective collision proxies overlap. To determine whether this is the case or not, the geometry aspect also requires the location of the collision proxies, which is identical to the object location.

Aspects also have to be able to interact with each other. For example, when the collision system detects a collision it has to resolve it by moving the two participating objects apart. It cannot do this directly, as a collision is a physical interaction between two objects and we want it to behave acurately. Therefore it has to generate the forces necessary to separate the two objects and apply these forces via the physics aspect in the next simulation step.

Another difficulty which arises when trying to represent objects is, that not all of them have to posess all three aspects. Imagine a very simple environment containing two objects: A ground plane and a sphere which is floating above the plane. When the simulation starts, gravity should cause the sphere to fall down. Both the sphere and plane are solid, so the sphere should collide with the plane, bounce a few times and come to rest once its kinetic energy has been consumed.

To translate this example to the aspect categories: Both the plane and the sphere would contain a visual aspect, as we would like to see these two objects displayed on the monitor when the simulation is running. It is also clear, that we need a geometry aspect for both objects, as both objects were stated to be solid. Now, what about the physics aspect? If both objects would posess a physics aspect, both objects would have to respond to the force of gravity. That means both, the sphere and the plane, would begin to fall once the simulation is started. This behavior is

obviously undesirable. We want the plane to be unmoveable and the only way to achieve this is to omit its physics aspect, as it is responsible for controlling motion. The absence of the physics aspect implies absence of motion.

The reason why some objects in the virtual world lack certain aspects comes from the fact that we are just modelling real world. Surface of the Earth in the example above is modelled by a plane. So, as our model is just an approximation to the real world, we have to bend the natural laws of the universe a bit to accurately represent it.

Now that we have looked at the concepts necessary to represent an object in a 3D environment, we have to extend these ideas to be able to represent agents within this system.

3.2 The Agent Aspect

Our definition of environment already states that an agent is an object. Therefore, it makes sense that an agent can posess the same aspects as an object. In order to represent the additional capabilities of an agent, we introduce the *agent aspect*. The agent aspect is the brain of our object and contains all the necessary control logic. It has the ability to perceive the environment using perceptors, think about what actions to perform next and actually perform them using its effectors. Using this modular approach it is possible to turn almost any object into an agent and vice versa. Section 4 will deal with the agent aspect in greater depth.

3.3 Scene Graph

Scene graphs represent the de-facto standard when it comes to representing spatially-oriented data. In the early nineties, scene graphs were popularized by SGI Open Inventor [Wernecke, 1994] as a user-friendly way to composite objects within a 3D world [Reiners *et al.*, 2002]. SGI Open Inventor was the basis for the *Virtual Reality Modelling Language* (VRML).

What makes scene graphs so attractive? Considering just the visual aspect of representing a scene, a scene graph helps in hiding low-level graphics API details from the user by offering him a high-level concept [Walsh, 2002]. For example, he can tell the scene graph that he would like to display a sphere with the radius 5, instead of having to tell the graphics API how to scale, position and display a group of triangles representing a unit sphere in order to get the same result. Instead of having to focus on *how* to make the system do something, he can focus on *what* it is he wants to do.

3.3.1 Conceptual Representation

Another important property of scene graphs is that they aid in structuring the 3D world by organizing the objects it contains into hierarchies, both on a conceptual and on a spatial level. For example, imagine a room which contains a table, a chair and a bed. The concept of the room *containing* these objects can be expressed through a parent-child relationship yielding the tree depicted in Fig. 2 a).
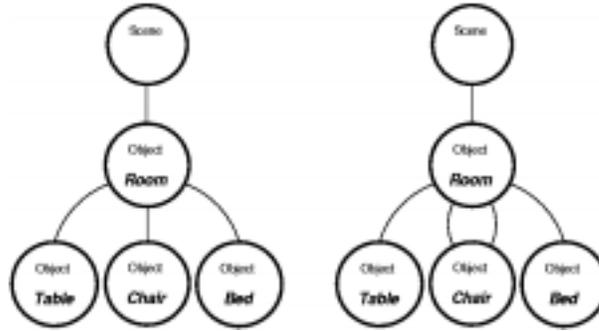
Figure 2: a) Conceptual hierarchy   b) Conceptual hierarchy with instances

The scene node is the root of the scene graph. We see that the relationship of the chair belonging to the room manifests itself in the form of a simple link between the parent (room) and the child (chair). This form of representing such a relationship gives rise to a resource sharing scheme. When we add a second (identical) chair to the scene, the scene graph only has to create another link to the chair object to reflect that change in the scene (Fig. 2 b)).

3.3.2 Representing Spatial Relationships

It was stated above, that the scene graph is also capable of representing spatial relationships, but from the above descriptions we still do not know where the objects are located within the room. Before being able to tackle this problem, we have to look at the term *location* more closely.

A location in a 3D space is defined by a position vector $< t_x, t_y, t_z >$, which is the translation from the origin and the orientation of the object at that position. The orientation is usually specified in the form of a rotation matrix. Using homogeneous coordinates this yields the following two matrices [Foley *et al.*, 1989]:

$$M_{translation} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M_{rotation} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1}$$

The use of a homogeneous coordinate system allows us to concatenate both matrices using matrix multiplication:

$$M_{transform} = M_{translation} \cdot M_{rotation} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

A transformation matrix such as the one above has the ability to convert from one coordinate space to another. For our world representation we have to cope with two different spaces: the *model space* and the *world space*. A 3D model consists of a number of points, so-called vertices. These vertices can be connected with other vertices to form triangles or polygons describing the surface of the object. Each vertex has a positional attribute which is a 3D coordinate. In order to allow this object to be independent of its location in the world, the positional attribute is specified in the objects coordinate space, the model space. The location of an object is given in the form of a transform, the object's *world transform*. Given a homogeneous position in model space in column vector form, $\mathbf{p_{model}}$, and the world transform M_{world}, we can convert it to world space simply by right-multiplying the position vector:

$$\mathbf{p_{world}} = M_{world} \cdot \mathbf{p_{model}} \tag{3}$$

The column vector form is necessary for the multiplication with the matrix to be defined. The way the right-multiplication works is best illustrated when we split up the transform and use parenthesis:

$$\mathbf{p_{world}} = M_{world} \cdot \mathbf{p_{model}} \tag{4}$$

$$\mathbf{p_{world}} = M_{worldtranslation} \cdot M_{worldrotation} \cdot \mathbf{p_{model}} \tag{5}$$

$$\mathbf{p_{world}} = M_{worldtranslation} \cdot (M_{worldrotation} \cdot \mathbf{p_{model}}) \tag{6}$$

So, the position in model space is first rotated and then translated. Instead of just concatenating translation and rotation matrices, we can also concatenate several transforms. Each transform will convert from one coordinate space to another. This allows the chaining of coordinate frames and gives rise to the concept of *local transforms*. A local transform is always relative to a coordinate space. If the coordinate space is the identity space, the local transform is the world transform. The idea of local transforms allows us to add spatial relationships to a scene graph. This is done by introducing *transform nodes*. Using this new node type we can augment the scene graph for the previous example. The result of this augmentation can be seen in Fig. 3.

Every node within the scene graph knows about its local and world transform. But, the only nodes which actually posses these transforms are the transform nodes. The object nodes (Room, Table, Chair, and Bed) only know this information semi-implicitly. The behavior of retrieving local and world transforms is best described inductively:

$$LocalTransform(x) = \begin{cases} \text{identity matrix} & \text{if } x \in Scene \\ \text{identity matrix} & \text{if } x \in Object \\ \text{local transform matrix} & \text{if } x \in Transform \end{cases} \tag{7}$$

$$WorldTransform(x) = \begin{cases} \text{identity matrix} & \text{if } x \in Scene \\ WorldTransform(Parent(x)) & \text{if } x \in Object \\ \text{world transform matrix} & \text{if } x \in Transform \end{cases} \tag{8}$$
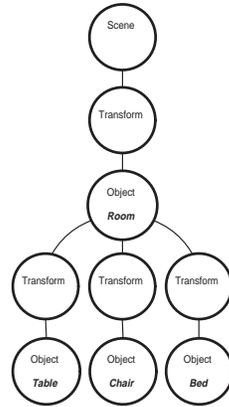
Figure 3: Conceptual and spatial hierarchy

The above behavior clearly shows that only transform nodes need to know about concrete local and world transforms. The corresponding matrices are data members of the actual transform objects. The world transforms are updated during a scene graph traversal which takes place every simulation step by concatenating the local transforms. However, many nodes (as we will see) do not need to specify spatial relationships.

Thus, we chose to express spatial relationships through explicit transform nodes. As world transforms are always recalculated from the local transforms, we can move entire subtrees without disturbing their spatial relationships. For example, when we modify the parent transform of the room, it would move the room and its child objects around, but the position of the objects relative to the room would remain constant. This makes assembly and reuse of complex objects very easy.

3.3.3 Aspect Representation

All aspects should also be reflected in the scene graph structure. The most flexible solution is to represent each aspect as a node in the scene graph. As all the aspects are (in a sense) synchronized via the object location, a node representing that location would be an ideal candidate for the aspects parent node. This role will fall to the transform node mentioned above. In the room example, we would replace the object nodes with the aspects making up that object. All edges attached to the object node would go directly to the parent transforms. These changes can be seen in Fig. 4.

4 Simulation

In this section we add the ability to act to objects and present how all the aspects interact with each other. We will also illustrate how it is possible to add the necessary flexibility so that the
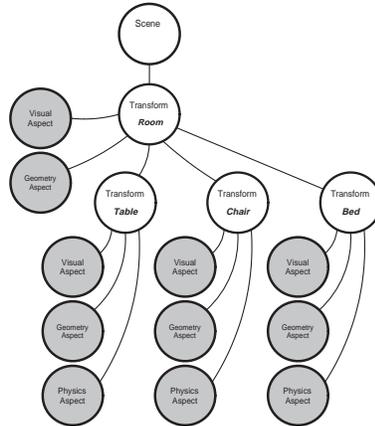
Figure 4: Scene graph with aspects (in gray)

simulator is able to provide more than a single specific simulation.

4.1 Agent Aspect

The agent aspect is the node in our scene graph which distinguishes agent objects from other world objects. According to our agent model it has to perform three basic functions:

- Perceive the environment through perceptors

- Decide which actions to perform next to satisfy its goal

- Perform actions through effectors

The perceptors and effectors represent the agent's interface to the environment. In order to provide a flexible solution we chose to model and implement these agent capabilities as individual classes, rather than having each agent aspect deal with them internally. Thus, the agent does not have to directly access the scene graph structure. Instead, it can receive all the necessary information through its perceptors and perform all actions through effectors. This relationship is illustrated in Fig. 5.

The approach of modeling perceptors and effectors separately is beneficial because it allows us to hide away the implementation details of the sensory and action models from the agent, which as a result only has to deal with comparatively high-level concepts. This design decision also makes the specification of agent abilities extensible. It is very easy to add new perceptors or effectors to an existing code-base. For example, we have only focused on simulating agent vision, but it would be very easy to add an acoustic perceptor, which reacts to sound objects which live in the virtual world represented by the scene graph. Alternatively, an even more exotic perceptor might create a rendered image of the virtual world, which the agent aspect has
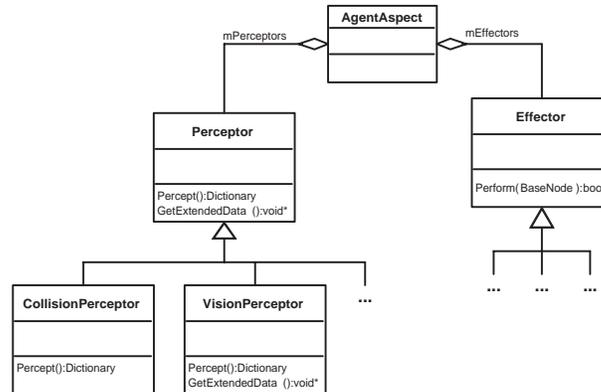
Figure 5: UML diagram showing the relationship between agent aspects, perceptors, and effectors

to analyze to get its sensory information. Thus, this design could be exploited to simulate actual physical devices.

The agent aspect controls which perceptors and effectors it needs. When an agent aspect is attached to an object in the simulation, it performs an initialization procedure. During this procedure it can request specific instances of perceptor and effector classes to be added to itself. The simulator has knowledge about all classes which are managed by the system (more details in the next section) and can therefore accept a requested class by returning a new instance of it or deny the request if the class does not exist. Only when all requested classes are accepted, will the agent aspect be allowed to participate in the simulation. This procedure is necessary, because it guarantees that agent aspects only encounter functionality, which they know how to deal with. For example, if an agent relies on the information provided by a SoundPerceptor and this class is unknown to the system, then the agent will not be allowed to run, as it would never receive the data it requires. Effectors are treated in an analogous way.

4.1.1 Perceptors

We already mentioned above, that a very strong focus has been given to the visual aspect. This is not only the case for the scene graph, but also for the perceptors. The primary ability, which is necessary to navigate through a 3D environment is vision. A *Perceptor* class and its subclasses have the ability to process the scene graph structure to extract sensory information, which is made available to its client (the agent aspect). Every agent receives its own instance of a concrete perceptor class, so it is possible for the perceptor to store internal state. Before the agent aspect executes its thinking process, it can query all its perceptors for information about the world.

Concrete perceptors are derived from the base Perceptor class as illustrated in Fig. 5. Thus, the actual implementation of the perception algorithms can change freely without breaking the agent aspect code. This design is very similar to the *Strategy* design pattern outlined by Gamma

et. al. [Gamma *et al.*, 1995]. As we would like to add arbitrary perceptor implementations to the system, the interface they expose to the agent aspect is crucial. How do perceptors return their data to the agent aspect? The interface has to allow the arbitrary passing of data from the perceptor to the client. In order to strike a compromise between simplicity, safety and generality two mechanisms are made available. The first mechanism is just a dictionary of string and value pairs. This allows easy access to individual data and is sufficient for simple perceptors. But, what if we would like to write a perceptor which returns a rendered image? It would certainly be possible to somehow encode the image into the name-value pair semantics of the dictionary, but a more general (but also unsafe) way to pass data would be beneficial in this case. We provide this through a function which returns a C++ void-Pointer (untyped) to a memory location. As the agent aspect has knowledge about which perceptors it deals with, it also has knowledge about how to interpret the data it returns. Therefore, it is able to cast the void-Pointer back to its original type and use that data structure. This second approach to passing data is useful, when the perceptor not only wants to pass data, but actual objects, back to the client.

Example 1 *Two implemented perceptors from our simulator framework:*

CollisionPerceptor *The CollisionPerceptor allows an agent to receive information about when it has collided with another object of the world. The physics engine is responsible for detecting these collisions and will update the collision perceptor, if present, for each object involved. The only data the CollisionPerceptor of an agent will return is the path in the object hierarchy corresponding to the respective collision partner. This is done using the name-value pair return facility. The "colidee"-entry holds the above information.*

VisionPerceptor *The vision perceptor is a bit more complex than the CollisionPerceptor as its functionality is not provided by an existing subsystem. The VisionPerceptor models the eye of an agent through the same kind of viewing volume we used for the camera frustum. Unfortunately, just culling objects against the VisionPerceptor's frustum is not enough, as it does not handle occlusion. Therefore, we have to perform extra visibility-checks for every object within the view frustum. This is done by tracing rays through the scene. The ray-testing is performed recursively, beginning at the root of the scene graph. The ray is always tested against the bounding-box of a node, before its interior is tested.*

4.1.2 Effectors

An *Effector* has the ability to modify the contents of the scene graph (even create new objects). Every agent aspect can request a number of effectors during its initialization. All concrete effector implementations derive from a common Effector base class, as illustrated in Fig. 5. Thus, we also have an open design, which can be easily customized. Again, the Strategy design pattern comes to mind [Gamma *et al.*, 1995]. What kind of effectors the agent needs is highly dependent on the kind of simulation one wants to perform. For example, in a RoboCup soccer simulation, we might have (among others) a KickEffector, a MoveEffector, a DashEffector and a

CatchEffector (for the goalie). For every action an agent wants to perform he needs to request its accompanying effector. This design facilitates simulation growth, being able to successively add new functions without breaking old functions. It would even be possible to add new functionality in parallel to old functions. Looking at RoboCup again, it would be possible to add experimental functionality through new effector classes. The old effectors would still be available, so this kind of experimentation would not break existing code. Existing agents could slowly migrate to the new interfaces.

4.2 Control Aspects

With all this freedom inside the agent aspects, we still have to address how the actual simulation is controlled. Basically, the legality of the agents actions somehow has to be enforced. For example, a field player requesting and using a CatchEffector doesn't really make sense, as this capability is reserved for the goalie. A simple analogy gave rise to the solution we actually ended up implementing into the simulator. When looking again at the game of soccer, we have an entity on the field which is responsible for making sure the rules of the game are followed: the referee. Thus, it makes perfect sense to employ a similar "construct" to watch over simulations. We call these entities *control aspects* and they are very similar to the agent aspect. In fact, on an implementation level they are equivalent, control aspects are just specialized agent aspect which posses some advanced perceptors and effectors, giving them the ability to analyze the entire scene graph.

Some simulations might be too complex to be watched over by a single control aspect. Going back to the soccer analogy, we also have two line referees which help the field referee. Thus, more than a single control aspect could be employed to monitor the simulation. This makes it easy to extend this functionality. For example, if the rules each control aspect enforces are mutually exclusive it is possible to provide several flavors of a simulation. Imagine a game of soccer with and without the offside rule, only influenced by the presence or absence of an "OffsideControlAspect".

Control aspects have the ability to register a few custom perceptors and effectors. For example, when an agent aspect is added to the simulation it is interesting for the control aspect to perceive which perceptors and effectors the agent tries to request. Based on this the control aspect also has a special effector which allows him to disqualify an agent from the simulation. Thus, aspects and effectors are much more, than just interfaces of the agent to the environment. They are also the basis for an event-like system, where perceptors act as the sinks for events.

4.3 Putting it all together

We have looked at the different aspects of our simulation objects now in quite some detail. Some of the interactions between them have already been hinted at, but the big picture about how they all interact with each other to result in the desired simulation have yet to be presented. The scene graph unifies and triggers this simulation procedure.

4.3.1 Warming up

In the beginning the scene graph does not contain any world objects. It is only composed of the scene node as its root.

The simulator is initialized by registering a host of perceptor and effector classes. After this step is realized, the actual simulation can be initialized. At first the above discussed control aspects are added to the simulation. This can be one or more aspect, depending on the complexity of the task. No control aspect is needed for purely physical simulations. At this stage the world can be populated with objects and their corresponding aspects. During this step every agent aspect performs its initialization procedure, requesting perceptors and effectors. After the world is initialized, we can now start the simulation process.

4.3.2 Simulating

The simulation is performed in a so called *run-loop*. Every iteration of the loop corresponds to a frame being displayed by the simulator. The production of a single frame involves the interaction of all aspects to update the scene graph. At the beginning of the current iteration the world is in a legal state. This means, it contains only agent aspects which were not disqualified and the transform nodes correspond to their corresponding physics aspect's location.

The first group of aspects, which get updated by the simulator are the agent aspects. They use their perceptors to get feedback about the current world state. Each agent processes this information resulting in one or more effector being triggered to achieve its current goal. The usage of perceptors is usually monitored by a control aspect, as there often is a number of limited effector usages allowed in a simulation. For example, in the RoboCup Soccer Simulation you can only execute a single `dash` command per cycle, but you could issue a `turn_neck` command in parallel [Chen *et al.*, 2001].

The use of effectors changes the state of the environment. A move effector will apply forces onto the physics aspect of an object, for example. This brings us to the next step, where the physics engine resolves the resulting object motion using the geometry and physics aspect. At this point, collision perceptors are also notified when applicable. Once this update pass terminates, the location of all objects already reflects the state of the next simulation iteration, bringing us to the final aspect, the visual aspect.

Updating the visual aspect of the scene graph objects begins by locating the camera. Based on this, the scene graph is culled (geometries and lights), resulting in a very conservative estimate on what needs to be displayed. Then a rendering procedure is utilized to bring the simulation on the screen and the next iteration can begin.

4.4 Parameterizing the Simulation

One of the main features of our simulator is the ability to provide more than just a single simulation. The entire architecture, the way we represent a simulation, has been built with this aspect in mind. The aspects and their interactions are responsible for what simulation is performed
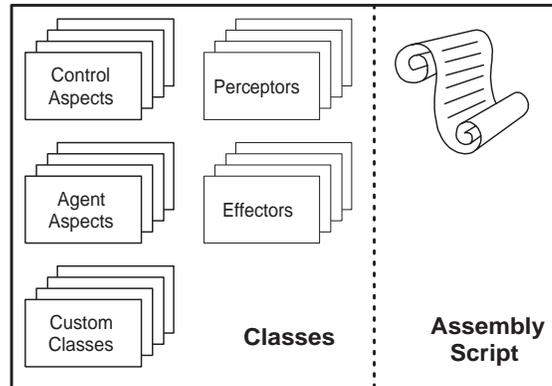
Figure 6: Schematic view of a simulation description

once the simulator is running. Thus, by modifying the concrete aspect implementations and the structure in which they are assembled within the scene graph one also is able to influence, even completely change, the actual simulation. Through modeling all the interaction with other objects via perceptors and effectors we have also introduced the necessary level of abstraction to decouple the implementation details from the aspects and the simulator. This means, the only knowledge the simulator requires is how to deal with the different aspect types. It is, however, completely ignorant about what the concrete aspect implementations do. The simulator only needs to treat individual aspects on a conceptual level.

The notion presented above allows us to parameterized the simulator with a *simulation description*. We have identified a simulation to be composed of two parts, the *classes* which are required to build the simulation and the *structure* in which instances of these classes are combined to form the scene graph. Fig. 6 shows a schematic view of a simulation description. It contains a series of classes, which are added to the class pool of the simulator. These classes are mainly implementations of `AgentAspect`, `ControlAspect`, `Perceptor`, and `Effector` interfaces. In addition to this, the simulation description might also contain other custom classes, which have to be added to the scene graph (e.g. new visual aspects). However, this information alone would be useless, as we still need to know how the simulation is modeled using instances of these classes. This is done using a so-called *assembly script*. It describes the structure of the scene graph and the initial parameters of the objects within the scene graph.

This design requires a tremendous amount of flexibility on the implementation side, as the simulator has to be able to instantiate concrete classes without having any knowledge about them. We also need a way to be able to add new classes to a simulator that can remain static and constant. Part of this flexibility was achieved by integrating Ruby [Matsumoto, 2002] as scripting language into our simulator. For the sake of brevity we omit the details here, for details on our class object system and the object hierarchy in our simulator see [Kögler, 2003].

5 Conclusions and Future Work

In our approach we have provided a flexible way to represent 3D simulations using a scene graph architecture, modeling the aspects of objects in the virtual world and their interactions with the environment directly. By integrating a class object framework it is possible to extend the simulator at runtime, allowing it to be parameterized with simulation descriptions, ultimately giving the simulator the ability to provide more than a single simulation. This makes our simulator ideal as a platform for trying out new simulations.

When performing 3D simulations, a good visualization component is also necessary. In order to provide a flexible simulation component, we have implemented a real-time lighting solution on the basis of the *OpenGL* graphics API. The visualization component can operate in two distinct modes, ambient lighting (fast, but low quality) and local lighting (slower, but high quality), allowing it to scale based on simulation needs and hardware requirements.

The simulator we developed was implemented in a highly portable fashion to support different operating systems. Recently Opcode [Terdiman, 2002] – a library used for collision detection and restricting ourself to Microsoft Windows up to now – has been ported to Linux. The various libraries employed such as ODE, Boost, DevIL, or SDL are freely available. We are convinced that our simulator framework would be an ideal basis for a next generation Soccer Server. Many problems have been solved with the system developed, yet a number of issues still remain.

5.1 Distributed Simulation

The current design of the simulator is very monolithic. We have a single application, which is responsible for the physics simulation, agent simulation and visualization, it is not a server system yet. The current RoboCup Simulation League Soccer Server allows for a distributed simulation. Our most favoured approach is to solve this problem at the scene graph level. The idea would be that every node in the scene graph can either be local or remote. For example, plain objects are always managed by the server. So, they are local to it, whereas they would be remote for a client. Now, changes to these objects have to be synchronized across the network connection to ensure that the state of the simulation is identical (or near identical) for all processes involved.

As the server is responsible for the simulation state, physics, sensing and actual execution of actions should be performed on the server side. On the other hand not all simulation tasks have to be performed by a single client, so the aspects can also be split up. For example, only the server needs the geometry and physics aspect, as it is responsible for running the dynamics simulation. Only a client which actually wants to display the simulation needs the visual aspect. Together with the capability of distributing the application, the decoupling of the visualization and simulation components should be investigated. This would introduce the concept of a stand-alone monitoring client capable of processing the visual aspect of all the objects.

One can see, that the way we chose to represent the simulation by modeling the different aspects of objects will greatly ease the creation of a distributed simulator, as the various parts of the simulation process are already decoupled from each other. This decoupling is one of the

primary strengths of the simulator design, as it allows different aspects to be changed without disturbing the others.

5.2 Soccer Simulation

Up to now, we have only created a simple and small example simulation. A new soccer simulation has to be created. To this end, some important questions have still to be discussed, as the simulator design will encourage exploration and experimentation. How should the soccer agents be represented in 3D? As simple spheres, or using more complex geometric primitives? Through the addition of complex joints it would also be possible to create articulated structures, such as agents with two legs. The design of the simulator allows for such changes to be made easily without breaking too much other code (if any). So there is still work to do, but the current Soccer Server was not built in a day, either.

References

[Chen *et al.*, 2001] Mao Chen, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Jan Murray, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. *RoboCup Soccer Server*. The RoboCup Federation, April 2001. Manual for Soccer Server Version 7.07 and later.

[Ferber, 1999] Jacques Ferber. *Multi-Agent Systems - An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[Foley *et al.*, 1989] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics – Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1989.

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

[Howard *et al.*,] Tom Howard, Artur Merke, Oliver Obst, Martin Riedmiller, and Patrick Riley. New soccer server initiative. Work in progress. Available in the Soccer Server Repository.

[Kitano and Asada, 1998] 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 (IROS)*, pages 419–424, 1998.

[Kögler, 2003] Marco Kögler. Simulation and visualization of agents in 3D environments. Diploma thesis, Fachbereich Informatik, Universität Koblenz-Landau, 2003.

[Matsumoto, 2002] Yukihiro Matsumoto. Ruby: The object-oriented scripting language. `http://www.ruby-lang.org/en/`, 2002.

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

[Reiners *et al.*, 2002] Dirk Reiners, Gerrit Voß, and Johannes Behr. OpenSG: Basic concepts. In *1. OpenSG Symposium OpenSG 2002*, 2002.

[Terdiman, 2002] Pierre Terdiman. Opcode: Optimized collision detection. `http://www.codercorner.com/Opcode.htm`, 2002.

[Walsh, 2002] Aaron E. Walsh. Understanding scene graphs. *Dr. Dobbs Journal*, July 2002.

[Wernecke, 1994] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley, Reading, MA, 2nd edition, 1994.

Available Research Reports (since 1998):

2003

3/2003 *Marco Kögler, Oliver Obst.* Simulation League: The Next Generation.

2/2003 *Peter Baumgartner, Margret Groß-Hardt, Alex Sinner.* Living Book – Deduction, Slicing and Interaction.

1/2003 *Peter Baumgartner, Cesare Tinelli.* The Model Evolution Calculus.

2002

12/2002 *Kurt Lautenbach.* Logical Reasoning and Petri Nets.

11/2002 *Margret Groß-Hardt.* Processing of Concept Based Queries for XML Data.

10/2002 *Hanno Binder, Jérôme Diebold, Tobias Feldmann, Andreas Kern, David Polock, Dennis Reif, Stephan Schmidt, Frank Schmitt, Dieter Zöbel.* Fahrassistenzsystem zur Unterstützung beim Rückwärtsfahren mit einachsigen Gespannen.

9/2002 *Jürgen Ebert, Bernt Kullbach, Franz Lehner.* 4. Workshop Software Reengineering (Bad Honnef, 29./30. April 2002).

8/2002 *Richard C. Holt, Andreas Winter, Jingwei Wu.* Towards a Common Query Language for Reverse Engineering.

7/2002 *Jürgen Ebert, Bernt Kullbach, Volker Riediger, Andreas Winter.* GUPRO – Generic Understanding of Programs, An Overview.

6/2002 *Margret Groß-Hardt.* Concept based querying of semistructured data.

5/2002 *Anna Simon, Marianne Valerius.* User Requirements – Lessons Learned from a Computer Science Course.

4/2002 *Frieder Stolzenburg, Oliver Obst, Jan Murray.* Qualitative Velocity and Ball Interception.

3/2002 *Peter Baumgartner.* A First-Order Logic Davis-Putnam-Logemann-Loveland Procedure.

2/2002 *Peter Baumgartner, Ulrich Furbach.* Automated Deduction Techniques for the Management of Personalized Documents.

1/2002 *Jürgen Ebert, Bernt Kullbach, Franz Lehner.* 3. Workshop Software Reengineering (Bad Honnef, 10./11. Mai 2001).

2001

13/2001 *Annette Pook.* Schlussbericht "FUN - Funkunterrichtsnetzwerk".

12/2001 *Toshiaki Arai, Frieder Stolzenburg.* Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing.

11/2001 *Kurt Lautenbach.* Reproducibility of the Empty Marking.

10/2001 *Jan Murray.* Specifying Agents with UML in Robotic Soccer.

9/2001 *Andreas Winter.* Exchanging Graphs with GXL.

8/2001 *Marianne Valerius, Anna Simon.* Slicing Book Technology — eine neue Technik für eine neue Lehre?.

7/2001 *Bernt Kullbach, Volker Riediger.* Folding: An Approach to Enable Program Understanding of Preprocessed Languages.

6/2001 *Frieder Stolzenburg.* From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking.

5/2001 *Oliver Obst.* Specifying Rational Agents with Statecharts and Utility Functions.

4/2001 *Torsten Gipp, Jürgen Ebert.* Conceptual Modelling and Web Site Generation using Graph Technology.

3/2001 *Carlos I. Chesñevar, Jürgen Dix, Frieder Stolzenburg, Guillermo R. Simari.* Relating Defeasible and Normal Logic Programming through Transformation Properties.

2/2001 *Carola Lange, Harry M. Sneed, Andreas Winter.* Applying GUPRO to GEOS – A Case Study.

1/2001 *Pascal von Hutten, Stephan Philippi.* Modelling a concurrent ray-tracing algorithm using object-oriented Petri-Nets.

2000

8/2000 *Jürgen Ebert, Bernt Kullbach, Franz Lehner (Hrsg.).* 2. Workshop Software Reengineering (Bad Honnef, 11./12. Mai 2000).

7/2000 *Stephan Philippi.* AWPN 2000 - 7. Workshop Algorithmen und Werkzeuge für Petrinetze, Koblenz, 02.-03. Oktober 2000 .

6/2000 *Jan Murray, Oliver Obst, Frieder Stolzenburg.* Towards a Logical Approach for Soccer Agents Engineering.

5/2000 *Peter Baumgartner, Hantao Zhang (Eds.).* FTP 2000 – Third International Workshop on First-Order Theorem Proving, St Andrews, Scotland, July 2000.

4/2000 *Frieder Stolzenburg, Alejandro J. García, Carlos I. Chesñevar, Guillermo R. Simari.* Introducing Generalized Specificity in Logic Programming.

3/2000 *Ingar Uhe, Manfred Rosendahl.* Specification of Symbols and Implementation of Their Constraints in JKogge.

2/2000 *Peter Baumgartner, Fabio Massacci.* The Taming of the (X)OR.

1/2000 *Richard C. Holt, Andreas Winter, Andy Schürr.* GXL: Towards a Standard Exchange Format.

1999

10/99 *Jürgen Ebert, Luuk Groenewegen, Roger Süttenbach.* A Formalization of SOCCA.

9/99 *Hassan Diab, Ulrich Furbach, Hassan Tabbara.* On the Use of Fuzzy Techniques in Cache Memory Managament.

8/99 *Jens Woch, Friedbert Widmann.* Implementation of a Schema-TAG-Parser.

7/99 *Jürgen Ebert, and Bernt Kullbach, Franz Lehner (Hrsg.).* Workshop Software-Reengineering (Bad Honnef, 27./28. Mai 1999).

6/99 *Peter Baumgartner, Michael Kühn.* Abductive Coreference by Model Construction.

5/99 *Jürgen Ebert, Bernt Kullbach, Andreas Winter.* GraX – An Interchange Format for Reengineering Tools.

4/99 *Frieder Stolzenburg, Oliver Obst, Jan Murray, Björn Bremer.* Spatial Agents Implemented in a Logical Expressible Language.

3/99 *Kurt Lautenbach, Carlo Simon.* Erweiterte Zeitstempelnetze zur Modellierung hybrider Systeme.

2/99 *Frieder Stolzenburg.* Loop-Detection in Hyper-Tableaux by Powerful Model Generation.

1/99 *Peter Baumgartner, J.D. Horton, Bruce Spencer.* Merge Path Improvements for Minimal Model Hyper Tableaux.

1998

24/98 *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE Worldwide.

23/98 *Peter Baumgartner, Norbert Eisinger, Ulrich Furbach.* A Confluent Connection Calculus.

22/98 *Bernt Kullbach, Andreas Winter.* Querying as an Enabling Technology in Software Reengineering.

21/98 *Jürgen Dix, V.S. Subrahmanian, George Pick.* Meta-Agent Programs.

20/98 *Jürgen Dix, Ulrich Furbach, Ilkka Niemelä .* Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations.

19/98 *Jürgen Dix, Steffen Hölldobler.* Inference Mechanisms in Knowledge-Based Systems: Theory and Applications (Proceedings of WS at KI '98).

18/98 *Jose Arrazola, Jürgen Dix, Mauricio Osorio, Claudia Zepeda.* Well-behaved semantics for Logic Programming.

17/98 *Stefan Brass, Jürgen Dix, Teodor C. Przymusinski.* Super Logic Programs.

16/98 *Jürgen Dix.* The Logic Programming Paradigm.

15/98 *Stefan Brass, Jürgen Dix, Burkhard Freitag, Ulrich Zukowski.* Transformation-Based Bottom-Up Computation of the Well-Founded Model.

14/98 *Manfred Kamp.* GReQL – Eine Anfragesprache für das GUPRO–Repository – Sprachbeschreibung (Version 1.2).

12/98 *Peter Dahm, Jürgen Ebert, Angelika Franzke, Manfred Kamp, Andreas Winter.* TGraphen und EER-Schemata – formale Grundlagen.

11/98 *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.

10/98 *Jörg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.

9/98 *Dieter Zöbel.* Schedulability criteria for age constraint processes in hard real-time systems.

8/98 *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.

7/98 *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.

6/98 *Ulrich Furbach, Michael Kühn, Frieder Stolzenburg.* Model-Guided Proof Debugging.

5/98 *Peter Baumgartner, Dorothea Schäfer.* Model Elimination with Simplification and its Application to Software Verification.

4/98 *Bernt Kullbach, Andreas Winter, Peter Dahm, Jürgen Ebert.* Program Comprehension in Multi-Language Systems.

3/98 *Jürgen Dix, Jorge Lobo.* Logic Programming and Nonmonotonic Reasoning.

2/98 *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme.* Zeitstempelnetze in technischen Anwendungen.

1/98 *Manfred Kamp.* Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.