Multi-Agent Systems and Virtual Reality for Interactive Prototyping

F. Harrouet, P. Reignier, J. Tisseau
Laboratoire d’Informatique Industrielle
Ecole Nationale d’Ingénieurs de Brest — C.P. n° 15, 29608 BREST Cedex (France)
harrouet,regnier,tisseau}@enib.fr — http://www.enib.fr/LI2/

ABSTRACT
This paper introduces a Virtual Reality platform based on a dynamic multi-agent programming language. These tools have been designed to show that simulating a Multi-Agent System in a virtual environment with dynamic properties can be used for Interactive Prototyping. This kind of prototyping has to be considered when the designed system cannot be described as a whole but as a set of autonomous components with many interactions. Due to the fact that these interactions are very complex to model before simulating the system, we propose to let the designer enter inside the system and dynamically build, tune and mend the model.

After making out a list of requirements to achieve this goal, we present our tool. Then, an example shows, through a simple application, what an Interactive Prototyping session looks like.

Keywords: Multi-agent systems, virtual reality, simulation, dynamic language, interactive prototyping.

1 INTRODUCTION
The aim of this paper is to point out the fact that a multi-agent approach in a virtual reality environment, as performed with our oKis/AlEvé platform, can be run as a tool for dynamic/interactive prototyping.

The classical spiral, very often used to build a prototype of a system to be designed, consists in the following loop:
1. to generate the prototype,
2. to test the prototype,
3. to adjust the prototype according to the test’s results,
4. to go to step 2 while needed.

This approach is quite efficient in many cases where the described system can be seen as a whole corresponding to one global model. However, when the system becomes complex (i.e. made up of multiple parts in interaction), this global model is much harder to design and to tune, due to the fact that every possible interaction cannot be listed in an exhaustive way. According to this remark, our main goal is to introduce a tool which allows to handle intrinsically distributed systems to dynamically and interactively model their behavior.

If the simulated system is complex, then its global behavior cannot be fully described. Therefore, it would be easier to try to give its components the best behavior when peculiar situations happen to them. This can be done only when the simulation of the system is running because we are not able to know in advance every possible configuration of the system. Thus, the prototyping loop can disappear and the user and the designer of the system have to be present inside the simulated system. The use of a tool allowing to perform this kind of prototyping should consist in:

- describing an initial state of the system to work on as a set of components with their own behaviors,
- letting the tool simulate the system’s global behavior which results from interactions between components,
- allowing the user to modify and mend behaviors and structures during run-time.

In section 2, we will describe what we exactly define as Interactive Prototyping. The functionalities required by a tool to achieve this goal will be presented section 3. Then, a possible solution, illustrated by an example, will be shown section 4.

2 INTERACTIVE PROTOTYPING AND VIRTUAL REALITY

Nowadays, Virtual Reality is a term which is very commonly used, in many contexts and sometimes with different meanings. We will try here to find out the main meaning of this term that suits to Interactive Prototyping. This will be done considering first the animation aspect, then the simulation aspect, and finally the interactive point of view.

2.1 3D animation
Virtual Reality is usually seen as a graphic animation in a 3D world. In this case, people only have to watch the animation but can neither do anything else nor interact with it. Sometimes, the user can travel in the virtual world, choosing its own way to look at different parts of the system under many points of view. In this case, the animation is qualified as "walk through animation".

The quality of such a tool can be resumed to the quality of the rendering (3D images on large screens or head-mounted displays, 3D sounds, . . . ). The user can only travel and look around. Therefore, this is not intensive enough to be run for Interactive Prototyping. Nevertheless, this kind of tool can be useful for architectural and artistic designing [1, 2].

2.2 Interactive simulation
Virtual Reality can also be seen as a tool for Interactive Simulation. It means that in the 3D world, the user will not only be able to move but also to interact with the components of the represented system according to predefined parameters. Thus, in this situation, the only way to act on the animation, is to change the values of these parameters and nothing else.

The designer of such an animation has to think about which parameters will be accessible to the user. Then, he must give the user a way to control them. A very common way to control these parameters is to use a classical graphic user interface (control panels made up of buttons, switches, sliders, . . . ), but in 3D worlds we often use specialized devices to control geometric parameters (joysticks, 3D mice, data gloves, trackers, force feedback actuators, . . . ). The more accurate and easy to use these devices are, the better the quality of this tool is. This kind of animation can be useful to install premises [3], to train people to work in a specific environment (SOFI project [1]), or to work in tele-operation [4]. However, the only changes to be allowed must be predefined by the designer before the user acts, and consequently, they are very often structural (as moving objects).

2.3 Interactive prototyping
We try here to get rid of the main limitation of the above kind of tool: a restricted expression of changes, due to the fact that the designer should think about every possible change the user could try. Since prototyping represents the action of designing, building the model of the system, the previous tools cannot be run for this. They can be used to test the prototype in several predefined situations, but they do not allow to build such a prototype. The designer builds the system, the user tests it and the designer re-builds the system until the user is satisfied of his tests’ results. This approach can be described as “prototyping” but cannot be qualified as “interactive” (it corresponds to the prototyping spiral seen section 1).

ISAS’99, volume 3, pages 50–57, Orlando (USA), August 1999
Then, if we want to have an Interactive Prototyping process, we must give the user the opportunity to substantially change the behavior of the system’s components, whenever and however he wants. This can be done in order to try many behaviors against each other, to simulate defaults or mistakes in a process, to watch how the system reacts to an unknown situation, to simulate the reconfiguration of an industrial process, and many operations which cannot be done on a pre-established simulation.

Therefore, if we want to give the user the full control of his experiments on the system, he has to be able to use the same expressions and ways to act than those used by the components in the system. This means that if a language is used by the designer to describe the system’s components and their behavior, the user has to use the same language during the experiment. In this case, more than testing the component’s behavior, he will be able to tune or to mend them in situation, while running them.

Thus, we can say that the user is in “immersion through the language” so that there is no longer any limit between him and the designer of the simulated system. This is the main meaning we give to Interactive Prototyping, and that’s the way we consider Virtual Reality too (as much as being in sensitive immersion). According to our own opinion, the three main aspects of Virtual Reality can be summarized as follows:

- Animation → “what the model is”
- Simulation → “what the model does”
- Interactive Prototyping → “what the model becomes”

Further explanations on our systemic approach of Virtual Reality and Interactive Prototyping are presented in [5].

2.4 Multi-agent systems

This kind of system is a set of components which are not only static objects but which also have a dynamic behavior. More than just looking like real world’s objects, the virtual components must behave as real ones.

In this kind of virtual world, the designer cannot know in advance how the system will change during the simulation. The global evolution of the system is the consequence of many interactions between its components. These ones react according to local information, so that it is very difficult to deduce a global result from so many local data. In this case, the only way to determine the global behavior of the represented system is to let it work.

Thus, the simulated system is a Multi-Agent System in which each component is an agent endowed with its own goal and behavior. It implies that the behavioral/dynamic aspect of the virtual components must be taken under consideration as much as their structural/static aspect. The designer of such a system cannot model it as a whole but has to describe each agent’s behavior using the following sequence:

- perception, in the virtual environment (other agents or objects),
- decision-making, according to its own perception, state and goal,
- action, in the environment and on itself.

In such a tool, the user can have a very important role. He can introduce changes and make the simulated system go in a state which may never have been reached without its intervention. A possible way, for the user, to act in the virtual world, is to use an “avatar” to perform several actions, and to see it as an agent in that world. But more than that, the user must be able to dynamically change every part of the system by rewriting their behavior when needed.

3 REQUIREMENTS FOR INTERACTIVE PROTOTYPING

In the previous section, we have shown that the same formalism should be used by the designer to describe the multi-agent simulation and by the user to experiment and tune the system. Then, we will try to find out which properties should be found in a development environment to achieve this goal.

3.1 3D worlds and sensitive immersion

The main visible aspect of Virtual Reality concerns the sensitive immersion in a 3D environment. High-level concepts like 3D geometry, kinematics, cameras, scenes, viewers or manipulators can be found in 3D libraries such as Open Inventor [6], OpenGL Optimizer [7], Fahrendorf [8]. Ilog Vision [9] . . . These libraries are useful to build a Virtual Reality tool but they do not offer sufficient abstractions to be used by an end user.

However, there are Virtual Reality platforms that can be run by end users, such as Virtual Builder II [10], Minimal Reality Toolkit [11] and its Environment Manager [12], Clovis [13], Virtual Design [14], Dose [15] or Dive [16]. With these platforms, the concept of 3D autonomous object exists so that the end user, building a virtual world, can concentrate its work on the application more than on the technical/graphical aspects. A large range of specific devices are managed by these tools. Different kinds of behavior can be predefined, from simple callbacks to high-level intentions.

3.2 A dynamic programming language

In subsection 2.3, we have reached the conclusion that the user should be in “immersion through the language” and should describe the behavior of many components. Thus, we must use, to design the simulation, a language which is both object-oriented and based on run-time code rewriting. Nevertheless, for many languages the qualifier “dynamic” or “dynamically extensible” does not represent the point we are interested in.

Most of these languages, and especially Java, could imply that it is possible to rewrite the code of an application at run-time but it is far from granted. These languages allow to load and use extension libraries that may be compiled after the application was built. Thus, the main application must be designed to explicitly load such libraries. Moreover, these ones can only extend what was designed to be extended in the main application.

For example, in Java, it is possible to use a ClassLoader to load a compiled class that must extend a more abstract class managed by the application. If it does not extend a predefined class, then the application will only know that it is an Object and will not be able to use it very efficiently. By using ugly hacks, we can try to load many times a given class with several changes in its code. Then, different classes (with the same name) are obtained in the application but it is impossible to change the code of an existing class.

The C/C++ languages that are not qualified as “dynamic” allow exactly the same operation in an even easier way (using calls to dlopen(), dladdr() and dlclose() functions — see the UNIX man pages). Actually, you can design for example, a C++ application managing an abstract class Vehicle and let many other designers create many specific classes extending Vehicle (Car, Lorry, Truck, . . .) in dynamic libraries. The only thing to know about these libraries is that each one provides its own predefined function (say for example: extern "C" Vehicle * newVehicle(...) ;) that the main application can use to obtain a new instance of a specific vehicle. If a class is no longer used, its library can be unloaded, and eventually, reloaded after code modifications and a new compilation. As with Java, different classes with the same name can be loaded.
The main limitation to these extension libraries lies in the fact that it is only possible to load classes which inherit from the predefined classes of the application. They are explicitly loaded and used in specific points of the program. To dynamically introduce a totally different class, the main application should be modified to manage it. Thus, this kind of application can only be extended in the way the designer allowed it.

Interpreted languages such as Scheme or Prolog use similar representations for code and data so that the code might be changed at run-time as easily as data are. Therefore, this kind of languages can be used to design self-re-writing programs but the syntax and the style used in this case have nothing in common with "classical" programming. More than the fact that the programming style may be difficult to handle for a non-specialist user (in comparison with imperative languages), object-oriented extensions of these languages must be studied with a particular attention. Actually, these extensions can be implemented using classes (every instance of the same class share the same code) or as prototype languages (each instance has its own code), but it is much harder to use a mix of these two approaches. This is a very important point of our objective because we want the user to be able to change entire classes or only several instances during Interactive Prototyping. We call this property "instance granularity". For us, this expression means that the most specific behaviors are not only described in very specific classes but can also be given to particular instances (which then differ from their original class). The reason for this choice is a simple projection of what we do in reality; when someone makes changes on his own car, he does not change every car of the same model. But afterwards, the changes on one object seem to be better than the original version, it can be useful to generalize this new behavior to the entire class too. Thus, the prototyping language must offer an easy and clear way to allow changes in both cases.

3.3 Active objects, concurrency and scheduling

In subsection 2.4, we have declared that more than designing a global control for all the components in the simulation, it is better to let these components "live" and interact to produce themselves the global behavior of the simulated system. This means that each component should be given its own autonomy in the execution of its behavior. Thus, more than being passive objects whose methods are called from an external control, the components should be active objects, "living" in concurrency, and deciding on their own to trigger actions according to their local perception.

This kind of execution could imply the use of object-oriented and multi-threaded programming languages such as C++ with Posix threads or Java. But this must be used with a lot of care. Actually, we are not performing a simple animation but a simulation of a system made up of many concurrent objects. In these circumstances, the tool used to activate the objects must not introduce a bias in the simulation. The sharing of the execution time has to be as fair as possible. In a discussion during the SMAGET’98 meeting (CEMAGREF, October 1998, Clermont-Ferrand, France), Jacques Ferber made an oral intervention concerning this problem. He pointed out the fact that the designer of the simulation of a distributed system must not only describe the behavior of each component of the system but also how these objects interact to make the way the scheduling is done in the simulation. This care would allow the designer or someone else to find out in the scheduler an artifact that could drive the simulation in a state which would have never been reached without it.

An illustration of this problem stands in the fact that the priority of a thread is not interpreted in the same way under UNIX and under Windows. In the first case, we are sure that a thread with a low priority will not be active as long as other threads with higher priorities can be activated. But in the second case, the priority of a thread only represents a probability to be activated. Moreover, the scheduling of Java threads uses a cooperative policy. This means that a high priority thread is responsible for letting the others become active (using the yield() function). Then, a thread can keep the entire activity for itself. By using another ugly hack, it is possible to simulate a round-robin policy (a high priority thread sleeping during a few milliseconds in an infinite loop), but the solution is not very stable. Actually, in a small Java program using this hack with a lot of threads, it occasionally happens that the activity is given many consecutive times to the same thread, during a few seconds or more, and is then given to the other threads without any visible reason. Such a scheduling can be sufficient for parallel computing but must not be used to simulate distributed systems.

The problem of the scheduling policy is the same with Posix threads which are scheduled in a cooperative way too. The round-robin policy is only a theoretical approach for processes with the superuser privileges (due to the fact that threads’ activity is managed with processes’ activity — see the UNIX man pages). Therefore, it must be handled with care. Even if it is possible to simulate a round-robin policy (with a hack similar to the previous one), there is another scheduling problem due to the fact that Posix threads are stored in queues (First In – First Out). This means that in a set of same priority threads, these ones will always be activated before those in the same group (except for those which are blocked). This could be very detrimental to the fairness of the simulation. For example, if a set of active objects are regularly in competition for the same resources, those which have been created first have an advantage over the next ones because they always react before them. Therefore, with this sequence, the scheduler introduces an artificial and unwanted priority between the active objects that could bias the simulation.

However, the queues used to manage the threads ensure a very interesting property: we are sure that every thread of the same priority will be activated the same number of time (except for those which are blocked). A good solution could consist in a round-robin keeping this property while breaking the unwanted sequence. This could be done with a scheduling policy which allows letting the execution of a thread quite simple but requires the building of an entirely new scheduler.

Many parallel programming environments exist, allowing to build active object applications. These environments consist in libraries (Poggi & Rimassa’s library [17], Para++ [18] or in extensions of existing object-oriented languages (sC++ [19], Parsec [20], Jade [21], ICC++ [22]). Most of these environments are based on an event driven execution relying on synchronization primitives such as the "rendez-vous". They can be formally described (this is the case for sC++ [22]) or that a theoretical study allows to prove some properties about the execution or allows to detect inconsistencies such as "deadlock". But the entire model only relies on the synchronization primitives and no detail is given about the scheduling policy. Then, these environments are very useful for parallel computing but do not fit very well the simulation of distributed systems.

3.4 Agents and communication

On the assumption that an active object environment with the scheduling policy described figure 1 is available, we must now study how the interactions between these active objects can be performed.

The simulated system is made up of many components whose interactions determine the global behavior. Therefore, in this kind of application, the communication means have to be studied with a particular attention. The most simple communication mode which can be used consists in synchronous method invocations, as usual in imperative object-oriented languages. This kind of communication implies that the receiver receiving the method call must execute it; the entire decision-making has been
made by the caller. The caller waits until the method is ended (in fact, the time needed to execute the method is spent on the caller’s activity). The main care to be taken consists in synchronization with critical sections around shared resources to ensure that many active objects cannot perform some antagonist actions at the same time. This is usually done with mutual exclusion semaphores (hidden behind the synchronized keyword in Java).

However, in Martin Carroll’s article [24], the author explains that it is very difficult to detect all the possible critical sections and then, in practice, a lot of them are very often forgotten. Thus, he introduces a pattern based on a mix between a synchronous method invocation and an asynchronous processing to design active objects. The asynchronous method invocation consists in putting a message in the receiver’s mailbox. Then, the communication between active objects becomes asynchronous. This implies that the execution of the service is no longer performed by the receiver when checking its mailbox. Then, the service does not need to be synchronized (only one execution at the same time), only the mailbox management requires a synchronization. Therefore, according to this point of view, this method invocation seems to be easy to use. Nevertheless, this implies that the active objects’ behaviors must be written in a very different way. Actually, the result of a service call may be returned a long time after the invocation was done, so that the caller may wait for this result or do something else while keeping an eye on the result delivery. Furthermore, each active object must manage the actions it decides to do on its own as well as those asked by the others (through its mailbox).

Despite the fact that the programming style is quite different from synchronous method invocations, however it allows to consider active objects as agents. The main difference between an object and an agent can be represented figure 2, which is directly inspired from Jacques Ferber’s figure [25, 26]. We can see here that method invocations are only used inside the agent itself, and that the communication between agents is performed with communication actions often called “speech acts”. This communication is based on messages and is incompatible with the previous pattern for active objects. The KQML formalism [27, 28] inspired by the theory of speech acts can be used to design the different kinds of messages to be exchanged. The very particular point that makes an agent different from an object is that its own intentions and goals are used to filter for communication actions. This means that the decision to perform a service is not only due to the caller but is also submitted to the receiver’s objectives. Thus, very high-level intentions can be given to such components, and, why not, consciousness [29] or emotions [30].

Moreover, concerning Interactive Prototyping, the most interesting thing relevant when describing the components of the simulation as agents relies on the modularity of this model. Actually, if the user is able to dynamically change a component’s behavior, the code describing this one must not be deeply dependent from the code describing the other components’ behavior. If communications between the components are restricted to message passing, then, internal changes will not affect the existing messages’ semantics. However, the dynamic properties of the system allow to create new kinds of messages for the new semantics which may be introduced by these changes.

The fact that an agent is seen by the others through its communication slots is a very important point concerning the multi-agent approach for Interactive Prototyping. The way an agent is made inside does not matter to the others. Thus, if the user drives an “avatar” to perform actions in the virtual world, then, he can be seen has an agent similar to the others in the simulation.

Consequently, the above requirements imply the use of an object-oriented programming language, with dynamic properties, ensuring the full control of concurrency, using many communication means between objects/agents and offering high-level abstractions for 3D environments.

4 OUR PROPOSAL: oRis/ARêVi

The preliminary study, previously summarized, drove us into realizing a tool to fulfill our needs. This is a Virtual Reality platform allowing to perform Interactive Prototyping on Multi-Agent Systems. It can be described as a dynamic, parallel and object-oriented programming language coupled with a high-level Virtual Reality toolkit.

4.1 The oRis language

The first step in our realization consists in designing the oRis programming language which fulfills many of the requirements described in subsections 3.2, 3.3 and 3.4.

oRis can first be seen as an interpreted object-oriented programming language whose syntax is very near from C++ or Java. Each time a concept used in oRis already exists in one of these two languages, we try to use a similar way to express it. However, oRis supports much less low-level details than these languages (only one integer format, only one float format, no character but a built-in string type, no pointer but object names, no array but a built-in multi-dimensional vector type, . . . ). Our language allows multiple inheritance and is strongly typed. Instances are dynamically created (as with Java) and are referenced through an explicit object name which represents the metaphor of an object pointer. This object name is automatically given to the instance and is made up of the class name and an integer identifier so that the user can easily “talk” to an instance. For example, Car.5->brake() is much easier to express than ((Car)objectPool[theIndex]).brake(). Unreferenced instances can be automatically deleted, but we can dynamically choose which instances are submitted to this “garbage collector”, and we can always explicitly destroy an instance. The language can be extended with C++ dynamic libraries making C++ objects, methods or functions accessible from oRis code. Thus, specialized C++ classes (GUI, 3D, network management, . . . ) can be embedded inside oRis classes and then extend the application fields.

One of the main characteristics of this language lies in the fact that all that can be expressed in the initial source files can also be expressed at run-time. The entire language is available on-line and can be very easily used to change the running program whenever we want. The new source code can be introduced to perform a direct action such as instance creation/deletion, method call, . . . Another purpose can consist in the rewriting of some existing code:

- global function rewriting,
- method rewriting in classes,
- method rewriting in instances (instance granularity).

In the same way, the new code can contain new declarations and definitions:

- new global functions,
- new classes

class Car : Vehicle /* ... */;
- new attributes and methods in classes

float Car::avgSpeed;
void Car::park(void) /* ... */;
- new attributes and methods in instances (instance granularity)

float Car.5::avgSpeed;
void Car.5::park(void) /* ... */;
There are many ways to introduce new source code in a running application:

- the user can write it in a dialog window,
- the program already running can build a string containing this new code and call the `parse()` function,
- the program already running can load it from a new source file,
- some specialized instances can supply this code from a specific origin such as network, ...

To make these modifications easier, a graphical browser allows the user to inspect every function, class, instance, method, of the application. With this tool, we can, for example, select an instance, change the values of its attributes, change its methods and try them. This browser is entirely written in oRis using both an introspection/reflection package and a GUI package provided by this language.

Another main characteristic of oRis lies in the fact that active objects can easily be created and controlled. If an object is endowed with a `main()` method, then it is considered as an active object. This method represents the entry point of its behavior and is automatically called by the scheduler (and restarted when ended). Moreover, the user can launch other activities in parallel with the active objects, and each active object can consist in several concurrent activities (using the `start primitive` — see figure 3). Of course, basic synchronization tools are available, such as mutual exclusion semaphores or low-level critical sections (which ensure no activity switching during code execution). Furthermore, three possible scheduling policies can be chosen:

- **Cooperative:** each activity explicitly gives up its execution,
- **Preemptive:** each activity is executed during a chosen time slice,
- **Deeply parallel:** an activity switch occurs between each oRis micro-instruction (slower but very useful to detect synchronization failures).

The way an activity is picked to be executed can either be a *serial* fashion (First In – First Out, should never be used) or a random one (for the fair round-robin see figure 1). A work about the role of the scheduler in a simulation has been made in [31].

Communication between objects can be performed by direct method invocations as usual when working with objects. A more specific mechanism lies in the use of "attribute callbacks" which is a concept similar to "fields" and "routes" specified in VRML [32] or "links" used in the Lightsing platform [33]. These are objects in charge of triggering an action just before the value of a specific attribute of a specified instance is changed, and another action just after. This allows, for example, to bound the possible values of an attribute or to propagate variations through a set of connected objects (loops in propagation are detected and broken). Thus, this can be seen as communication means which ensure that a set of objects keep consistent.

Concerning Multi-Agent Systems, the communication relies essentially on messages. A message contains at least the name of its emitter and can be extended in many classes as needed by the communication protocol. Such a message, after being composed, is able to reach the receiver’s mailbox and then, this latter is able to read and process it whenever it wants. This is an asynchronous "peer to peer" communication but these messages can be involved to perform a broadcast communication too. Each agent can dynamically declare itself to be sensible to some kinds of messages so that an appropriate callback is triggered when such a message is broadcasted. In this case, many receivers (not even known by the emitter) can react to a message broadcasting. These callbacks are triggered in a synchronous way, but they can simply consist in putting the broadcasted message in the receiver’s own mailbox to be processed as an asynchronous message.

Then, from now on, our oRis language allows to build concurrent simulations made up of objects, active objects and agents. With this tool we can achieve applications mixing low-level and high-level concepts, we can dynamically browse and change the content of a running simulation, and we can control the way the scheduler works. Thus, the following stage consists in embedding this language in a *Virtual Reality* environment to perform Interactive Prototyping experiences.

### 4.2 The ARÉVi platform

ARÉvi is first a *Virtual Reality* library made up of many C++ classes concerning 3D geometry, kinematics, visualization and interactions. These classes represent high-level concepts, which are not dependent on a specific 3D library, so that they can use different ones (*Open Inventor*, *Rog Vision*, *OpenGL*, ...). Some specific devices are managed to allow sensitive immersion (head-mounted displays, trackers, 3D glasses, data gloves, ...).

The oRis → C++ link has been used to make these classes available from our language. The main ARÉvi classes that can be used in oRis are shown figure 4. They can be described as follows:

- **Object:** root class in oRis,
- **ArEntity:** object with a 3D shape and kinematics,
- **ArLight:** light (many kinds),
- **ArCamera:** camera to choose a 3D point of view,
- **ArViewer:** window to display a camera image,
- **ArScene:** set of 3D objects to be shown,
- **ArUniverse:** universe containing everything concerning 3D.

`ArEvent` object sensible to hardware events (key stroke, mouse click, ...)

A particular point lies in the fact that the visualization and interaction means are considered as any other instance of the simulation. Therefore, they can be designed as agents with their own behavior. For example, we can use a viewer which decides on its own to change the display mode (normal, no texture, wireframe, ...) according to the measured performances (number of frames per second). We can also use a scene which automatically decides to integrate the entities near its geographic area and to reject the others.

Running a *Virtual Reality* application with ARÉvi consists in extending these oRis classes to describe objects/agents and interaction means, in letting them be activated by the scheduler, and then, in making interactions and changes with the specific devices or in "immersion through the language".

### 4.3 An example

An illustration of the use of oRis/ARÉvi in an Interactive Prototyping session will be given through a very simple example. It consists in the interactive building of mechanical gears and in the interactive link of these ones with a control process.

Before launching the simulation, we describe, in an oRis source file, two kinds of active objects: the *GearWheel* class and the *Engine* class. Each *GearWheel* has a 3D representation with its own radius and a behavior which consists in:

- the detection of neighbors (with sensors),
- the avoidance of collision with neighbors,
- the propagation of rotation to neighbors (using "attribute callbacks").

Each *Engine* is represented by a cube, has its own speed and a behavior that rotates the *GearWheel* to which it is connected.

Then, we launch the simulation, we create several *GearWheels* and several *Engines*, and we watch them "live" in a viewer as shown figure 5. With the mouse and the keyboard, we trigger callbacks which start/stop
Concerning measured performances, the example about KorSo simulating it. The chosen subject was quite simple to be that brings him to build, test and tune a system while allows a user to perform an behavior.

This example shows that our oRis/ARéVi platform allows a user to perform an Interactive Prototyping session that brings him to build, test and tune a system while simulating it. The chosen subject was quite simple to be explained in a few lines, but a real application about the KorSo production cell is described in [34, 35]. Concerning measured performances, the example about the gearings and the Petri net runs with an average rate of fifteen frames per second (on a SGI–O2 with a R10000 processor at 175MHz). This example is made up of approximately thirty agents. The simulation of the KorSo production cell runs at a similar rate but is made up of approximately three hundred agents. These similar performances can be explained by the fact that there are much more details in the 3D representations of our GearWheels than in those of our production cell. This points out the fact that the time spent in the 3D rendering is of the same order as the time spent in the execution of the agents’ behavior.

5 CONCLUSION AND FUTURE WORKS

Our objective was to show that a multi-agent approach in a virtual reality environment could be performed for Interactive Prototyping. The main point of Interactive Prototyping concerns the use of dynamic modeling capabilities in a virtual environment designed as a Multi-Agent System.

As far as we are concerned, the oRis/ARéVi platform we provide seems to be an ideal tool to interactively design and tune a distributed system. The complex behavior of such a system can be described by many agents whose interactions ensure a consistent representation of the global virtual world. The dynamic properties of our language allow the user to declare the model he is working on.

The next stage of our work relies on the use of the KQML formalism and the dynamic properties of our language to build behavior servers. This could be used by an agent to dynamically ask the environment to delivering, the behavioral primitives usually run in it. Of course, the use of this concept requires both a high-level dialog between agents, such as KQML, and the capability for an agent to rewrite its own code.

Another subject under study concerns the capability to run a session on many workstations to allow Interactive and Cooperative Prototyping. The low-level layer consisting in message exchanges between workstations is already running but we are still working on the way to keep the simulation consistent. Actually, running a distributed simulation of a system is quite different from simulating a distributed system. The updates between the different workstations can be very easily performed by sending directly some oRis code, but the policy in charge of these updates (as “dead-reckoning”) to ensure the graphical consistency needs to be studied with a particular attention.

REFERENCES

Repeat forever:
  ◦ while the set of same priority threads is not empty:
    ◦ pick and extract at random a thread from this set
    ◦ activate it during a time slice
    ◦ put it back in a second set
  ◦ swap the set of same priority threads with the second set

Figure 1: A fair round-robin

Figure 2: Structure and communication of agents
```java
{ 
    doSomething();
    start {doSomethingElse();}
    goOn();
}
```

Figure 3: Launching a concurrent activity

Figure 4: UML diagram of the main ARéVi classes

Figure 5: Simple interactions and Interactive Prototyping with ARéVi