Using GASP for Collaborative Interaction within 3D Virtual Worlds

Thierry Duval, David Margery

Using GASP for Collaborative Interaction within 3D Virtual Worlds

Thierry Duval and David Margery

IRISA — SIAMES project,
Campus de Beaulieu, F-35042 Rennes cedex
{Thierry.Duval, David.Margery}@irisa.fr

Abstract In this paper, we present GASP, a General Animation and Simulation Platform, whose purpose is to animate autonomous or user-driven agents, and we explain how it can be used for Collaborative Virtual Reality. First, we explain its architecture, based on the notion of simulation objects (or agents) associated with a calculation part (the behavior). Then we describe how it is possible to distribute efficiently our agents upon a network in order to share the amount of calculation between several computers. Finally, as the visualisation of a simulation is also a simulation object, we show that our architecture allows us to distribute several visualisations upon a network to share a 3D interactive simulation between several users.

1 Introduction

The construction of 3D virtual worlds full of intelligent or human-driven entities (or agents) may be a very complex task, which requires many complementary skills, such as 3D graphic programming for the visualisation, artificial intelligence knowledge for the entities' behavior, network programming in order either to distribute the computational weight, or to share the virtual worlds between several users, and software engineering for the global architecture of the system.

In order to reduce the complexity of the task, we believe that a framework should be provided to specialists of behavioral animation so that they can concentrate on the programming of the entities to be included in the virtual world. A run-time platform dealing with the other problems (3D visualisation, interaction and network distribution of the entities) should be associated with that framework. GASP is our attempt to achieve this goal.

The framework presents itself as basic classes a programmer has to reuse by inheritance to write his virtual world entities. Then, using a configuration file, he can decide to deploy these entities upon a network of workstations, and optionally act upon them using several interactive visualisations located on different workstations. The associated platform uses this configuration file to start the virtual world and ensure proper scheduling of the different entities as well as all the distributed aspects of the simulation and synchronisation.

With the associated platform, it is possible to distribute calculations to animate a complex virtual world and, at the same time, to distribute interactive

visualisations to interact with this world on a collaborative base. Within our framework, the same platform is used for both aspects. This contrasts with most of the work in the distributed virtual reality field which concentrates on collaboration (DIVE [1], MASSIVE [2], Community Place [3] or ARéVi [4]). This enables us to populate the world with more entities with complex behaviors than it would have been possible with frameworks where the only computing power available is the one available on the different users' workstations.

Our main concern in the conception of our platform is performance of interactive virtual worlds at run-time, unlike ARéVi [5] which addresses prototyping of virtual worlds during conception phase.

This paper is organised in the following way. First, we describe the framework provided and the associated concepts for distribution. We then talk about the way the platform distributes entities and visualisations over the network, in order to share the calculation weight of the simulation between several workstations and the interactions between several end-users. Finally, we talk about its technical aspects and possible improvements.

2 GASP Framework Overview: the Entities

The GASP [6] framework is an object oriented development environment allowing real-time simulation and visualisation of complex systems.

2.1 A Simulation : a Set of Entities with Different Frequencies

Each entity in the system is composed of one or more simulation objects. These simulation objects are composed of a set of named outputs, inputs and control parameters which constitute the public interface of the entity and of a calculus which is in charge of their evolution. This evolution happens at the frequency associated with each object or family of objects. Indeed, each simulation object may need its own frequency. For example a mechanical model of a car will need a 50 Hertz frequency to obtain a realistic mechanical simulation, but a behavioral model such as a car driver may only need a 5 Hertz frequency to simulate human behavior. We believe this aspect is important for virtual worlds populated with entities (in our example a car entity) made out of components of different nature and is to our mind one of the main differences between GASP and other distributed environments. Indeed, this enables a modular approach to building complex entities.

At each simulation step of an object, the calculus part will read the inputs it needs and calculate new outputs and a new private state for the object. Inputs can be connected to outputs of other objects at different stages in the simulation by either naming the objects to connect to or asking the controller for an object of the correct class. In other words, the evolution of a simulation object can be function of the entity's changing environment. Figure 1 shows a typical exchange between two simulation objects in the same simulation process: for each calculation, the `CB` object will ask its input for a new position value, maybe

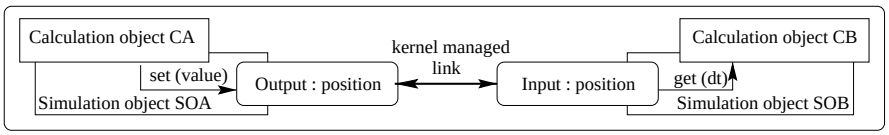

Figure 1. Typical exchange between two simulation objects in the same process

in order to follow the `SOA` simulation object. The kernel is in charge of ensuring that the value provided to the object is coherent with the value from the output. In the current implementation of GASP, this is done by fetching the value, but this could change without affecting the programming framework.

Due to the different frequencies the simulation object may have, an object may be asking for a value at a date where no value was produced. Therefore, every output is automatically able to provide a value at any asked date. This value is either calculated (by interpolation or extrapolation) or exact. Naturally, it is always possible for the asker to know if the answered value is an exact one or if it is a calculated one. In this last case, the system provides the difference between the date for which the value was asked and the date associated to the nearest (by mean of time) exact value produced by the calculation.

With such a mechanism, entities can ask for the value of other objects at the current simulated time. If the value has not yet been produced, an approximation will be calculated. This enables a modular approach, as simulated entities can be conceived without prior knowledge of their place in the simulation chain. This is quite different from traditional animation models such as those used in VRML [7] where the complete animation chain must be completely redesigned for each virtual world. This is of particular importance when cyclic dependencies are used which is the case in most multi-agent inhabited virtual worlds. For fine grained tuning, our simulation platform still enables VRML style coding, but we believe that for complex worlds, this is not a scalable approach.

2.2 The Distribution : Referentials and Mirrors

Our entities can be dispatched, at run time, across a network of heterogeneous workstations, in order to distribute the computational weight.

Each of the workstations will own one or several process making the calculations or watching some of the results produced by the global simulation.

Each process belonging to a simulation will own a particular simulation object: a controller, which schedules all the local simulation objects. This schedule is achieved by using several simulation frames, filled with references to the objects to simulate, according to their frequency.

So, within each process, there will be none, one or several "real entities": we call them "referentials", and none, one or several "ghost entities": we call them "mirrors", other people may call them "proxies" or "ghosts".

The presence of mirrors in a process is function of the inputs needed by the referentials of this process: if there is a referential `B` that needs for input the output of a referential `A` located within another process, then there will be a mirror of `A` within the process where `B` is located.

Mirrors are linked to their referentials with a data-stream connection: at each step of the simulation, a referential sends up to date values to all of its mirrors. Figure 2 shows a typical exchange between two simulation objects owned by two different simulation process: at each calculation step, a GASP mechanism of "pushing" will provide new output values to the `SOA` mirror, in order to enable the `SOB` referential to obtain its input value.

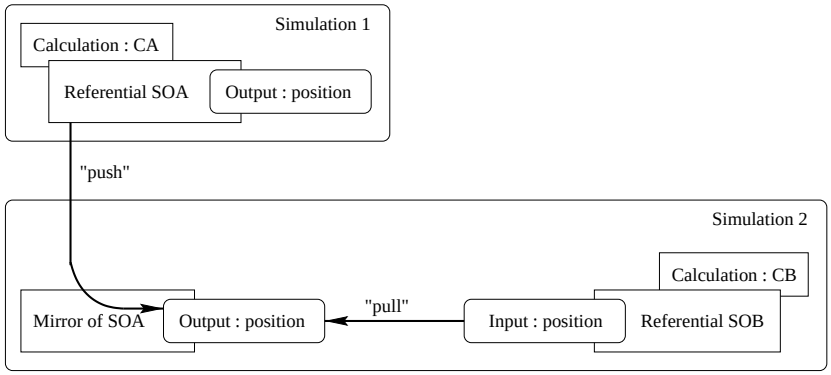

Figure 2. Typical exchange between two simulation objects in two different process

3 The Exchanges between Referentials and Mirrors

We are now going to explain the way referentials and mirrors communicate: this communication is valid because of the interpolation and extrapolation mechanisms we talked about in section 2.1. Thanks to them, we do not need a strong synchronisation between our different processes.

3.1 The Exchanges Are Synchronous

The data-stream connection is a synchronous one.

Our aim is to ensure the realism of our virtual worlds in term of consistence between the referentials and their mirrors in the different process, in order to avoid the distortion caused by the network latency as presented in [8] for the interactions. The proposed solution of [8] consists in placing the object to interact with on the same workstation than the end-user involved in this interaction. It is not enough for us, because we need consistency between referentials and mirrors in a more general way for the realism of the entities behavior, and because several users may want to share an interaction with the same virtual object (interaction level 3.2 of [9]).

This consistence is achieved by updating the data of the mirrors at each simulation step. To enable high interactivity, this updating typically occurs 50 times a second. This means that we need a large bandwidth network, as the amount of data that will have to transit on the network will be very important.

It also means we cannot adopt a wide area network dead-reckoning approach such as in NPS-NET [10] or ARéVi[4]. GASP ensures that the mirrors will always be the more "up to date" possible, by slowing faster process to adjust to the frequency of the slower one. So GASP approach is synchronous: at t time, each mirror will always own the referential values of the t - (dt + latency) time, where dt is the simulated time between two simulation steps for the referential, and where latency is the simulated time messages spend on the network. This latency can be parameterized to optimize the synchronization (latency can be positioned equal to zero to obtain a perfect synchronization, even if it will slow the global simulation).

Thanks to these data stream connections between referentials and mirrors, a mirror can always instantaneously provide the best values available to the referentials interested in them, without neither having to wait for the data to come from its referential, nor having to ask the referential to provide a new value (as we could do using a CORBA mechanism) which cost could be at least twice the cost of the network latency.

3.2 The Controllers Look for Informations

We use a subscription mechanism to ensure the synchronization between all our controllers. During the initialisation step of a simulation, each mirror will ask its controller to subscribe to its referential's controller.

So, a controller will know that it must receive informations from a set of other controllers, and at each step of simulation, it will look at the latest information from these controllers.

If a date associated to the last reception of information from a particular controller is too old, the controller will decide that newer informations from that controller are needed, and it will put that controller in the set of the controllers from which he absolutely must receive something before going on.

Then the controller will look at all the messages waiting for it from all the existing controllers (because some asynchronous informations such as events and messages may have been sent to it), and it will update the set of the controllers from which information is lacking, because it may have received some.

After that, if some informations from some controllers are still lacking, the controller will now wait for them, and only for them, then it will be able to go on the next simulation step.

3.3 Nearly Every Controller Subscribes to Another One

As process without mirrors could never have to wait for other controllers, their advance could become too important, so it would be problematic to dynamically add mirrors in such controllers, because this addition should then wait until other controllers have join them.

So, in order to slow down these controllers and then to synchronize them with the other ones in the same shared simulation, they subscribe to each controller

owning a mirror of one of their referentials. In such a case, the only information they will be waiting for is the date of the latest simulation step of these other controllers.

This does not solve the problem of a controller who would only own referentials without any mirrors anywhere. But what would be the use of such simulation objects in a shared simulation ? Of course, it could allow for example to dynamically make separations between a controller (or a group of controllers) and the rest of the simulation, so with our communication paradigm, one group could take some important advance on another one. To avoid this, we could force each controller to subscribe to a global scheduler, which would provide clock tics, or we could provide a new paradigm for distributed objects to complete the referential/mirror paradigm.

Another problem is the asynchronous diffusion of events and messages: as they are asynchronous, a controller does not know when it will receive such informations, so, at this time, we can only offer the guarantee that the time messages and events will take to reach their targets will be at most (dt + latency), where dt is the time between two simulation steps for the receiver's controller.

4 Sharing 3D Virtual Worlds

Each entity can be associated to a geometry (a visual 3D representation), and entities can be a compound of other entities ; in such a case, the visualisation of the compound entity results from the visualisations of its parts.

We provide a special simulation object called "Visualisation" which is in charge of the graphical representation of the objects of a simulation. It also allows the end-user to take control of some of the simulation objects.

With this simulation object, we can then instantiate several visualisation objects that can also be distributed upon the network on several workstations, not only in order to allow several people to see a graphical representation of the simulation, but also to allow these people to interact with the simulation, and therefore to cooperate in a shared 3D virtual world.

We have worked on a way to integrate interactors entities within visualisations, so that they could pilot the inputs of some simulations objects to benefit from our referential/mirror paradigm for synchronous interactions between several remote users of the same shared virtual environment. Now, it is possible to add interactors to simulation objects, so that these interactors can control the behavior of these objects, and then to share an interactive virtual world between several end-users. The way we do it is by dynamically adding some inputs to the simulation objects, and by linking these inputs to the outputs of some interactors. We use a generic way to achieve it nearly automatically, using the C++ inheritance mechanism, so that as many simulation objects as possible can benefit from these interactors.

5 How to Use GASP

Here, we are going to make a short state of the art about how to use GASP to develop a simulation application and how to distribute it upon a network.

5.1 From the Simulation Object Point of View

The GASP controllers' main task is to create and then to schedule a set of simulation objects, asking each of them to evolve. These simulation objects are instances of C++ classes which inherit from the GASP `PsObjetSimul` class.

This class provides some virtual methods that its subclasses must redefine:

- `PsCalcul * creerCalcul () ;`
  creation of the associated calculation object, an instance of a subclass of the `PsCalcul` class.
- `void initTableEntrees () ;`
  declaration of the inputs.
- `void initTableSorties () ;`
  declaration of the outputs.
- `void initTableParams () ;`
  declaration of the control parameters.
- `void traiterEvt () ;`
  management of the asynchronous events and their associated messages coming from any object of the simulation.

Inputs, outputs and control parameters types can be provided by GASP (basic types such as integer, float, long integer, long float, string) or extended by some simulation object to obtain complex types such as 3D referentials or quaternions.

A simulation object evolves thanks to its `PsCalcul` associated object, which provides also some virtual methods that its subclasses must redefine:

- `virtual void init () ;`
  initialisation of the internal data of the calculation object, this method enables the static connections between the inputs of its simulation object and the outputs of other objects.
- `virtual void calculer () ;`
  calculation of the new internal state and of the possible outputs of its simulation object from the previous internal state and from the values of the inputs of its simulation object.

5.2 From the Simulation Application Point of View

The `main` function of the simulation process must now be able to create new instances of the new classes that have been defined for the new simulation.

This is done by defining two methods of the `PsnControleur` class:

- `void initListeObjetsDerives () ;`
  responsible for the correspondence between the names (strings . . .) read from the configuration file and the real simulation objects types.

– `PsObjetSimul * associerCode (int i) ;`
responsible for the creation of the effective simulation objects types according to their type (represented here by an integer).

The name of the configuration file will then be provided to the main GASP function on the command line. This file describes the types and the names of the simulation objects initially in the simulation, their calculation frequency, and the process responsible for their evolution.

This file describes also the association between process and workstations (a workstation may "own" several process, a process can only be associated with one workstation).

So, the distribution of the simulation objects is only decided at run time, and can easily be changed, without needing any new compilation or binding.

5.3 From the Visualisation Point of View

In order to be visualised by our visualisation objects, a simulation object must inherit from a particular subclass of `PsObjetSimul`: `PsGeoMeca`, and must be associated with a geometrical file.

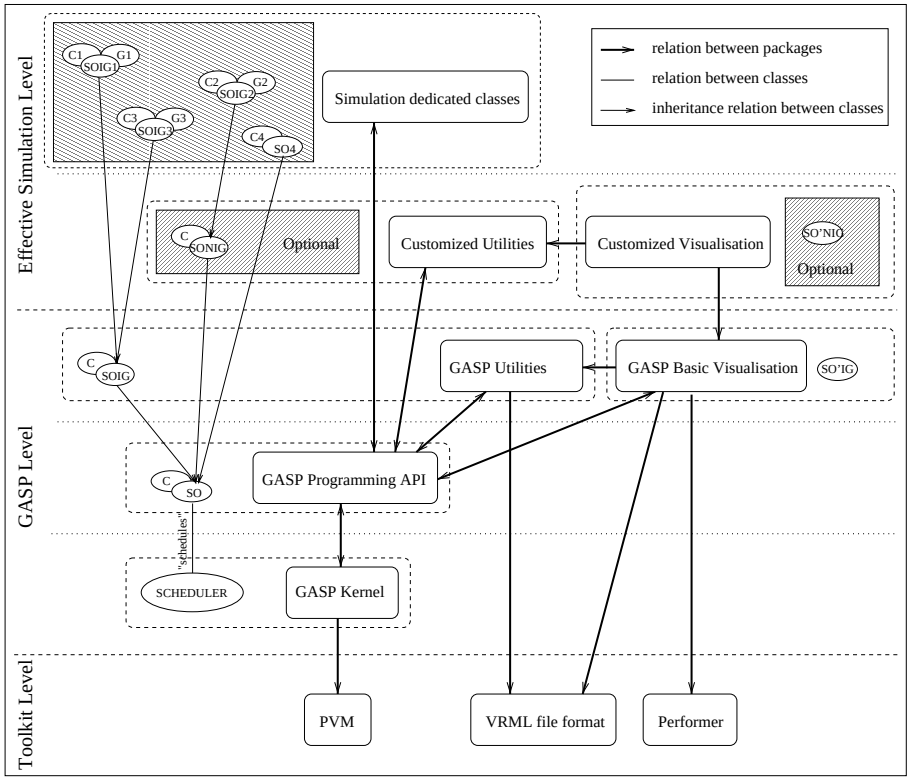

Figure 3. Example of a creation of a new simulation

5.4 Example Use of GASP for a New Simulation

Design of the New Simulation Package Assuming that the scheduler provided by the GASP kernel is only able to schedule simulation objects associated with a calculation object, the programmer of a new simulation object will have to create new classes that will inherit from this two classes: `PsObjetSimul` and `PsCalcul`, which are represented figure 3 by the `SO` and `C` classes provided by the GASP API.

First, if this programmer does not need visualisation, he can directly inherit from these classes, as for the `SO4` class of the figure 3.

Then, if he needs to provide visualisation or interaction for some of the new simulation objects, he can create new classes that will inherit from the `PsgeoMeca` class, represented figure 3 by the `SOIG` class (Simulation Object with an Interface for Geometry) which provides an interface to act upon its geometrical representation in terms of position, orientation and scale: it is possible to act upon a particular Performer DCS node associated to the geometry. That is the way the `SOIG1` and `SOIG3` classes have been created here, and then can be associated with some geometry.

Thanks to the classes provided by the GASP basic visualisation (such like the `SO'IG` class), it will then be possible:

- to visualise the geometries of the simulation objects,
- to see their dynamic evolution (modification of a Performer DCS node),
- to interact with these objects.

If the programmer needs some new ways to visualise or to interact with the simulation objects, he will be allowed to create its own classes, by providing new interfaces to the geometries. This possibility is illustrated figure 3 by the `SONIG` class (Simulation Object with a New Interface for Geometry) of the customized utilities and the `SO'NIG` class of the customized visualisation. For example, this new interface could allow to act upon a Performer color node associated with the geometry.

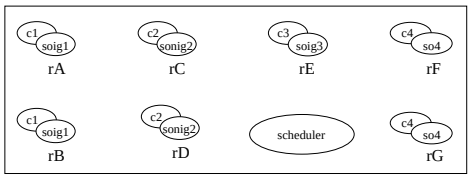

Figure 4. Run-time for a simple simulation

The Simulation at Run-Time If the simulation is a very simple one, that is to say that the configuration file places all the simulation objects in the same process, with no visualisation, there will be only one simulation process, which scheduler will have to schedule all the simulation objects.

For example, figure 4 shows seven simulation objects, called A, B, C, D, E, F and G and a scheduler. As there is only one process, the simulation objects are referentials, that is why they are presented as rA, rB, rC, rD, rE, rF and rG.

In order to visualise this simulation and then to allow two end-users to cooperate with the simulation objects, the configuration file will need the declaration of two visualisations, upon two different process on two different workstations.

The result is shown figure 5: the two visualisation process only own mirrors (with no calculation part), presented as mA, mB, mC, mD and mE, as the F and G objects do not own a geometry and then are not visualisable.

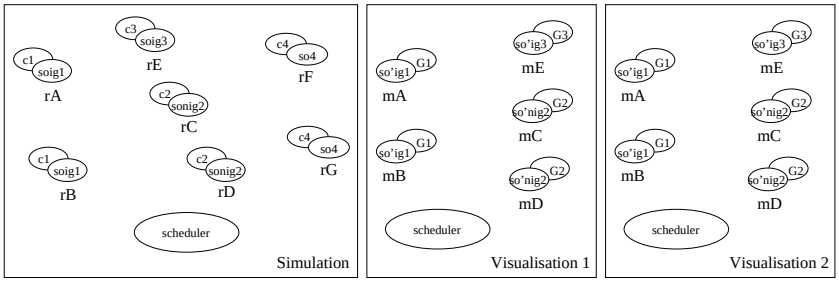

Figure 5. Run-time for a simple simulation with visualisations

Finally, this result can also be achieved by distributing the simulation objects upon several process on several workstations or CPU. Then, on each process, there will be a scheduler responsible for the referentials of its process, maybe some referentials (for example, the visualisation process will not own any referential for A, B, C, D, E, F or G), and maybe some mirrors, if the outputs of the simulation objects associated with these mirrors are needed by some of the referentials of the process.

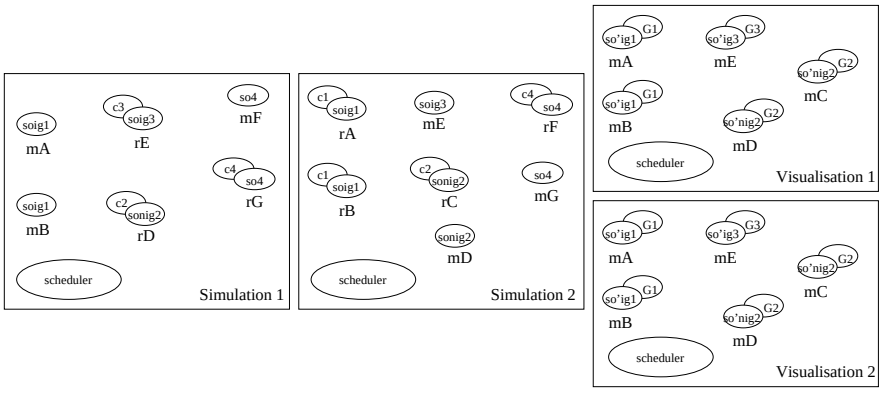

Figure 6. Run-time for a distributed simulation with two visualisations

A possible result is shown figure 6, where the first simulation is responsible for the D, E and G simulation objects, and the second simulation is responsible for the A, B, C and F simulation objects.

We suppose here that none of the D, E and G simulation objects are interested in the outputs of the C object, that is why there is no mirror of C in the first simulation process.

6 Technicals Aspects of our Platform

This platform relies on Performer for the real-time visualisation. Is is currently developed with Silicon Graphics workstations.

Distribution of calculations can rely on REACT (only on SGI multi-processors hardware) or on PVM for heterogeneous hardware UNIX platforms.

The mechanisms of communications between referentials and mirrors we have described here are based on PVM (Parallel Virtual Machine), it is that way we ensure the distribution of visualisations.

To visualise our simulation objects, we associate them to a geometry, which is a VRML 1.0 description.

All the calculations of the simulation objects are written in C++.

Previous versions of the GASP kernel have already been ported upon many UNIX systems and upon Windows NT, the only difficulty is for the visualisation, because of Performer, which is currently only available for SGI and Linux workstations.

7 Further Work

7.1 About the Use of User-defined Predictive Models

For a distributed simulation, the amount of data transmitted between the different nodes and the way concurrency between communication and calculation are handled are critical aspects of the overall performance. If a good approximation of the output values can be calculated, then the amount of data transmitted can be reduced and better concurrency can be achieved. In the current implementation of our simulation platform, these approximations are type based and system provided. We are now focusing on enabling user defined predictive models for these calculations. This will be possible on user-defined input/output type level or on an entity level.

7.2 About Physically Shared Collaborative Interactions

As we own a SGI reality center which allows group immersion, we are also studying the possibility to allow several users to interact with each other in front of the same physical visualisation, to benefit from a real collaborative immersion in a physically shared virtual environment. This is going to be realized by using techniques similar to these used for multi-modal applications.

8 Conclusion

We have presented here the architecture of GASP, our platform for animation and simulation. This platform can be useful for multi-agent simulations, as it allows a designer and a programmer to focus on the interesting part of the development of a virtual world : the behavior of the entities that populate that world.

As GASP allows entities to be distributed upon several workstations, it enables the construction and the execution of complex virtual worlds, with a lot of entities with complex and heavy weight calculation behavior.

What is very interesting here is that this programmer does not have to worry about the distributed aspects.

As GASP provides a generic 3D visualisation which is a simulation object that can also be distributed, we allow several end-users to share the same 3D virtual world, each of them has its own point of view and is able to interact with the entities of the world, so they can share interactions and then collaborate.

References

1. C. Carlsson and O. Hagsand, "DIVE — A platform for multi-user virtual environments," *Computers and Graphics*, vol. 17, pp. 663–669, Nov.–Dec. 1993.
2. C. Greenhalgh and S. Benford, "MASSIVE: A distributed virtual reality system incorporating spatial trading," in *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, (Los Alamitos, CA, USA), pp. 27–35, IEEE Computer Society Press, May30 June–2 1995.
3. R. Lea, Y. Honda, K. Matsuda, and S. Matsuda, "Community place: Architecture and performance," in *VRML 97: Second Symposium on the Virtual Reality Modeling Language* (R. Carey and P. Strauss, eds.), (New York City, NY), ACM SIGGRAPH / ACM SIGCOMM, ACM Press, Feb. 1997. ISBN 0-89791-886-x.
4. T. Duval, S. Morvan, P. Reignier, F. Harrouet, and J. Tisseau, "Arévi : une boîte à outils 3d pour des applications coopératives," *Numéro spécial de la revue Calculateurs Parallèles (coopération)*, pp. 239–250, juillet 1997.
5. P. Reignier, F. Harrouet, S. Morvan, J. Tisseau, and T. Duval, "Arévi : A virtual reality multiagent platform," in *Proceedings of the First International Conference on Virtual Worlds (VW'98), Paris, Lecture Notes in Computer Science, Artifial Intelligence series (LNCS/AI 1434)*, pp. 229–240, juillet 1998.
6. S. Donikian, A. Chauffaut, T. Duval, and R. Kulpa, "Gasp: from modular programming to distributed execution," in *Computer Animation'98, IEEE, Philadelphie, USA*, pp. 79–87, juin 1998.
7. G. Bell, R. Carry, and C. Marrin, "VRML 2.0 final specification," 1996.
8. M. Ryan and P. Sharkey, "Distortion in distributed virtual environments," in *Proceedings of the First International Conference on Virtual Worlds (VW'98), Paris, Lecture Notes in Computer Science, Artifial Intelligence series (LNCS/AI 1434)*, pp. 42–48, juillet 1998.
9. D. Margery, B. Arnaldi, and N. Plouzeau, "A general framework for cooperative manipulation in virtual environments," in *Virtual Environments '99* (M. Gervautz, A. Hildebrand, and D. Schmaltsieg, eds.), pp. 169–178, Eurographics, Springer, 1999.
10. M.R. Macedonia, *A Network software Architecture For Large Scale Virtual Environnements*. PhD thesis, Naval Postgraduate School, Monterey, California, 1995.