Urbi et Orbi: Unusual Design and Implementation Choices for Distributed Virtual Environments

Didier Verna Yoann Fabre
Guillaume Pitel
EPITA / LRDE, 14-16 rue Voltaire, 94276 Kremlin-Bicêtre cedex
mailto:didier@lrde.epita.fr
http://www.lrde.epita.fr

September 28, 2000

Abstract

This paper describes Urbi *et* Orbi, a distributed virtual environment (DVE) project that is being conducted in the Research and Development Laboratory at Epita. Our ultimate goal is to provide support for large scale multi-user virtual worlds on end-user machines. The incremental development of this project led us to take unusual design and implementation decisions that we propose to relate in this paper. Firstly, a general overview of the project is given, along with the initial requirements we wanted to meet. Then, we go on with a description of the system's architecture. Lastly, we describe and justify the unusual choices we have made in the project's internals.

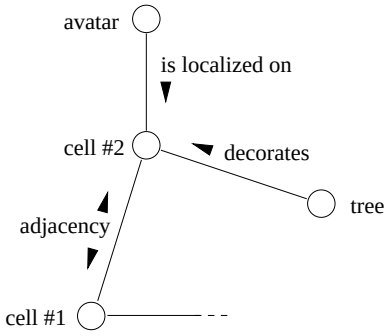
1 Introduction

This paper describes Urbi *et* Orbi, a distributed virtual environment (DVE) project that is being conducted in the Research and Development Laboratory at Epita. Urbi *et* Orbi is a virtual environment system: it provides users with a virtual world in which they can interact with objects or other members. In addition, Urbi *et* Orbi is a *distributed* virtual environment system: the world description is spread over several computers connected via a network.

Urbi *et* Orbi has been initially designed to be a *large scale* DVE: a potentially large number of participants could exist around the world, no-one having a complete control over the virtual world at any time. Because of this required scalability [1], there must not be other material requirements than a standard personal computer, with a connection to a local network or the Internet. This constraint brings up non trivial problems that the system must handle, in particular, to cope with network bandwidth and information propagation latency.

This paper is structured as follows: firstly, a general overview of the project is given, along with the initial requirements we wanted to meet. Then, we go on with a description of the system's architecture. Finally, we describe and justify the unusual choices we have made in the project's internals.

Figure 1: Part of a conceptual graph



2 Initial requirements

2.1 High Level Scene Description

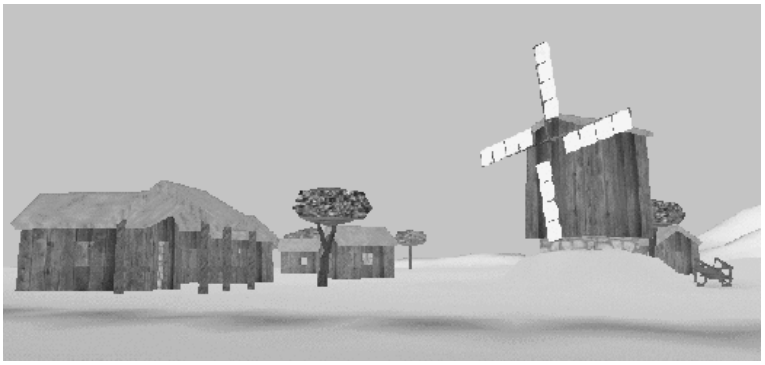
An important characteristic of Urbi *et* Orbi is that contrary to many other DVE systems, we wanted to provide rich semantics and high-level descriptions of the world components: forests are composed of trees of which there are several types, objects such as a house has an inside and an outside etc. We also wanted to avoid the *3D bias*: virtual worlds should be thought of as an abstract idea. Hence, thanks to a text terminal it should be possible for the user to walk around and interact with the world *textually*. With our approach, the world has enough semantics for a terminal to represent it faithfully, whatever its nature: textual, 3D rendering, etc. Currently, we have a textual terminal, which is actually a shell to the application kernel, allowing for textual action on the world components, and a 3D rendering engine that uses an OpenGL display.

Another benefit of rich descriptions of scenes is that a lot of extra optimizations can be performed, for instance related to the distance. Imagine a forest far off on the horizon: there is no need to compute the graphic rendering of every single tree, since it will most probably result in a green spot. The traditional answer is level of details (LOD) generated by mesh simplification: objects present different LOD according to the distance from the observer. Here, using a richer semantics, such as the variety of the trees and some of their characteristics, we may provide different views, ranging from the green spot to the fully detailed rendering, through approximations based on functions depending upon the nature of the trees and their spatial disposition, etc. The nature of the flat green spot is very different from the fully detailed description of the forest. Our “levels of details” not only imply a quantitative change of the semantics of the objects, but also a qualitative change. And again, since the observer may not have a 3D rendering engine, the type of the object, here “forest”, can be used to deduce the fact we don’t need to request further details.

In addition to the nature of the objects of the world, we also need relations between them. Of course, we need relations such as “is on” or “is adjacent to” but we also need non-spatial relations such as “is composed of” or “activates”. We are therefore naturally led to the notion of *conceptual graphs*.

Figure 1 illustrates a partial view of the world’s state: there are two geometrical cells, i.e. regions or zones of the world, a tree decorating cell 2 and an avatar somewhere in this cell. Yet in this simple example, there are three different relations involved: the link between the two cells stating that we may reach one from the other, the oriented edge “is localized on” specifying the presence of the avatar on a cell, and finally “decorates” which is similar to the previous relation but slightly different: “decorates” gives the additional information that the

Figure 2: A landscape from Urbi *et* Orbi



tree is unessential and can be passed over at first approximation (for instance because some more urgent updates need to be performed).

Please note that this approach is fully compatible with current solutions to reduce the volume of information flow: a hierarchical description of the world as proposed by Sugano *et al.* [16] and/or an aura management as proposed by Hagsand and Stenius [6] or by Hesina and Schmalstieg [8]. The nature of the data can depend on the scale on which the graph is consulted; the world must be also represented by a multi-scale graph.

2.2 Full Distribution

A specific feature of Urbi *et* Orbi, with regard to most other frameworks, is that each host which participates in the world is both a "server' and a "client" [10, 1, 16]. Therefore, all the computers run the same application which combines both server (propagating the information) and client tasks (rendering and interaction). This approach is similar to the ones of Frécon and Stenius [4] in Dive, and of Greenhalgh and Benford [5] in MASSIVE. It contrasts with client-server architectures.

As is the case for the world wide web, no single host knows the current state of the whole world, or even merely an outdated approximation: the knowledge is distributed across the machines, which only know the kingdom of their user, and the part of the world where their user are. This is so called "shared distributed data bases with peer-to-peer updates" according to [12].

Now, consider a scene with a windmill whose arms are rotating, as in figure 2. One can imagine at least two ways of publishing this information:

Distribution. The "owner" of the windmill sends frequent updates of the position of the arms to its observers.

Replication. Several windmills are actually "created", one per host. The implementation of the windmill is run on each machine.

The *distributed* approach is very adapted for avatars: because their behavior is unpredictable and there can be only one command center, the person who commands the avatar. Conversely, the windmill motion is fully predictable, and frequent updates of its position would waste the bandwidth. Here, the *replication* is a natural solution.

2.3 Group Communication

In order to reduce the amount of network communication, we have preferred, like Tramberend [17], group communication to unicast and broadcast solutions. We excluded Unicast because each time a machine needs to publish an update, it would have to send one message per connected host: *destinations* accumulate. We excluded Broadcast, since each time a machine publishes an update, all the machines would have to process the message, and maybe discard it, which results in useless additional work: *sources* accumulate.

Group communication is a way of implementing the notion of aura in DVEs, and allows us to define various groups equipped with different quality of service. Group communication has several benefits [17]. Firstly, it allows the deployment of several groups with the same members but with different protocols, hence various qualities of service. The possibility to choose the protocol is a means to control the load imposed on the network: urgent messages which must be delivered safely obtain most of the bandwith, while messages of little importance may be delayed or even lost.

The notion of group also helps in the design of the virtual world, since they constrain the programmer to structure its description and to partition it properly.

When a user enters the world, she first joins a spatial group and some other user within this group is elected to inform the newcomer of the current state of the world, more specifically, of the current state of the view which is pertinent to the newcomer. An economic consistency of the knowledge is obtained thanks to different policies on the distribution of the messages. For instance, elections are made with an extreme care, some actions require a causal ordering, and finally some unimportant tasks are just multicast with almost no quality of service. Furthermore, the possibility of choosing different qualities of service is a means to control the load imposed on the network: urgent messages which must be delivered safely obtain most of the bandwidth at a given time, while messages of little importance may be delayed or even lost.

3 Software Architecture

3.1 The Goal Language

Our requirements pertaining to the structure of the information and to the distribution system drove us to design Urbi *et* Orbi as a language-oriented system. The additional requirement that world management must be programmer-friendly naturally led to the concept of a scripting language. Several options were available, most prominently Java. However, because no language offered all the facilities we wanted to implement for a DVE, we decided to build our own language: Goal.

Goal is the vehicular language used throughout the project. “High level” Goal is used between the user and the environment to describe and modify the world, whereas “low level” Goal is exchanged between the machines through the network and between the different modules which constitute the user application. Since one of the main purposes of Goal is to offer an abstraction of the network to the programmer, its runtime naturally resembles a miniature OS kernel. Goal has several features that are worth mentioning:

Strong Typing. When Goal instructions are introduced in the environment, the Goal interpreter checks for errors before executing them and modifying the environment. The strong typing policy is a fundamental help in developing the complex features of the project, leading to safer code.

Figure 3: Code sample.

```
// file windmill.goal
@mill_shape = New Shape;                    //  1
@mill_shape <- Set file3DS = "mill.3ds";    //  2
@mill = New 3DGridObject;                   //  3
@mill <- Set shape = shape_mill;            //  4
@sails_shape = New Shape;                   //  5
@sails_shape <- Set file3DS = "sails.3ds"; //  6
@sails = New 3DGridObject;                  //  7
@sails <- Set shape = sails_shape;          //  8
@rotator = New Rotator;                     //  9
@rotator <- Set target = sails;             // 10
@rotator <- Set delay = 0.02;               // 11
```

Frame-Oriented. Goal is a frame language implementing the concept of class and inheritance. Since Goal is strongly typed the required notion of conceptual graph is implicitly supported by objects containing references to other objects. Each slot of the objects may be equipped with a daemon, i.e., a routine triggered when its slot is modified. Here, objects help modularize world descriptions and daemons help divide the tasks: when values are updated, there are usually a collection of actions to undertake, but some of them imply a global side-effect for several users spread over the network while others are purely local.

Distribution/Replication-Oriented. In a distributed environment, objects are usually replicated over the network and, when a user acts on such objects, a Goal routine is executed which may contain code for sending messages to distant replicates. Furthermore, objects can belong to several groups, which implies that objects' behavior at run-time may use several communication channels. Group communication should be a primitive of the environment provided to the programmer.

Dynamicity. Goal has been designed to make the virtual environment evolve. It allows to dynamically insert objects in the environment as easily as a web page is published on the Internet. Similarly, a new class can be defined by a user and loaded dynamically in the environment while running; this new class can then be accessed by another programmer and reused.

Reflexivity. All Goal entities can be introspected. With a class name, we can know the list of its attributes and methods; with a given object, we can get the name of its class and the values of its attributes.

Scripting. Goal is a scripting language, interpreted within MMk (Matrix Micro Kernel, see [15] for etymology), our application kernel. Although it is interpreted, thanks to the numerous optimizations (e.g. early binding, as in PostScript) its interpretation is fast. On the other hand, because the programs are scripts, code can easily migrate, which is necessary to properly model animated objects (consider again the windmill example).

3.2 A sample Goal script

In order to illustrate all these ideas, we provide a very simplified example of a Goal script. This code sample appears in figure 3, and actually implements the windmill example of figure 2.

The symbol `@` indicates that the instruction is only sent to the local virtual machine (alternatives are `.` and `!` to broadcast the instruction to the members of the communication group, respectively including and excluding the local machine). Lines 1 to 4 define the mill, lines 5 to 8 the sails, and lines 9 to 11 a rotator to make the sails turn and to manage their speed.

Apart from huge constant data like object textures, all values in the system are truly distributed, that is, created somewhere and then propagated via the network. Nevertheless, for the sake of the bandwidth economy, the remaining data is classified per priority. A high priority data is more likely to be updated than low priority data. The priority of the various attributes/objects is specified as part of their type: it is therefore completely and cleanly integrated into Goal and handled seamlessly by the application. For instance, a `3DGridObject` is an object that decorates a grid cell: its priority is medium.

Without entering in gory details, we also noted that shared values have a completely distinct status from replicated values. In particular, it is forbidden in Goal to set a shared value: one has to ask the object containing this value to perform that task. Then, the usual daemon mechanism is launched. This simple limitation, voluntarily introduced in Goal, appeared to save the world / objects developers from many errors, leading them to question the status (replicated or shared) of their values in a distributed world: there is a natural tendency to use parsimony. In the example above, the sails of the windmill have a fully predictable behavior and frequent updates of their position would waste the bandwidth. So, the *replication* is a natural solution, and each host has a local timer.

3.3 Software Components

The software is composed of three active modules as depicted in figure 4. Each module is responsible for an independent task and offers particular services.

MMk (or kernel) is the core of our software; it receives a flow of instructions, schedules their execution and manages the resources (files, memory, display and communication).

NET is in charge of the communications through the network; it handles the connection / disconnection and information transfer.

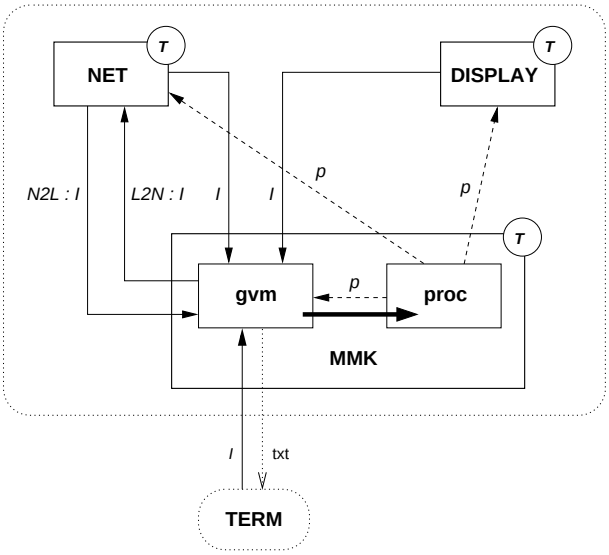
DISPLAY (or renderer, or navigator) provides the user with read access to the state of the world as known by MMk. It also allows the user to "write": when a user moves or activates an element of the world, feedback is given to MMk. For simplicity's sake the data circuit is not represented in figure 4 and will not be discussed.

In figure 4, the TERMinal is also represented; it is a shell (command interpreter) which allows direct textual communication with MMk, via Goal instructions. This has proven to be a great help in the design and implementation of virtual worlds.

Several other projects have based their approach upon a kernel, such as Maverik [9]. One of the most striking difference between the two projects lies in the fact that their approach is based upon modules, while we stressed on the importance of a high level language.

A central module of Maverik is its Spatial Management System (SMS), which is in charge of processing any 3D related thing. In Urbi *et* Orbi, there is no such module. Because we aim a generic approach of the information, we have tried to avoid any dedicated low level

Figure 4: Software components.



code for 3D in the kernel: the processing of 3D information is spread across the components (more specifically the renderer).

This results in a wide set of primitive operations that are bound to Goal instructions. The programmer then merely needs to attach such Goal instructions to her objects to access these primitives. For instance Goal objects, typically 3D objects, may be bound to a cell, which means the object is "in" the cell. Then, the programmer merely has to declare that her object `implements gridlistener[pos]`, in order to have automatic binding of the object to the proper cell. This makes heavy use of the daemons.

3.4 Multi-Threading and Asynchrony

Monolithic software is not adapted for interactive worlds: in order to provide the user with proper interactivity, tasks should be scheduled to favor human-visible operations. Fairness is also a strong requirement: neither rendering nor communication can be interrupted without degrading the interactivity of the whole application. Even if the traffic is high, the renderer must not freeze, and conversely, a complex rendering must not prevent the host from communicating.

Therefore, in Urbi *et* Orbi, each task runs in a separate *thread* (symbolized by the letter *T* in figure 4). The implementation, based on time slots, guarantees that we cannot enter a deadlock. Goal is sufficiently expressive so that programmers may introduce bugs in their Goal programs, such as deadlocks. However, thanks to the time slots, the system is still fair, and each task will be provided with the ability to make a step.

In addition to the real OS of the host, MMk handles concurrency between the modules, such as the renderer and the communication modules. Modules send requests to each other via MMk; contrary to procedure calls, the modules cannot hang while making a request. MMk delegates their execution to the proper modules, which run concurrently. Again, thanks to the time slots, it is also impossible for a module to hang.

4 Internal Design Choices

4.1 The First Prototype

A very popular way of building virtual worlds is to use the VRML language. A working group has specified an architecture for multi-user distributed VRML worlds, *Living Worlds*, which has already been implemented by Wray and Hawkes [18]. Another way is to reuse a dedicated language like NPSOFF by Zyda [19].

With the additional requirement of full distribution, we decided to implement the first prototype in Java, describe the world in VRML, and handle the distribution layer with an *Object Request Broker* (ORB) as in [3, 2]. Unfortunately, this prototype was very disappointing:

- firstly, the combination of Java, VRML, and an ORB turned out to be a major performance penalty.
- secondly, many difficulties arose when we wanted to process communication, display and user interaction simultaneously.
- finally, the VRML solution did not meet our requirements: we wanted extra object attributes without geometric semantics, and complex relations between objects.

4.2 Using Ensemble for Group Communication

In order to efficiently implement group communication, we decided to use Ensemble, a group communication toolkit developed at Cornell University by Hayden *et al.* [7]:

> "For a distributed systems researcher, Ensemble is a highly modular and reconfigurable toolkit. The high-level protocols provided to applications are really stacks of tiny protocol layers. These protocol layers each implement several simple properties: they are composed to provide sets of high-level properties, such as total ordering, security, virtual synchrony, etc. Individual layers can be modified or rebuilt to experiment with new properties or change the performance characteristics of the system. This makes Ensemble a very flexible platform on which to do research."

As mentioned by Macedonia and Zyda [12], system requirements such as strong data consistency, strict causality, very reliable communications and real-time interaction imply tough penalties on scalability. Consequently we decided to relax these requirements and to rely on different qualities of services provided by Ensemble. In particular, it is easily feasible with Ensemble to test distribution models, like for instance the one proposed by Ryan and Sharkey [14]. We also observe that combining group communication with asynchronous and partially reliable transfer protocols results in low-cost and efficient data exchange mechanisms.

4.3 Using *Objective Caml* for Grounding the Application

In the preceding section, we described some specificities of our language, Goal. This language has been built on top of *Objective Caml* [13, 11], because we found that most of our requirements were met, or at least made easier. Here are the most important of them:

***Objective Caml* is strongly typed.** As we want rich semantics in the world description, we need a strong typing system, more specifically, we need a strong typing for the objects and their attributes. It is well-known in compilation that the more you know about the semantics of the objects (which means the richer your typing system is) the more opportunities the system has to perform optimizations.*Objective Caml* provides this functionality.

Also, we need a flexible typing system which allows us to define families of objects with different attributes, hence we need classes. We want to be able to specialize some concepts, so we want inheritance. Again, *Objective Caml* provides this thanks to its powerful polymorphic typing.

***Objective Caml* and scripts.** Scripting languages have proven to be extremely useful both for extensions of the system and for programmer interface (e.g. shell scripts, TCL, etc.). The fact that scripting languages are interpreted is part of their success: they ensure very easy prototyping, dynamic extensions and modifications (no need to recompile and reinstall the software, which is ideal for distributed systems) and an extreme comfort for the programmer thanks to the interaction with the interpreter. *Objective Caml* actually provides script interpretation, byte-code compilation as well as native compilation.

Also, please note that *Objective Caml* can be easily interfaced with other languages; for instance, the 3D rendering in Urbi *et* Orbi is performed in C code. Thanks to this ability to mix compiled and interpreted code it is fairly easy to observe the behavior at run-time. Finally, the performance of the compiled executable outperforms Java.

***Objective Caml* is a functional language.** The functional paradigm has proven to be suitable to implement most of the transformations we apply on the distributed data, helping us to isolate the areas where imperative instructions are truly needed. *Objective Caml*, while providing all features that one can expect from imperative languages, fully supports the functional paradigm.

In addition, Ensemble is also based on *Objective Caml*, which made it even more attractive to us.

5 Conclusion

In this paper, we have presented the Urbi *et* Orbi project, its motivations and the rationale for our unusual design decisions. Specifically, we have detailed why we think that the *ad hoc* language that we have developed, Goal, is suitable to describe large and complex interactive virtual worlds. Goal is the natural high level interface to a distribution kernel, the MMk, which makes heavy use of Ensemble in order to ensure the coherence of the partial views each host has of the world.

The architecture of Urbi *et* Orbi is designed to face such difficulties while ensuring real-time rendering. Conceptors of worlds can then take care of the graphical aspect. To get immersive capabilities with Urbi *et* Orbi, the end-user can today buy cheap commercial 3D glasses which rely on OpenGL to produce 3D sensations. Our experiments were limited to a fast LAN (Ethernet 100MB) with PC equipped with 3D video cards: we typically reach 25 frames per seconds with high quality images (see figure 2) and excellent interactivity.

A key aspect of our projet is the intensive use of the functional language *Objective Caml*, which allows the developers to leverage powerful language support to attain high performance and flexibility. Urbi *et* Orbi differs from other systems in that it is mostly scripted.

The advantage is twofold. This feature addresses the real need of being able to develop components that may be dynamically inserted into a distributed virtual environment, and it also allows to dynamically adjust the configuration of the environment.

In its current state, the Urbi *et* Orbi project has proven that non standard tools can be of a great help in the design and implementation of a DVE. Comparing the two versions of the prototype has also demonstrated that with such tools, one can obtain even better performances than with fashionable languages such as Java or VRML. Lastly, several problems have been put under evidence thanks to the prototype, notably in the field of information structuring. A rewrite of the system, possibly by using languages dedicated to distributed systems, such as Erlang, is currently under study.

References

[1] Tapas K. Das, Gurminder Singh, Alex Mitchell, P. Senthil Kumar, and Kevin McGee. Developing social virtual environments using NetEffect. In *Proceedings of the 6th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE'97)*, IEEE Computer Society Press, pages 148–154, 1997.

[2] M. Deriggi, M. M. Kubo, A. C. Sementille, S. G. Santos, C. Kirner, and J. R. F. Brega. CORBA platform as support for distributed virtual environments. In *Proceedings of the IEEE Virtual Reality International Conference (VR'99)*, Houston, USA, March 1999.

[3] Stephan Diehl. Towards lean and open multi-user technologies. In *Proceedings of the International Symposium on Internet Technology (ISIT'98)*, Taipei, Taiwan, April 1998.

[4] Emmanuel Frécon and Møarten Stenius. DIVE: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments)*, 5(3):91–100, September 1998.

[5] Chris Greenhalgh and Steve Benford. MASSIVE: a distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (DCS'95)*, IEEE Computer Society Press, pages 27–34, Vancouver, Canada, May-June 1995.

[6] Olog Hagsand and Møarten Stenius. Using spatial techniques to decrease message passing in a distributed VE system.

[7] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, January 1998.

[8] Gerd Hesina and Dieter Schmalstieg. A network architecture for remote rendering. In *Proceedings of 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT'98)*, pages 88–91, Montreal, Canada, July 1998.

[9] Roger Hubbold, Xiao Dongbo, and Simon Gibson. Maverik – the Manchester virtual environment interface kernel. In *Third Eurographics Workshop on Virtual Environments*, 1996.

[10] Rodger Lea, Yasuaki Honda, and Kouichi Matsuda. Virtual Society: Collaboration in 3d spaces on the internet. *Journal of Collaborative Computer Supported Cooperative Work (CSCW)*, 6(2/3):227–250, 1997.

[11] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system*. INRIA, 1999. http://caml.inria.fr/index-eng.html.

[12] Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. *IEEE MultiMedia*, 4(1):48–56, January-March 1997.

[13] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

[14] M.D. Ryan and P.M Sharkey. Distortion in distributed virtual environments. In *1st international conference on Virtual Worlds*, pages 42–48, Paris, France, 1998.

[15] D. Schmalstieg and M. Gervautz. Implementing gibsonian virtual environments. In *Proceedings of the 13th European Meeting on Cybernetics and Systems Research*, pages 928–933, Vienna, Austria, April 1996.

[16] Hiroyasu Sugano, Koji Otani, Haruayasu Ueda, Shinichi Hiraiwa, Susumu Endo, and Youji Kohda. SpaceFusion: A multi-server architecture for shared virtual environments.

[17] Henrik Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality International Conference (VR'99)*, Houston, USA, March 1999.

[18] Mike Wray and Rycharde Hawkes. Distributed virtual environments and VRML: an event-based architecture. *Computer Networks and ISDN systems*, 30:43–51, 1998.

[19] Michael J. Zyda, Kalin P. Wilson, David R. Pratt, James G. Monahan, and John S. Falby. NPSOFF: An object description language for supporting virutal worlds construction. *Computer and Graphics*, 17(4):457–464, 1993.