In the Proceedings of the 2nd International Conference on Virtual Worlds (VW'2000), J.-C. Heudin (Ed.), Lecture Notes in Computer Science Series, Springer Verlag, LNAI 1834, pages 54-64, Paris, France, July 2000.

A Framework to Dynamically Manage Distributed Virtual Environments

Yoann Fabre, Guillaume Pitel, Laurent Soubrevilla, Emmanuel Marchand, Thierry Géraud, and Akim Demaille

EPITA Research and Development Laboratory, 14-16 rue Voltaire,
F-94276 Le Kremlin-Bicêtre cedex, France,
thierry.geraud@epita.fr,
WWW home page: http://www.lrde.epita.fr/

Abstract. In this paper, we present the project URBI ET ORBI, a framework to dynamically manage distributed virtual environments (DVEs). This framework relies on a dedicated scripting language, *Goal*, which is typed, object-oriented and dynamically bound. *Goal* is interpreted by the application hosted by each machine and is designed to handle efficiently both network communications and interactivity. Finally, we have made an unusual design decision: our project is based on a functional programming language, *Objective Caml*.

1 Introduction

This paper presents the current state of the URBI ET ORBI project, a framework to dynamically manage distributed environments, whose principle applications are virtual worlds. Virtual worlds are virtual 3D scenes inhabited with common objects (houses, trees, etc.) and avatars, in which one may walk around and interact. In addition, this virtual world is *distributed*: its full description is spread over several computers, connected via a network. There is no need for a single host to have full knowledge of the world.

As Das *et al.* (1997), we focus on virtual worlds for a large audience, therefore there are no other material requirements than a standard personal machine, with a connection to a local network or the Internet. Windows and UNIX platforms are supported. Scalability is required.

Compared to related projects the main characteristics of URBI ET ORBI are:

- **full distribution.** To support large scale environments, no host can be privileged. Therefore, we exclude the client/server paradigm; each machine hosts the same application and data is fully distributed.
- **mostly asynchronous.** Because there is no guarantee of network responsiveness, because the core of a DVE is essentially dealing with non deterministic events, all the communications in our implementation are asynchronous, including data management within each local application.

– **data management.** While developing our own virtual world we felt the need to supply the objects with rich semantics (type, behavior, etc.) in addition to the usual geometric descriptions. Therefore we developed a language, *Goal*, to describe the objects and their behavior including their distribution policy.
– **functional ground.** Finally, the whole architecture, from the distribution engine to the *Goal* interpreter, is implemented in a functional programming language, *Objective Caml*.

This paper is structured as follows. In section 2, we give an overview of our framework. We expose the way the information is structured and distributed. In section 3, we present the language *Goal*. We explain how it satisfies our requirements concerning the data structure; we point out its relationship to our distributed system, and we describe the software architecture in which the *Goal* interpreter plays a key role. Finally, section 4, presents our conclusion and future work.

2 Overview of the framework

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 (1998). Another way is to reuse a dedicated language like NPSOFF by Zyda *et al.* (1993). Unfortunately, both solutions do not meet our requirements: we want extra object attributes without geometric semantics, we want complex relations between objects, and we want a strong control on object distribution.

2.1 Structure of the information

We want to describe virtual worlds with rich semantics: forests are composed of trees, of which there are several types, an object such as a house has an inside and an outside, etc. We also want 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 *Goal* interpreter, 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 (*lods* for short) generated by mesh simplification: objects present different lods according to the distance from the observer. Here, using

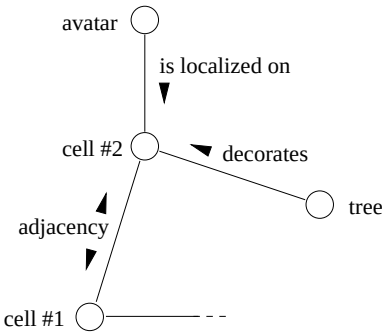

Fig. 1. Part of a conceptual graph.

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.

The natural implementation of values with specific semantics leads to the typing of values, more specifically, to the typing of the objects and their attributes of our worlds. 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.

We need a flexible type 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.

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*. Again, such relations should be typed, which simply means that attributes, or slots, of our objects must be typed.

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 additional information: the tree

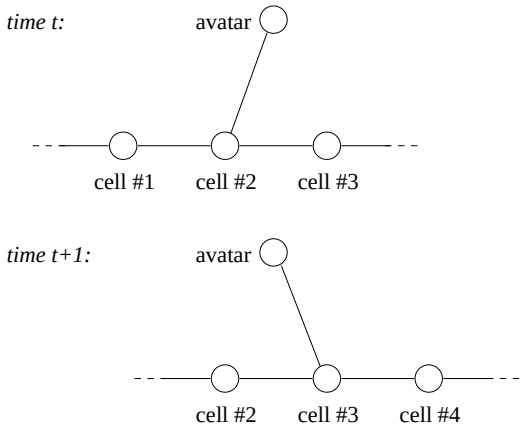

Fig. 2. Global graph modification and local evolution.

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

Because the world changes, the graph is constantly modified. The updates must be reflected in each local partial view of the world. They must be distributed.

The prototypal example of a graph incremental update is when an avatar moves and changes cell. Figure 2 shows such an example. The machine hosting the avatar has then to inform the world (namely the users seeing these cells) of the moves, to fetch needed information such as cell 4 and to flush useless information such as cell 1. There are two operations: the graph of the world is changing and the local graph of each host is evolving.

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.* (1997) and/or an aura management as proposed by Hagsand and Stenius (1997) or by Hesina and Schmalstieg (1998). 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 Distribution management

Our aim in the design of Urbi et Orbi is to provide a completely distributed environment. This approach is similar to the ones of Frécon and Stenius (1998) in DIVE, and of Greenhalgh and Benford (1995) in MASSIVE. It contrasts with client-server architectures.

In order to reduce the amount of network communication, we have preferred, like Tramberend (1999), group communication to multicast and broadcast solutions. 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. To this end, we rely on *Ensemble* [1], a group communication toolkit developed at Cornell University by Hayden (1998):

> "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 (1997), 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 (1998). We also observe that combining group communication with asynchronous and partially reliable transfer protocols results in low-cost and efficient data exchange mechanisms.

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 of world 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 the protocol 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.

Note finally that 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.

3 The language *Goal*

Our requirements pertaining to the structure of the information and to the distribution system drove us to build a new language. The additional requirement that world management must be programmer-friendly naturally led to the concept of a scripting language: *Goal*. "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.

[1] `http://www.cs.cornell.edu/Info/Projects/Ensemble`

3.1 Features of *Goal*

Strongly typed. 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.

Frame-oriented. *Goal* is a frame language implementing the concept of class and inheritance. Since *Goal* is strongly typed the notion of conceptual graph (Cf. section 2.1) 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 take, 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 (Cf. section 2.2) should be a primitive of the environment provided to the programmer. *Goal* provides a high level interface to *Ensemble*.

Dynamic. *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.

Reflectivity. 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.

Efficiency. Although *Goal* is an interpreted language, we have a very efficient virtual machine, built on the top of the one of *Objective Caml*, a strongly-typed functional programming language from the ML family; see Leroy, Rémy *et al.* (1999) for a description of this language.

3.2 Interpretation and *Objective Caml*

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.

The interpreted nature of *Goal*, together with the ease of extension, made it possible to have different types of terminals: obviously, a 3D terminal in charge of rendering the *OpenGL* information, but also a text terminal which is able to listen to the world, change it, control it, etc. Other kinds of terminals will be easy

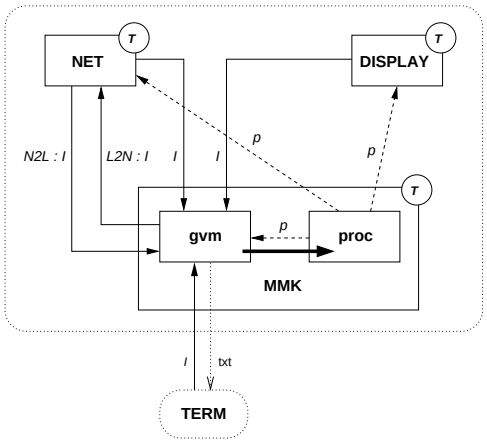

Fig. 3. Software architecture.

to develop; in particular, inferfaces with VR devices such as datagloves or head mounted displays. 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.

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. The development environment is also attractive: thanks to the 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*.

In addition, since our project is based on *Objective Caml*, *Ensemble*, also based on *Objective Caml*, appeared as an attractive choice for the distribution layer.

3.3 Overview of the software architecture

A major issue in DVEs design is to ensure that network communication does not degrade the interactivity between the machine and the user.

With this aim in view, our application is composed of three active modules, depicted by a bold rectangle in figure 3. 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). MMk includes the Goal Virtual Machine (gvm for short) to process *Goal* instructions.

NET is in charge of the communications through the network; it handles the connection/disconnection and information transfer. This module can be seen as a layer over *Ensemble*.

DISPLAY (or renderer, or navigator) provides the user with "read" access to the state of the world as known by MMK, and with "write" access: when a user moves or activates an element of the world, the display sends information to MMK. For sake of simplicity the data circuit is not represented in figure 3 and it will not be discussed touch.

Each task runs in a separate *thread* (symbolized by the letter *T* in figure 3). *Goal* is expressive enough for programmers to introduce bugs in their programs, such as deadlocks. However, the implementation, based on time slots, still guarantees that the system is fair: each task will be provided with the ability to make a step.

In figure 3, the arrows represent *channels*, i.e., queues of messages/requests, and not function calls. When a *Goal* instruction requires services (network, display etc.), the interpreter delegates the procedure call to the module proc. proc then sends the *procedure* *p* to the proper module. Technically, *p* is a closure: it contains both the procedure and the environment it needs. Then the module applies the closure: once the whole environment is filled, the procedure is executed. The receiving module is not blocked while awaiting further data: it still executes other routines. This mechanism could be considered as an *asynchronous execution transfer*.

The TERMinal is also represented; it is a shell (command interpreter) which allows direct textual communication with MMK, via *Goal* instructions.

Several other projects have based their approach upon a kernel, such as MAVERIK by Hubbold *et al.* (1996). One of the most striking differences between the two projects lies in the fact that their approach is based upon modules, while we stress the importance of the high level language, *Goal*. A detailed description of our architecture is given by Fabre *et al.* (2000).

3.4 Example

A sample file containing few *Goal* instructions is given in figure 4. It defines the interactive windmill depicted in the virtual landscape of figure 5.

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 13 a rotator to make the sails turn and to manage their speed.

Goal provides distribution primitives. Distributing data is the major challenge in large distributed worlds: ideally, any data that has to be shared or transmitted between hosts should be distributed. But, in order to perform an efficient distribution of the data we first have to classify it.

```
// 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
@listener = New RotatorListener;            // 12
@listener <- Set eventTarget = sails;       // 13
```

Fig. 4. Code sample.

Oversized objects, e.g. textures or classical 3D models (like the mill in the example of the previous section), are considered as part of a standard library that members must have available on their hosts (maybe via standard point-to-point file transport protocols). Since we aim at large distributed systems (several hundreds or thousands of participants), it is unreasonable to distribute such huge constant values to each newcomer.

All other values are truly distributed values, 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 VE application. For instance, a `3DGridObject` is an object that decorates a grid cell; its priority is medium.

Without entering the 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 VR-programmers 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, the sails of the windmill have a fully predictable behavior (they turn!) and frequent updates of their position would waste the bandwidth. So, the *replication* is a natural solution, and each host has a local timer.

Fig. 5. A virtual landscape with a windmill.

4 Conclusion and criticism

We have presented our project Urbi et Orbi, its motivations, the rationale for 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 over 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 5) 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.

Many tasks remain to be fulfilled. The priority management has not yet been implemented; the number of groups, their nature and qualities of services have not been completely established. These elements should be fixed in our current re-engineering of the prototype. Tests have only been carried out on a high speed local network; a full scale test, with an environment spread over large distances, still have to be performed. Nevertheless, the prototype behaves in a very satisfactory manner.

References

[Das *et al.* (1997)] Das, T.K., Singh, G., Mitchell, A., Kumar, P.S., McGee, K.: Developing Social Virtual Environments using NetEffect. In Proceedings of the 6th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, IEEE Computer Society Press. (1997) 148–154

[Fabre *et al.* (2000)] Fabre, Y., Pitel, G., Soubrevilla, L., Marchand, E., Géraud, T., Demaille, A.: An Asynchronous Architecture to Manage Communication, Display, and User Interaction in Distributed Virtual Environments. Submitted to the 6th Eurographics Workshop on Virtual Environments, Amsterdam, The Netherlands. (2000)

[Frécon and Stenius (1998)] Frécon, E., Stenius, M.: DIVE: A Scaleable Network Architecture for Distributed Virtual Environments. Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments). **5(3)** (1998) 91–100

[Greenhalgh and Benford (1995)] Greenhalgh, C., Benford, S.: MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. In Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE Computer Society Press, Vancouver, Canada. (1995) 27–34

[Hagsand and Stenius (1997)] Hagsand, O., Stenius, M.: Using Spatial Techniques to Decrease Message Passing in a Distributed VE System. In VRML'97.

[Hayden (1998)] Hayden, M.: The Ensemble System. Technical Report TR98-1662, Cornell University. (1998)

[Hesina and Schmalstieg (1998)] Hesina, G., Schmalstieg, D.: A Network Architecture for Remote Rendering. In Proceedings of the 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT'98), Montreal, Canada. (1998) 88–91

[Hubbold *et al.* (1996)] Hubbold, R., Dongbo, X., Gibson, S.: MAVERIK – The Manchester Virtual Environment Interface Kernel. In Proceedings of the 3rd Eurographics Workshop on Virtual Environments. (1996)

[Leroy *et al.* (1999)] Leroy, X., Rémy, D., Vouillon, J., Doligez, D.: The Objective Caml system. Manual Report, INRIA, Rocquencourt, France. `http://caml.inria.fr/index-eng.html` (1999)

[Macedonia and Zyda (1997)] Macedonia, M.R., Zyda, M.J.: A Taxonomy for Networked Virtual Environments. IEEE MultiMedia. **4(1)** (1997) 48–56

[Ryan and Sharkey (1998)] Ryan, M.D., Sharkey, P.M.: Distortion in Distributed Virtual Environment. In Proceedings of the 1st International Conference on Virtual Worlds. Paris, France. (1998) 42–48

[Sugano *et al.* (1997)] Sugano H., Otani, K., Ueda, H., Hiraiwa, S., Endo, S., Kohda, Y.: SpaceFusion: A Multi-Server Architecture For Shared Virtual Environments. In VRML'97.

[Tramberend (1999)] Tramberend, H.: Avocado: A distributed Virtual Reality Framework. In Proceedings of the IEEE Virtual Reality International Conference, Houston, USA. (1999)

[VRML (1997)] VRML'97: Proceedings of the 2nd International Conference on the Virtual Reality Modeling Language. (1997)

[Wray and Hawkes (1998)] Wray, M., Hawkes, R.: Distributed Virtual Environments and VRML: an Event-Based Architecture. Computer Networks and ISDN systems. **30** (1998) 43–51

[Zyda *et al.* (1993)] Zyda, M.J., Wilson, K.P., Pratt, D.R., Monahan, J.G., Falby, J.S.: NPSOFF: An Object Description Language for Supporting Virtual Worlds Construction. Computer and Graphics. **17(4)** (1993) 457–464