# CORBA Visualization Platform

Thierry Benoist, W. T. Hewitt, Nigel W. John

Manchester Visualization Centre

## Abstract

Virtual Reality applications mainly fall into two categories: on one hand, applications handling a virtual environment in which users interact, on the other hand scientific visualization applications handling post processing of data. The former makes use of proprietary network protocols to allow remote users to share the same experience, whereas the latter makes use of other proprietary network protocols to spread the computation load over several computers. This research aims at using a standardized network architecture, CORBA, to develop a VR application capable of handling a multi user virtual environment, in which scientific simulation occurs. The computation power necessary to run the simulation is distributed across several machines, once again through CORBA. CORBA does allow very heterogeneous platforms to communicate; this paper unveils the latests results in validating and testing this new kind of heterogeneous visualization environment.

## 1 Introduction

VR[1] has become in the last ten years an abundant source of excitement. The constantly growing power of computers allows us to achieve each year new levels of performance, endowing this field of study with an extraordinary dynamism. The growing success of Reality Centers in academia as well as in industry opens more and more research and development opportunities based on VR technologies.

Furthermore, the introduction of Internet in the mass market gave birth to distributed VR, whose concept is to allow remote users to share the same virtual experiment. This union between networking and VR is fairly recent and different problems arise due to the wide range of 3D hardware and 3D software libraries.

In short, the VR world is for the time being very heterogenous, no real effort being made to bridge the gap between different hardware/software solutions.

The current paradigm in distributed graphics is to share an environment among users, each graphics workstation obviously renders the scene via its own graphics subsystem, the CPU time being reserved for altering the virtual environment. This can be done through user interaction, or through calculation results, a process known as post processing, if you want to simulate a phenomenon for instance.

The problem in the former statement is that you do not have access to a lot of computing power on a graphics workstation or even on a graphics server such as SGI Onyx2s[1].

For jobs requiring a lot of computing power, the approach currently taken consists in first doing the calculation on a HPC[2] facility and then viewing the results of the simulation through a VR application.

The goal of this work is to present a brand new approach to distributed VR using as a reference model computational steering, see Figure 1. This paper addresses the issues raised in this short introduction, therefore investigating a solution to solve heterogeneity in distributed VR applications and integrating HPC in those very applications.

We will first present an overview of related work to set a context to this research, then the general aims of the projects will be discussed. Finally the current achievements as well as the future work will be presented.

[1] Virtual Reality

[2] HPC: High Performance Computing

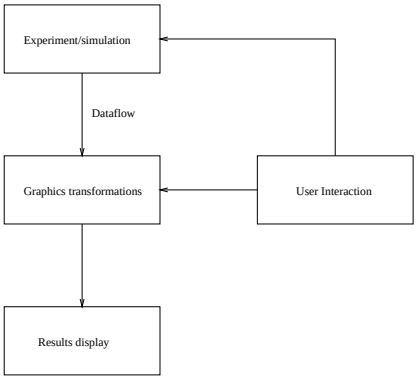


Figure 1: Computational steering
Computational steering is a reference model where the user interactions influence the rendering process and the simulation process. Influencing the rendering process might be for instance viewpoint changing whereas influencing the simulation might be a real time change of an experiment parameter.

# 2 Related Work

In order to set a proper context for this thesis, we will review here the research being conducted in this area and relate it to the research presented in this report.

Let us underline first the two main different types of research conducted in VR:

The first approach consists of defining a virtual environment possibly shared, where many users can interact. In this area the research aims at reducing the network lag by implementing alternative strategies.

The second approach is scientific visualization. You have a simulation running on a supercomputer, which generate a huge dataset. Why not then using a VR facility to project you and your dataset into an immersive environment? There you can tamper your dataset at will.

Now that we have briefly sketched the two main approaches in VR research, let us be more specific about them.

## 2.1 Shared VR environments

Today, workstations capable of handling millions of polygons a second are so cheap, that a brand new range of applications involving high performance 3D is born. Let us have a closer look at one of them, for example, the Maverik[4] and Deva[5][3] software.

Maverik is a system for managing display and interaction in virtual reality applications. It is designed to address the challenges of VEs where many objects have real geometry and therefore facilitate interaction. This is in contrast to reliance upon techniques such as texture mapping, popular in computer games, which provide the illusion of complexity, but do not afford interaction with the majority of the environment.

Maverick is a very proven product as it has been widely tested and upgraded over time. However it has the following drawbacks:

- It is written in C: the lack of object methodology makes it harder to maintain, and sometimes awkward to use when programming Maverick applications in C++
- It is monothread. Hence it will not gain any advantage from running on a parallel machine
- It has no network capabilities by itself. Although this point has been addressed by Deva, an extra library allowing different instances of maverick, or any other Deva-compatible viewer, to interact. Of course, that induces an overhead as you need to implement an interface between your viewer and your communication layer.

Let us now say a little bit more about Deva[9].

The Deva Project aims at developing a multi-user VR system to support distributed complex virtual environments. The Deva system aims to address two specific issues: complexity of behavior, and geographical distribution. These two factors are actually very strongly related in as much that the means by which behaviour is described in a virtual environment affects the techniques used to distribute that behaviour around a multi-user virtual environment.

The tandem Maverik - Deva is a fairly well proven solution, although still under development, which is used in a lot of different projects such as [6], [7] and [8]to achieve distributed VR.

## 2.2 Scientific Visualization

All the previous systems were dealing with user interaction in a distributed VE[3]. The user was inter-

[3] Virtual Environement

acting with the environment and other users. As you want a lot of users, most of the above applications will run on typical workstations. Therefore, their aims are different than those in scientific visualization.

Here, we are dealing with a dataset generated by some simulation, and we want to make the most of it. Therefore immersion can help in understanding the data as you can watch them from every angle, in 3D. Also you can apply transformations on the set of data, so that you can focus on a particular aspect. That is what products like AVS and Covise[14] are about.

Open GL Vizserver is also relevant. It presents a way to use the network to export Scientific visualization rendering.

#### 2.2.1 Scientific Visualization Toolkits

Scientific Visualization toolkits like AVS[10][11] and NAG Explorer[15] are popular visualization application software and development environments.

They accept the data produced by instruments, or by scientific and engineering simulation software and support a comprehensive set of data types. They create a visual display of your data in a variety of forms using a wide array of visualization techniques. Typically, they employ modular architecture comprised of many separate, yet tightly integrated, sub-systems that each provide important capabilities.

Modules are the building blocks. A module is an independent computing element (C or FORTRAN) which is represented by a rectangular icon on the screen. Each module performs a specific visualization function or set of functions. Alternatively, you can build your own modules to extend or customize the software. External programs or subroutines written in C or FORTRAN can be converted into modules.

However, most of these toolkits do not support distributed modules easily.

#### 2.2.2 OpenGL Vizserver

SGI has developed OpenGL Vizserver[2], a technical computing solution designed to deliver advanced visualization capabilities and performance to the desktop. OpenGL Vizserver allows users to view and interact with large data sets from a desktop system at any location in an organization. OpenGL Vizserver transmits compressed images from the Onyx2 frame buffer, resulting in the fastest possible update to the client desktop system see Fig 2.

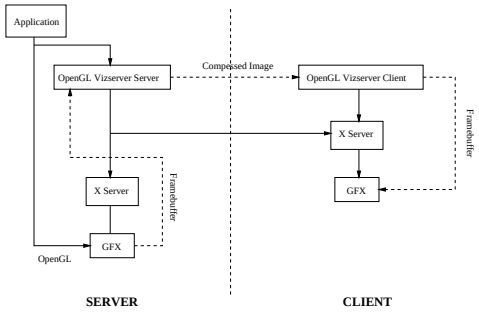


Figure 2: Vizserver architecture

In other words, Vizserver is a good solution to export an OpenGL rendering over a network, similar to exporting a XWindow display. However, it has the following limitations: it does not work in stereo, it requires a sustained network bandwidth - 100 base T recommended- , as it sends over the network the content of the Onyx2 pipe's framebuffer and finally the serverside is only available for Onyx2 computers, although you can export the rendering to any openGL platform, Irix, Linux and SunOS being currently supported.

### 2.3 The CORBA visualization Model

If you consider the software designed for shared VEs, then it is clear that many of the issues related to sharing VEs have been, or are being, solved. What we try to introduce here is a new concept of shared VR, using interactively supercomputers to generate environments.

We are not addressing network lags or high end research on existing VR applications. What we try to do here is to validate a concept that may open at a later stage opportunities for further research and development. In fact, all the software presented above used their own proprietary network layer and run on graphic workstations or graphic supercomputer. What if you could use a network protocol so generic that it could be used anywhere, and what if we could run the part of the code which is not

linked to graphics on a supercomputer like a Cray T3E?

The second kind of software identified, dealt with scientific visualization. Once again a lot of work has been done in this area, and software like AVS and COVISE are neatly adapted. What we will do here is to render into the VE so that you can see the phenomenon actually happen, which may add to the understanding as much as just analyzing results. You can of course generate data while you watching the simulation that you can browse later using AVS or COVISE. Analyzing datasets with AVS or COVISE will tell you what happened as this work will tell you how it happened.

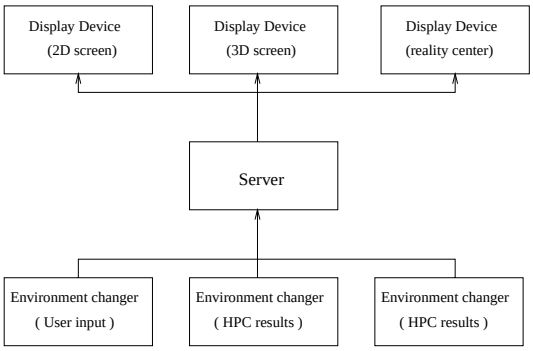


Figure 3: Main overview of the system, seen as a link between HPC and Display devices

# 3 Aims

## 3.1 Introduction

This section presents the goals of this research, highlighting the key issues and the features that are to be achieved.

This research tries to fill a gap in HPC, namely interactive HPC. HPC is tightly linked to batch computing as it is the only true way to ensure the full use of the machine at any time, thus giving the best cost performance ratio. Hence most of supercomputers are today used in batch mode, and scientific visualization then allows the browsing of the results.

The subsequent lack of interactivity is significant as you can only browse 'frozen' results. One of the goals of this research is to make an interactive use of supercomputers, for instance user input such as changing parameters are taken into account real time.

This is one of the biggest difference between visualization and VR. In visualization you try to get the most of the representation of a given set of data, while in VR you have the notion of environment, something you can see evolve and react with. This project intends to merge those two concepts.

Hence the application resulting from this research, at the crossroads of HPC, scientific visualization, VR and simulation must deal with the following key issues:

- extended portability, in order to link HPC, with network and display management.
- Modularity, as the code will be run on different machines: it is very unlikely to see a supercomputer such as a T3E with a graphic pipeline. So it becomes necessary to break the code into several components or modules. The modules will have to be adapted to their hosting computer while communicating via a common interface.
- fast and transparent network layer for the previous modules to communicate.

Therefore the key features the system should provide are:

- Portability. The software should run on a wide range of computers from T3E to Linux PCs
- Scalability. The software should run on any number of machines simultaneously
- Real time high end simulation. The application should be able to run any simulation on a supercomputer and display the results real-time.

A high level figure showing the main strategy is shown on Figure 3. These features are described in the following sections.

## 3.2 Developing a high end Simulation platform

Simulating a reactive environment is the ultimate purpose of this project. Simulation is addressed is as follows:

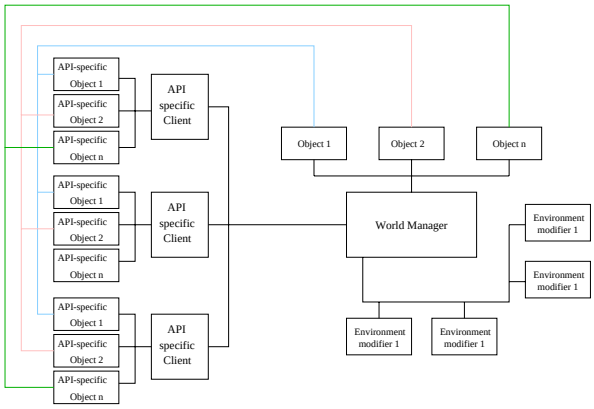


Figure 4: Illustration of the system from a lower level. Each box represents one module. We can see here the environment hosted by the server then all the modules connected to it.

First, you create an empty environment and wait for the graphics clients to connect. The environment is then populated by objects. Eventually, some modules, also called environment modifiers, alter the objects.

Typically modulesare to be optimized for parallel computing using MPI or pthreads for instance and graphic clients are to be optimized for the hardware they are running on.

As it has being said previously, the application will run the same with any number of modules and graphic clients connected on it. The structure of the resulting architecture is shown in Figure 4.

# 4 Achievements

## 4.1 Introduction

This section describes currently implemented features.

In the first place, it is important to understand the client-server paradigm. Here, a class named *Worldsmanager* plays the role of the server. It can be seen as an embodiment of the virtual world. It has knowledge of every single object in the environment. It provides the only way to create, access and transform objects.

The server has no graphics ability of any kind, as we want it to be independent from any graphic library so that it can be compiled on any platform, not only graphics workstations. Of course any transformation occurring on a given object must result in a change that is seen on a display device. Hence it is the role of the server to forward any transformation to all the graphic clients, instantiation of the class *World*. We will see in greater detail how servers and clients interact at a later stage. What is obvious from the previous paradigm is the importance of networking.

As it has been said, the server is the one and only way to create, access and transform objects. All those operations are run in some modules - or environment modifiers, remotely. As we aim at developing a distributed system on various platforms, we need a homogeneous network layer to link all those platforms together with the minimal hassle and the maximum efficiency.

The current implementation consists of:

- the deployment/port of CORBA.
- the implementation of generic clients, section .
- the implementation of a server.
- a basic module management.

## 4.2 Platform independent network layer

### 4.2.1 Introduction

As we discussed earlier, the network layer is a key issue of any distributed application.

We want an efficient, programmer friendly, portable environment. In other words, we do not want to reinvent the wheel by implementing a specific protocol and all the portability tests "is this host big Indian ?" and everything of the like.

As the implementation of this project is in C++, a good answer to those concerns is CORBA.

### 4.2.2 CORBA deployment

The first part of the work was to support a maximum number of CORBA platforms. Once a platform is made CORBA-compliant, you can start developing a port of the main code.

Most main code ports required no code modification at all thanks to the efforts that have been made to keep the C++ code clean, and also mostly thanks to CORBA. Hence, the key issue is to port a CORBA ORB itself. Therefore it is relevant

to choose an ORB with numerous supported platforms.

ORBaccus[12] proved to be a good choice in this regard. It is also opensource software, so it can be ported where needed. A lot of ORBaccus resources can also be found on the web. Finally ORBaccus has proven itself fast and reliable. It is the solution adopted by the US military for their ballistic missiles division and by NASA for their Hubble telescope amongst others.

ORBaccus has also proven itself easy to deploy on most platforms except for the Cray T3E under Cray UNICOS/mk.

Making the T3E CORBA-compliant involved:

- Rewriting all low levels CORBA functions
- Patching the IDL compiler
- Patching the JThreads/C++ library. ORBaccus being multithreaded, it makes use of a thread library built upon pthreads.
- Patching the CORBA name service.

Eventually, all the following platforms have been made CORBA-compliant:

- CRAY T3E under UNICOS/mk
- SGI Origin 2000s and Onyx2s under IRIX 6.5
- Linux PCs (debian & redhat)
- IBM SP under AIX 4.

The clients have been successfully tested on SGI O2000s, Onyx2s and Linux PCs using GeForce and fireGL cards under Xfree4.

## 4.3 Implementation

### 4.3.1 Introduction

Figure 5 shows a mid-level UML class diagram, illustrating the internal architecture of the system and the different interactions it deals with.

At startup, the server is initiated and the clients register themselves. Each client represent an opened window on the virtual environment. The actual environment is stored on the server, whereas the perceived environment is stored on the client. Here is a straightforward analogy: think of a tree in

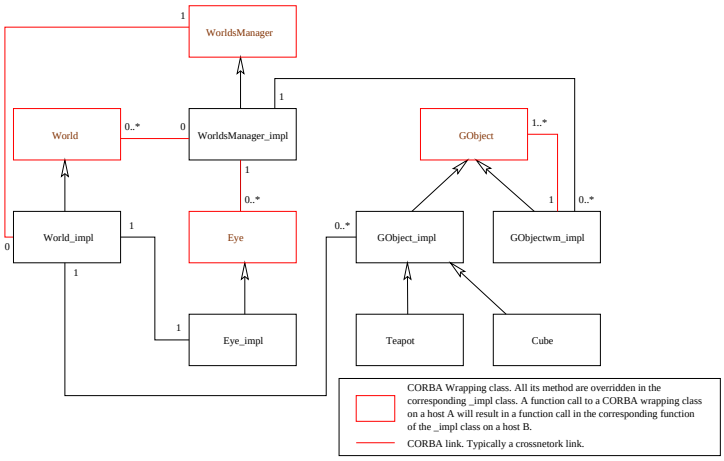


Figure 5: UML Class diagram of the system. The Server *Worldsmanager_impl* is linked through CORBA to the graphics clients *World_impl* via the CORBA wrapping classes *World* and *Worldsmanager*. Each client owns a viewpoint *Eye_impl* shared with the server through *Eye*. The server's representation of the world is a list of objects encapsulated in the *GObjectwm_impl* classes. To every single object on the server is attached a list of *GObject*s each of them being the remote representation of this very object on a client. At this stage, such an object can only be a *Cube* or a *Teapot*.

a field. This is data about the environment, hence the fact there is a tree in the environment will be stored in the server. What you see when you look at that tree is not the actual tree. You see the interpretation the brain makes of the signals traveling along the optic nerves. The perception of the same object in terms of shape and colour for instance may vary. A fly and an eagle would not have the same vision on the very same tree. Therefore, in our paradigm, the fly and the eagle are two different clients. Of course at some level the perceived environment and the actual environment are to be consistent.

When we create an object, the server tells all the clients that there is something new to be displayed and each client does just that but in its own way. What is important here is to understand that for each object *GObjectwm_impl* created on the server, there is a link to every representation *GObject_impl* on every client.

Now that this structure is in place, we can notice it is also adapted to handle object transformations. Let us consider this tree again. The role of the server is to convey information to the input of the display pipe of the client. If the server is informed that the tree has changed in some way, then it will

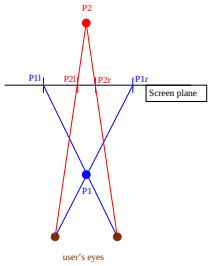


Figure 6: Stereoimpression.
Both views are projected on one frame simultaneously, the images being interlaced columnwise.

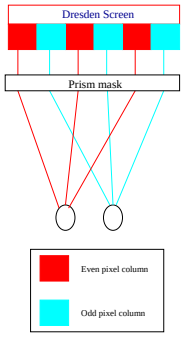


Figure 7: Dresden 3D display

contact all the clients to tell them so. What you will see afterwards depends entirely on the client.

The source of the information 'the tree has changed' will be a module. The means to convey the information from the module to the server and from the server to the clients is CORBA.

### 4.3.2 Client features

As we said earlier, the client's role is to interpret and display an environment. It has a proper data structure representing its vision of the world, but only a server can update this data structure by issuing a CORBA request.

The characteristics of a client are numerous, among them:

- type of vision (2D or 3D)
- Optics specification (Angle of vision, perspective specification,...)
- type of renderer (OpenGL, DirectX, Custom renderer...)
- type of rendering (shading, lighting...)
- rendering capabilities. In other words, which objects and which transformations the client can handle.

The 3D display device that we have used is a Dresden D4D Screen [13]. To achieve stereo impression as shown on Figure 6, a prism mask has been mounted on the panel of the screen. Using this prism, the left eye of the user can only see even columns of pixels, whereas the right eye can only see the odd columns, as shown on Figure 7.

Let us stress that all clients are multithreaded. One thread deals with the X event loop and the OpenGL calls. Some other threads -the number varies- deals with the network. As their respective domain is varied, graphics and network components rarely interact directly, the overhead due to synchronization is thus reduced to a minimum. They interact through directive FIFO[4] stacks - one per object, as shown on Figure 8. The network threads pushes in the stack the incoming directives (e.g. transformations), whereas the OpenGL thread, when it comes to the refreshing code, pops all the transformation from the stack of each object. Applying transformations to an object is implemented through a 4x4 matrix $M$ that is owned by the object.

This matrix reflects all the transformations applied to the object. Hence, applying a new transformation consists in updating this matrix by multiplying it by the relevant transformation matrix $T$. If the transformation to be applied is local, the update to perform is $M = MT$. If the transformation is global then the update to perform is $M = TM$

### 4.3.3 Server features

As we have seen in the previous section the client's role is to handle incoming directives from the CORBA layer and generate a graphic output. The server's role is to handle inputs from the modules and generate directives for the clients. So whereas clients are based half on CORBA, half on a specific graphic API, the server is based only on CORBA. Therefore it can be compiled and run on

[4] FIFO: First In First Out queuing model.

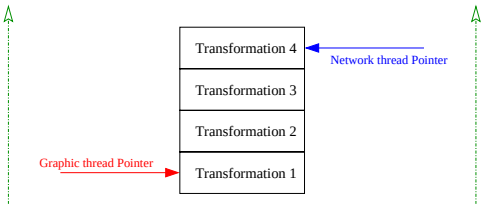


Figure 8: Directive stacks
The figure above shows a directive stack for a given object. The green arrows show the evolution direction of the pointers. The network pointer drawn in blue points at the last received transformation while the graphic pointer drawn in red points at the next transformation to apply to the object.

any CORBA enabled platform with no need of any extra library. So far servers have been successfully installed and tested on IBM SP, SGI Origin2000 and Onyx2, Linux PC, Cray T3E.

As the server makes use of native multithreading, it is beneficial to run it on a multi processor machine with a decent pthread library. The typical platform for a server would be an SGI Origin2000, although it works fine on a single CPU Linux PC.

Basically the server is a router. It has knowledge of all the connected clients, and of all the IORs of those clients objects. Therefore calling a translation on a local server object for instance, will initiate such a loop:

- *while there is still a client to consider after this one*
  - *send a CORBA translation call to the corresponding object on that client*
  - *consider the next client*

Therefore all the modules point to the server, and the server points to all the clients.

As for the raw features, the server in its current state of development handles simple object creation and transformation, namely cubes and teapots, makes use of native multithreading, it is portable and scalable. Thread numbers and all data structures are dynamic. You can connect as many clients as you want. It is also evolutive. All the structure is in place to handle more objects types, included non trivial objects such as DXF files.

#### 4.3.4 Modules Features

There is two main kinds of modules: interactive and global modules. Interactive modules are modules dealing with user inputs for instance. They obey a push model, which means that any request coming from such a module will be executed at once. Global modules are not yet implemented and are discussed in the future work section.

The case studies in the next section shows how a interactive module operates.

### 4.4 Case study

For the time being, the system only accepts interactive modules. Those modules obey the push model, the server having no control over them. An interactive module will execute faster than a module obeying the pull model as the server does not have to handle modules synchronization, reducing networking overheads. Each module is executed in the same 3D environment, however they have no knowledge of one another whatsoever. Each module runs a self contained simulation in its own time frame: the notion of time is not common to the whole environment but proper to each module.

Interactive modules can be used when one module is enough to describe a whole simulation. They can also be used for comparison purposes. Suppose you wrote the same piece of code with threads and MPI and you want to compare them visually. As they are not synchronized, the modules will run at maximum speed, and you will be able to actually see the difference in efficiency.

You can also test the impact of distributing a module by running the same module simultaneously locally and remotely.

Two modules have been implemented to demonstrate interactive module use. It is important to say that those modules were C++ programs and only a few extra lines of code were needed to turn them into multithreaded corba modules.

The first module to be developed was a teapot race. You specify a number of 'contestants' and the module creates the corresponding number of teapots, creating a thread to handle each. Afterward, each thread advances randomly its teapot and sleeps for a random amount of time then repeats the operation. There is no synchronisation whatsoever between the threads in the module. They can freely access the corba ressources, the synchronization being dealt with at the server level.

The second module to be implemented was a basic multithread resolution of the n-body problem.

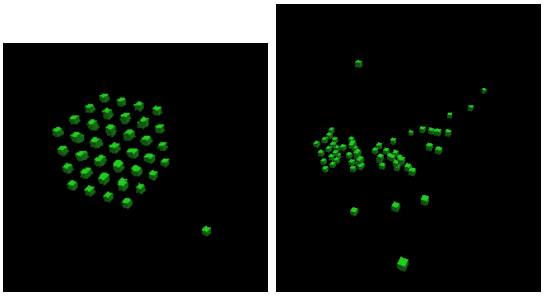

Figure 9: This example shows how 64 1kg material points - represented by small cubes- contained in a cubic meter get distorted by the gravity field of a single passing 1000 tons cube. The 1000 tons cube starts attracting the other cubes. But doing so it draws them closer and closer to each other until a point where it no longer applies the predominant force. Then, 1kg cubes will deviate each other quite violenty, resulting in a pretty quick and wide spread.

Each body has got a thread working out speed acceleration and position from the position and mass of all the other bodies. This does require synchronization on the module level, the point being that any thread concurrency model will work just fine with this system. Each thread computes data for the next instant then waits for all the other threads to be done. When this occurs, a main thread advances the clock and starts all the other threads over again. An example of this module running is shown on Figure 9.

# 5 Future Work

## 5.1 Objects Representation Improvements

For the time being, an object is represented by what it can do, such as rotation or translation. This interface is essentially one-way, from the module to the object, and therefore does not allow the module to read any information it may need to generate transformations. In other words, currently, only the module creating the object knows about its properties through local structures.

Therefore, the CORBA representation of an object is to be refined in adding 'properties'. Object properties can be anything, from physical parameters such as mass, velocity, charge, to diverse parameters such as name, type.

## 5.2 Module Management Improvements

Numerous improvements need to be done in this area.

Global modules support is to be written. Those modules obey a pull model. A module manager in the server will 'ask' each module to run a pass at a given frequency, which can be considered as the time differential *dt*.

Another issue to deal with is the problem of priority. When modules are running, you may wish to have them respect a precedence constraint, for example. Suppose for instance, you are simulating an object with variable mass, with magnetic and gravitational fields applied on it, for instance a magnetic levitation train. Such a problem can be solved using three modules, each of them dealing with Lorenz/Laplace forces, Newton forces, and with mass changing. It is clear that you want the action of gravity and magnetism to be simultaneous. You will also want the new mass computed after each gravity-magnetism pass. Therefore, modules will have to be endowed with a priority. In the previous example, gravity and magnetism have the same priority, whereas the mass changing module has a higher priority.

To sum up, the module manager will have to respect two rules. On one hand all modules with equal priority are run concurrently and their outputs are mixed before refreshing the display device. On the other hand, when a module with a priority n is run, every module with a greater priority will have already perform its pass for the current *dt*.

# 6 Conclusion

This paper demonstrates that a strong development basis has been built for further work. The network layer works across diverse platforms, from the Cray to PCs through SGI supercomputers. Clients and servers have been developed and tested successfully with excellent frame rates, even with an extensive network use through CORBA. Technically speaking, a multi user heterogeneous CORBA environment has been set up successfully with limited module support. However it is still under heavy devel-

opment which prevents any significant testing on heavy datasets. The next step will be to add more features into the system, especially at the module level, so that thorough testing can be conducted.

## References


[1] SGI, Onyx 2 technical information, *http://www.sgi.com/onyx2/*

[2] SGI,OpenGL Vizserver information, *http://www.sgi.com/software/vizserver/overview.html*

[3] AIG homepage, http://aig.cs.man.ac.uk

[4] Maverick homepage, http://aig.cs.man.ac.uk/systems/Maverik/index.html

[5] Deva Homepage, http://aig.cs.man.ac.uk/systems/Deva/

[6] Pettifer S.R., West A.J, Crabtree A."Tales of the Distributed Legible City", in proc. of 2nd Annual I3Net Conference, Sienna, Italy October 1999.

[7] Pettifer S., West A., Crabtree A. & Murray C., "Designing shared virtual environments for social interaction", in proc. 3rd Workshop on Human Computer Interaction, Kings Manor, York University, 1999.

[8] Marsh J, Pettifer S. and West J., "A technique for maintaining continuity of perception in networked virtual environments", in proc. UKVRSIG'99, Salford, September 1999.

[9] Pettifer S.,"An operating environment for large scale virtual reality", PhD Thesis, The University of Manchester, 1999

[10] AVS homepage, http://www.avs.com

[11] International AVS Center, http://www.iavsc.org

[12] ORBaccus website, CORBA resources, http://www.ooc.com

[13] A. Schwerdtner and H. Heidrich, "Dresden 3D Display: A Flat Autostereoscopic Display", Electronic Imaging / Photonics West 1998, San Jose, California, 1998

[14] COVISE Homepage, http://www.hlrs.de/structure/organisation/vis/covise/

[15] Numerical Algorithms Group (NAG) homepage http://www.nag.co.uk