(1 100801) xx

CINVESTAV

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

UN EDITOR DE AMBIENTES VIRTUALES DINÁMICOS

Tesis que presenta:
HUGO IVÁN PIZA DÁVILA

Para obtener el grado de:
MAESTRO EN CIENCIAS

En la especialidad de:
INGENIERÍA ELÉCTRICA

Guadalajara, Jal. Septiembre del 2002

CINVESTAV

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

A DYNAMIC-VIRTUAL ENVIRONMENTS EDITOR

A thesis presented by
HUGO IVÁN PIZA DÁVILA

To obtain the degree of
MASTER IN SCIENCE

In the subject of
ELECTRICAL ENGINEERING

Guadalajara, Jal. September, 2002

UN EDITOR DE AMBIENTES VIRTUALES DINÁMICOS

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:

Hugo Iván Piza Dávila
Ingeniero en Sistemas Computacionales
Instituto Tecnológico de Colima

Becario de CONACYT, expediente No. 157981

Director de Tesis:
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, Septiembre del 2002

A DYNAMIC - VIRTUAL ENVIRONMENTS EDITOR

Master of Science Thesis
In Electrical Engineering

By:

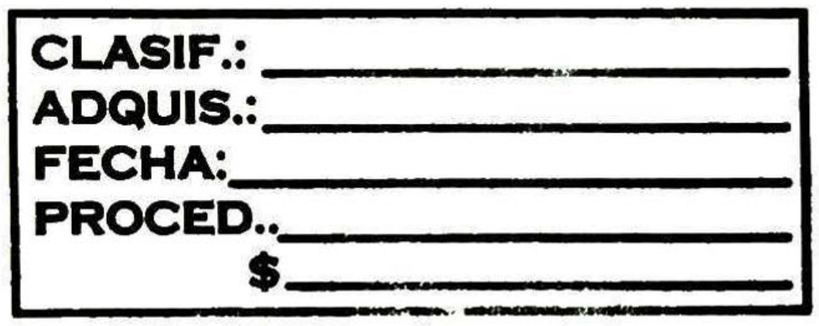
Hugo Iván Piza Dávila

Engineer in Computer Systems
Instituto Tecnológico de Colima

CONACYT no. 157981

Thesis Director:
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, September, 2002


CLASIF.:
ADQUIS.:
FECHA:
PROCED..
$

Agradecimientos:

A mis padres

A mis asesores y compañeros

A todos los profesores del CINVESTAV-GDL.

A todas las personas que contribuyeron en el desarrollo del proyecto

Al CONACYT.

Becario de CONACYT No. 157981

Resumen

Hoy en día, existen muchos ambientes de programación que asisten en el desarrollo de aplicaciones basadas en agentes. Sin embargo, la mayoría de éstos no satisfacen aspectos tales como interfaz gráfica de usuario e integración de aplicaciones. En este trabajo, se propone una plataforma con espacio 3D útil para integrar y administrar aplicaciones cooperativas distribuidas. Tal plataforma se basa en una *arquitectura de agentes móviles* y ha sido diseñada específicamente para el desarrollo de ambientes virtuales (AV) dinámicos. Muchas aplicaciones del mundo real pueden ser administradas por nuestra plataforma, tales como: trabajo cooperativo asistido por computadora, comercio electrónico, servicios de mensajería, juegos en red, sistemas de entrenamiento, etc. Nuestra plataforma proporciona un conjunto de herramientas útiles para desarrollar un editor de AV, donde los usuarios pueden configurar AV dinámicos y ejecutar simulaciones. Un AV dinámico se caracteriza por tener entidades independientes al usuario con un comportamiento específico que les permiten alterar el estado del AV. Existen diversas aplicaciones de editores de AV, tales como, oficinas virtuales, diseño de mundos artificiales, simulación de ambientes naturales y humanos, y desarrollo de juegos interactivos. En este trabajo se presentan una serie de aspectos a considerar en el diseño de editores VE. Para ilustrar las facilidades proporcionadas por nuestra plataforma, se implemente un juego de estrategia. Este juego cumple algunos de los requerimientos de un editor de AV.

Abstract

Nowadays, several programming environments that assist to develop agent-based applications are available. Nevertheless, issues such as graphic user-interface and application-integration are not addressed in most of these environments. In this work, we propose a 3D-Space platform useful to integrate and manage distributed cooperative applications. Such a platform is based on a *mobile agent architecture* and has been designed specifically for the development of dynamic virtual environments (VE). Several real-life applications are manageable by our platform, including: Computer Supported Cooperative Work, e-Commerce, Messaging Service, Networked Games, training systems, etc. Our platform provides a number of tools necessary to develop a VE Editor, where users may configure dynamic VE and run simulations. A dynamic VE is characterized for having user-independent entities with a specific behavior that enables them to alter the state of the VE. Applications of VE Editors include virtual offices, design of artificial worlds, simulation of natural and human environments, and development of interactive games. A list of issues to be considered in the design of VE Editors is presented in this work. To illustrate the facilities provided by the platform, a strategy game is implemented. Such a game fulfils some of the requirements of a VE Editor.

CONTENTS

Chapter 1

Introduction

1.1 Objectives

Describe briefly the problem this work intend to address, explain the solution proposed –remarking the activities to perform and the results expected in the completion of the work– and display the organization of the thesis.

1.2 Problem description

The problems of interest in this work are the following: creation and management of cooperative systems through a middleware, and use of Virtual Reality to create an enhanced interface for such middleware.

Due to the evolution of electronic devices and the need for providing a better human perception of computer systems, the development of interactive systems generated across Virtual Reality have become more popular day after day.

Several research projects addressing VR issues enable users to take part of a previously defined virtual environment where they can walk and fly through in three dimensions, meet other users and interact with virtual objects. These projects show a low level of dynamicity when creating virtual worlds. Most of them define a virtual world where the state of an object is not affected by the action of another one, thus an evolution of the virtual world is not evident

The aim of the present work is to develop a tool supporting the configuration of dynamic virtual environments useful for cooperative work. This kind of environments are constituted by virtual objects which are assigned a specific behavior enabling

them to interact human – autonomously with each other in the evolutionary environment (see Chapter 3).

This tool –called VE editor– intends to allow, on the one hand, programmers to develop, classify and make accessible virtual objects, and on the other, users to integrate such objects within a dynamic virtual world. Applications of our work include virtual offices, design of artificial worlds, simulation of natural environments (jungle, forest, sea, moon ...), human environments (circus, traffic in the city, sports practice ...), and development of interactive games.

1.3 Objectives

The first objective of this work is to implement a platform useful to create cooperative dynamic systems of any sort where a 3D scenario can be useful. The second objective is to study the problems to implement a Virtual Environments Editor and create a prototype of such an editor.

1.4 Solution proposed

To achieve the first objective[1], the creation of a generic architecture that would offer all services needed to manage cooperative applications is proposed. This solution uses the agents paradigm and is strongly based on a previous work of our team described in [PUGA:01, TOSCANO:00]. The architecture proposed is constituted by a Coordinator agent, a number of Clients and a number of Applications from different nature. The Coordinator comprises all the services provided by our platform. The Client agent stands for the interface between the platform and a user. It provides a 3D-space where users may interact with each other and make use of shared Applications and services provided by the platform. In addition, this platform includes a Consistency service in charge of keeping all the users aware of any change occurred in the 3D space, and addressing a number of VR issues. Finally, our architecture introduces the concept of Mobility that makes possible the transfer of the Applications to be used by remote users.

---

[1] The responsibility of the first part of this work is shared with Fabiel Zúñiga another member of our team. His email is fzuniga@gdl.cinvestav.mx

The second objective involves the design and implementation of a VE-Editor capable to allow inexperienced users to create their own cooperative systems, only by dragging objects from a menu and drop them in a specific virtual world. Examples of possible real-world applications include virtual offices, ecosystems (jungle, forest, sea, moon ...), human environments, simulators, interactive games, etc.

1.5 Organization of the thesis

The description of our work is organized as follows:

1) Chapter 2, titled Foundations, reviews some background related to Distributed Systems and Virtual Reality; besides, it describes briefly a number of technologies available today useful to implement platforms supporting the creation of Dynamic Virtual Environments; a comparison of such technologies is performed in [TOSCANO:00]
2) Chapter 3, called Virtual Environments, provides a definition, classification and characteristics of VE as our work conceive them; in addition, it presents the state of the art where we describe and criticize six research projects going in the same direction than to ours
3) Chapter 4 introduces a platform that provides the means to implement VE editors, we call it GeDA–3D; this chapter intends to emphasize the innovations added to the model architecture; besides, it introduces the competencies of the new community of agents, describes deeply two underlying services provided: Consistency and Mobility; and the end of the present, a formal verification of the Consistency Service is performed using Petri Nets formalism
4) Chapter 5 provides the conception of a VE editor for GeDA-3D and the components necessary to implement such editor; the body of this chapter identifies a number of issues to be considered in the development of VE editors for GeDA-3D
5) Chapter 6 describes deeply the case study implemented, including the problem description, rules of the game, design of the solution and implementation.
6) Chapter 7 summarizes present and future work, and recommendations
7) The last chapter lists the references used to develop this work

Chapter 2

Foundations

2.1 Objectives

Introduce a summary containing the concepts that became the foundations for our work; present some technologies useful for the development of such work; describe briefly formal techniques designed to model complex computer systems

2.2 Introduction

This work is multidisciplinary; it involves different areas of computer science. Thus, this chapter is devoted to overview basic topics related to: virtual reality (VR), distributed systems (DS), software technologies useful to develop systems, and finally, a couple of formal methods commonly used to perform the modeling and verification of complex systems where concurrency and communication are fundamental issues.

2.3 Distributed Systems

In this section, the definition and the characteristics of Distributed Systems are provided; in addition, some useful technologies that support the development of distributed applications are described.

2.3.1 Definition

A Distributed System (DS) consists of a collection of autonomous computers interconnected across a network and equipped with special software designed to maintain some shared state [MULLENDER:95, COLOURIS:96]. DS enables computers to coordinate their operation and to share resources of the system

(hardware, software, data). In addition, DS provides users a perception of a single, integrated computing facility, hiding complexities implicit in a computers interconnection.

2.3.2 Characteristics

A Distributed System intends to address six main issues.-

1. *Resource sharing*: Allow a number of hardware components and software entities to be shared usefully in a DS
2. *Openness*: Extend the system by adding new resource-sharing services without disruption or duplication of existing ones.
3. *Concurrency*: Since many users simultaneously invoke commands or interact with application programs, and many server processes run concurrently, each responding to different requests from client processes, a DS should allow several process in a DS to be executed independently or in parallel
4. *Scalability*: Operate effectively and efficiently at many different scales, ranging from a DS consisting of 2-workstations and a file server to a WAN containing several hundred workstations and may special-purpose servers.
5. *Fault-tolerance*: Keeping the system away from incorrect results or incomplete processing when faults occur in hardware or software. Two approaches are considered: hardware redundancy and software recovery.
6. *Transparency*: The DS is perceived to the user and the application programmer as a whole rather than as a collection of disjoint components

Several techniques that allow the communication and synchronization of remote processes within a Distributed System are available today. Four of them are described in this work.

2.3.3. Sockets

Sockets are a TCP/IP-based mechanism that allows programs to communicate, either on the same machine or across a network [SOCKETS]. Each machine on a network is uniquely identified by some IP address and also has a number of ports that allow handling multiple connections simultaneously.

The socket operation is: A program that intends to receive a connection from another one, asks the operating system to create a socket and bind it to some port. The program then remains listening to the socket created to receive incoming connections. The other program (caller) also creates a socket for communicating with the receiver. The caller needs to specify the IP address and the port number of the receiving end. If success, the two programs establish a communication through the network using their sockets and may exchange information, each by writing to and reading from the socket created.

2.3.4. RMI

Java offers the Remote Method Invocation (RMI) as a communication mechanism. RMI behaves as the RPC of procedural languages: it allows a process (object) to invoke a method on an object that exists in another address space (on the same machine or a different one). More specifically, RMI allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides remote communication between programs written in the Java programming language.

There are three processes that participate in supporting remote method invocation.

1. *Server:* process that creates remote objects; it makes references to them accessible and waits for clients to invoke methods on them. The remote object is an ordinary object in the address space of the server process.
2. *Client:* process that gets a remote reference to one or more remote objects in the server and then invokes methods on them.
3. *Object Registry:* a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, it can be referenced using its name through the Object Registry.

Two kinds of classes are distinguishable in Java RMI.

1. A *Remote class* is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:
    a. Within the address space where the object was constructed: the object is an ordinary object which can be used like any other object

b. Within other address spaces: the object can be referenced using an object handle.

2. A *Serializable* class is one whose instances can be copied from one address space to another. An instance of a Serializable class is called a *serializable object*. In other words, a serializable object is one that can be marshaled.

2.3.5 DCOM

The Component Object Model (COM) refers to both a specification and implementation developed by Microsoft Corporation which provides a framework for integrating components [COM:95]; in other words, it is a software architecture that allows applications to be built from binary software components [COM:99]. COM defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. The true power of COM is revealed when you remove it from the context of a single machine and spawn it on a network.

Distributed COM (DCOM) is an extension to COM that enables software components to communicate directly over a network in a reliable, secure, and efficient manner [DCOM:96, DCOM:97]; it allows you to share objects that reside on two separate machines. This means you can create an object in one application or DLL and then call the methods of that object from an application that resides on a different computer. When you are making these calls, the application server is loaded in the address space of the server machine and does not consume resources on the client. In particular, DCOM maps method calls down to standard RPC calls and then marshals the data passed as parameters between machines.

2.3.6 CORBA

The Common Object Request Broker Architecture (CORBA) [OMG:95a] is an open distributed object computing specification of an *architecture* and *interface* that provides interoperability between objects allowing an application to make request of objects (servers) in a heterogeneous, distributed environment and in a way transparent to the programmer.

CORBA was developed by the Object Management Group (OMG) [OMG]. The OMG was founded by a group of commercial vendors in 1989. The OMG wanted to develop a common way to interact with distributed objects. To support this effort the OMG developed an object model, the OMG/OM. The OMG/OM is the underlying specification for all OMG compliant technologies. It is not part of CORBA, but the CORBA Object Model is an implementation of the specification.

2.3.6.1 OMA

The OMG defined an architecture called Object Management Architecture (OMA). The OMA is a high-level vision of a complete distributed environment and outlines general technical guidelines that should be followed by every component within the OMA. The OMA Reference Model defines the categories of components necessary for the OMG to realize these goals. CORBA is the specification of one of these component categories: the Object Request Broker.

The OMA is composed of four components that can be roughly divided into two categories:

1. System – oriented components
    a. *Object Request Broker (ORB).* It is a communication infrastructure that allows or facilitates object communication. The ORB relays object requests across distributed heterogeneous computing environments.
    b. *Object Services.* Object services are low-level system type services (object persistence, transaction capabilities, security, etc.). This collection of system – oriented services allows application developers to construct applications without having to 'reinvent the wheel'. Support for object services must be included in all ORB environments and platforms. Examples: naming service, trading service and lifecycle management.
2. Application – oriented components
    a. *Common Facilities.* Common facilities are high-level, application-oriented services (such as mail and printing facilities). Unlike Object Services, support for common facilities is discretionary; they are oriented towards end – user applications. Example: Distributed Document Component Facility (DDCF).

b. *Application Objects*. Application objects are developers' programs, legacy systems and commercial software. These objects make use of the other three component categories.

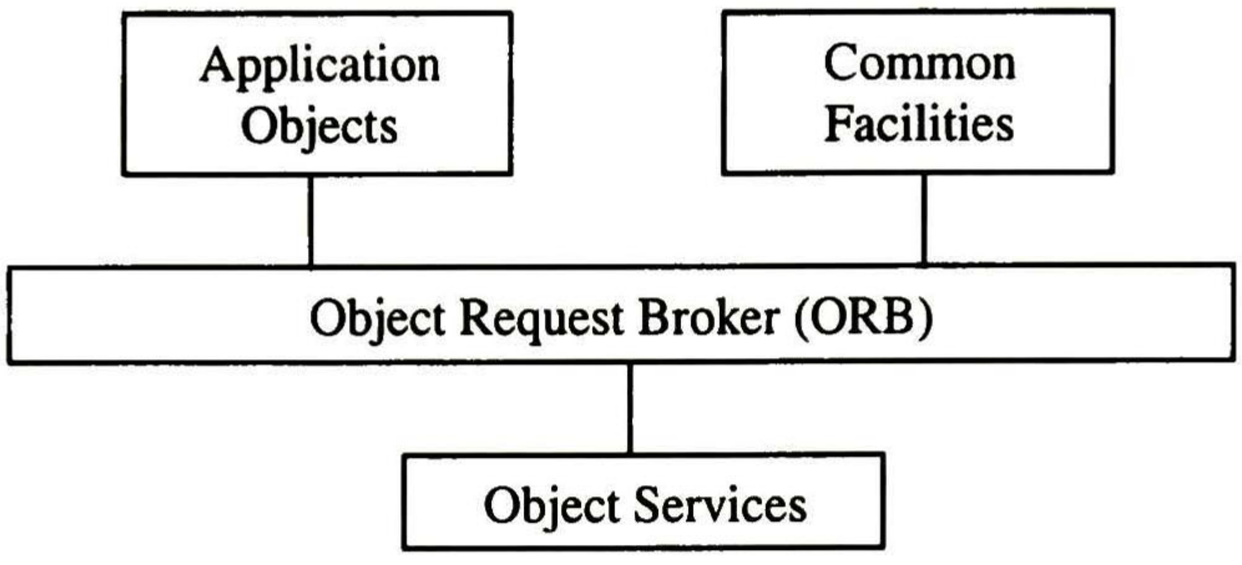


Figure 2.1 Object Management Architecture (OMA)

2.3.6.2 Architecture

The ORB manages interaction between clients and object implementations (servers). This includes the distributed computing responsibilities of location, referencing and 'marshalling' of parameters and results. Clients issue requests and invoke methods of object implementations.

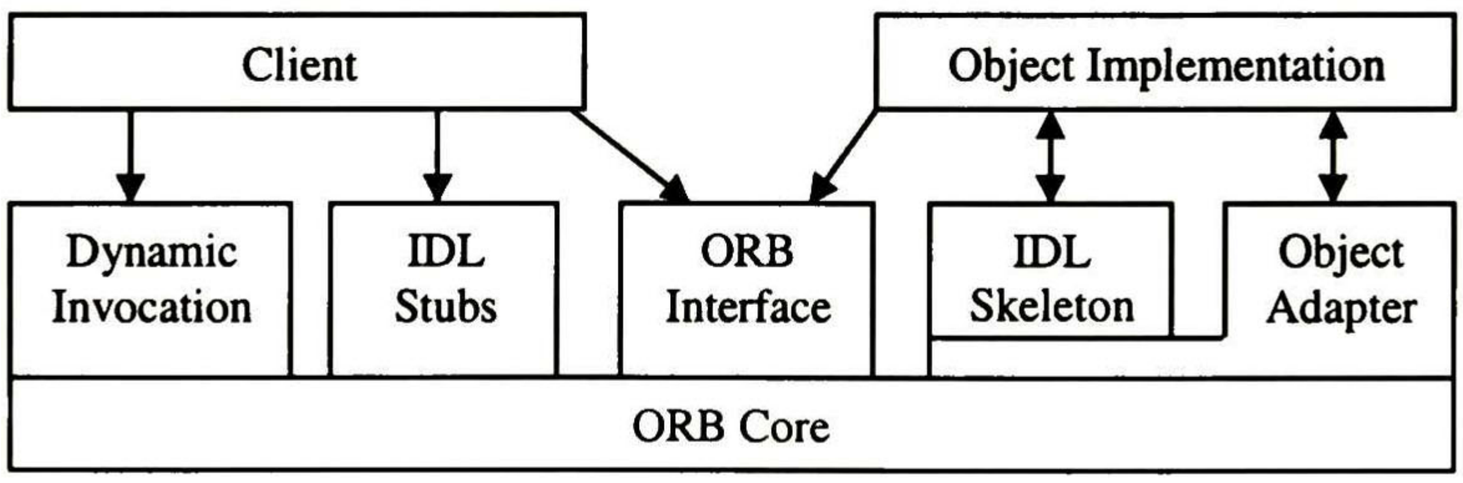


Figure 2.2 CORBA Architecture

The client side architecture provides clients with interfaces to the ORB and object implementations. In consists of the following interfaces:

- *Dynamic Invocation*. Allows for the specification of requests at runtime. This is necessary when object interface is not known at run-time. Dynamic Invocation works in conjunction with the interface repository.
- *IDL Stub.* This component consists of functions generated by the IDL interface definitions and linked into the program. The functions are a mapping between the

client and the ORB implementation. Therefore, ORB capabilities can be made available for any client implementation for which there is a language mapping. Functions are called just as if it was a local object.

- *ORB Interface.* The ORB interface may be called by either the client or the object implementation. The interface provides functions of the ORB which may be directly accessed by the client (such as retrieving a reference to an object) or by the object implementations. This interface is mapped to the host programming language. The ORB interface must be supported by any ORB.
- *ORB core.* Underlying mechanism used as the transport level. It provides basic communication of requests to other subcomponents.

The implementation side interface consists of the ORB Interface, ORB core and:

- *IDL Skeleton Interface.* The ORB calls method skeletons to invoke the methods that were requested from clients.
- *Object Adapters (OA).* Provides the means by which objects implementations access most ORB services. This includes creation and interpretation of object references, method invocation, security and activation. The object adapter actually exports three different interfaces: a private interface to skeletons, a private interface to the ORB core and a public interface used by implementations. The OA isolates the object implementation from the ORB core. The CORBA specification envisions a variety of adapters, each providing specific services. The Basic Object Adapter (BOA) is the most generic of the Object adapters.

2.4 Virtual Reality

In this section, the definition, characteristics and classifications of Virtual Reality are provided; in addition, the Distributed VR concept is introduced; finally, some useful technologies that support the development of VR systems are described.

2.4.1 Introduction

Due to the need for finding a workspace totally interactive generated through technology, in the late 80's, computer applications entered a new age where 3D solutions arrived to replace gradually 2D approaches and drawings.

Virtual Reality is the simulation of real/artificial life and sensorial mechanisms of humans with the help of computer technology. As a result of such simulation, a Virtual Environment (VE) is generated and can be experienced visually in the three dimensions of width, height, and depth in order to provide the users a feeling of immersion. VE may additionally provide an interactive experience visually in full real-time motion with sound and possibly with tactile and other forms of feedback.

The simplest form of VR is a 3D image that can be explored interactively at a personal computer, usually by manipulating keys or the mouse so that the content of the image moves in some direction, rotates by some angle or zooms in or out.

Virtual Reality and Artificial Reality (AR) are not the same. VR tends to emphasize the possibility of simulating the real world with a cognitive purpose. AR emulates nonexistent environments and fictitious scenes considered impossible since they are not compliant with physical laws. AR only intends to explore the expressive capabilities of the media supporting interaction instead of comparing the evolution of the VE with respect to real life.

2.4.2 Characteristics of a VR System

Every VR System is supposed to include the following features.-

- *Immersion.* Property that allows a user to have the feeling of being within a three-dimensional world
- *Existence of a reference point.* Helps to determine the location and observation position of the user within the virtual world
- *Navigation.* Property that allows a user to change his/her position of view.
- *Manipulation.* Enables the interaction with and transformation of the virtual world

2.4.3 Classification of VR

Immersion stands for the feeling of the human to be within a VE interacting with virtual objects. The degree of immersion varies according to the use of specialized devices enabling a better perception of and interaction with the VE. Today, the term 'Virtual Reality' is also used for applications that are not fully immersive. This way, VR can be classified as follows:

2.4.3.1 Immersive VR

- Allows virtual worlds to be presented in full scale and properly to the human size.
- Enables the perception of depth and the sense of space with stereoscopic viewing.
- Realistic interactions with virtual objects via data gloves and similar devices useful for manipulation, operation, and control of virtual worlds
- Devices such as head mounted displays (HMD) provide a natural interface for the navigation in a 3D space and allows for look–around, walk–around, and fly–through capabilities in virtual environments
- Networked applications allow for shared virtual environments

2.4.3.2 Non-immersive VR

- Mouse or similar devices – controlled navigation through a 3D environment on a graphics monitor
- Stereo viewing from the monitor via stereo glasses, stereo projection systems, and others.
- Photographs for the modeling of three 3D worlds and pseudo look–around and walk–trough capabilities on a graphics monitor.

2.4.4 Distributed VR

The idea behind distributed VR is the following: a virtual environment runs not on one computer system, but on several. The computers are connected over a network (possibly the global Internet) and people using those computers are able to interact in real time, sharing the same VE.

There are a number of obstacles to be overcome in achieving this goal.-

- The fact that we want people to be able to access the VE from their homes means that we have to be able to run over relatively limited – bandwidth links
- The fact that we want to run over the Internet means that we have to tolerate a certain amount of latency in the delivery of update information.
- Finally, the fact that people are running on different computer systems with different hardware and different software means that we must design the system for portability.

Each of the computers participating in the simulation is called a "host" On each host there is a number of "entities" (objects in the VE) that communicate their changing state by sending "update messages". The specific entity that corresponds to a human participant's virtual body is called an *avatar*.

2.4.5 VR – Oriented technologies

This section intends to introduce and describe briefly some software technologies widely used to develop dynamic virtual environments. Such technologies range from low – level specifications to libraries implemented by some programming languages and high – level interpreted languages.

2.4.5.1 VRML

VRML stands for Virtual Reality Modeling Language. It is a human – readable scene description language for describing 3D shapes and interactive environments [VRML:1, VRML:2]. VRML was formerly designed to create a friendly user – interface for the World Wide Web. Its goal is to provide a rich 3D interactive graphical environment allowing the user to define and interact with static and animated worlds. VRML incorporates 3D shapes, colors, textures and sounds to produce a VE where a user could walk and fly through. VRML is an interpreted language. That is, commands written in text are parsed by a browser and then displayed on the user's monitor. The browser may be a plug-in for a web browser (Cosmo Player) or a helper application. Many of these worlds can be found on the web today. The current specification, VRML 2.0, supports JAVA, sound, animation, and JavaScript. As opposed as VRML 1.0, it allows the world to be dynamic.

Characteristics of VRML:

- Capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images.
- VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms.
- Supports an extensibility model that allows new dynamic 3D objects to be defined allowing application communities to develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D application programmer interface (API) features.

There are four main components of a VRML file:

- Header for the file. Required for the interpreter to know that the file is written in VRML. Every VRML file must have this as its first line. E.g. #VRML V1.0 ascii
- Nodes. Commands that define shapes and properties in VRML. The four basic predefined shapes are the *cube*, the *sphere*, the *cylinder*, and the *cone*. Some other nodes are:
    - Transform. Moves, scales, and rotates objects
    - AsciiText. Defines ascii text to be displayed
    - Grouping Nodes. Used to group information together, perhaps for reuse
- Fields. Attributes that the various nodes have, such as width, height, and depth.
- Comments. They make code more easily understood in later use. Any text on a line that starts with "#" is a comment. It can be used at any point in the line.

2.4.5.2 OpenGL

OpenGL stands for Open Graphic Library. It is a low – level 3D graphics library specification. [WRIGHT:00] defines OpenGL as "a software interface to graphics hardware" Ever since it was released, OpenGL has been considered the assembler of graphic computers and has been adopted as the official API for the development of portable real – time interactive 3D applications.

Introduction

It emerged as an initiative by SGI (Silicon Graphic Inc.) to create a single, vendor – independent API for the development of 2D and 3D graphics applications. Before OpenGL was introduced, many hardware vendors had different graphics libraries. This situation made it expensive for software developers to support versions of their applications on multiple hardware platforms, and it made porting of applications from one hardware platform to another very time – consuming and difficult. SGI saw the lack of a standard graphics API as an inhibitor to the growth of the 3D marketplace and decided to lead an industry group in creating such a standard.

The OpenGL API is the result of this work, it was designed for use with the C and C++ programming languages but there are also bindings for a number of other programming languages such as Java, Tcl, Ada, and FORTRAN.

OpenGL makes available to the programmer a small set of geometric primitives: points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered into the frame buffer.

Characteristics of OpenGL

- Provides fast and complete 3D hardware acceleration
- Handle graphics in a very low level
- OpenGL specification is operating system and windowing system independent
- Provides a *procedural* interface. The program should specify an appropriate sequence of commands to set up the location, view, color and geometry for every shape desired to be drawn
- The efficiency and portability relies on the fact that OpenGL acts as an *interface* between the programmer and the operating system. OpenGL doesn't perform the drawings but instead it indicates the SO what to do

The event loop

OpenGL programs often run in an *event loop*. After the program is started, some initialization code may be executed and then program falls into an infinite loop accepting and handling events. Events include operations such as key pressed, mouse movement, mouse button pressed, mouse button released, window reshaped, exposed and displayed. A reshape event occurs anytime the program window is resized. An expose event occurs when the program window is initially displayed and also any time that it is brought to the foreground. A display event occurs after one or more of the other events are handled.

OpenGL Primitives

The programmer is provided the primitives depicted in Figure 2.3 for use in constructing geometric objects. Each geometric object is described by a set of vertices and the type of the primitive to be drawn. Whether and how the vertices are connected is determined by the primitive type.

With OpenGL, all geometric objects are actually described as an ordered set of vertices. OpenGL commands are not necessarily executed as soon as they are issued. It is necessary to call a certain command (glFlush) at the end of a sequence of drawing commands to ensure that all previously issued commands are executed and so, all objects in the scene are drawn.

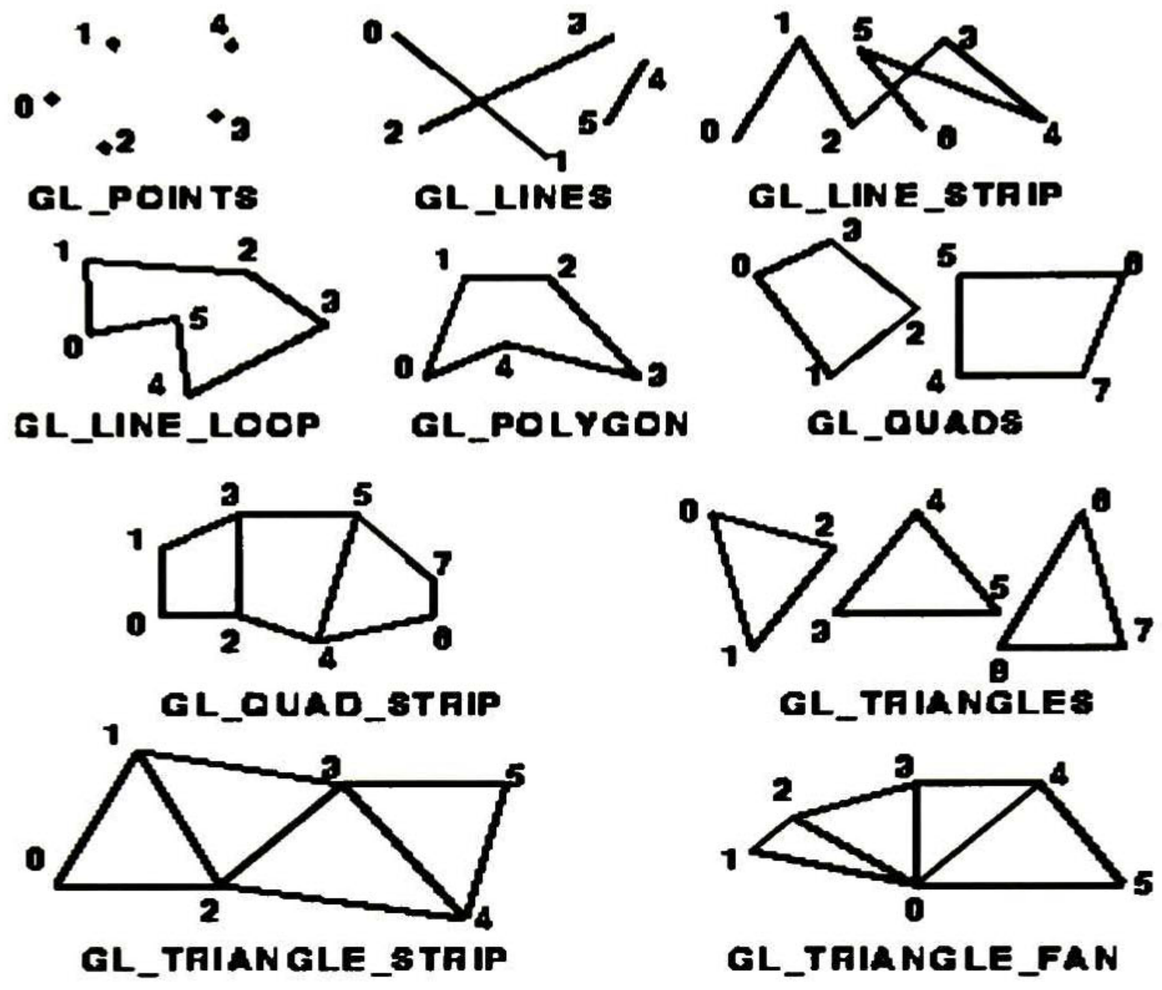


Figure 2.3 OpenGL primitives

State variables

As a geometric primitive is drawn, each of its vertices is affected by the current OpenGL "state" variables. These state variables specify information such as line width, line stipple pattern, color, shading method, fog, polygon culling, etc.

Some state variables refer to OpenGL capabilities that are either enabled or disabled. Other state variables refer to a certain mode chosen from a fixed set of modes. Lastly, there are state variables that are set to certain values (integers, floats, etc). Each state variable has a default value. The values of the state variables, whether set by default or by the programmer, remain in effect until changed.

The Pipeline

A pipeline stands for a process that involves more than one single step to be finished. OpenGL operates through a pipeline which starts as soon as a command is issued and ends when a 3D object is displayed on the screen. Figure 2.4 shows a

simplified version of the OpenGL pipeline. As an application makes OpenGL API function calls, the commands and their data (vertex data, texture data ...) are stored in a command buffer. When the buffer is flushed (emptied), either by the programmer or by the driver's design, the commands and data are passed to the next stage in the pipeline.

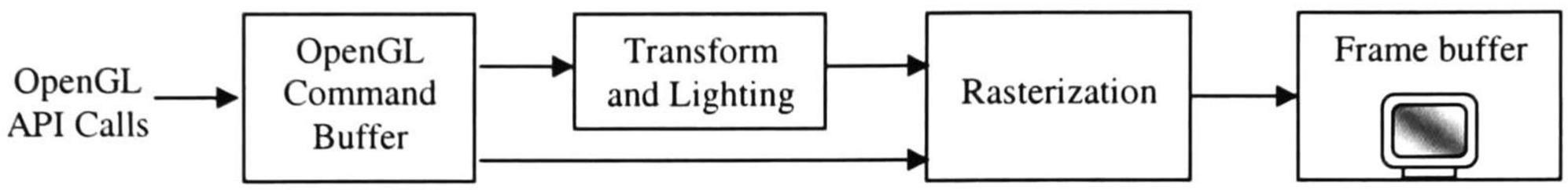


Figure 2.4 A simplified version of the OpenGL pipeline

Vertex data is usually transformed and lit initially. The *transform* is a mathematically intensive operation where points used to describe an object's geometry are recalculated for the given object's location (coordinate) and orientation (angle). *Lighting* calculations are performed to indicate how brightly the colors should be at each vertex. Once this stage is complete, the data is provided to a *rasterizer* in charge of converting projected primitives and bitmaps into pixel fragments in the *frame buffer*. The frame buffer is the memory of the graphics display device.

2.4.5.3 Java3D

Java3D is an API that provides a hierarchy of Java classes for writing applications designed to display and interact with three – dimensional graphics [JAVA3D]. Java 3D API is part of the Java Media suite of APIs, which in turn is part of the overall Java API efforts. Java 3D API is a joint collaboration between Intel, Silicon Graphics, Apple, and Sun. The initial reference implementations of the Java 3D API are layered on top of existing lower–level immediate–mode 3D rendering APIs, such as OpenGL and Direct3D. The API provides a collection of high – level constructs for creating and manipulating 3D geometry and structures for rendering that geometry. Java 3D provides the functions for creation of imagery, visualizations, animations, and interactive 3D graphics application programs.

The programmer works with high-level constructs for creating and manipulating 3D geometric objects. These geometric objects reside in a *virtual universe*, which is

then rendered. By taking advantage of Java threads, the Java 3D renderer is capable of rendering in parallel. A Java 3D program creates instances of Java 3D objects and places them into a *scene graph* data structure. The scene graph is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered. A Java 3D virtual universe is created from a scene graph. A scene graph is created using instances of Java 3D classes. The scene graph is assembled from objects to define the geometry, sound, lights, location, orientation, and appearance of visual and audio objects.

A common definition of a graph is a data structure composed of nodes and arcs. A node is a data element, and arc is a relationship between data elements. The nodes in the scene graph are the instances of Java 3D classes. The arcs represent the two kinds of relationships between the Java 3D instances. The most common relationship is a *parent – child* relationship. A group node can have any number of children but only one parent. A leaf node can have one parent and no children. The other relationship is a *reference*. A reference associates a *NodeComponent* object with a scene graph Node. NodeComponent objects define the geometry and appearance attributes used to render the visual objects. A Java 3D scene graph is constituted by Node objects in parent – child relationships forming a tree structure. In a tree structure, one node is the root and no cycles are present.

2.5 Formal methods

Several formal methods designed for modeling complex computer systems are available today. In this work, we introduce two of them, Process Algebra and Petri Nets. Both formalisms are useful to model systems where communication, concurrency and parallelism are present and considered underlying factors in design–time. Not only do they allow systems to be represented formally – using numbers, equations – but also graphically. In addition, Petri Nets provides mathematical techniques to verify if a system fulfils certain properties.

2.5.1 Process Algebra

Process Algebra [BERGSTRA:01, MILNER:89] is a formal technique for describing complex computer systems, especially those involving communicating and concurrently executing components. This subject touches many topic areas of

computer science and discrete math, including logic, concurrency theory, specification and verification, operational semantics, algorithms, complexity theory, and algebra.

A system is modeled as a process, often specified as the parallel composition of a number of other processes, the components. Components are often described by recursive equations, using sequential and alternative composition. These equations can have data parameters, which serve as state variables. Such a system is represented by an interpreted graph with

- Labeled edges. Vehicle for information passing
- Nodes. Agents with input / output ports where information is received / sent through

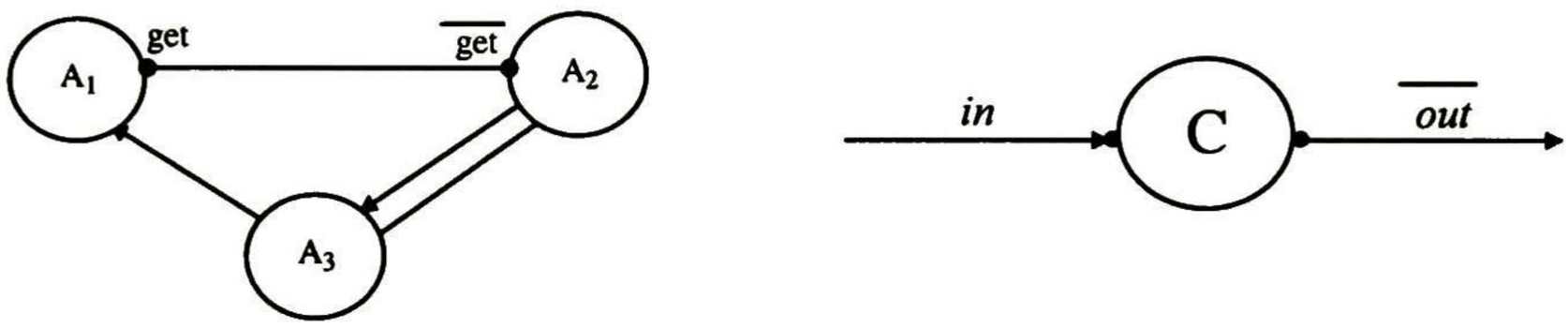


Figure 2.5 Two simple systems modeled with process algebra

Three kinds of communication are distinguishable in process algebra:

1. Between the system and the environment
2. Between two components of the system (A_2, A_3)
3. Within a single component

Consider an agent *C* (Figure 2.5) as a cell with the ability of storing a piece of data. Its behavior can be described with the following recursive equation:

$$C \stackrel{\text{def}}{=} in(x).C'(x)$$

$$C'(x) \stackrel{\text{def}}{=} \overline{out}(x).C$$

Here *in* and *out* are so – called atomic actions, which can be externally visible actions, or which synchronize with corresponding actions in different components.

A variable receives the scope according to its occurrence …

- in the input prefix: the scope is the whole agent expression that starts with such prefix

- as a formal parameter in the left side of the equation the scope of *x* is the whole equation

Process algebra includes 4 main operators:

1. **Prefix** (.) Models sequences of actions. in(*x*).$\overline{P}$ out(*x*).Buff(*y*)
2. **Alternative** (+) Useful for non determinism. in(3) + in(6) (it expects either 3 or 6 as input)
3. **Composition** (|) P | Q is as system where both P and Q act independently but they can also interact through complementary ports, usually called inner actions. (get and get in Figure 2.5)
4. **Restriction** (\ ) Restricts the number of agents allowed to connect to a specific port or set of port. In the expression (A | B | C) \ {get, put}, actions *get* and *put* are hidden (protected) from agents other than A, B and C

2.5.2 Petri Nets

Petri Nets [ZIMMERMAN:01] is a graphical and mathematical modeling language; which is appropriate to model systems characterized as being concurrent, parallel, asynchronous, distributed, nondeterministic and/or stochastic. Petri Nets has been under development since the beginning of the 60's, where Carl Adam Petri defined the language. It was the first time a general theory for discrete parallel systems was formulated. The language is a generalization of automata theory such that the concept of concurrently occurring events can be expressed.

A Petri Net consists of *places*, *transitions*, and *arcs* leading from places to transitions and vice versa. Places are represented by circles, transitions by boxes and arcs by arrows. An arc leading from a place *p* to a transition *t* is called either an input arc of *t* or an output arc of *p*; p is an input place of *t*, and *t* an output transition of *p*. Conversely, an arc leading from a transition *t* to a place *p* is called either an input arc of *p* or an output arc of *t*; p is an output place of *t*, and *t* an input transition of *p*.

Places can contain *tokens*, represented by dots. The marking of a place *p* stands for the number of tokens in *p*. The marking of a Petri Net represents the current state of the modeled system and is given by the number (and type if the tokens are distinguishable) of tokens in each place.

Transitions are considered active nodes since they model activities which can occur eventually, thus changing the state of the system (the marking of the Petri Net). Whenever an action occurs a transition is said to fire. When a transition fires, it removes tokens from all its input places and adds some at all of its output places. A series of firing gives rise to a marking evolution.

Sometimes arcs include a number (usually integer) which represents the weight of the arc. When that number is not present we assume a unitary weight. If an arc *a* leads from a place *p* to a transition *t*, the weight of *a* indicates the number of tokens to be removed from *p* whenever *t* fires. Similarly, if an arc *a* leads from a transition *t* to a transition *p*, the weight of *a* indicates the number of tokens to be added to *p* whenever *t* fires.

A transition *t* is only allowed to fire if it is *enabled*, which means that the marking of every input place *p* of *t* is greater than or equal to the weight of the arc connecting *p* with *t*. In few words, a transition is enabled when there are enough tokens available in its input places. In such case, we say that the preconditions for the activity are fulfilled.

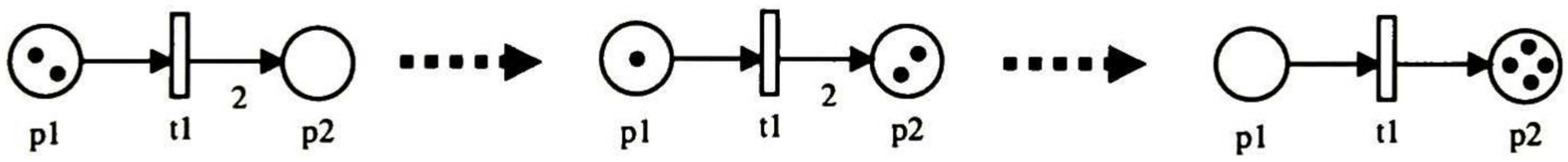


Figure 2.6 An example of a marking evolution

Figure 2.6 shows a marking evolution caused by a sequence of firings of t_1. Notice that when the net reaches the third marking, transition t_1 can no longer fire. In addition, place p_1 remains unmarked (with no tokens) forever. In such situation, t_1 and p_1 are said to be *dead*.

Some properties of modeled systems are interesting to be analyzed for fulfillment:
Boundness.- A Petri Net is *b – bounded* iff there exists an integer *b* such that the marking of every place is never greater than *b*.

- Liveness.- A Petri Net is *live* iff no transition becomes eventually dead

- Deadlock–freedom.- A Petri Net is *deadlock–free* iff at least one transition is never dead

2.6 Conclusions

In this chapter, we have reviewed some basic concepts related to Virtual Reality and Distributed Systems since they serve as a background of our work. In addition, we have introduced some technologies useful to develop static and dynamic virtual environments which can be shared across a heterogeneous network. A comparison of these technologies is not performed in this work but in [TOSCANO:00], a previous work of our team, which has served as a basis for us to choose an appropriate set of technologies to implement our work. Finally, we have described briefly a couple of methodologies useful to model and verify systems with concurrency, communication and parallelism.

Chapter 3

Virtual Environments

3.1 Objectives

Provide a definition of Virtual Environment as we conceive it; identify its main characteristics; classify Virtual Environments; and criticize some research projects going in the same direction than our case study.

3.2 Introduction

This chapter is divided in two parts. In the first one, we introduce a definition of Virtual Environment as our work intends to conceive it, and list a number of components constituting a typical VE. According to the enhancements provided, two kinds of VE are distinguished and characterized. Besides, we present the major components constituting the base infrastructure of both kinds of VE.

In the second part, we present the state of the art of our work. Here, we criticize and compare related projects with ours, emphasizing some of their drawbacks which we intend to solve.

3.3 Virtual Environment

The concept Virtual Environment is closely related to Virtual Reality [Ref 2.4.1]. The main goal of the latter is to implement Virtual Environments capable to picture real life or some sort of artificial life in a computer, with the help of devices allowing user interaction. We define VE as an interactive simulation of 3D objects arranged together in a three – dimensional space (virtual world) and capable to provide a natural human – perception.

3.3.1 Components

A general VE must be constituted by the following elements:

- A virtual world, that is, a 3D space where a user can navigate
- A database of 3D objects existing in the virtual world and keeping some physical properties
- A language or library supporting the creation and display of 3D objects inside such navigable virtual world in a non – complex way
- Output devices that display the contents of the virtual world and provide user (observer) a feeling of immersion in the virtual world
- Input devices allowing a user to interact with and manipulate virtual objects
- Controls to change a user location and observation position in the virtual world

3.3.2 Classification

Literature related distinguishes two classifications of VE: single–user and multi–user, and static and dynamic. A static single – user VE is not worth to analyze, since it adds no features to the general VE described in the previous section. Thus, we focus our attention in describing the main features and components constituting multi–user and dynamic VE.

3.3.3.1 Multi – user VE

A VE is usually distributed across a wide area network and shared by different users. Every user is assigned a graphical representation in the virtual world, commonly called *avatar*. Users may meet and interact with each other and with all other objects constituting the environment. To design a multi–user VE it is necessary to take into account the following issues, in addition to the ones mentioned in &3.3.2:

- A ***DBMS*** to store the state (in the VE) and geographical location (in the network) of users navigating
- A ***consistency*** *service* in charge of keeping all the users aware of every change performed in the shared VE. Such service synchronizes, validates and submits all these changes.
- A ***collision detector*** in charge of handling the physical meeting between a user and a solid object or between two users. This detector is used to prevent two different solid objects from share the same virtual space
- In real world it is available just ***a communications link*** with a low bandwidth

- ***Mechanisms for grouping*** the users according to their virtual location
- ***Protocols*** for transmitting/multicasting changes in the VE

3.3.3.2 Dynamic VE

A Dynamic VE is constituted by static and dynamic objects. Examples of DVE include: virtual ecosystems, games, virtual offices, etc. A dynamic object not only is assigned physical properties, but certain behavior that enables it to interact autonomously with the virtual world. In this scenario, the main role of users is to create and add dynamic virtual objects within a VE.

In a Dynamic VE, the state of a virtual object can be changed as a consequence of a non–deterministic action performed by another object. This means, that, the environment evolves in non–deterministic ways. The design of a dynamic VE involves identifying the following elements added to the ones listed in 3.3.2:

- Natural laws governing the actions performed in a specific virtual world
- A set of static objects and non – static objects
- A set of behaviors, including learning algorithms, than can be assigned to virtual objects in order to become non – static
- A mechanism to dynamically assign such behaviors to dynamic virtual objects
- A data structure to classify objects, thus enabling inheritance of properties and behaviors
- A mechanism (or interface) allowing non – static objects to communicate and interact with each other and with static objects
- A semantic in charge of keeping congruency in the VE; this involves validating the inclusion of certain objects within a virtual world
- A data structure storing the current relationship and dependencies between the objects, and capable to dynamically change as a consequence of changes in the VE

3.4 Related work

Several research projects address virtual reality issues in the same direction than we do. This section intends to show the difference between our work and some projects that we consider relevant, emphasizing some advantages and drawbacks present in

those projects. We are very sorry if our choice of systems reviewed does not include someone the reader consider very important.

3.4.1 AVIARY

AVIARY [AVIARY:97] is a high level generic multi-user virtual environment allowing several applications and several users to interact in the same virtual world. The applications being able to be managed by AVIARY cover a vast field going from the simple tools usable in the environment until broad activities covering the major part of this environment. The virtual world in AVIARY defines a set of attributes assigned to the entities of the world in a certain instance; it also specifies a set of constraints, which govern entities behavior, and supports a multiple inheritance hierarchy that allows new worlds to be defined in terms of existing worlds. It allows the management of several applications which define a behavior to the objects of the VE; moreover, it manages users in the same way than applications; finally, AVIARY can manage several virtual worlds equipped with different laws at the same time. Drawbacks: all the communications between objects are carried out by using UDP protocol, which results in an unreliable end-to-end communication; collisions detection between artifacts present in a zone is performed by a centralized server storing the EDB (Environment Database); has limited support for replications.

3.4.2 MASSIVE

MASSIVE [DVR:2] (Model Architecture and System for Spatial Interaction in Virtual Environments).- It is a multi-user, multi-media distributed VR system built on top of an underlying implementation of the Spatial Model of Communication. There may be any number of worlds with portals to move between worlds; there are textual, graphical and audio client programs allowing users to communicate by graphical gestures, typed messages or real-time packetised audio. Drawbacks: runs only on Sun and SGI platforms; usually works with up to about 10 users; not a general-purpose VR application development environment.

3.4.3 DIVE

DIVE [DIVE:93].- The SICS Distributed Interactive Virtual Environment is a fully distributed heterogeneous VR system, where users navigate in a 3D space and may interact with other users and applications in the environment. DIVE is the most

complete RVD system and more used with current time. Indeed, it is available free what makes of him a tool very much used in the educational establishments. It offers several mechanisms to manage multi-user environments and develop behaviors of virtual objects. Drawbacks: the principal language of specification of the behavior, Tcl, has weak expressive capacity and poverty of data structures; no collisions detection presented in the implementation; during user navigation, only borders of the virtual objects are displayed.

3.4.4 dVS

dVS [DVR:2] is a platform-independent software environment for virtual reality applications. dVS is a virtual world simulation software tool which allows the non-programmer to easily develop virtual worlds, animate them with intelligent and realistic properties, and experience them in partially or fully immersive modes. Drawbacks: allows management of only one world; works with up to 10 users.

3.4.5 VREng

VREng [DVR:2] (Virtual Reality Engine) is an Interactive and Distributed 3D Application allowing navigation in Virtual Environments connected over the Internet using IP Multicast Technology and RTP/RTCP protocol; VREng allows its users to make a visit of Virtual Worlds (rooms, campus, museums, workshops, landscapes, networks, machines,...). Visitors may interact with each other through their avatars. They may also communicate by exchanging text (Chat), audio and/or video channels, shared white boards and interact with objects in the 3D environment like Web panels, virtual workstations, documentation on-line, MP3 sounds, MPEG audio/video clips, and distant applications and servers; Drawbacks: not a general-purpose VR application development environment; no behaviors are defined to virtual objects.

3.4.6 VIPER

VIPER [TORGUET:98] (Virtuality Programming Environment).- A generic object-oriented platform enabling multi-user virtual environment management and, more generally, the development of collaborative Virtual Reality applications. VIPER proposes two programming levels to the developer of distributed virtual environments. The first level totally hides the distributed aspects of the application

and allows the developer define new entities and behaviors. The second level enables the specification of new virtual worlds, allowing the developer to redefine distribution schemes proposed by VIPER. Drawbacks: since the communication is performed by multicast (groups of communication) specialized equipment, such as routers, is required

3.5 Conclusions

In this chapter, we have focused our attention in analyzing the main characteristics, components and classifications of Virtual Environments. This theory provides more foundations for the development of our core work: a dynamic virtual environment for GeDA-3D.

In addition, we presented some research projects having a similar direction than ours and emphasized some of their drawbacks we are willing to solve. Only VIPER and AVIARY supports migration of behaviors from one machine to another; none of the projects described above allows management of applications with different nature, but only VR – oriented applications, and in most cases, VRML – based virtual worlds. Finally, no project described above address both the aspects of congruency and dynamic object – dependency

Chapter 4

GeDA-3D

4.1 Objectives

Introduce the architecture proposed for GeDA-3D; describe the competencies of the new community of agents; analyze deeply the operation of the Consistency Service and the Mobility Platform; describe technically and formally a session of a user of GeDA-3D.

4.2 Introduction

The experience of our team in VR started in 1998 with the construction of a virtual office running on a PC network [RAMOS:98]. In the year 2000, after a brain storm, our team decided to create a generic platform to facilitate the implementation of distributed systems with different nature involving a VR-based interface. This architecture should act as a software layer providing a number of constant features useful to develop such systems. Thus, GeDA-3D was born and the first works are described in [TOSCANO:00] and [PUGA:99].

As mentioned in the solution proposed (see 1.3), part of our work involves designing and implementing GeDA-3D taking as a basis our team's previous work, but providing some enhancements to support new services, mainly a Virtual Editor described in chapters 5 and 6 which conforms the second part of this work.

In this chapter, we first describe the competencies of the new community of agents; then, the proposed architecture for GeDA-3D is introduced; the core of this chapter includes a deep analysis of two important functionalities provided by GeDA-3D: the

Consistency Service and the Mobility Platform. In order to show the way GeDA-3D handles some VR issues, we depict a visual example with four users meeting in a 3D–space provided by our platform. In addition, we describe technically a session of a user of GeDA-3D, including the typical actions he/she may perform. Finally, the operation of the client and some services are both modeled using Petri Nets formalism.

4.3 Community of agents

Multi-agent systems [SYCARA98] comprise intelligent elements independent to users called agents [BRENNER98, HYACINTH96], which have specific skills, perform specific tasks concurrently, are able to interact with each other, and react against environmental changes. The agent paradigm was used to design GeDA-3D in order to take advantage of these features. The agent's characteristics are used to help users manage cooperative distributed applications.

The GeDA-3D architecture can be described as a community of different sort of agents:

- Coordinator: in charge of the management of the various distributed systems integrated to the architecture and the end-users connected.
- Application: represent cooperative distributed applications integrated to the architecture.
- Client: act as an interface between the architecture and the end-user, in the form of an immersive virtual environment.

These agents are described more in detail in the next sections.

4.3.1 Coordinator

This agent is distributed and is constituted by different services. The services are bound through our transport layer later described. The architecture proposed for the coordinator allows more services to be easily added. The six services available in this version are:

1. *Look Up*: resolves the reference of all services and applications currently available in GeDA-3D

2. *Users*: manages the operations occurring in the users database and validates users logins
3. *Applications*: manages the operations occurring in the applications database and validates applications logins
4. *Consistency*: addresses and validates all the changes performed in the virtual environment
5. *Chat*: provides point to point messaging between users connected
6. *Security*: provides mechanisms to prevent GeDA-3D from intruders with knowledge of the services references and its public methods.

4.3.2 Application

One of the advantages of GeDA-3D arises from the fact that it allows external contributions, that is, anyone can develop applications from different nature which results in an extension of the services available through GeDA-3D.

Every application developed for GeDA-3D should follow some defined templates and include three main elements:

1. *Server*: binds the application to GeDA-3D and provides public methods
2. *Mobile interface*: stands for the user interface. It is cloned and sent to the client requester as many times as necessary. Once in execution, the Mobile Interface invokes Server public methods
3. *Host*: synchronizes with the client requester to send the mobile interface via a Socket.

4.3.3 Client

This Client has an important role in our platform. The Client provides the means for the user to be part of GeDA-3D and to take advantage of the services provided by the coordinator and the various developers.

The Client comprises three main elements:

1. *Interface*: provides a shared VE where users connected may navigate, interact with other users and launch distributed applications.
2. *Listener*: receives every change performed within the shared environment and messages from other users.

3. *Host*: synchronizes with the application provider to receive the mobile interface via a Socket.

The shared VE included in the Client Interface can be considered as a set of virtual objects arranged in a 3D-space. Some of them (walls, ceiling, floor, light bulbs...) are static and build the virtual world, in this case a 4-wall room; the rest of the objects are virtual representations of user or applications, and we call them *avatars*. Users can navigate within the 3D-space through their avatars. A user avatar has the ability to perform *motions* –move up, down, forward, backwards, left or right– and *rotations* –turn left or right a certain amount of degrees–.

4.4 Architecture

In order to find every service bounded to GeDA-3D, we take advantage of CORBA Naming Services. Since our platform works over a TCP/IP network, communication carried out between agents is point to point. As we can see in Figure 4.1, all the services provided by the coordinator have a number of replicas which depends on how critic the services are. In addition, both the Consistency Service and the Client Interface keep a database of the VE, including the state of the entities currently present.

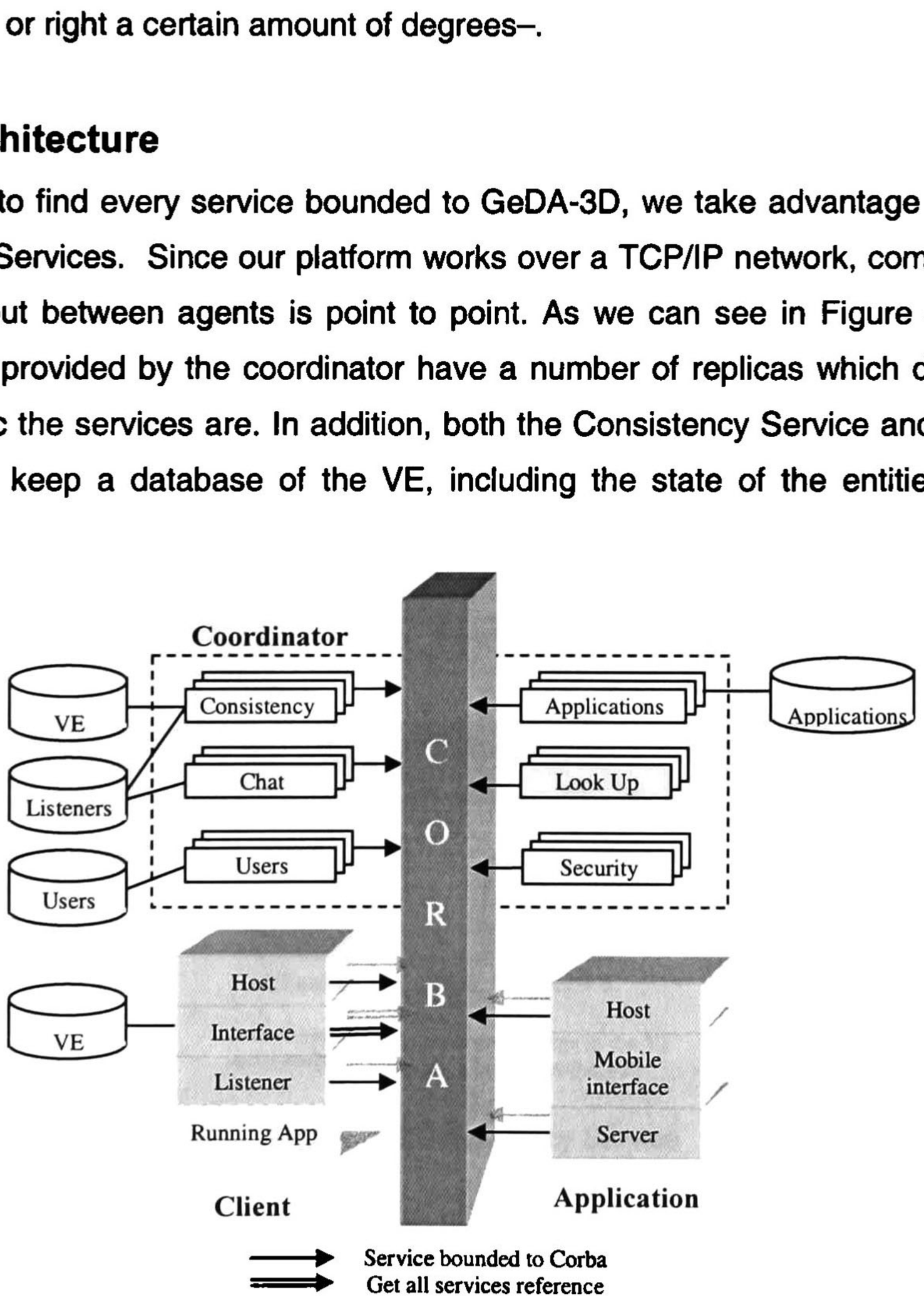


Figure 4.1 Architecture of GeDA-3D

The Chat Service, as well as the Consistency Service, includes a list storing the references of the Clients Listeners which makes possible to forward either changes performed in the VE or user messages to the users connected.

4.5 Consistency Service (CS)

As mentioned in 4.3.3, the Client provides a non-immersive virtual environment, that is, a 3D-space where all the users and applications connected to GeDA-3D meet and can interact with each other. Since this virtual space is common to all the users, every change performed by an avatar is immediately notified to the rest of the users. The changes include: appearances or new connections, motions, rotations and disappearances or logouts. When an avatar enters the virtual environment (VE), an *appearance* of an avatar is performed and it is assigned a tuple <position, angle>. The position is represented by a tuple (*x*, *y*, *z*) and gives the current location of the avatar. The angle takes a value between 0º and 360º and represents the field of view of the avatar (agent) representing the user. A displacement of an avatar in the VE results in a change of its 3D-coordinate, and it is called a *motion*; similarly, a change of direction of an avatar results in a change of its angle, called *rotation*. When an avatar leaves the VE, a *logout* is performed.

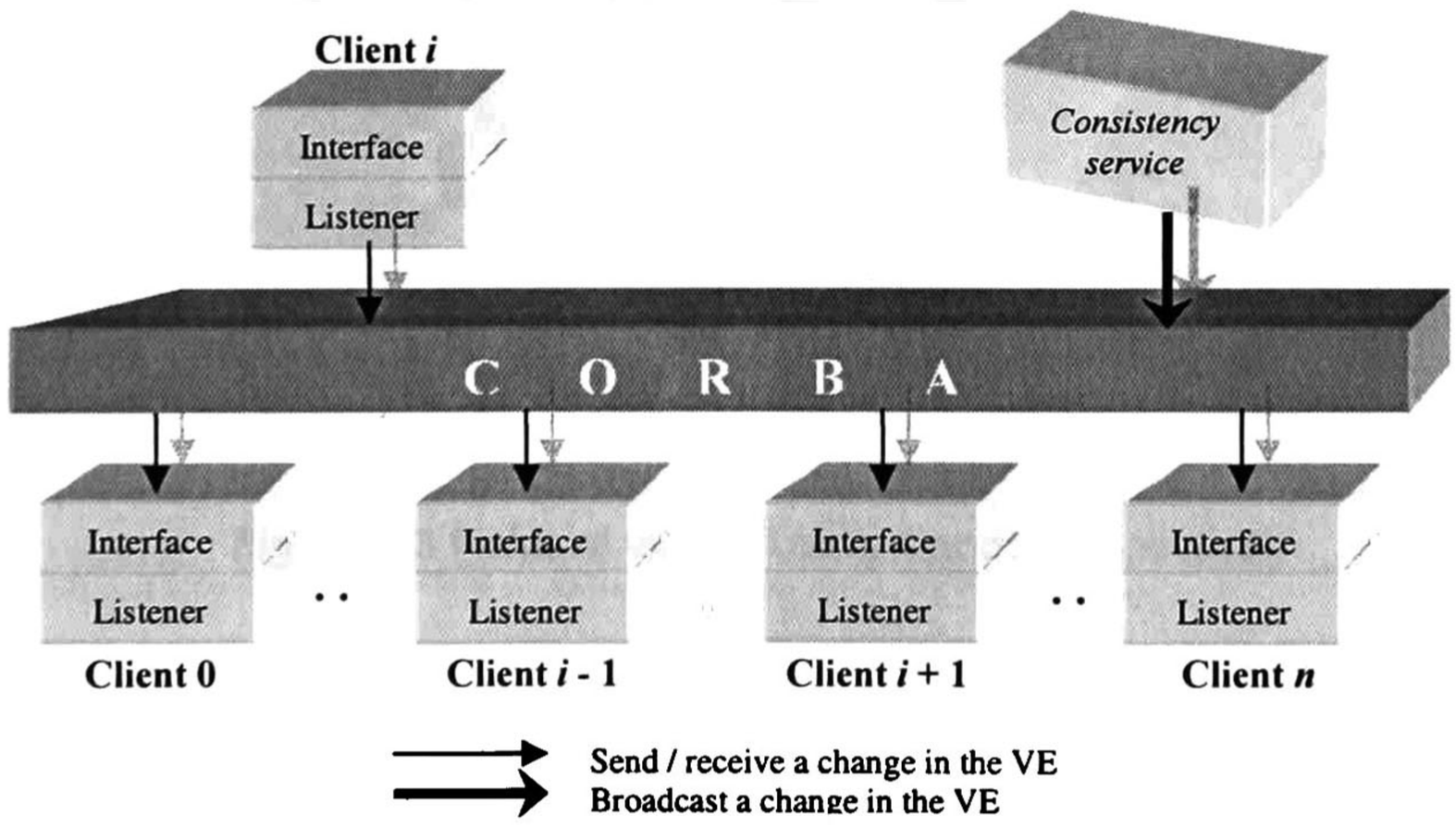


Figure 4.2 Consistency Service

As mentioned above, the main objective of the Consistency Service is to address all the changes performed by the avatars belonging to the VE. The operation of the CS is as follows. As soon as a client, for instance *C*, is both authorized to take part of

GeDA-3D and assigned a virtual layout (avatar), a reference from its listener is sent to the CS where all the listener references are stored. Immediately, the CS resolves a tuple <3D coordinate, angle> representing a virtual space free from collisions for the user avatar; after that, the CS indicates the appearance of this avatar to all the clients connected (except *C*) and sends back the environment to *C*. Whenever an avatar performs either a motion or a rotation, the new values are sent to the CS, where they are broadcasted to the rest of the clients. The general operation of the CS after changes occurred in the VE is depicted in Figure 4.2.

4.5.1 Example

Assume there are four users connected to GeDA-3D. Here, we represent the corresponding avatars of users 1, 2 and 3 by a hexagon, a diamond and a cube, respectively. The field of view of the four users in a certain moment of the navigation is shown in Figures 4.3 and 4.4.

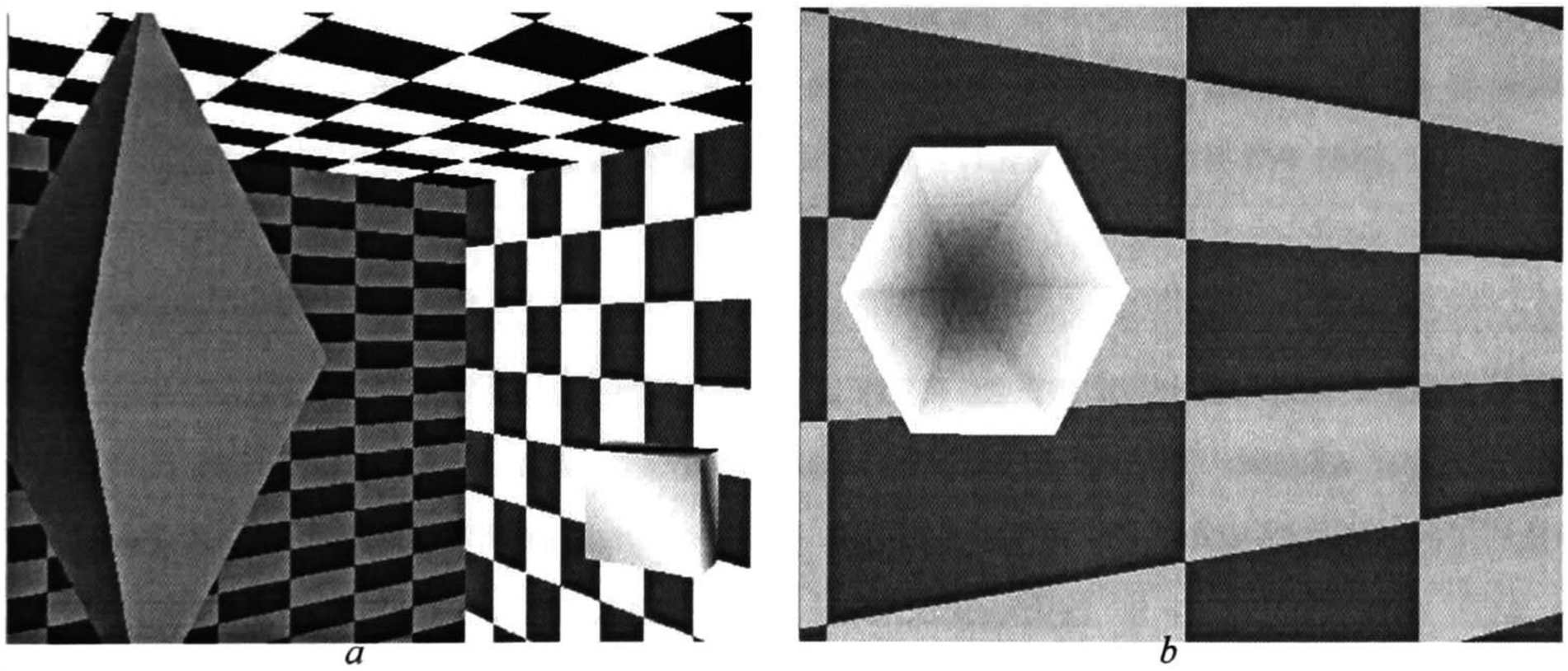


Figure 4.3 Fields of view of Users 1 and 2, respectively

At this very moment, a 4-wall room, a sphere and 4 avatars constitute the shared VE. All the users share the same VE, but it is shown to them according to their field of view, that is, the location and the angle of their avatars.

For instance, Figure 4.4a illustrates the field of view of user 3 (the cube). Its field of view reaches the ceiling partially, the lower-side of the diamond and the whole hexagon. This cube is not visible from user 2 (Figure 4.3b) because the cube is considered to be behind the diamond which is staring at the hexagon.

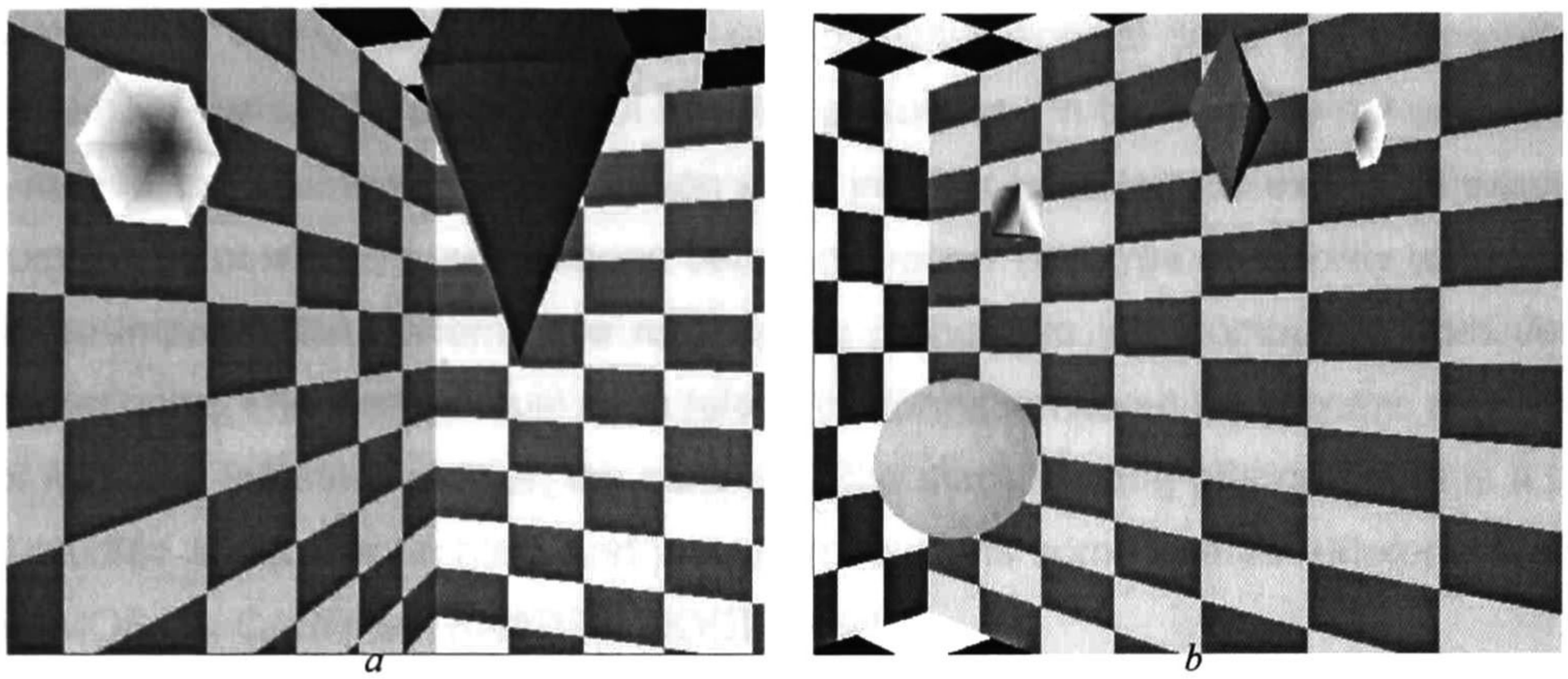


Figure 4.4 Fields of view of Users 3 and 4, respectively

In Figure 4.4b, the spatial arrangement of the avatars mentioned above is appreciated. This figure illustrates the view of user 4 staring at users 1, 2, and 3 from a far enough distance. Any change occurred in the VE and arisen in user 1 is evident for the rest of the users, since the hexagon is reachable from all users' fields of view. Conversely, the first change performed by user 4 is not evident for the rest.

4.6 Mobility Platform

The wide spread of interconnected networks, such as the Internet, has imposed the needs for new paradigms and technologies. In this scenario, mobile agents are proposed as a model for addressing the requirements of large-scale distributed applications. Ever since computers were interconnected, it emerged the idea of taking advantage of these connections, not only for messages exchange but for moving entities. In the early days, only single data were moved; but mobility has evolved in such a way that nowadays it allows moving code and execution control / environment.

The first stage of such evolution includes file mobility, for instance, the FTP protocol; after that, remote procedure calls (RPC) [BIRRELL:98] were introduced, where execution flow is moved. Later, the idea of moving code was proposed. Two types of migrations were considered upon the facilities of the operating system. The first one, called *soft mobility*, allows moving a process among two different sites. The

receiving site starts the execution of the received process. The second type of migration is called *hard* mobility; this one demands special skills to the operating system, in special the possibility of freezing a process. In this mobility not only code is moved, but also the whole execution state, in order to restart the execution exactly from the point where it was stopped before migration This type of mobility is used in DS to improve the performance reallocating process to idle computers from very loaded ones. The system must keep relevant information about the process migrated for instance the stack pointer, the content of the stack, among others. There is a lot of studies about this problem and proposed solutions some interest references are [RAMOS:98, CABRI:99, KNABE, FUGGETA:98].

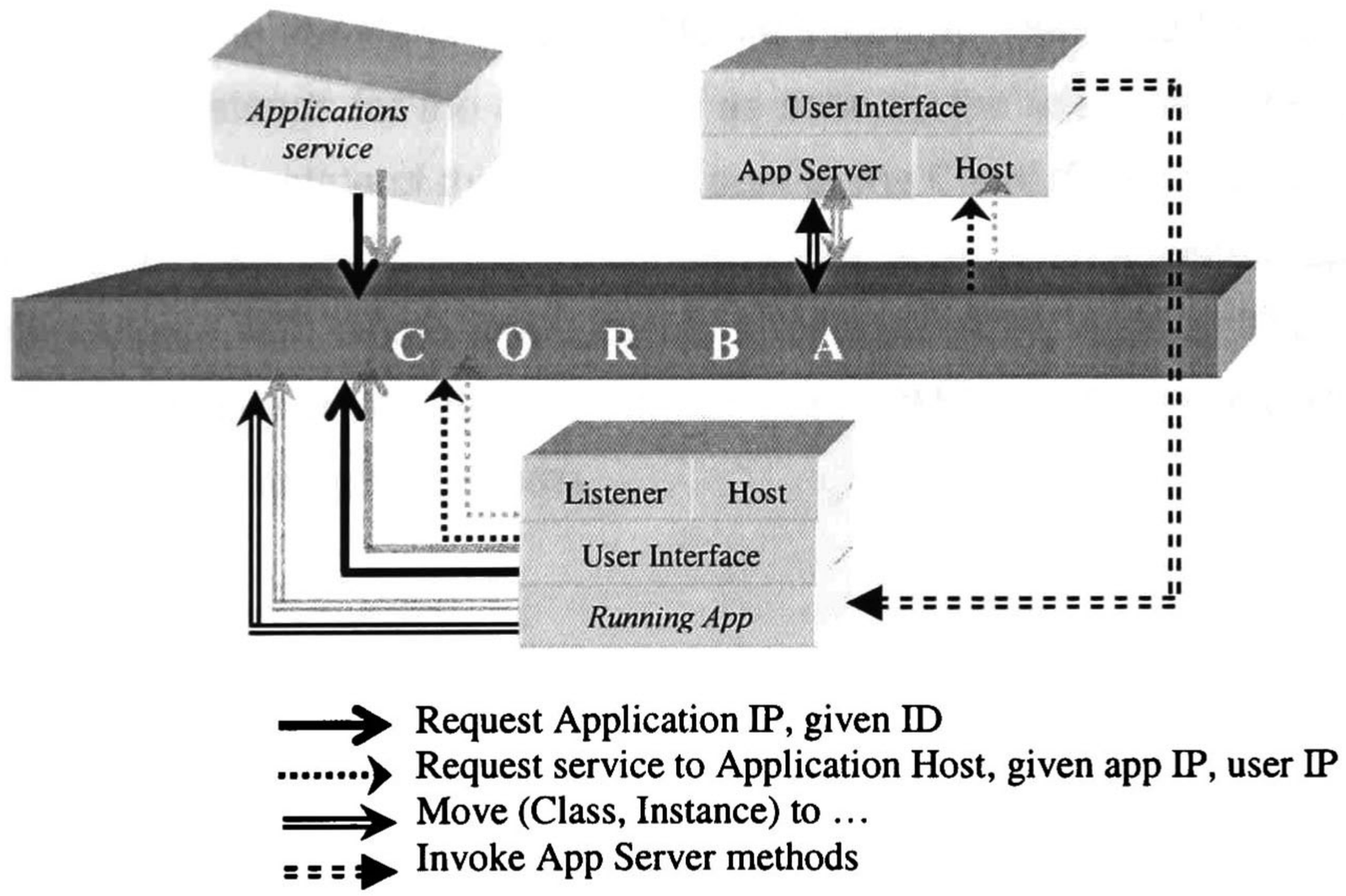


Figure 4.5 Mobility Platform

As mentioned above, GeDA-3D is constituted by a community of static (as opposed to mobile) agents, but also has the capability of managing mobile agents and it provides a platform where mobile programming is supported and performed with certain ease. GeDA-3D satisfies the underlying aim: allow the users to run locally remote applications. In this context, a mobile agent is not forced to remain in the system where it was started, but it has the ability to move from a system to another containing an object (agent, resource) to interact with [OMG:00].

Whenever an agent travels across the GeDA-3D network, both its state and source code are moved. The agent state includes some parameters determining which process resumes the agent execution as soon as the application arrives to the host requester. Figure 4.5 depicts the way GeDA-3D performs agent mobility, from the moment an application is requested by a user. Both Client and Application agents include a service called Host behaving as an agent system. As [OMG:00] establishes, an agent system refers to a platform enabling agent creation, interpretation, execution, transference and completion.

The Client requests the IP address of the Application selected; immediately later, the Client requests the Application's Host service to synchronize with its own Host service and start the transference of the Mobile Interface. Here, the Mobile Interface is the agent and stands for the application as seen by the user. After this, both the source code and the state of the agent are sent to the Client. Once this transference ends, the agent resumes its execution on the Client side. Such a system agent-host can communicate with others agent systems for transferring agents. The set of all the agent systems is called Region – taken from [OMG:00]. Figure 4.6 shows the interconnection of two agent systems.

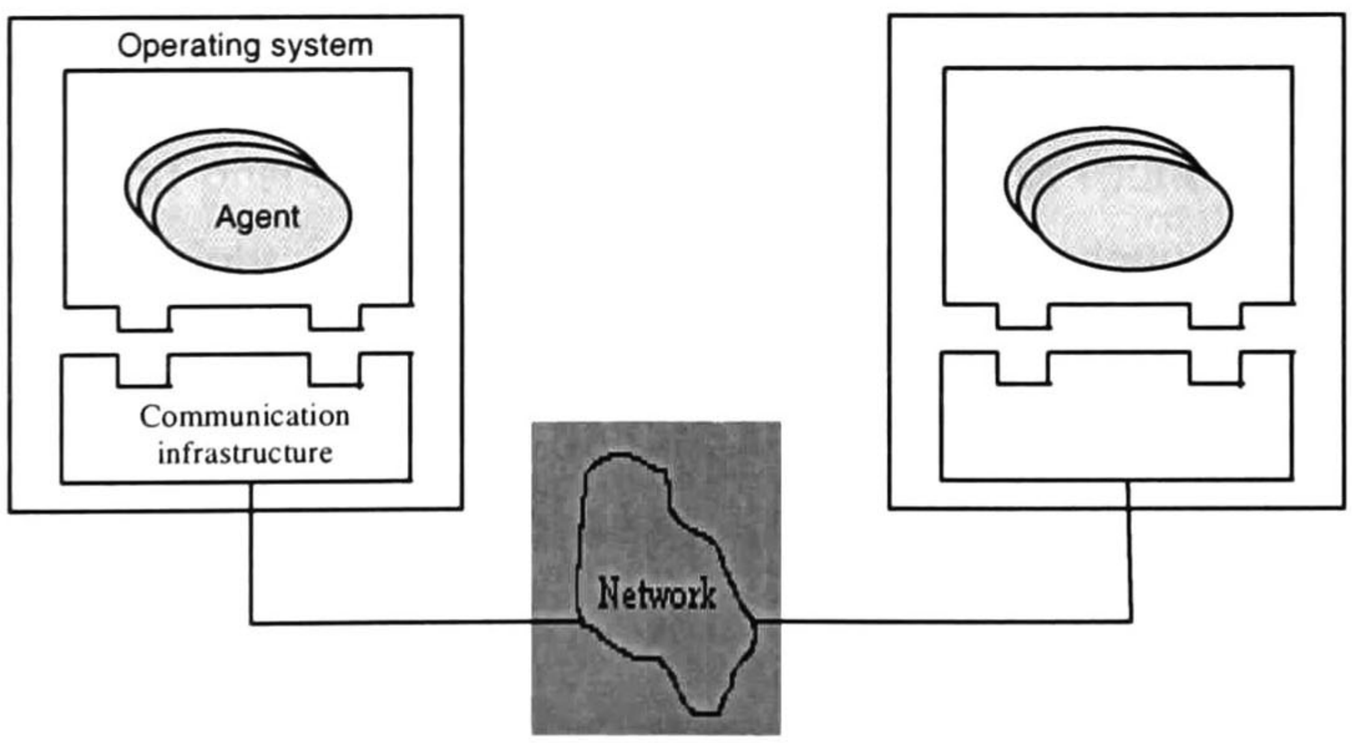


Figure 4.6 Agent system to agent system interconnection

A communications infrastructure provides communications transport services (e.g. CORBA), naming, and security services for an agent system. Serialization techniques are used before an agent is sent across the network, and deserialization when the agent is received from the network. The key to storing and retrieving agents is to represent the state of an agent in a serialized form that is sufficient to

reconstruct the agent. Notice that the serialized form must be able to identify and verify the classes from which the fields were saved. For not object-oriented agent systems, the agent state refers to the extraction of runtime data for the agent, and the classes refer to the code implementing the agent [OMG:00].

4.7 Implementation

This section introduces a series of structures, classes and interfaces designed for the development of our platform; in addition, a session in GeDA-3D is explained in a technical level, emphasizing some methods used to send/receive motions and rotations to/from the Consistency Service

4.7.1. Onthology

To make possible the development and deployment of 3D-shapes, our platform provides an onthology consisting of a series of classes useful for the development of virtual environments. Such onthology introduces the concepts of *entity*, *primitive* and *vertex,* described below. A 3D-shape based on this onthology is treated as an *entity* consisting of *primitives* and *vertices*.

An *entity* stands for a geometric body characterized for having three dimensions: width, height and depth. Examples of entities include: cylinders, spheres, cubes, cones, pyramids and prisms. Notice that some entities are rounded and others are made up of only straight edges. An entity includes a *type* value to determine if it represents a stra ight body or a rounded one. In the former case, the entity is constituted by one or more primitives. In GeDA-3D, a 3D-shape is constituted by one entity; in our VE Editor, more than one entity conform a 3D-shape typically.

A *primitive* is a uni- or bi-dimensional shape defined by a succession of points and lines connecting the points. Examples of a primitive include: lines, squares, triangles, line strips, circles and polygons in general. A primitive is constituted by one or more vertices. A primitive includes an identifier useful to determine whether and how the vertices are connected. A *vertex* is a point in the space, characterized by a 3D-coordinate (x, y, z) and a RGB color (red, green, blue).

4.7.2 Virtual Objects

As said in 4.3.3, the shared VE provided by GeDA-3D is constituted by a set of virtual objects, each representing a user, an application or a part of the room. A virtual object in GeDA-3D is represented graphically by a 3D-shape (or entity).

4.7.2.1 Attributes

The state of a virtual object in the shared room is defined by its location and dimensions of width, height and depth. The location of a virtual object within the shared room is defined by the current coordinate of its *mass center*. The 3D-space occupied by a virtual object is bounded by its dimensions forming the shape of a bucket. The dimensions of a virtual object are conformed by six values storing the smallest and largest values along X, Y and Z-axes of the object: minX, maxX, minY, maxY, minZ and maxZ. These values are illustrated in Figure 4.7.

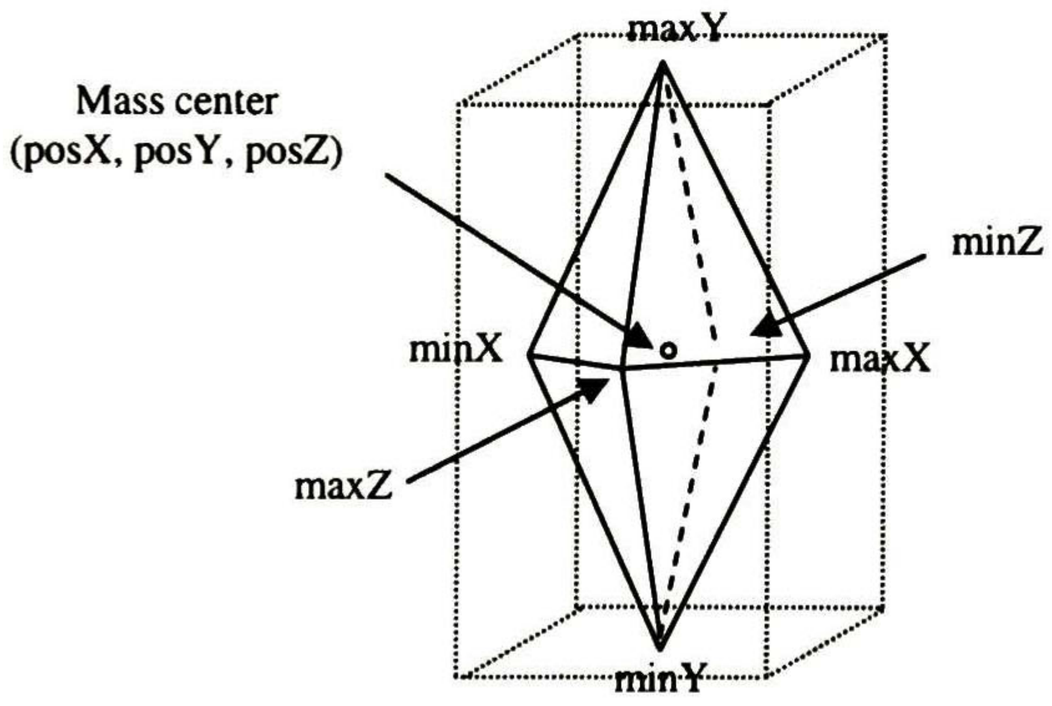


Figure 4.7 The 3D-space defined by an object in GeDA-3D

The initial dimensions of a virtual object are obtained from the dimensions of its 3D-shape; for every further rotation, new dimensions are calculated using a class called Dimensioner. The collision detector uses these values (coordinate, dimensions) to prevent all the objects from sharing the same 3D-space.

4.7.2.2 The Virtual Object Interface

GeDA-3D provides a library of 3D-shapes (diamonds, spheres, prisms, cubes...) useful to build virtual worlds and to be chosen as avatars. Every 3D-shape is assigned a unique numerical value. The development of a 3D-shape involves implementing the following methods defined by the Virtual Object Interface:

- set Coordinate. Change the current coordinate (posX, posY, posZ) of the object
- get Current Coordinate. Get the current value of (posX, posY, posZ)
- set Angle. Change the current angle value of the object
- get Dimensions. Get the original dimensions of the 3D-shape
- get Entity. Get the geometric body to be displayed on the screen

4.7.3. Description of classes employed

This section describes the structures and classes necessary for the client to send actions and receive updates from the Consistency Service.

Object Addition. This structure stores the information sufficient to display and identify a virtual object in the virtual room. It contains the following attributes:

- UserAppID. A numerical identifier assigned to the user or the application.
- Type. A character defining whether the virtual object takes part of the environment or it represents either a user or an application
- ID. The concatenation of the two attributes described above. This value identifies uniquely every virtual object in the room.
- EntityID. The numerical identifier of the 3D-shape employed
- Angle. A float value ranging from 0 to 360º. It represents the current visual direction of the virtual object
- PosX. A float value indicating the location of the object along the *x*-axis
- PosY. A float value indicating the location of the object along the *y*-axis
- PosZ. A float value indicating the location of the object along the *z*-axis
- Dimensions. This attribute stores the current dimensions of the object.

Object Motion. This structure stores the information necessary to send avatar motions from a client to the CS and vice versa.

- ID. The identifier of the avatar performing the motion
- PosX. A float value indicating the new location of the object along the *x*-axis
- PosY. A float value indicating the new location of the object along the *y*-axis
- PosZ. A float value indicating the new location of the object along the *z*-axis

Object Rotation. This structure stores the information necessary to send avatar rotations from a client to the CS and vice versa.

- ID. The identifier of the avatar performing the rotation
- Angle. A float value indicating the new angle
- Dimensions. The new dimensions

Society. An array of Object Additions.

Object Action. This structure stores the action performed by an avatar. Such an action includes motions and rotations.

- ID. The identifier of the avatar performing the action
- ActionID. The identifier of the kind of action performed:
    - 'M' : Motion
    - 'R' : Rotation
- Angle. Stores the new angle of the object after a rotation
- PosX. The new location of the object along the *x*-axis after a motion
- PosY. The new location of the object along the *y*-axis after a motion
- PosZ. The new location of the object along the *z*-axis after a motion
- Dimensions. The new dimensions of the object after a rotation

Actions Buffer. This class includes a Vector storing the actions to be sent to the Consistency Service. The Actions Buffer publishes the following methods enabling the client to perform operations on the Vector:

- void **addMotion** (Object Motion). Receives an Object Motion from the client, converts it to an Object Addition, and adds it to the Vector
- void **addRotation** (Object Rotation). Receives an Object Rotation from the client, converts it to an Object Addition, and adds it to the Vector
- boolean **hasActions** ( ). Returns *true* if the Vector has Object Additions
- Object Action **getAction** ( ). Gets the first (oldest) Object Action from the Vector and deletes it
- void **emptyBuffer** ( ). Remove all the elements from the Vector

Action Sender. This class is an agent that keeps on reading the Actions Buffer. If the Buffer is not empty, then the Action Sender gets the first Object Action and sends it to the Consistency Service as an Object Motion or an Object Rotation:

```
Object Action OA = Actions Buffer.getAction ( );
If (OA.actionID == 'M')
   Create a new Object Motion OM using some attributes of the OA
   ConsServ.moveUserObject (Service Name, OM);
If (OA.actionID == 'R')
   Create a new Object Rotation OR using some attributes of the OA
   ConsServ.rotateUserObject (Service Name, OR);
```

Dimensioner. This class is useful to recalculate the dimensions of an object given the current angle. As said before, a virtual object has a number of attributes including coordinates, angle and dimensions. In addition, a virtual object is assigned a 3D-shape. All the 3D-shapes are initially 0° rotated and have static dimensions. Such dimensions are available to the developer because the 3D-shapes implement the Virtual Object Interface (see 4.7.2.2). When a user selects a 3D-shape to become his/her avatar, the Client creates a new Dimensioner class taking as a parameter the dimensions of the 3D-shape selected:

```
Dimensioner D = new Dimensioner (myAvatar.Dimensions)
```

After every rotation, the dimensions of the virtual object (avatar) changes and they are obtained using the following method of the Dimensioner class:

- `Dimensions getMyDimensions (angle)`

Section 5.5.1.4 depicts an example of new dimensions obtained after a rotation.

4.7.4. A session in GeDA-3D

After introducing the structures and classes necessary to send/receive actions to/from the CS, now we proceed to describe technically a session in GeDA-3D including the methods of the CS and the Listeners employed. Such a session starts with a user login, includes motions, rotations, updating, and ends with a user logout.

4.7.4.1 Login

A user introduces his/her account number and password. If the data are valid then the GeDA-3D client launches a service called *Listener*. This service enables a user to receive the changes performed within the virtual room from the Consistency Service.

The user chooses a 3D-shape from a list to become his/her avatar. The dimensions of the object chosen are obtained and assigned to the initial dimensions of the avatar. An Object Addition (OA) is then created with the dimensions and ID of the avatar chosen, and the ID of the user. Both the OA and the name of the service are sent to the Consistency Service (CS) where the OA is assigned a pair (coordinates, angle) free from collisions:

```
Object Addition = ConsServ.addUserObject (Service, Object Addition)
```

The resulting OA is broadcasted to all the users connected:

```
Listener.addObject (Object Addition)
```

The OA sent is added to the database of every client

The client requests the CS a database storing the Object Additions currently living in the virtual room:

```
Society = ConsServ.getSociety (User ID);
```

NOTE: User ID tells the CS not to send the Object Addition with ID equals to the requester's

The virtual room is finally displayed for the user. There, the user may perform different actions, including motions, rotations and logouts. In addition, he/she is aware of the actions performed by other users at the moment.

4.7.4.2 Motion

A temporary coordinate (posX1, posY1, posZ1) is calculated according to the kind of motion carried out, as follows:

Move up	posY1	=	posY	+	step;
Move down	posY1	=	posY	–	step;
Move left	posX1	=	posX	–	step x sin (angle + 90º)
	posZ1	=	posZ	+	step x cos (angle + 90º)

Move right	posX1	=	posX	+	step x sin (angle + 90º)
	posZ1	=	posZ	−	step x cos (angle + 90º)
Move forward	posX1	=	posX	+	step x sin (angle)
	posZ1	=	posZ	−	step x cos (angle)
Move backwards	posX1	=	posX	−	step x sin (angle)
	posZ1	=	posZ	+	step x cos (angle)

where,

step is the amount of units to advance in one motion

angle indicates the direction of the motion

The motion is then checked for collisions: verify if the new space occupied by the object overlaps with the space occupied by any other object in the virtual room (see Section 5.5.1.4). If not, the motion is considered valid and the temporary coordinate becomes the actual coordinate (posX, posY, posZ).

A new Object Motion is created with the new coordinates and added to the Actions Buffer. Finally, the room is re-displayed.

4.7.4.3 Rotation

A temporary angle is calculated according to the kind of rotation carried out:

Rotate left	angle1	=	angle	−	angle step
Rotate right	angle1	=	angle	+	angle step

where,

angle step is the amount of degrees to turn in one rotation

Temporary dimensions are calculated, given the angle obtained:

```
myDimensions1 = Dimensioner.getMyDimensions (angle1);
```

The rotation is then checked for collisions. If no collisions detected, the motion is considered valid, the temporary angle becomes the actual angle and the temporary dimensions become the actual dimensions.

A new Object Rotation is created with the new angle and dimensions, and added to the Actions Buffer. Finally, the room is re-displayed.

4.7.4.4 Sending actions

Periodically, the Action Sender reads the content of the Actions Buffer. For every item read, the Action Sender removes it from the list and invokes the method:

```
MoveUserObject (Service Name, Object Motion)
```

if the item read is an Object Motion, and the method:

```
RotateUserObject (Service Name, Object Rotation)
```

if the item read is an Object Rotation, and the method:

NOTE: Service Name tells the CS not to broadcast the Motion / Rotation to the requester.

The CS then replaces the current values of the object with the new ones received and broadcasts them to the rest of the users connected, using either

```
Listener.moveObject (Object Motion)
```

or:

```
Listener.rotateObject (Object Rotation)
```

4.7.4.5 Log out

The Object Addition database is emptied. The client requests the Consistency Service to remove the Object Addition:

```
ConsServ.removeUserObject (User ID);
```

The CS broadcasts all the other clients a request to remove the Object Addition:

```
Listener.removeObject (User ID);
```

NOTE: User ID tells the CS (or Listeners) which Object Addition is to be removed.

The Client sends the Users Service a request to be logged out as a user:

```
Users.disconnect (user ID);
```

Finally, the Listener service is destroyed.

4.8 Formal Modeling

To model the behavior of both the Client and the Coordinator, from the moment a login is performed till the user starts navigation and sends messages, the Elementary Object System (EOS) methodology is used [VALK]. The System Net represents the Coordinator Services and the Object Net stands for the Client.

Our formalism proves that given a finite number of clients and applications connected the whole operation of GeDA-3D ends successfully, which means that the

number of reachable states of the system is finite and it never reaches a state where any single task no longer operates.

Here, the whole operation of a user is modeled at the moment his/her connection to GeDA-3D is requested. This way, two Petri Nets are included, one representing the Client's behavior, and the other, the coordinator services. Since the coordinator manages all the clients connected, the former is modeled as the Object Net of the EOS, and the latter, stands for the System Net where the Client travels through.

As a graphical convention, thick places/arcs are said to store/transport Object Nets, and narrow places/arcs, are said to store/transport ordinary tokens. For a better understanding of the Petri Net depicted in Figure 4.8, the transitions labels are described in order of appearance.

	=	a user attempts to Login to the Users Service
<nv>	=	user information is Not Valid (incorrect password, user ID inexistent or user already connected)
<v>	=	user information is Valid
<sa>	=	Send an Addition request to the CS to perform an appearance in the virtual shared room
<as>	=	the Addition request is successfully Sent to the CS
<gr>	=	get the current state of the shared Room

If the grey–shaded place (q_5) is marked, then the user is using the shared virtual environment provided by GeDA-3D. In this moment, the user may perform the following operations concurrently:

mr	=	start a Motion or a Rotation, one at a time (press key)
<su>	=	Send Update to the CS: the result of the motion/rotation
<us>	=	the Update is successfully Sent to the CS
mr'	=	stop the current Motion/Rotation (release key)
<u>	=	update the VE after a change performed by another user
<a>	=	Add a new user connected
<r>	=	Remove a user from the list

sc	=	start a conversation with a user (a chat window is created)
ec	=	end a conversation (a chat window is closed)
<sm>	=	Send a plain – text Message to a user connected
<rm>	=	Receive a Message from a user connected
<ms>	=	the Message is successfully Sent
<se>	=	Sending Error (the target user logged out)
<q>	=	Quit the program (close window)
<sr>	=	Send a Removal request to the CS to perform a logout in the virtual shared room
<rm>	=	the Removal request is successfully Sent to the CS
<lo>	=	Log out from the Users Service

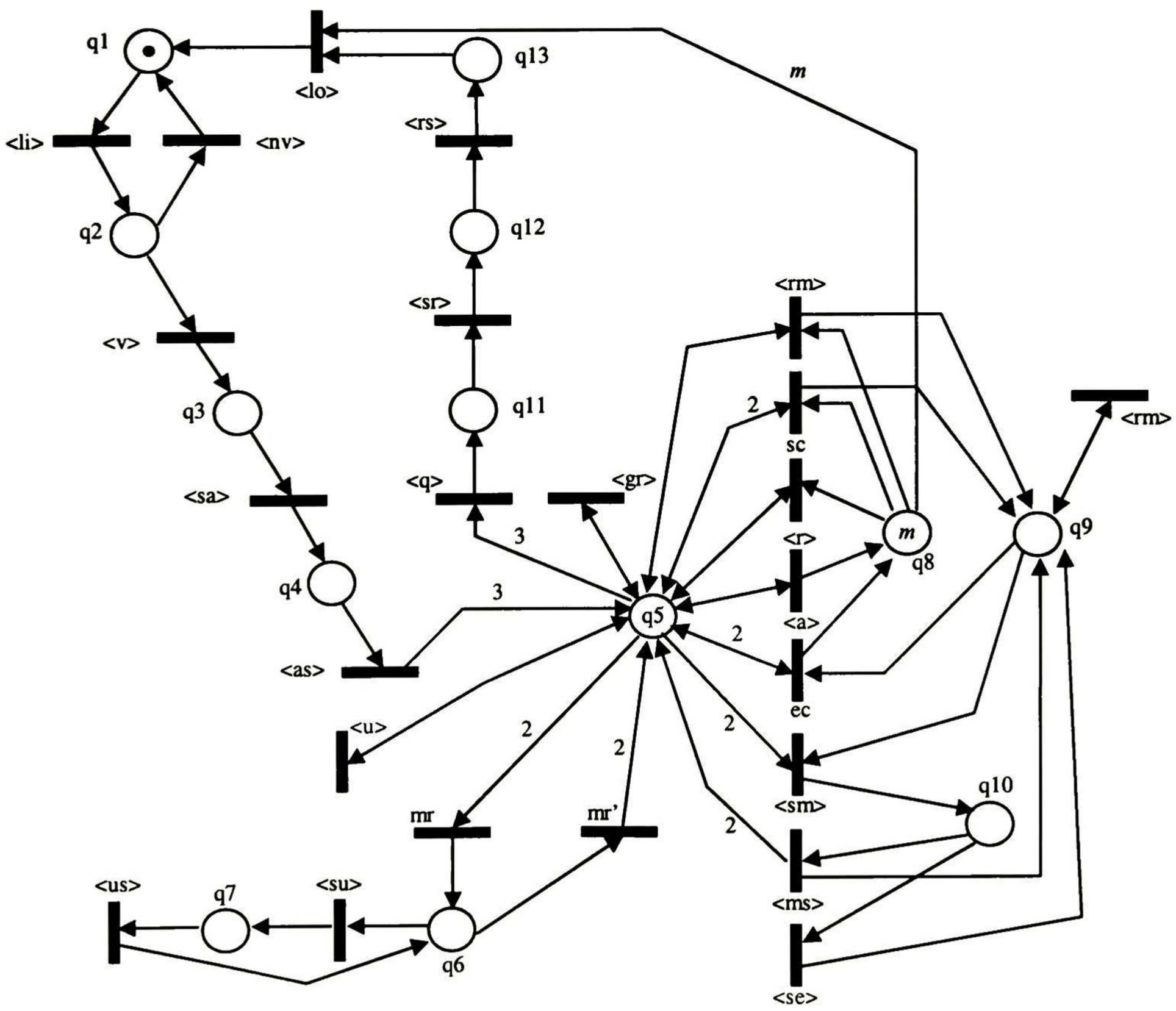


Figure 4.8 The Client

Every labeled transition in the Coordinator model synchronizes with some transition in the Client model. Place p1 stores the clients not connected to GeDA-3D. Initially,

the marking of p1 is equal to a finite natural number n. Place p8 contains the clients currently connected to GeDA-3D. Places p2, p9 and p16 represent a permission to use the Users Service, Consistency Service and Chat Service, respectively. They are initially marked since these services are available when nobody is connected.

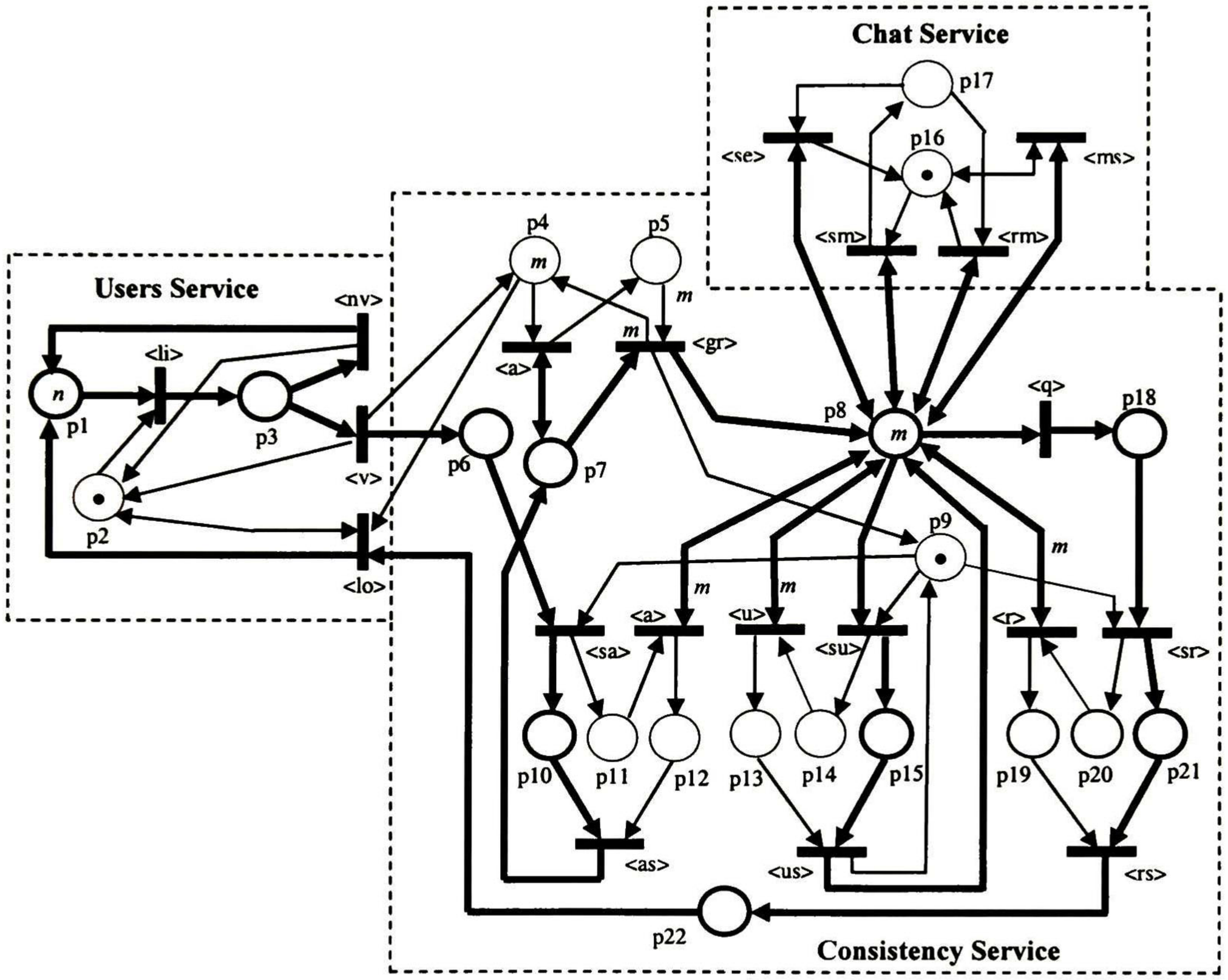


Figure 4.9 The Coordinator

The kth firing of a Petri Net generates the (k+1)th marking. In Adaptive Petri Nets, such firing also generates the (k+1)th structure of the Net. That is, the marking in a certain moment of the evolution define the weight and existence of the arcs at that moment. Always before a firing is performed, the structure of the Net is already defined.

Both System Net and Object Net behave as Adaptive Petri Nets, since the weight and existence of some arcs vary with respect to the marking of some adjacent

places. For instance, in Figure 3.1, the weight of the arc connecting place q8 with transition labeled <lo> is equal to the marking of q8.

4.9 Conclusions

In this chapter, we introduced the architecture of a platform supporting the development of VE editors, which we call GeDA–3D. Such architecture is an enhanced version of the model architecture described in [TOSCANO:00].

GeDA–3D provides a 3D–space where users (represented by avatars) may meet, interact with each other and launch shared applications from different nature; here, we have analyzed deeply the way our agent–based platform manages the changes occurred in such 3D–space with the help of a Consistency Service, and also the technique employed to allow the mobility of remote applications. In addition, we have modeled formally both the behavior of a client from the moment of his/her login, and the operation of the CS after a user performs any action affecting the 3D–space. The Mobility Service is modeled formally using an extension of Petri Nets and described in the work of my co–worker.

Our platform provides facilities for the development and sharing of VE editors since it makes available a useful template that enables developers to share applications of their own across GeDA-3D network, and also a series of services and libraries for the use of developers, including: the Consistency Service described in this chapter, a collision detector, an object rotator, a method that determines the order of displaying objects, an onthology consisting of a series of classes useful for the development of virtual environments, and a library of basic 3D-shapes -based on such onthology- useful to design complex virtual objects,

Chapter 5

VE Editors for GeDA-3D

5.1 Objectives

To accomplish the objective to introduce the Virtual Editor for GeDA-3D, the presentation was split in two parts: a) identify the main components of a VE Editor designed to work for GeDA-3D; b) introduce some issues to be considered during the design and the implementation of such editor.

5.2 Introduction

One of the main services provided by GeDA-3D is a VEE (Virtual Environments Editor). This module is the interface that our platform offers to the user and/or creator of any sort of VE.

This chapter is devoted to describing the main requirements to develop our VE editor for GeDA-3D. Unlike Chapter 3, some important issues are considered for designing different kinds of Virtual Environments. First, this chapter starts by describing briefly how our VE editor for GeDA-3D is conceived. Then, the elements constituting our VE editor are presented and explained. After that, some issues to be addressed in the development of VE editors for GeDA-3D are described.

This chapter introduces a number of basic concepts related to VR that must be considered among the requirements for VEE. Some of them are already handled by our platform; unfortunately, some others, because of time, are intended to be addressed by our tool in the future.

5.3 Conception of a VE Editor for GeDA-3D

The VEE for GeDA-3D should allow any inexperienced user to create or modify VE by adding static and/or dynamic VO to a specific virtual world, using his/her own 3D-application. The process of creating or modifying a VE consists in dragging VOs from a menu, and dropping it inside the area displaying the virtual world. This "drag & drop" process implies the use of peripheral devices such as the mouse and the keyboard.

Every dynamic VO is supposed to have its own behavior. Thus, once the creator finishes his/her creation (all the desired objects are included in a virtual world) the user (creator) can then run a simulation of such a world which means than all the VO become alive interacting with the other elements present in the VE created. If the virtual world chosen represents a sort of ecosystem, then it will evolve alone; otherwise, it will evolve according to the actions taken by the users (for instance a Virtual Office).

Because of the management service of GeDA-3D, virtual objects can be shared across all the components of the distributed system.

5.3.1 The interface proposed for the VE Editor

Since the goal is to allow any inexperienced user to use our system, the VEE is presented to the user like any editor having:

- *A rectangular space* –initially blank– where a VE is to be created.
- *A list box*, for a user to choose a virtual world (see section 5.4). Once a virtual world is chosen, it is drawn on the rectangular space
- *A toolbar* used to pick a VO and drop it on the space where the virtual world is displayed
- *Navigation buttons* that allow the observer to walk forward & backwards, move left & right, rise, descend and rotate left & right.
- Useful *control buttons* to start, pause and stop the simulation

An approximation to such interface is depicted in Figure 5.1

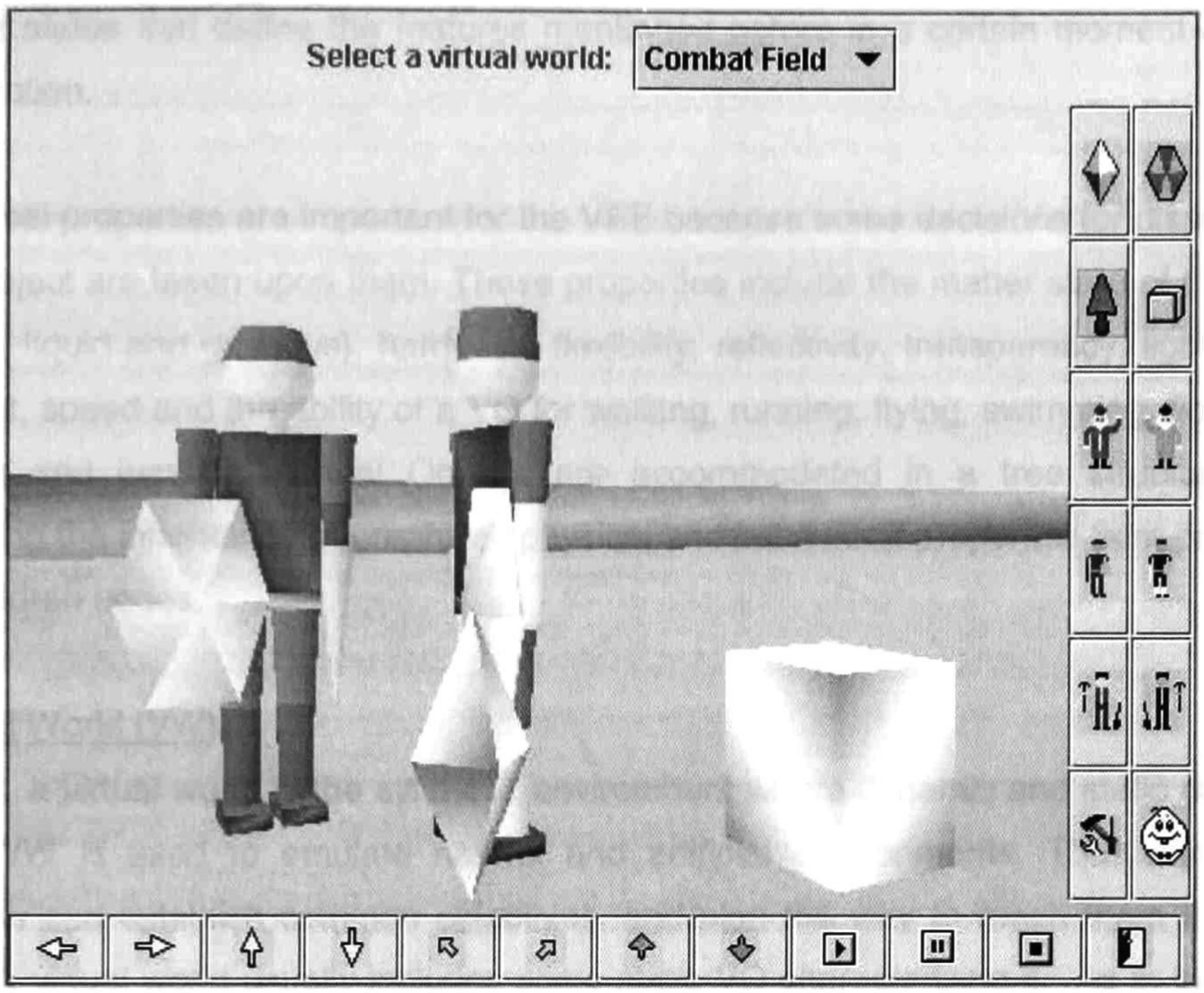


Figure 5.1 The VE Editor Interface

5.4 Concepts concerning the VE Editor for GeDA-3D

This section introduces some of the concepts which must be taken into account in the design of the VEE for GeDA-3D:

Virtual Object (VO)

A VO is a 3D entity that lives in a virtual world. Two types of virtual objects can be identified:

a) *Static VO:* constituted by a *graphical representation* and physical properties
b) *Dynamic VO*: include also a *behavior* that enables them to interact with each other and modify their environment.

The behavior of a dynamic VO refers to the actions it may perform. These actions are subordinated to the physical properties inherent in a VO and also to the laws ruling the VE. When we talk about the physical representation of a VO we include not only its shape (or shapes), orientation, texture, color and dimension, but also a

set of states that define the features mentioned before in a certain moment of the interaction.

Physical properties are important for the VEE because some decisions for displaying the object are taken upon them. These properties include the matter state of the VO (solid, liquid and gaseous), hardness, flexibility, reflectivity, transparency, lightness, weight, speed and the ability of a VO for walking, running, flying, swimming, floating, diving and jumping. Virtual Objects are accommodated in a tree structure for allowing the inheritance of graphical, physical and behavioral properties from parents to children nodes.

Virtual World (VW)

In VR, a virtual world is the synthetic environment where dynamic and static objects live. VW is used to emulate natural and artificial environments. Thus, dynamic objects can establish common objectives, and plan the way to reach them in their VW. A virtual world usually includes some static VO characterizing it. As in the real world, a VW imposes a set of physical laws ruling the environment. Such laws include: lighting, gravity, coloring, living media (air, water, vacuum ...), inertia, pressure and humidity. In our case of designing a VE, these laws are important because they determine the way the environment is displayed, and govern the addition and interaction of virtual objects.

Observer or Creator

The observer refers to the user who builds and navigates within the virtual world. A VE editor has to address the way an observer performs additions and selections of objects in a 3D space by clicking on a 2D media. In our case, the VE editor should support user navigation, which determines the location and orientation of the virtual world to be displayed.

Validating service

This Validating service is in charge of verifying if all the changes that occur within the virtual world are considered valid; if so, that service also displays them properly through the output device. This service has to understand the results provided by actions included in the behavior of dynamic virtual objects.

Semantics service

This service interprets both the physical laws of the virtual world and the physical properties of a virtual object, then executes a pattern-matching algorithm in order to determine whether a virtual object is allowed or not to be part of a specific world. For instance, a whale is not allowed to be in a living room unless it is a toy whale.

Reorganizer

Typically, the state of a VO is subordinated to the state of another. For instance, if a virtual human is riding a horse, then the current position of the human depends closely on that of the horse. However, a human might change his activities after getting off the horse. This means that the relation among the horse and human must be changed. To keep track of this dynamic of relations, we propose a dynamic tree structure. This structure could store the current relationship and dependencies among the objects, and is capable of evolving with the changes in the VE.

5.5 Issues to be considered in the development of a VE Editor for GeDA-3D

This section is devoted to fulfilling the second goal of this chapter, that is, to describe and classify a number of common problems encountered in the development of our VE Editor for GeDA-3D. Some of them are closely related to the design of virtual worlds, others have to do with virtual objects, while some are related to the user interface and the rest are specific to the VE Editor.

5.5.1 VE Editor

The design of a VE Editor for GeDA-3D should consider some services capable to solving the following problems:

5.5.1.1 Order of objects display

In most of the existing VR–oriented languages and tools, the display of the 3D-objects in a virtual space is performed sequentially, which means that an entity positioned at the end of the display list is always drawn in front of all the others, no matter if it is supposed to be far away from the observer. To handle this, it is convenient to calculate the Euclidian distance from the observer to every entity. The

Euclidian distance stands for the shortest distance between two objects in the space and is obtained by calculating the magnitude of a vector that starts at the position of the observer and leads to the coordinates of an entity's mass center. Once all the distances are obtained, we sort the objects in a descendant way according to their Euclidian distance value with respect to the observer. This process is repeated every time the observer changes either position or orientation.

5.5.1.2 Removal of non visible objects

The position and orientation of an observer determines the portion of the virtual world to be displayed at once. In most cases, some objects are not visible in a certain instant of the navigation. Hence, the display of such objects can be ignored resulting in better performance of the VE Editor. An object is built from a set of polygons and primitives; sometimes a primitive is located in front of the viewer, but sometimes it is hidden behind or in the middle of the object. In the latter case, the primitive falls outside of the observer's field of view, and does not need to be drawn (see example described in 4.5.1). In OpenGL, the elimination of graphics primitives that would not be seen if rendered is called *culling*. *Backface culling* removes the front or back face of a primitive so that it isn't drawn. *Frustum culling* eliminates whole objects that would fall outside the observer's view.

5.5.1.3 Constant updating

Whenever a dynamic virtual object requests an action that modifies partially or totally the viewing of the environment, such change has to be displayed immediately after it is validated and performed. A VE editor should include a service that listens to all the actions –including user navigation– and constantly repaints the virtual world, providing the illusion of animation.

5.5.1.4 Resolution of collisions

In real life, solid objects are not able to share the same physical space. Such natural law has to be considered in the design of Virtual Environments emulating the reality. Thus, the platform considers a service that validates the motion or rotation of all the objects in the virtual world –including the observer– in order not to allow them to share the same 3D space. This is how this service works in our platform.

If the new coordinates and dimensions acquired after a motion or rotation of an object *ob1* fulfill the six following conditions for another object *ob2* within the virtual room:

```
ob1.PosX + ob1.MaxX >= ob2.posX - ob2.minX
ob1.PosX - ob1.MinX <= ob2.posX + ob2.maxX
ob1.PosY + ob1.MaxY >= ob2.posY - ob2.minY
ob1.PosY - ob1.MinY <= ob2.posY + ob2.maxY
ob1.PosZ + ob1.MaxZ >= ob2.posZ - ob2.minZ
ob1.PosZ - ob1.MinZ <= ob2.posZ + ob2.maxZ
```

Then *ob1* and *ob2* overlap. In other words, they share the same 3D – space resulting in a collision; whereby, such action is not considered valid and the new values should not be submitted. This comparison should be performed whenever an object intends to move or rotate.

The current coordinate of the *mass center* only changes after a valid motion. Since the rotations in GeDA-3D are performed along the y – axis, the values of minY and maxY never change after a rotation, unlike the values of minX, maxX, minZ and maxZ.

To obtain the new dimensions after an object rotates, the *y*-axis is ignored, as if we were staring at the object from the ceiling of the virtual room (see Figure 5.2). Each dimension is characterized for having a radius – the distance between the dimension coordinates and the mass center – and an initial angle with respect to the *x* – axis. The value of the angle changes after every rotation and it is calculated using trigonometry (sinus and cosines).

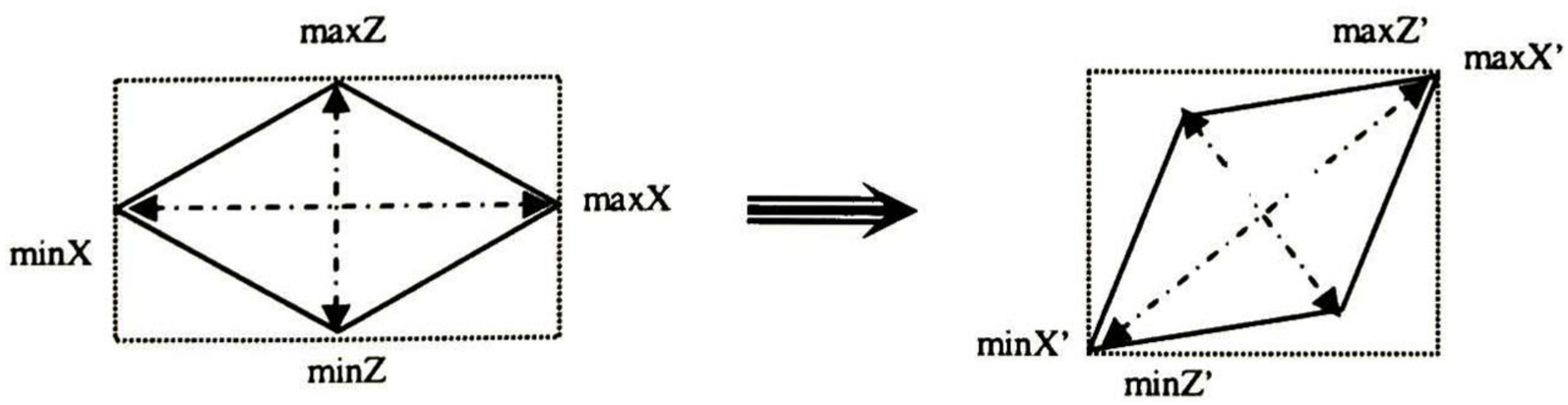


Figure 5.2 An example of new dimensions acquired after a rotation

5.5.1.5 Congruency

One of the aims is that the system be able to simulate the behavior of real environments ranging from ecosystems to virtual offices. Hence, the nature of the environment and the objects our tool intends to manage is quite variable. The nature of a virtual object defines the kind of world(s) it can belong to, and is determined through its physical properties and capabilities. For these reasons, a VE Editor should include a congruency service determining whether a VO is allowed to be part of the in-construction virtual world. This objective is reached with the help of semantics used to interpret the nature of the VO and compare it against the physical laws of the world.

5.5.2 Virtual World

As we describe previously in this chapter, a VW is the synthetic world where VO live, and it is constructed by one or several users. The design of an editable Virtual World should consider the following issues and their solutions:

5.5.2.1 Lighting source

This issue is useful for adding realism to our scenes. Most scenes in the real world are illuminated by a white light containing an even mixture of all the colors. Under white light, therefore, most objects appear in their proper or natural colors. Nevertheless, this is not true in all environments.

Real objects don't appear in a solid or shaded color based solely on their RGB (Red, Green and Blue) values. Unless an object emits its own light, it is illuminated by three different kinds of light: *ambient*, *diffuse*, and *specular* [WRIGHT:00]. Ambient light is the one that doesn't come from any particular direction. It has a source, but the rays of light have bounced around the scene and become directionless. Objects illuminated by ambient light are evenly lit on all surfaces in all directions. Diffuse light comes from a particular direction but is reflected evenly off a surface. Even though the light is reflected evenly, the object surface is brighter if the light is pointed directly at the surface than if the light grazes the surface from an angle. Specular light is directional too, but it is reflected sharply and in a particular direction. A highly specular light tends to cause a bright spot on the surface it shines upon, which is called the *specular highlight*.

No single light source is composed entirely of any of the three types of light just described. Rather, it is made up of varying intensities of each. Thus, a light source in a scene is said to be composed of three lighting components: ambient, diffuse, and specular. Just like the components of a color, each lighting component is defined with an RGBA value that describes the relative intensities of red, green, and blue light that make up that component.

5.5.2.2 Physical laws

Every virtual world is characterized by a set of physical laws which determine what kinds of objects are allowed to be part of the world, according to the properties and capabilities of such objects. Physical laws include issues such as: lighting intensity, gravity, humidity, pressure, living media and dimensions.

5.5.2.3 Dynamic group management

Since a virtual world is made up of both static and dynamic virtual objects, one of its competencies is to manage the relationship and dependencies among them, thus the virtual world should include a tree structure. Every time an object is included in the world, eliminated from it or changed in its dependency from one object to another, the tree structure has to be modified properly. In order to keep up–to–date such structure, a virtual world should include a reorganizer (see Section 5.4) too. Our tool intends to assist the developers of virtual worlds in supporting this dynamic group management.

5.5.2.4 Projection

When designing virtual worlds, it is very common to use perspective projections, rather than orthographic projections. In orthographic projections, all objects that have the same dimensions appear the same size, regardless of whether they are far away or nearby [WRIGHT:00]. This type of projection is most often used in architectural design, CAD (computer aided design), or 2D graphs.

The perspective projection adds the effect that distant objects appear smaller than nearby objects. Objects nearer to the front of the viewing volume appear close to their original size, but object near the back of the volume shrink as they are projected to the front of the volume. This type of projection gives the most realism for

simulation and 3D animation. The width of the back of the viewing volume does not have the same measurement as the front of the viewing volume. Thus, an object of the same logical dimensions appears larger at the front of the viewing volume than if it were drawn at the back of the viewing volume (see Figure 5.3). A *frustum* is a pyramid–shaped (from the narrow end to the broad end) viewing volume that creates a perspective view [WRIGHT:00].

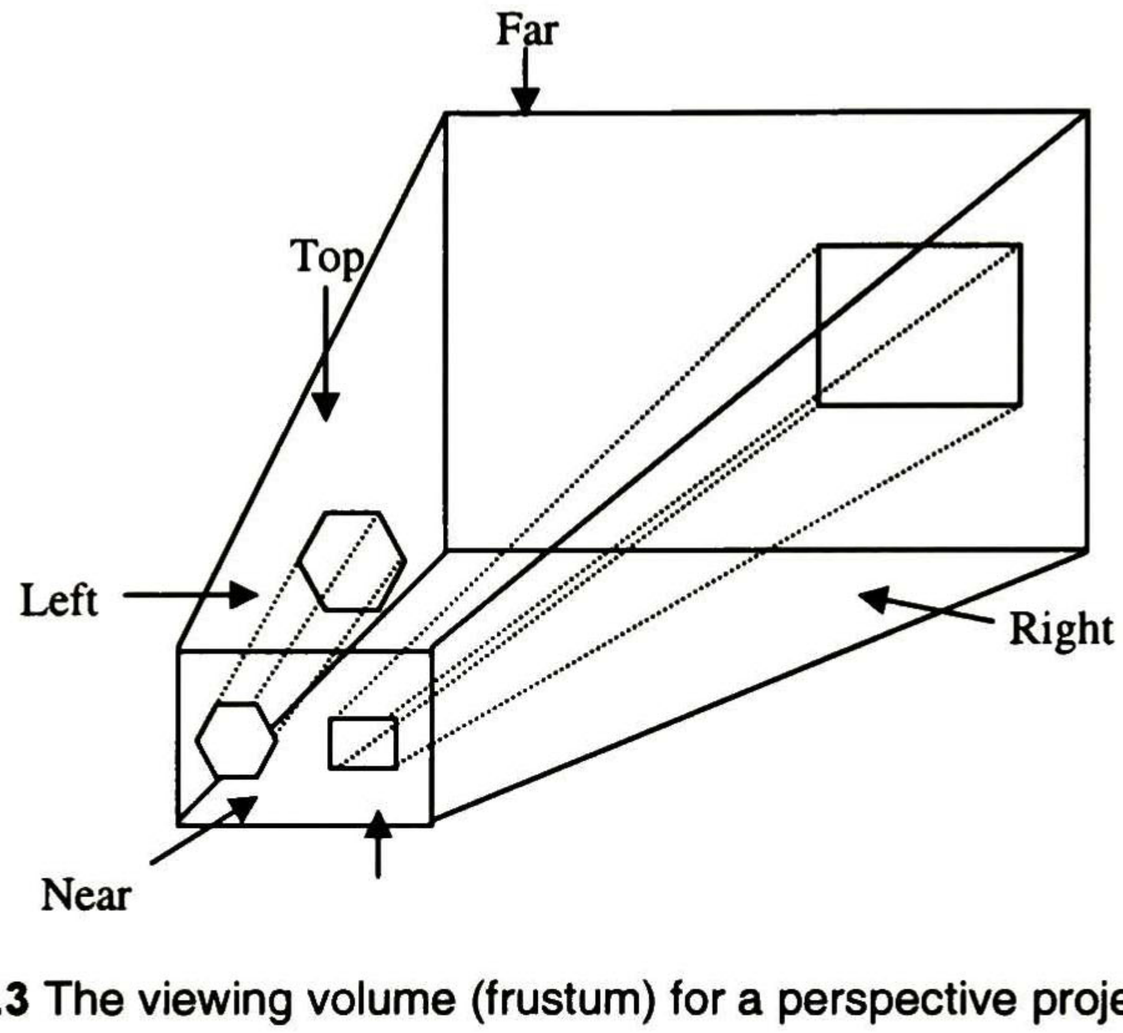


Figure 5.3 The viewing volume (frustum) for a perspective projection

5.5.3 Virtual Objects

The design of static and dynamic virtual objects should both consider the issues described in the next sections.

5.5.3.1 Motion

Whenever an object changes its coordinates (*x*, *y*, *z*) we say that the object performs a motion. An object motion is a consequence of either the interaction with other objects, the dependency with a moving object, the action of its own behavior or the commands of the user it represent (when the object is an avatar).

Often, a motion involves the change of more than one component (axis) of the current coordinates; in most cases, the values of *x* and *z* change together after a motion step. Considering the same amount of steps, the absolute distance traveled by a virtual object after a motion involving the change of one axis, should be the same than if the motion involves more than one axis. Our platform handles this

problem employing trigonometry to obtain the new values of all the axes involved in the motion, according to the current angle of the object. Figure 5.4 depicts an example of two consecutive motions performed by the same object. In the first one, the cube translates along the *x* axis. But in the second the object moves along two axes: *x* and *z*. Notice that the object appears smaller after the second motion.

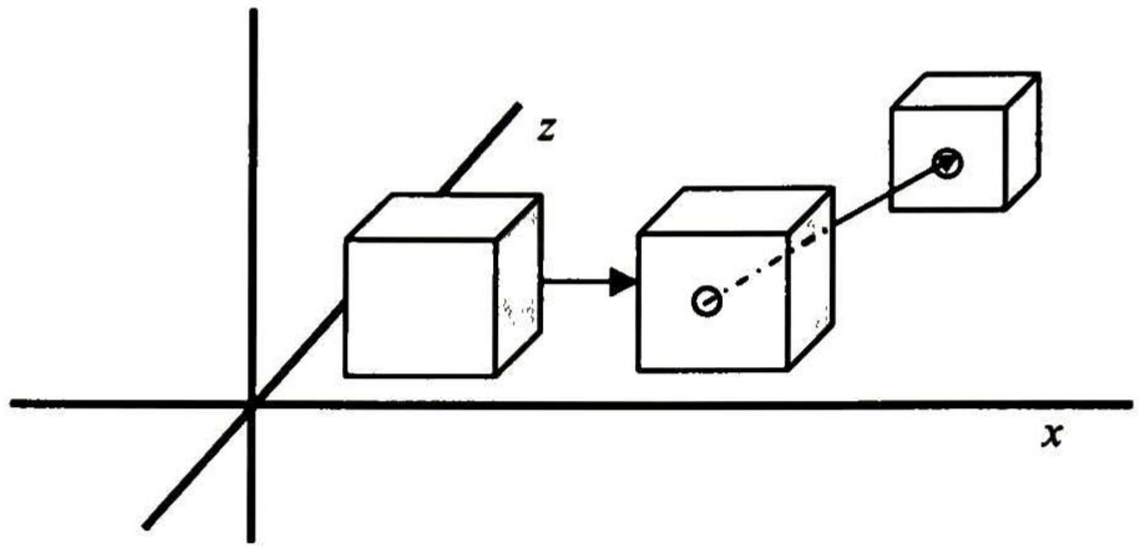


Figure 5.4 Example of two consecutive motions

5.5.3.2 Rotation

A virtual object is not only assigned 3D coordinates, but also an angle that indicates its orientation: where the object is staring at and heading to. As well as in a motion, after a rotation, all the vertices that define the object have to be recalculated. GeDA-3D takes advantage of sinus and cosines to obtain the new coordinates of every vertex rotated, according to the current angle of the object. Figure 5.5 depicts an example of a 3D object performing a 30º rotation around the *y* axis.

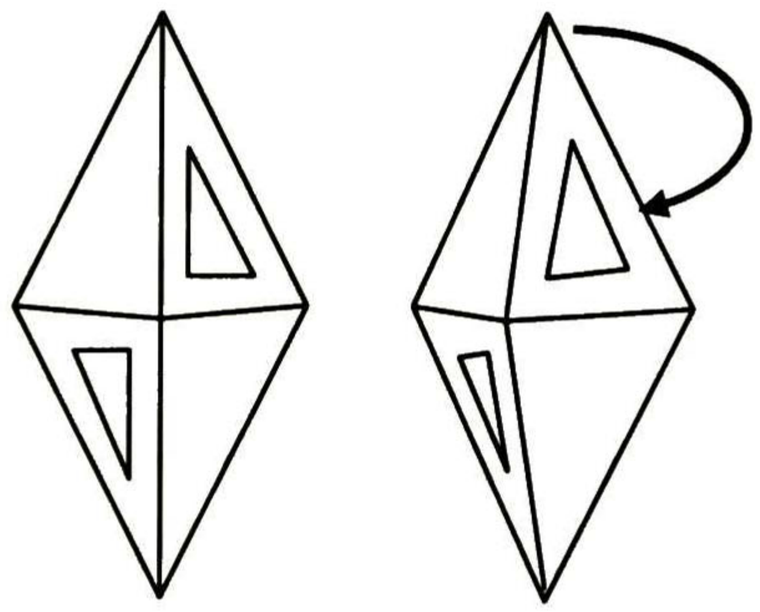


Figure 5.5 Example of an object rotation

5.5.3.3 Physical properties and capabilities

The physical property of a VO determines not only its graphical representation, but also defines the properties of the material that the VO is supposed to be made of. For instance, opaqueness, hardness, reflectivity, lightness, shininess, smoothness and transparency (see Section 5.4). To achieve this, texturing, lighting and coloring techniques are used.

In Section 5.5.2.1, we talked about the different kinds of lights and their effects in the scenes of the real world. Objects have a color of their own too, and it is defined by its reflected wavelengths of light. When we use lighting, we do not describe polygons as having a particular color, but rather as consisting of materials that have certain reflective properties. Instead of saying that a polygon is yellow, we say that the polygon is made of a material that reflects mostly yellow light. We are still saying that the surface is red, but now we must also specify the material's reflective properties for ambient, diffuse, and specular light sources. A material might be shiny and reflect specular light very well, while absorbing most of the ambient or diffuse light. Conversely, a flat colored object might absorb all specular light and not look shiny under any circumstances. Another property to be specified is the emission property for objects that emit their own light.

The capabilities of a VO are closely related to its behavior, because they determine issues such as: the number of actions a VO can perform, which worlds are suitable for the object to subsist, how speed it can move, whether it can swim, fly or walk, and so on.

5.5.3.4 Reachable states

Because of the dynamics of the Virtual Environment, the useful life of an object is often concerned with a set of reachable states. The graphical representation of an object is directly related with its current state. The software implementation of such graphic is called an *avatar* (explained more carefully in Chapter 6). All the objects have an initial state, which changes every time an action is performed by or over the object. Actions include walking, fighting, turning around, creeping, sitting, jumping, swimming, speaking, and interactions with other objects. Usually, after an

interaction, the states of the VO involved are modified. The life cycle of an object comes to an end when it reaches the final state.

5.5.3.5 Behavior

The behavior is the dynamic side of a virtual object. It comprises the set of actions an object can perform, including the way it reacts to changes in the environment. The behavior is subordinated to the capabilities of the VO. The behavior is closely related to Artificial Intelligence, because it is often implemented using learning algorithms, such as, genetics algorithms, neural networks, adaptive algorithms, and so on. One of the principal aims of our VE Editor is to provide an interface enabling the virtual world to understand the actions proposed by the behavior of a VO.

5.5.4 Observers or Creators

The design of a VE Editor involves not only the identification of some issues concerning the virtual environment, but also the role that a user is going to play and how his/her actions affect the VW.

5.5.4.1 Navigation in the virtual world

Once the simulation of the world is started, the VE Editor should then allow an observer to perform actions involving a motion –walking, turning around, flying- through the whole environment. The navigation includes actions such as: moving left & right, rising, descending, walking forward & backwards, and rotating right & left.

5.5.4.2 Placing an object in the virtual world

Our VE Editor intends to allow users to pick a Virtual Object and drop it into a 2D space representing the virtual world. The main challenge here is to manage the conversion of a 2D coordinate (pixel in a screen) to a 3D coordinate (location in a virtual world), where depth is added (*z*-axis). Once we are able to convert the pixel coordinate to a 2D-virtual coordinate (ignoring *z*-axis), this problem can be solved in many ways.

Since our tool allows an observer to navigate within the virtual world, he is assigned a coordinate too. Whenever we wish to add a VO, we may assign the current *z*-coordinate of the observer, but increased by one or two, perhaps –because it is

farther than the observer–. In our case study, we provide the user a view of the virtual world top–down initially, which means that the *y*-axis is ignored. Thus, we convert the current (*x*, *y*) of the monitor to its corresponding (*x*, *z*) point of the virtual world. Once the simulation is started, the front view is restored and so, the *y*-axis is considered.

5.5.4.3 Selecting an object from the virtual world

Sometimes, is desirable for a user to be capable of picking an object and manipulating it to his/her convenience. This is a similar problem to the one described above. We have to perform a conversion from the monitor coordinates to the world coordinates in order to find out which object(s) are selected.

5.5.5 Implementation

Some of the issues introduced in section 5.5 are successfully addressed by the VR-oriented language employed, such as, selecting an object from the virtual world, projection, lighting source and removal of non-visible objects. Others are already implemented in the platform, including: order of objects display, constant updating, resolution of collisions, motion, rotation, navigation in the virtual world. The rest are intended to be handled by a VE-Editor for GeDA-3D: placing an object from the virtual world, behavior, reachable states, physical properties and capabilities, dynamic group management, physical laws and congruency

5.6 Conclusions

In this chapter, we have focused our attention in describing how we conceive a VE Editor for GeDA-3D, including its main components. In addition, we reviewed some issues to consider in the development of such editor and the techniques used by our platform to address some of the issues. The elements of a VE Editor for GeDA-3D are summarized and depicted in Figure 5.5.

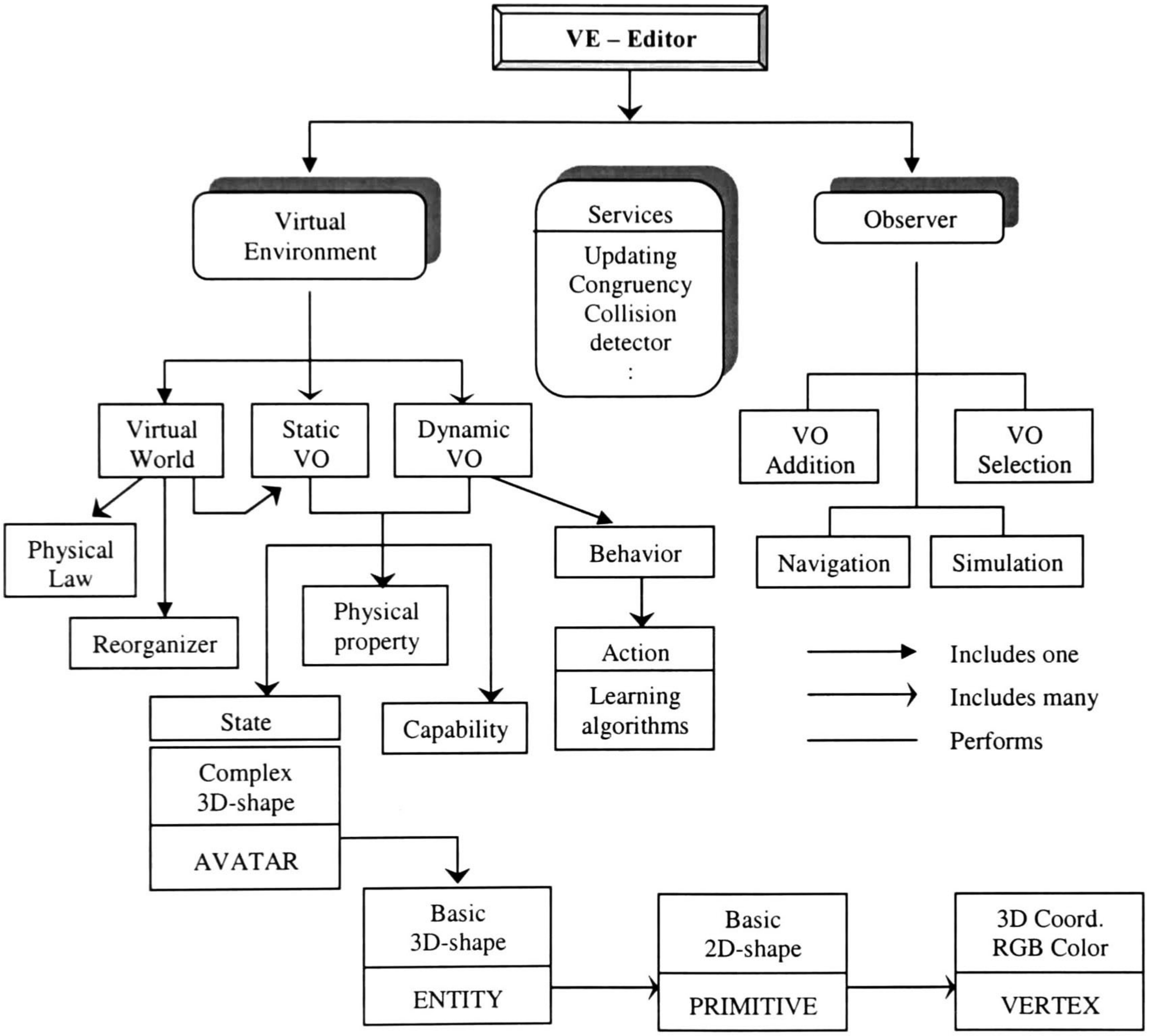


Figure 5.5 Components of a VEE

Chapter 6

Case Study

6.1 Objectives

Introduce the 3D Combat Game developed as a case study of a VE Editor for GeDA-3D, including conception, operation, rules and implementation details. Describe the issues fulfilled by the game as a VE Editor.

6.2 Introduction

In Chapter 5, a series of issues to be taken in account in the development of VE Editors for GeDA-3D was summarized. In addition, for every requirement analyzed, whether it is already implemented in the platform, handled by VR – oriented languages or intended to be fulfilled by the VE Editor, was specified.

As a Case Study, we develop a networked 3D Combat Game which satisfies some of the issues referred above, mostly, the ones handled by the platform and by the language employed. It works over GeDA-3D and takes advantage of the services provided by our platform. This game allows remote users to create a shared dynamic VE: a middle-ages combat environment

Since the goal of the Case Study is to show the facilities to implement VE Editors provided by the platform, instead of creating a very exciting game, the operation and interface of the Combat Game is rather simple.

This game is about two empires encouraged to fight against each other for power. Every empire is constituted by a set of soldiers and miners commanded by a king, each having a virtual representation in the environment. The objective is to conquer

the rival's kingdom in order to establish a universal kingdom, ruled by the winner king. Such a game was first conceived in the DAI course with the aim of testing the behaviors of different empires equipped with different learning algorithms

6.3 Conception of the Game

The conception of the 3D Combat Game considers the following issues:

- The combat environment is shared by a number of remote GeDA-3D users, and it is built by them as a whole.
- A user may choose a virtual object from a list and drop it somewhere in the combat environment (virtual world).
- Some virtual objects are static, such as rocks and trees
- Some of them are dynamic: kings, soldiers, miners, farmers. In such a case, VO include a behavior, implemented with adaptive algorithms [ZUNIGA:02] enabling them to perform some actions (fight, walk, gather gold, defend) affecting the state of the VE. We refer to these dynamic VO as *agents.*
- Whenever an object is successfully added to the environment, the number of objects available decreases; the rest of the users are immediately notified about such changes
- As soon as there are no virtual objects left, a simulation of a battle between the two empires can then be started by any user.
- Once the battle starts, the users can navigate across the environment and appreciate the changes taking place
- This game runs on a series of cycles
- A cycle starts whenever the *referee* sends the state of the territory to both empires; in turn, they send back a set of actions (possible null) they decided to perform (using learning algorithms)
- As soon as the referee receives the actions of both empires, it validates and performs them (if valid). Then, the referee sends back the new state of the territory and a new cycle starts. The main operation of the editor is depicted in figure 6.1

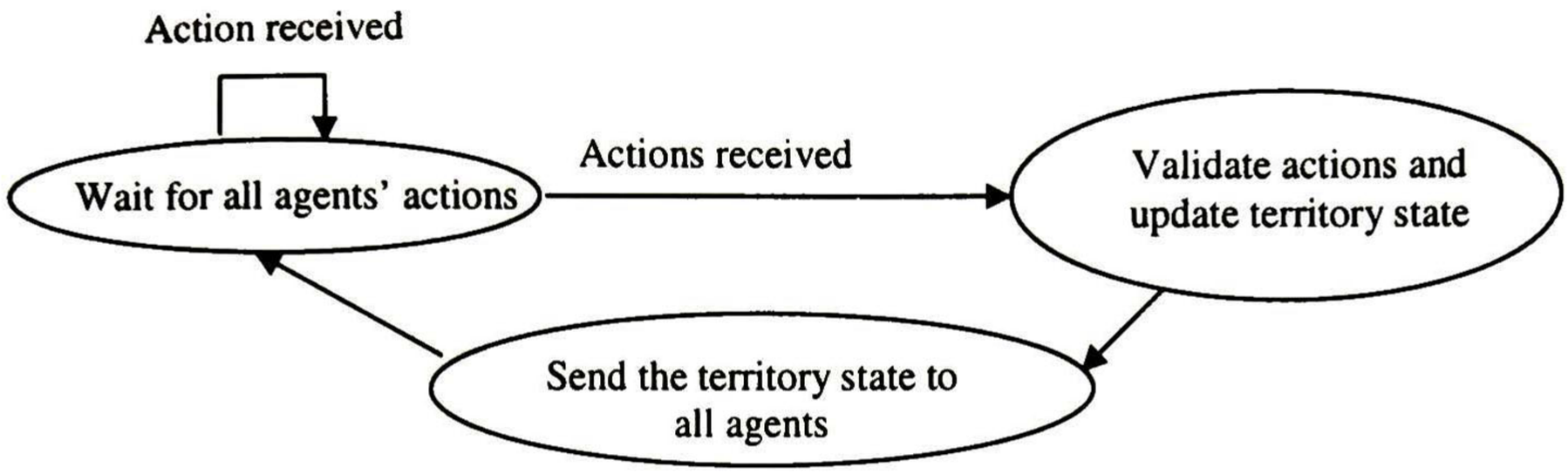


Figure 6.1 Operation of the editor

6.4 Rules of the Game

As mentioned in 6.2, this game consists of two empires fighting against each other to get the power of the whole territory. Every empire is constituted by a set of soldiers and miners commanded by a king. Once a king is killed, its empire no longer performs any action, and the other is declared winner. The rules of the game are listed below.

1. The territory is divided by a matrix of cells, each may lodge a king, a soldier or a miner, one at a time
2. The territory contains certain amount of gold: every cell either provides or not a gold unit
3. A cell can also contain a static object (tree, rock)
4. Every agent (soldier, miner or king) is capable to walk from one cell to another contiguous free one
5. The soldier is supposed to be faster than the miner and the latter faster than the king. Therefore, in a cycle, a soldier may walk three cells, a miner two cells, and a king only one.
6. A soldier is capable to:
    a. attack a near enemy
    b. defend himself from a near enemy attack
7. A miner is capable to extract the unit gold located in its own cell
8. Every unit gold extracted is used to add a new soldier or miner to the empire
9. The operation of the game is divided by execution cycles, where every agent may perform at most one action

10. When the king is attacked, the agents commanded by him can no longer perform actions.
11. The game is over as soon as a king is killed
12. If one soldier attacks a miner enemy who doesn't move, then the miner dies
13. If one soldier attacks a soldier enemy who:
    a. attacks him back, then both dies
    b. moves to another cell, then no one dies
    c. defends himself from him, then no one dies
    d. defends himself from another soldier, then the second dies
    e. performs no actions, then the second dies
14. The decision making of an agent is governed by its behavior algorithm assigned

6.5 Implementation

Part of this work involves the development of adaptive algorithms which comprise the behavior of the dynamic objects (agents) present in the game. This is not described in detail here, but in [FABIEL:02]. Instead, this section intends to explain in a high level a huge part of the implementation related to the development of and navigation within virtual environments. This technical part involves the design of an onthology (see Section 4.8) which makes possible to develop and display 3D objects.

6.5.1. Onthology

The onthology of our VE Editor inherits all the concepts defined in the onthology of GeDA-3D, but adds one element at the top level: the *avatar*.

In our VE Editor, an *avatar* is the graphical representation of a virtual object (soldier, miner, king...) having some state. If an object moves, rotates or performs any other action (attack, dig, defend) then the avatar will have to be redrawn with different coordinates, angle or state. A new state acquired involves performing local changes to the avatar components. An avatar is basically constituted by a unique identifier, a coordinate, an angle and a list of entities (3D-shapes).

Our VE Editor includes a library of basic 3D-shapes (boxes, diamonds, spheres, stars, horizontal/vertical cylinders) useful to build more complex 3D-shapes, the *avatars*. All these 3D-shapes are classes implementing the following methods:

- Void setCoord (Coord3D). Assigns the current coordinate of the entity with respect to the avatar. Coord3D is a tuple (posX, posY, posZ).
- Void setAngle (Angle). Assigns the current angle of the entity. Angle is a float value with respect to the avatar.
- Entity getEntity ( ). Returns a graphical representation of the 3D-shape.

6.5.2 VObject and DVObject interfaces

Unlike GeDA-3D, our VE Editor only displays *avatars* in the 3D scene; therefore, every static or dynamic virtual object intended to be part of the game needs to be treated as an avatar.

For the development of the game, we first developed a generic virtual object –called VObject– that indicates all the actions a static or dynamic virtual object may perform. The methods implemented are the following:

- Void **setCoord** (Coord3D). Sets the new coordinate of the avatar. This method is called after an object is first created, and whenever the object moves –in case of a DVO–.
- Coord3D **getCoord** ( ). Returns the current coordinate of the avatar.
- Void **setObserverPostion** (Coord3D). Changes the observer position –after a motion / rotation of the observer–. Useful for reordering the entities that make up the avatar: display first far entities, display last near entities
- Dimensions **getDimensions** ( ). Returns the current dimensions of the avatar.
- Avatar **getAvatar** ( ). Returns the graphical representation of the object in such a way that the editor is capable to understand. This method is called after an object is first created and whenever the object performs an action altering its state.

The implementation of a DVO involves defining some methods enabling the object to change its state and its angle. DVObject interface extends the VObject interface in order to implement the following methods added to the ones listed before:

- Void **setAngle** (Angle). Sets the new angle of the avatar. This method is called whenever a DVO changes direction
- Integer **getAngle** ( ). Returns the current angle of the avatar as an integer value
- Void **setState** (State). Changes the current state of the DVO. Such a state defines the graphical representation of the avatar. For instance, a soldier can walk, attack using the sword or defend himself with the shield. For every action taken, at least one part of the soldier's body changes position and direction.

VObject is actually a common interface for all the objects present in the game who implement it. This interface enables the editor to treat in the same way all the objects present in the combat environment, no matter if it plays a soldier, a tree or the moon.

6.5.3 Avatar actions

The combat environment includes static VO like the trees, rocks and pieces of gold, but also dynamic VO (agents) such as the soldiers, miners and kings. These ones are considered dynamic because they perform actions that eventually alter the state of the environment. The behavior of an agent is defined as a set of learning algorithms who choose whether and which action is best to be performed given some circumstances of the game. An action should include the following data before it is sent to the referee:

- An unique identifier of the object performing the action. Example: "A_S1" is the name of the soldier 1 from empire A.
- The name of the action: "Defend", "Attack" "Move" …
- The (*i*, *j*) coordinates of the target cell(s) (3 cells at the most). Example, a soldier may attack to someone lying on cell (3, 4); a miner may walk over cells (15, 6) and (15, 7).

Once the referee has received the actions proposed by all the agents of both empires, it then validates them and keeps only the valid ones. The actions filtered are sent to the observers. The VE Editor interprets the actions; then, it turns 2D cell coordinates into 3D virtual coordinates and turns action names into graphic simulations.

6.6 Requirements fulfilled by the game

The 3D Combat game not only performs the order of avatars display according to the current position of the observer, but it also orders the entities that make up every avatar. For instance, whenever the observer is positioned against a soldier, the shield will always appear in front of the soldier covering part of its waist and the fist behind the shield. This can be appreciated in figure 6.2.

The removal of non – visible objects is successfully performed by the VR–Oriented language. The constant updating is assisted by a Consistency Service which enables all the users to share the same combat environment and keep aware of all the changes. The collision detector Service is provided by the platform.

The projection and lighting source is successfully solved by the language employed. The motions and rotations are performed in a more complex way, for it involves the change of state of an object. For instance, when a soldier moves from one cell to another, the display of four different avatars is involved in the motion, in order to provide an illusion of walking to the observers. If a soldier intends to walk left, right or behind, a rotation is performed first. This game includes only 90°, 180° or 270° rotations.

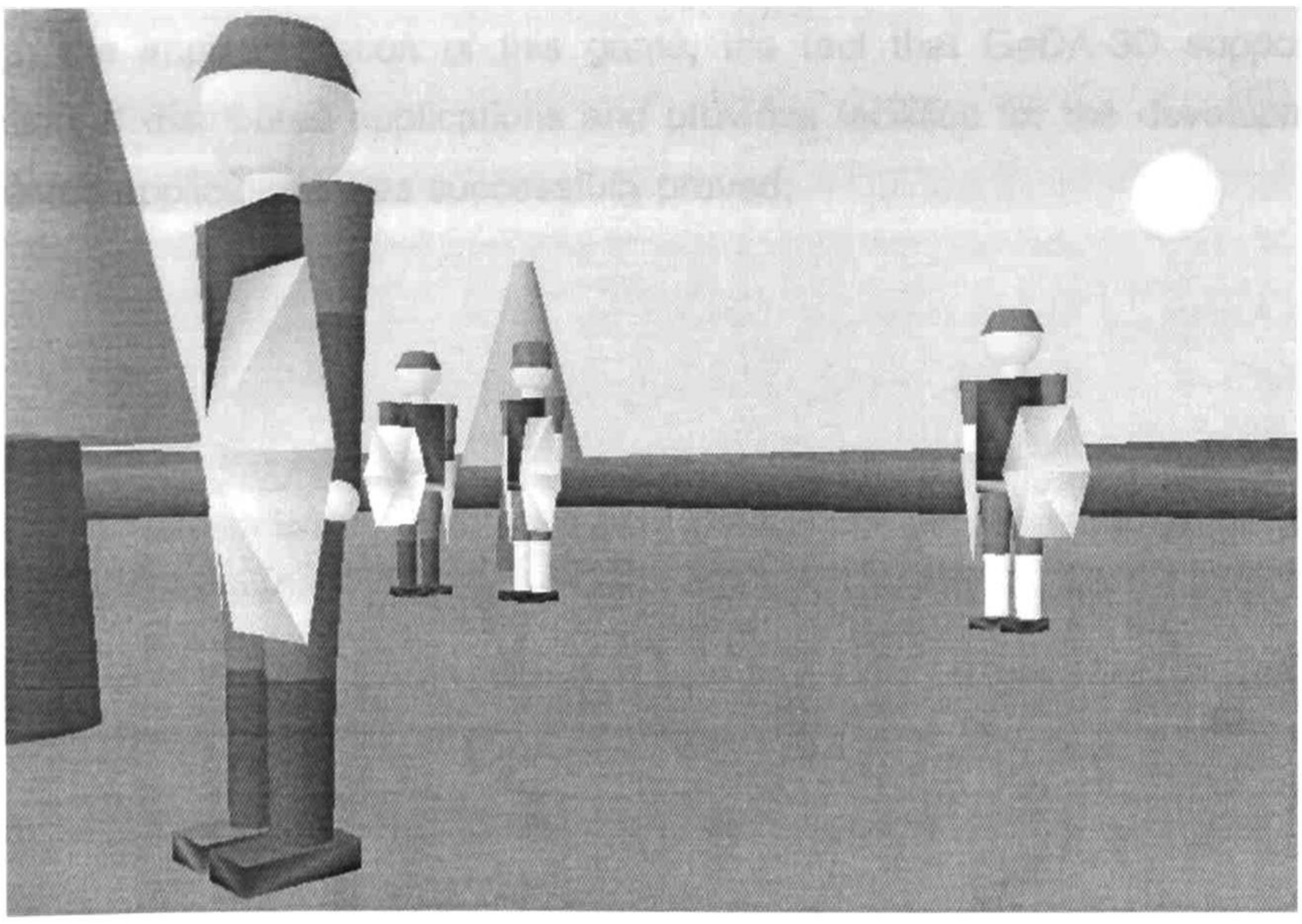


Figure 6.2 A view of the 3D Combat Game

An observer performs the navigation in the same way as he does in the 3D–space provided by the platform: he/she may move left, right, up, down, forward, backwards, rotate left and right. But in this case, these motions are not sent to any other user. The observer can also click on an avatar and get information, such as, the identifier, the object name and last action performed. A user can place objects to the virtual scene before the simulation is started. In this moment, the scene appears to be in 2D, for it is viewed from the top to the bottom. With such a view, the users can easily select a free 2D space from the whole combat environment to place an object.

6.6 Conclusions

In this chapter, we introduced the 3D Combat Game as a case study of our VE Editor for GeDA-3D. We first explained the way this game behaves as a VE Editor, after that we described how a user conceive such game and described the rules of the game. In the next section some details of implementation were presented and the onthology was introduced. The onthology is a common language used by the developers of virtual objects and the editor in charge of displaying them. Finally, the last section summarized some issues (introduced in Chapter 5) that our Case Study considered and fulfills successfully. Some of them are already addressed by the platform and used by the game.

Through the implementation of this game, the fact that GeDA-3D supports the integration of distributed applications and provides facilities for the development of VR-oriented applications was successfully proved.

Chapter 7

Conclusions

7.1 Objective

Describe the current state of the project by summarizing the work actually developed in this term and the work left to do in a future research.

7.2 Present work

Based on a generic architecture described in [TOSCANO:00], a middleware application, GeDA-3D, was developed and presented in this work. GeDA-3D is useful to implement distributed cooperative applications from different nature.

The present work was divided in two phases. The first one involves the following tasks: analyze and criticize a previous work introduced in [TOSCANO:00, PUGA:01], propose some changes, and implement the new architecture for GeDA-3D. In the second phase, the design and implementation a VE Editor for GeDA-3D is included.

Due to the nature of applications the platform intends to manage, the design of GeDA-3D follows the agent paradigm. The agent paradigm facilitates the design and development of complex systems, since it provides higher and more humanly abstractions of things.

Thus, GeDA-3D can be thought as a multiagent system made up of a community of agents: Coordinator, Client and Application. The Coordinator is constituted by a number of services distributed across GeDA-3D** network.

A service that involves more complexity and analysis is the Consistency Service. It is in charge of keeping all the users aware of every change performed in the virtual space. Such changes are performed by users avatars and primarily include motions, rotations, logins and logouts. The whole operation of the Consistency Service is modeled using Petri Nets formalism, specifically Elementary Object Systems.

The Coordinator includes other services such as Chat Service that enables users to exchange plain – text messages point to point; the LookUp Service hides all the complexity of communication since it performs the task of locating all the services, applications and clients currently connected to GeDA-3D**; the Users Service and the Applications Service are in charge of handling the issues related to validate logins of users and applications and Manage the operations performed to the users and applications databases.

The Client manages all the issues related to VR; it includes a 3D space where it displays all the changes received from the Consistency Service; the Client also provides means for the users to navigate through the space and avoids sharing the same virtual space with a collision detector.

Our platform includes a Mobility Platform enabling remote applications to be sent to a user who request them via a socket. Such transfer includes a bytecode (class) of the application and the current state (instance). This whole operation of this platform is modeled with an extension of Petri Nets, and included in the work of my partner.

Since its conception, GeDA-3D was meant to support the development of VE Editors; nevertheless, the present work is not intended to design and implement a full VEE, but only to identify the main problems in constructing a VE Editor. This fact leads us to review some concepts related to VR and propose original solutions to solve some problems, including the development of a dynamic group management and a congruency service

In order to apply the ideas proposed in the work, a practical and simple example of a VE-Editor for GeDA-3D has to be considered. Hence, a 3D-Combat Game which fulfills most of the requirements of a VE Editor was developed as the Case Study.

This application takes advantages of the services and tools provided by the platform, GeDA-3D. In addition, it introduces an onthology useful to develop and display virtual objects. This ontology helps to establish a common language between the developer of VO and the VE editor.

The 3D Combat Game allows different users (observers) to add static and dynamic virtual objects (agents) to a shared combat environment. When the simulation is started, the agents (soldiers, miners, kings), governed by their behaviors, remain performing actions that alter the state of the environment. During the simulation, the observer may navigate and watch the combat from different angles and locations. The 3D Combat Game allows different users (observers) to add static and dynamic virtual objects (agents) to a shared combat environment. When the simulation is started, the agents (soldiers, miners, kings), governed by their behaviors, remain performing actions that alter the state of the environment. During the simulation, the observer may navigate and watch the combat from different angles and locations.

The implementation of the case study shows that there is still a lot of work to do, but the accomplishment of the objective is feasible and it is worth further research. At the end of this work, we achieved the objectives established at the beginning, which includes: re-design / implement GeDA-3D and study the possibility of implementing a VE Editor for GeDA-3D.

7.3 Future work and recommendations

Concerning to the architecture, the mobility of applications is performed using sockets, which means that the IP addresses of both the source and target hosts are provided in order to find the location of the user and successfully sends the application. This situation is not desired if we wish to hide the communication complexity considering that some user hosts have no public (routable) IP addresses since they belong to networks where a DNS server assign dynamically local IP address. Sometimes, a host with a public IP address belongs to a network with proxy servers that restrict the point-to-point communication between their hosts and external hosts. This issue is successfully addressed by the LookUp Service of GeDA-3D since it takes advantages of the Naming Service provided by CORBA. If both the class (es) and the instance of the application could be included as formal

parameters of a CORBA – object method, we could do without the IP address since it is internally handled by CORBA and, therefore, get rid of the sockets service.

A complete and successful VE Editor involves the implementation of all the issues outlined in chapter 5, indeed, resulting in a complex task. My aim in a future research project will be involved in the development of a Reorganizer agent using Dynamic Group Management (DGM) and a Congruency Service assisted by semantics as research project. As described previously, this work will be helpful in our VEE in providing more dynamicity and reality in a VE. That is, a DGM would help to reorganize the relationship between the static and dynamic VO after any change occurred in the evolving world. A Congruency Service would determine which VO may live in a specific world, according to the physical laws of the world and the physical properties of the object. The present work is only the first part of a huge and complex research and implementable project not ever done before.

Chapter 8

References

[AVIARY:97] "The AVIARY Distributed Virtual Environment", David Snowdon and Adrian West. The 2nd UK VR-SIG conference, 1st December 1994, Theale, UK.

[BERGSTRA:01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, Editors. Handbook of Process Algebra. http://carol.wins.uva.nl/~alban/Handbook-free/

[BRENNER:98] W. Brenner, R. Zarnekow, and H. Wittig. "Intelligent Software Agents". Springer, 1998.

[BIRRELL:84] A. D. Birrell e B. J. Nelson, "Implementing Remote Procedure Calls", ACM Trans. On Computer Systems, vol. 2, pp. 39-59, February 1984.

[CABRI:99] G. Cabri, L. Leonardi, F. Zambonelli. "Mobile Agents Technology: Current Trends and Perspectives" Università di Modena, 1999.

[COLOURIS:96] George Coulouris, Jean Dooimore, Tim Kindberg. Distributed System.Concepts and Design. 2nd Edition. Edit. Addison-Wesley. USA, 1996.

[COM:95] Microsoft Corporation. The Component Object Model

Specification, Version 0.9, October 24, 1995 [online].
http://www.microsoft.com/Com/resources/comdocs.asp

[COM:99] Microsoft COM Technologies. 1999.
http://www.microsoft.com/com/tech/com.asp

[CORBA:1] Overview of CORBA.
http://www.cs.wustl.edu/~schmidt/corba-overview.html

[CORBA:2] CORBA Tutorial
http://www.cs.umbc.edu/~thurston/cbatop.htm

[DCOM:96] Microsoft Corporation. Distributed Component Object Model Protocol-DCOM/1.0, draft, November 1996 [online].
http://www.microsoft.com/Com/resources/comdocs.asp

[DCOM:97] The Distributed Component Object Model.
http://www.dalmatian.com/dcom.htm

[DIVE:93] C. Carlsson and O. Hagsand, "DIVE - A Platform for Multi-User Virtual Environments, Computers and Graphics 17(6), 1993

[DVR:1] Distributed Virtual Reality - An Overview.
http://ece.uwaterloo.ca/~broehl/distrib.html

[DVR:2] Distributed Virtual Reality Systems
http://www.doc.ic.ac.uk/~np2/virtual_reality/distributed.html

[FUGGETA:98] A. Fuggeta, G. Picco, G. Vigna. “Understanding Code Mobility“.IEEE Transactions on Software Engineering, vol 24, No. 5, pp. 352-361, may 1998.

[GOPALAN:98] Gopalan Suresh Raj. A Detailed Comparison of CORBA,

DCOM and Java/RMI.
http://www.execpc.com/~gopalan/misc/compare.html

[HYACINTH96] Hyacinth S. Mwana. "Software agents: An overview". Intelligent Systems Research. AA&T, BT Laboratories, 1996.

[JAVA] http://www.java.sun.com

[KNABE] Frederick Knabe P. "An Overview of Mobile Agents Programming". Universidad Católica de Chile, Casilla 306, Santiago 22, Chile.

[LOPEZ:97] Ernesto López Mellado. Introducción a las Redes de Petri. Universidad Autónoma de Nuevo León. Octubre 1997

[MILNER:89] Robin Milner. Communication and concurrency. Edit. Prentice Hall, 1989.

[MULLENDER:95] Sape Mullender. Distributed Systems. 2nd Edition. Edit. Addison-Wesley. USA, 1995

[OMG] http://www.omg.org/

[OMG:00] Mobile Agent Facility Specification. January 2000. OMG Specifications.

[OMG:95a] Common Object Request Broker Architecture, OMG, July, 1995.

[OMG:95b] Common Object Services Specification, OMG 95-3-31, 1995

[OPENGL] http://www.sgi.com/software/opengl/

[PUGA:99] Félix F. Ramos C., Ma. Eugenia Puga N., Silvia I. Toscano G. A CORBA based Architecture for a Virtual Office. First Edition, accepted poster on memories of the WRV'99 2 Workshop Brasileiro De Realide Virtual.

[RAMOS:98] Félix Ramos, David Garduño, Iván Romero, José Luis Márquez. Arquitectura de una Oficina Virtual Distribuida. eMemories of the CIE98 congress, CINVESTAV México, 1998.

[RMI] Java RMI Tutorial. http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

[SOCKETS] Perl, Sockets and TCP/IP Networking. http://www.perlfect.com/articles/sockets.shtml

[SYCARA98] Katia P. Sycara. "Multiagent systems" American association for artificial intelligence. Summer 1998.

[TORGUET:98] Patrice Torguet. VIPER: Un modèle de calcul réparti pour la gestion d'environnements virtuels. L'Universite Paul Sabatier de Tolouse (Sciences). 17 fèvrier 1998.

[TOSCANO:00] MC Silvia Toscano Garibay. Ambiente genérico virtual distribuido. 19 - Septiembre - 2000

[VALK] Rüdiger Valk. "Petri nets as token objects: An introduction to elementary object" Universität Hamburg, Fachbereich Informatic.

[VINOSKI:98] Steve Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, *IEEE Communications Magazine*, February, 1997.

[VR] Virtual Reality: A Short Introduction. K – P. Beier. http://www-vrl.umich.edu/intro/

[VRML:1] VRML 2.0. Basic Concepts. http://deslab.mit.edu/DesignLab/courses/13.016/visualization/second/introconcepts.html

[VRML:2] A brief tutorial of VRML. http://www.vrml.org/technicalinfo/specifications/eai_fdis/part1/introduction.html

[WRIGHT:00] Richard S. Wright, Jr. Michael Sweet. OpenGL SuperBible, Second Edition. Waite Group Press USA, 2000

[ZIMMERMAN:01] Armin Zimmerman. Petri Nets. 2001. http://pdv.cs.tu-berlin.de/~azi/petri.html

[ZUNIGA:02] Adaptive algorithms applied to the VE Editor for GeDA-3D

ANIVERSARIO
Cinvestav

Centro de Investigación y de Estudios Avanzados del IPN

Unidad Guadalajara

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó la tesis: A Virtual Environments Editor del(a) C. Hugo Iván PIZA DÁVILA el día 27 de Septiembre de 2002

DR. JOSE LUIS LEYVA MONTIEL
INVESTIGADOR CINVESTAV 3B
CINVESTAV GDL GUADALAJARA

DR. LUIS ERNESTO LÓPEZ MELLADO
INVESTIGADOR CINVESTAV 3A
CINVESTAV GDL GUADALAJARA

DR. FÉLIX FRANCISCO RAMOS CORCHADO
INVESTIGADOR CINVESTAV 2A
CINVESTAV GDL GUADALAJARA