Towards the next generation of Human-Computer Interface

David N. Snowdon, Adrian J. West*, Toby L.J. Howard

Advanced Interfaces Group
Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL
United Kingdom

Tel: +44 61 275 6251
Fax: +44 61 275 6236
Email: ajw@cs.man.ac.uk

Presented at 'Informatique'93: Interface to Real & Virtual Worlds', pages 399-408, 24-26th March 1993, Montpellier, France.

Keywords: *Virtual Reality, VR, generic virtual world, parallelism, computer graphics, transputer, object-oriented techniques, autonomous objects, human-computer interfaces, AVIARY*

Abstract

The techniques of Virtual Reality would seem to have great potential for revolutionising some currently problematic areas of human-computer interaction. The Advanced Interfaces Group at the University of Manchester are currently engaged in addressing major issues in the use of these techniques for real applications. Part of this work is the construction of an environment capable of hosting a wide range of virtual-reality based models, and providing a coherent applications interface to them. This paper details the underlying software model of the AVIARY implementation.

*Author for correspondence

1 Introduction

This paper focuses largely on the implementation of the AVIARY system. While our motivation and the underlying philosophy of the system are described in detail in [1], it is necessary to introduce these briefly in order to place the implementation in context.

2 The AVIARY project — a high level view

The techniques of Virtual Reality(VR) would certainly seem to herald a revolutionary advance in the human-computer interface (HCI) enabling many of the current limitations, particularly in 3 dimensional interaction, to be addressed. The basis of this advance could perhaps be most correctly viewed as the ability to engage the powerful human perceptual mechanisms directly, rather than relying almost exclusively on human cognitive capabilities.

Progress is being made in the underlying technology of VR, but significant questions remain as to how it can be applied to serious problems of significant complexity; how practical interfaces will actually function, and the successful models of HCI which can be based upon them. It should be emphasised therefore that we are concerned here not with a demonstration of the virtual reality technology per-se, which is assumed, rather, we are concerned with the issues of generic support of a wide range of substantial, real applications.

In order to assist our work on applications, we have proposed a general model of the virtual reality environment which we call AVIARY. AVIARY is an environment that supports multiple concurrent virtual worlds, applications, users, and the facilitation of interaction between them. The model is under-pinned by higher level philosophical and users conceptual models of the virtual totality which lend coherence to a system of diverse worlds and applications.

The single conceptual model is important. Different applications will require different virtual worlds. It is obvious that all conceivable properties cannot be accommodated within a single world, either due to performance limitations, or to contradictions within the world model. From this we recognise that many different virtual worlds will be needed. Our aim is to provide a coherent experience of the environment as a whole, and to facilitate interaction between such worlds.

The following sections detail the current implementation model of the AVIARY system. AVIARY is designed to be platform independent, and is currently distributable over our transputer racks and sun network. Our future plans include porting AVIARY onto the 1.2Gflop, 32 processor virtual shared memory KSR supercomputer in the department.

3 Design Philosophy

In the AVIARY model everything is an object (in the Object Orientated Programming sense of the word). Every object is autonomous and executes concurrently. Splitting the virtual world into objects provides convenient units of parallelism and because objects generally represent something which can be seen (or heard, felt etc) in the virtual world they are conceptually easy to deal with.

At this point it is necessary to define some of our terms, particularly so due to the confusion that reins when discussing object-oriented virtual reality systems, and the real world itself. Things (such as rooms, menus or users) which are presented to the user in a virtual world are referred to as artifacts. Objects which directly cause artifacts to be present in the virtual world are known as demons. The methods defined for the demons give the artifacts a behaviour. In some cases (such as walls, floors or furniture) this behaviour may be non-existent or minimal. In other cases an artifact may represent an application and its

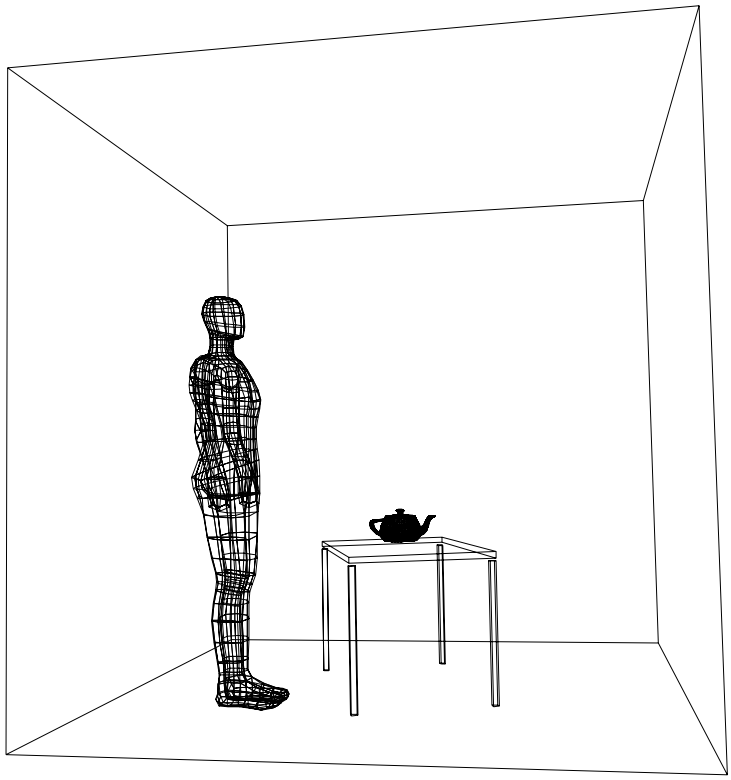

Figure 1: A Simple Virtual World

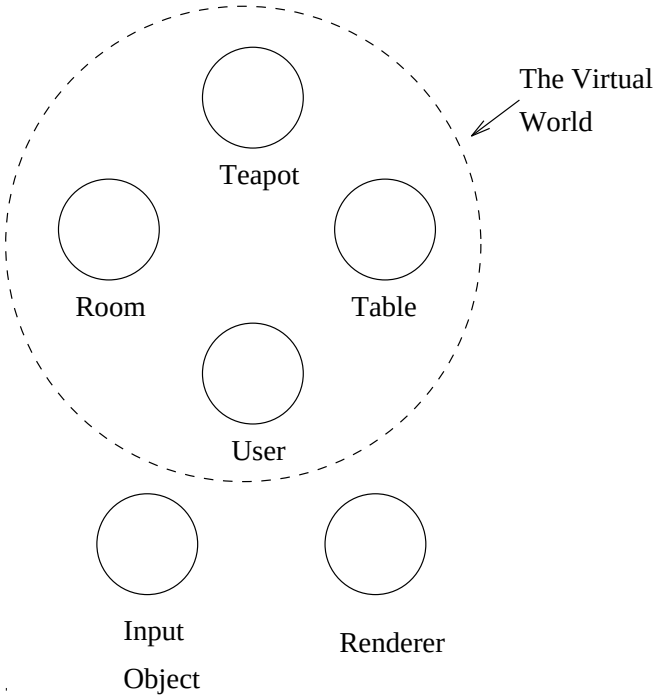

Figure 2: AVIARY representation of the Simple Virtual World.

demon may give it a complex behaviour.

Users are represented in exactly the same manner as other artifacts. A demon representing a user will typically communicate with input objects (which provide an interface to input devices). The demon will have a behaviour which translates user input into actions in the virtual world and which communicates with output objects (which control output devices) to give the user feedback.

Figure 1 shows a virtual world that is very simple, but sufficient to communicate the distinctions with which we are concerned. The world consists of a room with a table, a teapot on the table and a user. Figure 2 shows how this world is represented in the AVIARY model.

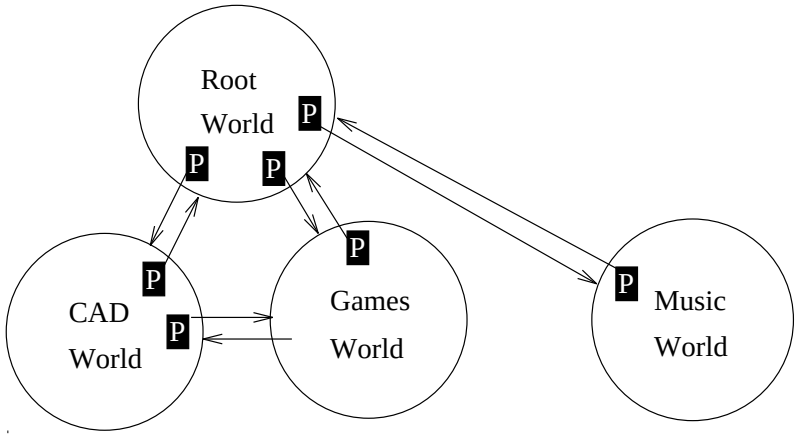

Figure 3: Worlds connected by portals.

This world is represented by four demons: the room, the table, the teapot and the user. The room has very minimal behaviour – it contains only enough code to display its walls. The table and teapot are slightly more complex and can be moved by the user. The user demon controls an input object (a data-glove, for example) and interprets the input as commands from the user which cause it to perform some action in the virtual world. It also controls a renderer which displays the world to the user.

Clearly a collection of interacting demons is not sufficient to create a Virtual Reality system; other objects are also necessary both to provide required functionality and to make the system more usable. Input and output objects have already been mentioned. Other objects are also present which provide services to client objects and these will be described in the implementation section.

4 Handling multiple virtual worlds

As emphasised in the introduction, the aim of AVIARY is not to demonstrate the feasibility of VR interface technology per-se, which is assumed, rather it is designed to provide generic, high level support for large scale, real applications. As a variety of world models are required, an interesting aspect of the design is the concurrent behaviour of multiple virtual worlds.

As each world has a set of properties and constraints on those properties. Each artifact in a world will contain the properties specified by the world and conform to its laws thus presenting a consistent interface to the user. Multiple worlds may be concurrently active, rather like a Graphical User Interface where several windows may be open, each one running a different application. Using this analogy the presence of a user in a virtual world is equivalent to the input focus of a Graphical User Interface on a particular window. Since we have a system which may concurrently support a number of possibly different virtual worlds, the question arises whether users may freely move from world to world. We believe this is a necessity and to do so is entirely consistent with the philosophy of the model. Special objects called ‘portals’ are provided to support movement between worlds. A portal allows a user, or any other object, to move into another virtual world. Only one-way portals are provided as primitive objects since two-way portals can be constructed by placing a one-way portal in each of the two worlds to be connected.

Portals allow virtual worlds to be connected in a directed graph, as shown in Figure 3, where the portals are indicated by ‘P’. For example a hierarchy could be built using a ‘root’ world with portals to worlds representing certain categories (such as CAD or games). These ‘sub worlds’, no different in implementation to any other world, contain portals to the worlds in the particular category along with portals which allow the user to get back to the ‘root’,

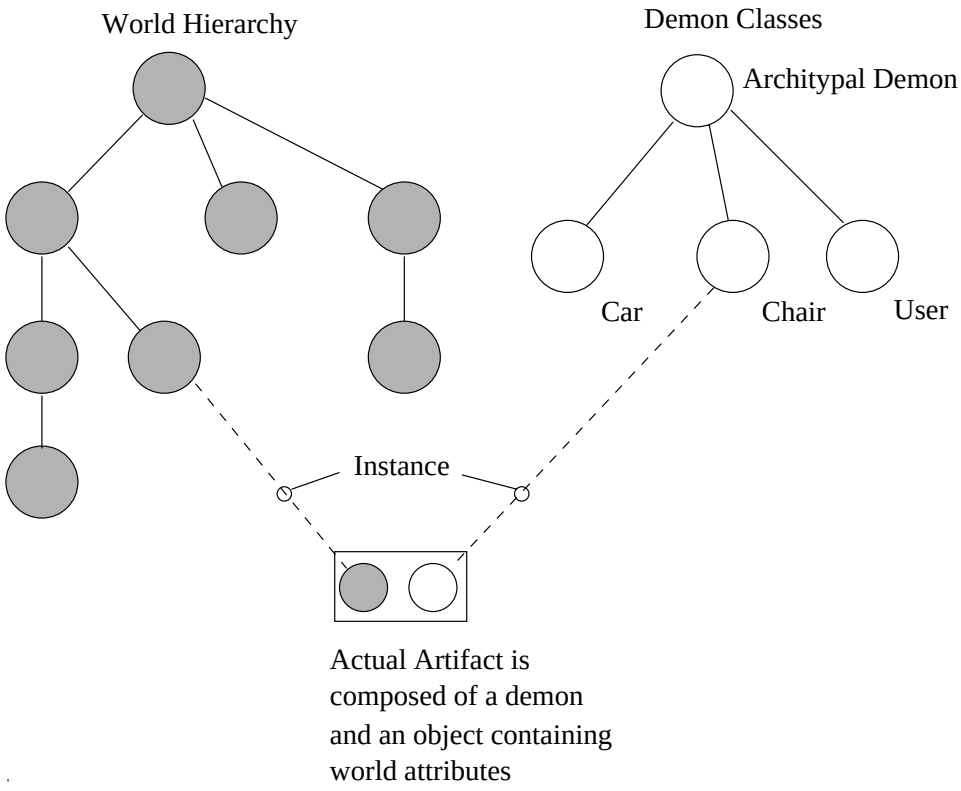

Figure 4: The structure of an artifact.

and the next level back up the hierarchy. Portals can appear as artifacts in a virtual world, they may have a simple graphical representation, such as a door, or a more complex one, such as a window which displays a rendered scene of the the world to which the portal provides entry.

We must also consider how the hierarchy of virtual worlds is to be represented and how demons capable of existing in these worlds can be constructed. The decision to permit movement between worlds does, however, lead to a number of potentially difficult situations. One particular problem is how an artifact (such as a user) in one world may be moved to another and behave in accordance with the laws of the new world – although clearly it may not be sensible or desirable to move some artifacts into a world with completely different laws. Also, some applications may not want their artifacts to be moved into another world; in this case the demon controlling the artifact may protest if an attempt is made to move it to another world.

One solution is to represent the hierarchy of worlds as a multiple inheritance lattice of classes. Each class representing a world defines the set of (possibly inherited) attributes that an artifact in that world may possess and methods which operate on them in accordance with the laws of that world. Demons are defined by other classes, which define the attributes (such as a graphical representation) specific to that class of demon. A demon contains a reference to an instance of the class representing the world it is currently in, as shown in Figure 4. When a demon moves between worlds an instance of the world class of the new world is created for it. The new instance replaces the existing one, attributes which the worlds have in common are copied, and suitable default values are given to the attributes not possessed by the demon in its previous world.

5 The implementation of AVIARY

Since Virtual Reality has a high computational requirement, AVIARY is designed to allow the exploitation of parallel hardware or distributed systems. In order not to place too many

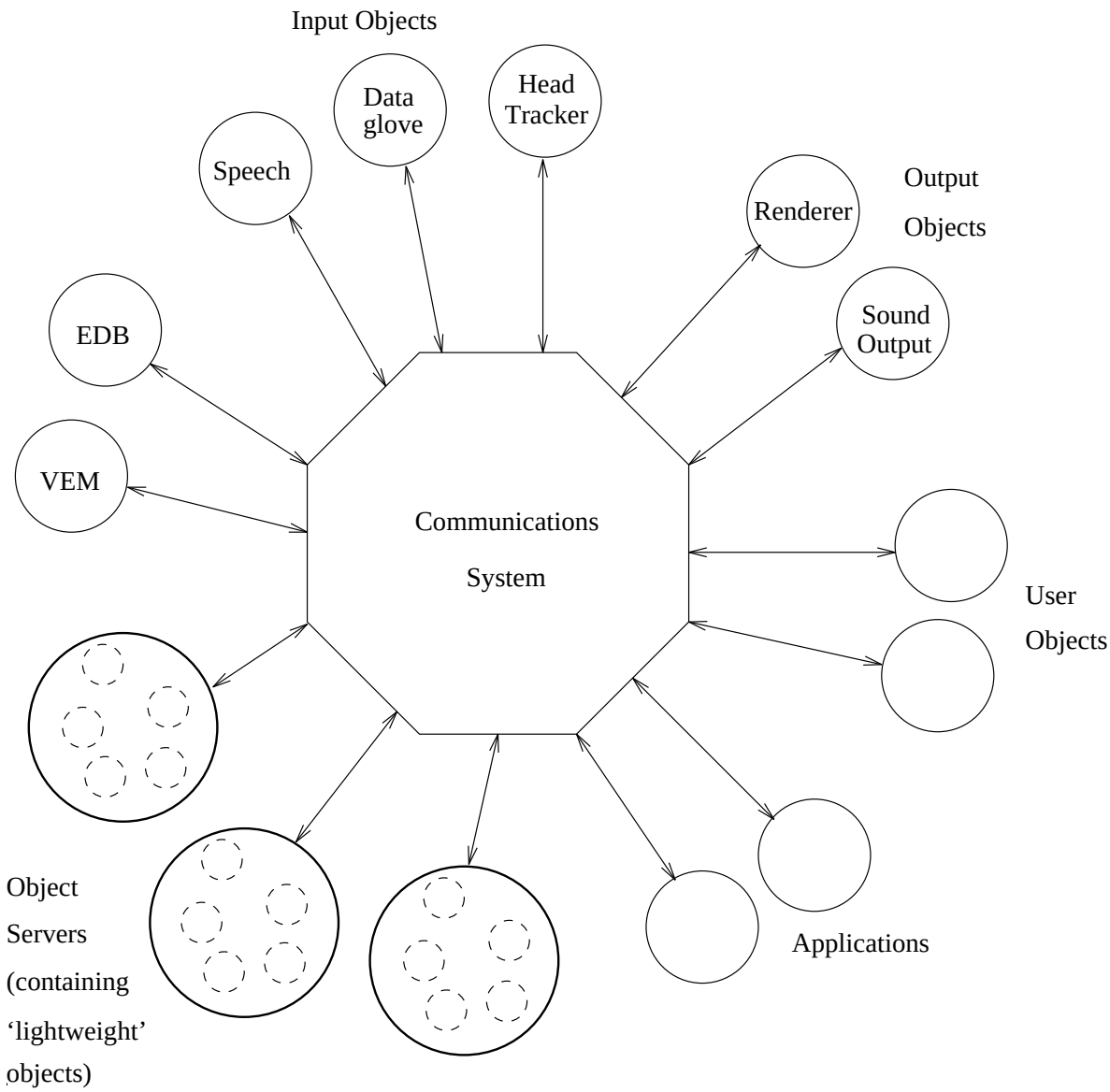

Figure 5: The main components of AVIARY.

restrictions on the nature of any particular kind of parallel hardware, the implementation assumes little about its capabilities. Specifically, there is no requirement for shared memory. This should allow the software to map onto a variety of hardware architectures.

Figure 5 shows the main components of AVIARY, which are connected by the communications system in the centre of the diagram. The other main components are the Virtual Environment Manager, the Environment Database and the Object Server. The main components of AVIARY are described in the following sections.

5.1 The Communication system

The implementation is composed of loosely connected objects which can execute concurrently. On multi-processing hardware objects may reside on different processors. Objects communicate via the communication system which allows them to send messages to one another without needing to know on which processor the object is currently residing. If objects migrate between processors, then their movement is tracked by the communication system which will automatically route messages accordingly.

Although figure 5 shows the communications system as a central entity this is purely for clarity of presentation. The actual implementation is completely distributed, thereby avoiding the possibility of bottlenecks occurring. The communications system tracks object

creation and destruction allowing objects to be created dynamically when needed.

Apart from device drivers, the communication system is the component which is most likely to vary in implementation on different architectures. On shared (or virtual shared) memory architectures the implementation of the communications system should be much simpler, since the message is not actually transmitted and the system simply needs to keep a queue of messages for each object. Implementations of the communications system currently exist for a network of transputers and a network of SUN workstations connected by ethernet. Bridge processes have been written which allow messages to be exchanged between the different networks.

On distributed (or loosely coupled) systems, messages between objects must be converted to a stream of bytes by the sender and the data structure re-created by the receiver. AVIARY utilizes a system (described in [2]) which allows this process to happen automatically for arbitrary data structures.

5.2 The Virtual Environment Manager

The Virtual Environment Manager (VEM) is a non-replicated object which provides services which must be consistent throughout the system. Ideally the services provided should not require much processing or communications bandwidth in order to prevent the VEM from becoming a bottleneck. One such service generates unique identifiers (IDs) for objects in the system. Upon object creation, the object creating the new object makes a request of the VEM for a new object ID. This is then assigned to the newly created object. It is necessary for object IDs to be unique because the communications system uses them to address messages between objects. If AVIARY were to be expanded to cover a Wide Area Network then having a single VEM would be impractical. In this case each site might have its own VEM which would prefix IDs with a site address to ensure that IDs were still unique across the network.

Other services provided by the VEM include notifying holders of shared objects of updates to the objects, and assigning integer identifiers to message selectors.

5.3 The Environment Database

The Environment Database (EDB) is an object which provides some ‘spatial management’ services to other objects. It performs coarse collision detection when demons change position and informs the objects involved if a collision occurs. Renderers can also use the EDB to determine which artifacts are visible from a given viewpoint.

There are several reasons for using such a centralized collision detection service. A major reason is this prevents the need for each demon to inform every other demon in the world when any attribute (eg position, size, shape, orientation) of the artifact it represents, that affects its relation to other artifacts in 3D space, changes. Also, each demon can operate independently until it is informed that a collision has occurred thus simplifying the implementation of demons and preventing waste of compute power by replicating the same collision checks every time an artifact moves. Because different artifacts may react differently when hit the code that handles collisions must reside in the controlling demon rather than centrally in the EDB.

To prevent the EDB from needing to perform n^2 collision tests, for n artifacts, the EDB organises the volumes containing the artifacts into a hierarchy of bounding extents of the type described in [3]. These extents use sets of planes to enclose an object and can be made to fit arbitrarily tightly for convex surfaces. This use of a hierarchy of bounding extents is similar to that described in [4] - where bounding boxes are organised into a binary tree with leaf nodes containing the bounding spheres of objects. Unlike [4], only one form of

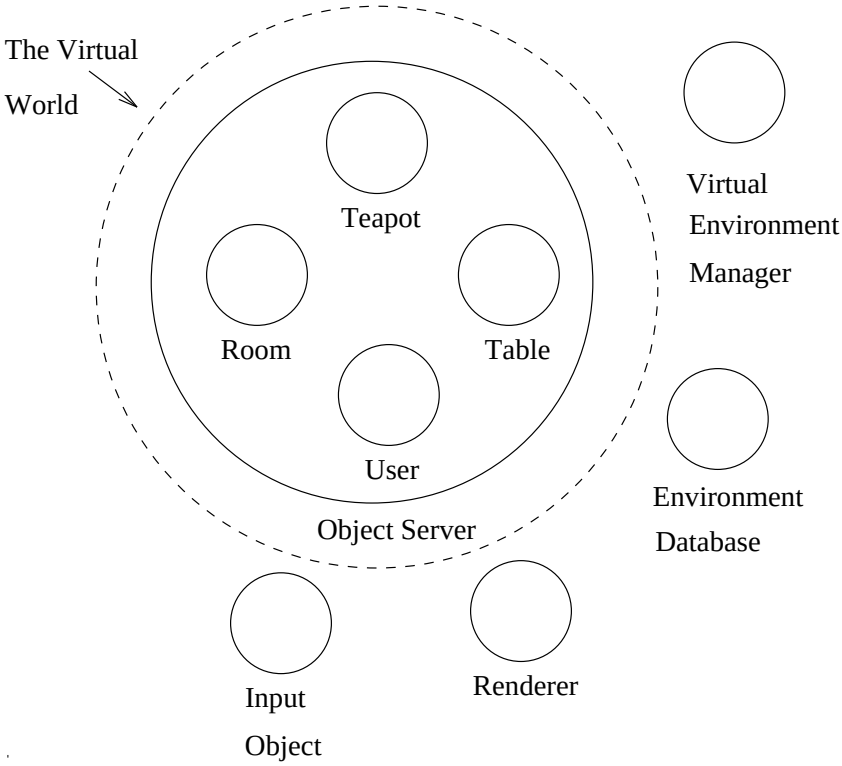

Figure 6: A more detailed representation of the Simple Virtual World.

extent is used and the tree is traversed depth first rather than via a list threaded through the nodes of the node when performing updates.

Collision checking is only performed when a demon sends a message informing the EDB that an artifact has moved, so that the extent of the moved object need only be checked with a small number of other extents. If an extent collision has occurred then the demons involved are notified by the EDB. It is then up to the demons how they handle the collision and whether any more detailed collision checks are performed.

5.4 The Object Server

Although it is quite possible and in some instances may even be desirable, to implement a demon from scratch, it would be better if the environment provided some means to ease this task. This is the function of the Object Server, which provides an execution environment for demons. It receives messages from other objects, handles execution of the relevant methods and provides simple scheduling facilities. The Object Server can be replicated as many times as desired. Using facilities provided by the Object Server, 'lightweight' object classes can be defined using multiple inheritance. Classes defined in this way only need specify data required for each instance and the actions required when a message is received.

Figure 6 shows how the representation of the simple virtual world might look using the Object Server to provide an environment for the demons. Also shown are the other main AVIARY components that are needed to make the system work – the Virtual Environment Manager and the Environment Database.

Of secondary importance to the system architecture are components including input devices which monitor real-world inputs, output devices such as renderers (significant though the issues or bandwidth are), and sound.

In addition, there may be one or more applications which create and manipulate demons to provide their interface.

5.5 Distributing the virtual environment

Although AVIARY is designed for distribution, introducing parallelism into the implementation creates problems, the most significant of which is the possibility of several concurrent processes requiring simultaneous access to the same data. A common situation might be that the application wishes to manipulate an object, at the same time that a renderer process needs to interrogate the object in order to display it. This problem can be partially alleviated by careful design of messages between objects. If updates are sent as single messages (rather than as a collection of simpler messages) then objects are less likely to be 'visible' in indeterminate states. Care must also be taken with the implementation of objects to prevent invalid states becoming 'visible' to other objects.

Another class of problem is due to the communication delays between objects on remote machines. These issues have been addressed in distributed object-orientated systems and solutions include migrating objects so they reside on the same machine and allowing multiple copies of objects. A description of a distributed Smalltalk system having some relevance to this problem is given in [5]. Another possibility for improving performance where an object's state is changing in a predictable manner for a period of time is for remote objects to employ 'dead reckoning' techniques to produce the objects state.

For example: an artifact is moving at a constant velocity. Its demon sends the current position of the artifact and a vector specifying its velocity in 3D space. Other objects then remotely compute the artifacts instantaneous position until the demon sends another position update which synchronises the objects. This approach would be suitable for certain artifact behaviours, and allows updates to be sent at lower frequency than would be acceptable if the intervening states were not calculated, thereby reducing the load on the communications system. This could be implemented by exporting explicit models of object behaviour which are then simulated or by employing 'proxy' objects which model this behaviour but are executed as ordinary objects. Both solutions would require changes to the way in which objects communicate and research is underway to evaluate these techniques.

5.6 The programming environment

There are several interesting constraints on the programming environment of the system itself that are worthy of discussion. In order to get the necessary speed it is essential that the core functions of the system should be written in an efficient compiled language. We would also like to be able to write the definitions for both the worlds, and the artifacts within them, in a language which provides support for object-oriented constructs. To ease prototyping it would be convenient if we could alter methods associated with worlds and objects at run-time. Also, we do not want to have to re-compile portions of the core system in order to change the definitions of worlds and their objects.

Since the only communication between objects is by message passing, as long as a single message format is used objects in the system need not be defined in the same language. Thus the standard system objects (such as the EDB, VEM, Object Server and renderers) could be written in an efficient compiled language and the world and artifacts within them could be described in another language which would provide an interface to the facilities provided by the Object Server. If the language used as this interface is interpreted or incrementally compiled then code fragments may be sent as messages between Object Servers allowing code updates to be sent as an ordinary communication.

Using an interpreted language would seem to sacrifice performance in order to gain flexibility. However, this need not be so. Common functions are made available as primitive operations. Functions written in the interpreted language then 'glue' together these efficient building blocks which implement computationally expensive operations.

6 Conclusions

In this paper we have presented AVIARY, the software architecture of a system providing high level virtual environment support for substantial, real applications. Our approach provides a generic extensible environment, onto which a range of applications, both existing and anticipated, can be mapped. The implementation has been presented which is based on objects which communicate by message sending. The implementation allows objects to run on a heterogeneous network of processors. Currently SUN workstations and INMOS transputers are supported, but it is planned to extend the implementation to support other architectures such as the KSR 1 at the University of Manchester's Centre for Novel Computing.

Acknowledgements

It is a pleasure to be able to thank our colleagues, who have generously shared with us their insights and enthusiasm. We are grateful to Jim Garside, Alan Murta and Roger Hubbold. Special thanks to Gareth Williams for his kind help in preparing Figure 1.

References

[1] A.J. West, T.L.J. Howard, R.J. Hubbold, A.D. Murta, D.N. Snowdon, and D.A. Butler. AVIARY - a generic virtual reality interface for real applications. In *Virtual Reality Systems, sponsored by the British Computer Society,*, May 1992.

[2] Peter J. Crowther. Forage. Project report, Department of Computer Science, University of Manchester, April 1989.

[3] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *ACM Computer Graphics*, 20(4):269–278, August 1986.

[4] Robert Webb and Mike Gigante. Using dynamic bounding volume hierarchies to improve efficiency of rigid body simulations. In *CGI'92 in Tokyo*, 1992.

[5] John K. Bennett. The design and implementation of distributed smalltalk. In *OOPSLA '87*, pages 318–330, Seattle, WA 98195, October 1987. Department of Computer Science, University of Washington, ACM.