ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

DEVA: An Operating Environment for Large Scale Virtual Reality

Steve Pettifer and Adrian West

Department of Computer Science

University of Manchester

Technical Report Series

UMCS-99-10-1

DEVA: An Operating Environment for Large Scale Virtual Reality[1]

Steve Pettifer and Adrian West

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
srp@cs.man.ac.uk

15th Feburary 2000

[1]Refereed by: Jon Cook

Abstract

Improvements in processors and graphics hardware are making Virtual Reality (VR) increasingly attractive as means of human/computer interaction. Although there are compelling stand-alone demonstrations of specific aspects of the field, there is little in the way of generic software that supports and integrates the needs of large or complex virtual environments. Future VR systems will require 'large scale' issues to be addressed. These include complex graphics and behaviour of objects and large numbers of applications and geographically distributed users.

However, beyond the graphics and networking challenges, a core issue is the specification of the environments themselves. For sophisticated environments, more appropriate methods are required than the conventional programming tools provided by existing VR systems. For virtual environments, this specification approaches a description of the metaphysical properties of the space being described and the objects it contains. In such a specification, the relationship between the user and the environment is an important consideration.

This thesis argues that by building a framework for describing virtual environments upon such a metaphysically orientated specification of the environments, that takes explicit account of the relation between perceiver and the perceived, many of the existing problems of VR can be successfully addressed.

A VR systems architecture is developed that supports investigation of this perceptual divide between perceiver and perceived, and is demonstrably capable of facilitating these investigations in the context of 'large-scale' VR as defined above.

Chapter 1

Introduction

This technical report describes Deva, a multi-user Virtual Reality (VR) system aimed at supporting "large scale" Virtual Environments (VEs). In this report we say why we believe VR is important; define "large scale" VEs and the challenges they raise; and report upon the design and implementation of a system we are constructing to address these challenges.

In designing an earlier system [Sno96], we observed that many important optimisations are highly application specific, and no one choice of internal data struture for the VR system could easily accommodate them. Importing application data into the VR system lost the direct access to application specific knowledge needed for some substantial optimisations. Furthermore, interactions undertaken in the VR system required the application to mirror the VR systems data structures so that the VR system and the application could remain synchronised. For example, editing objects in the VR systems PHIGS hierarchy required those edits to be mirrored in the application, which thus had to maintain a similar hierarchy.

These difficulties at the graphics, interaction and spatial management level, led us back to the drawing board to engineer an approach to these issues. The resulting system, the Maverik VR micro-kernel forms the basis of our larger scale VR systems venture - DEVA - described in this document. Maverik frees applications from the burden of conforming to the internal structure of the VR system, by using the application data structures directly. Maverik has been released Freely as part of the GNU project.

Maverik provides the support for graphics, spatial management and interaction - a role traditionally tackled by graphics systems, and thus provides for a "view onto the world". This aspect of the VR task has been deliberately separated from the semantics of the worlds content: namely the behaviours of the objects it contains, the impositions of laws and constraints, the distribution of the world amongst many participants, management of multiple applications, and the management of multiple worlds that simultaneously reside in the system. It is this management of "content" that is the role of Deva.

At the time of writing, Deva's framework is in an advanced state of construction, and has been used for public demonstrations of shared VEs at several European exhibitions [1]. In this introductory chapter we describe our motivation for VR as a tool in HCI, and identify the vision of "Large Scale VR" which our work seeks to address.

1.1 What Virtual Reality brings to HCI

Any interaction with a computer is in some sense an interaction with a "virtual reality", a fact not wasted on those who market software. However, the common expectation raised by the term is of at least a 3D interface. 3D presentations are by no means new, and this raises the question of how VR is distinguished from what for the past few decades has been known as "interactive 3D graphics". Research in interactive 3D graphics is important, but we must identify the distinguishing features of VR from the outset in order to avoid a great deal of confusion, and to clarify our aims.

[1] see http://aig.cs.man.ac.uk/systems/Maverik/ledgible.html

As terminology is hotly disputed, one way forward is to identify features of VR that are genuinely novel, which facilitate activity that cannot be achieved by other means. The area of VR where such a consensus exists, is where users are "immersed" in the environment by means of a head-mounted display and tracking system. This is therefore an appropriate area to seek the defining characteristics of VR, and hence to discover what it may have to offer that is a genuinely new contribution. This is be no means to say that VR is exclusive to immersive technology, or that a 3D desktop is not "VR", for clearly there is significant overlap of concerns in these related environments. However, the immersive environment is that in which the unique character of VR is most readily apparent, and where distinctions are most readily apparent.

In immersive VR it is possible, in principle, to acquire psycho-motor skills, such as learning to juggle or learning to ride a bicycle; this is clearly not the case with a desktop screen/keyboard/mouse. Manipulations can be performed in new ways by "directly" interacting with objects in the scene; reaching out, picking up, and manipulating items with actions that are physically more akin to interaction in the real world than they are to traditional mouse/keyboard interactions. The user can thus bring far more of their psycho-motor skills to bear on the interaction. For example, if the head is turned through 180 degrees, the user "feels" that the objects they were looking at are now "behind" them, and hence spatial awareness within the environment is improved. VR is in this respect more akin to flight simulation, though goes further by allowing the user to leave the cockpit and experience their body as being part of the simulated environment. Hence, genuinely new modes of interaction are available, though the technology itself remains that of 3D graphics augmented by position sensing.

A number of points can be drawn from these observations. Firstly, there are novel possibilities for HCI offered by VR that are not available in other technologies. Precisely how valuable these will be, and what kinds of activities will benefit from them, is still largely a matter of conjecture; that they are novel is however undisputed. An obvious supposition is that tasks that are naturally cast as taking place in a 3D environment with naturalistic interaction, such as design review, or team rehearsal of maintenance tasks, are likely candidates to benefit from the VR approach. Second, VR is characterised more by new psychological (and perhaps sociological) phenomenae, than by technology. It is the depth and "quality" of experience made possible by being "present" in the environment that is potentially radical. One psychologist has characterised the distinction as the difference between "standing on the beach watching someone swim, and actually swimming". These are substantial new factors, but are difficult to measure or analyse quantitatively, and hence difficult to discuss. Thirdly, if the new features of VR are predicated on the quality of psychological "immersion" or "presence", then the factors central to maintaining this experience are of key significance. For example, one may be led to hypothesise that quality of rendering is secondary to continuity of experience: people can "get by" without their glasses, but an immersive experience of a world that learches along at 1 frame every few seconds is unacceptable. Thus, interactive performace becomes more critical for immersive VR than it is for the 3D desktop, or for VR's "near neighbours" of physical simulation modelling and computer animation, being more intimately bound to the requirements of direct human interaction. Techniques used to create the virtual environments are at least bounded by significant real-time constraints, and by factors of human perception and interaction not yet fully understood.

1.2 Large Scale Virtual Environments

Virtual Reality is still a new area of research; the computing power necessary to generate non-trivial virtual environments has only recently become commonly available. Hithertoo, the preoccupation of VR researchers has been to demonstrate the possibility of useful VR, and hence with achieving presentations of VEs at interactive framerates through immersive peripherals. These crucially important issues of presentation are however only one side of the story. Given the ability to perceive a VE, we are subsequently faced with the issue of meaningful content for it. The issue of what is being perceived, and what can be done with it, becomes a central issue of interest. The challenge becomes the modelling of the realities that are to be perceived, and this is a substantial undertaking. To appreciate the complexities of this task it is necessary to investigate the requirements that a system supporting future VR based applications will need to fulfill. We characterise these requirements as "Large Scale" VR, in the sense of being substantially more complex and demanding than current demonstrations in the following specific ways:

Complexity of graphical display. Future VR applications will surely strain the limits of a systems ability to cope with the graphical complexity of their environments. Though the illusion of complexity can be achieved with "wall-papering" techniques such as texture maps, this is not adequate if the VE is to be composed of real objects offering real possibilities for interaction. Real geometry must be available for a much larger portion of the visual complexity of the scene.

Current systems rely on importing geometric (and occasionally behavoiural) models from such formats as VRML and VRML97. These models are predominantly represented by the universal currency of polygonal meshes, with behaviour defined by scripts and animations which are imported into the VR system. Models of an industrial complexity, such as process plant, rapidly swamp a VR systems ability to cope with the number of polygons that it must import. An oil rig model from the VRLSA project represents over half a gigabyte of polygonal information. Applications that cannot readily export or update a polygonal representation of the requisite complexity, such as those dealing in deformable surfaces or volumetric techniques, are poorly catered for by the paradigm of exporting polygonal models into the VR systems own data structures.

Number of objects in the environment The world of our everyday experience is characterised by large numbers of objects being available to us, each offering the possibilities for interaction. Future VR applications will surely desire to emulate this richness, populating environments with many different forms of objects simultaneously. One current limitation is that existing formats for scene description do not encourage the use of the optimized spatial management routines that are necessary for VEs with large numbers of heterogeneous objects.

Number of applications represented. In the 2D desktop environment we understand the need to run several applications simultaneously, and to provide methods of interworking between them. As VR comes of age, it will surely move away from the notion of a single dedicated application, and support multiple applications and interworking between them. Unlike the 2D desktop that we can oversee from our 3D perspective, in a 3D environment we do not have the possibility of a 4D overview of the many applications that may be active. New metaphors will therefore be required for interworking applications.

Richness of behaviour. Currently, the behaviours of objects in VEs are described in terms of geometric transformations, and are often defined on a per object basis. Another common mechanism is to provide hooks for the programmer to attach code (e.g. TCL scripts or Java) to individual objects. This works well in small-scale environments with few objects, or limited scope for behaviour. However it does not scale well to more realistic application environments - an average room contains far too may objects and opportunities for interaction than could readily be coded by scripts attached to each of them. Complexity of behaviour in the VE is thus about making large numbers of sensible interactions possible. This kind of broad behavioural complexity in the VE can be distinguished from the focussed behavioural complexity desired of an AI. For example, our concern in designing systems to support future VR applications is not that a model robot in a VE has complex behaviour, as that is an application issue, rather it is that the environment can easily be made to offer more than just the movement of the robot and its four play blocks.

Number of users. A particular strength of VR is the possibility of sharing a VE with several users; the sense of "presence" for which VR researchers strive is also a key to interactions between people. Thus VEs offer a vehicle for cooperative working that is fundamentally different to video conferencing.

The inclusion of more than one user in an environment introduces issues of communications and parallelism. More subtle are requirements for social interaction in the VE. Increasing the number of users increases the complexity of social situations and demands more sophisticated support by the enabling technology.

Geographically dispersed users. If shared VEs can be made effective, then there will surely follow a call to allow participants to be located at different geographical sites. As with video conferencing this permits shared activities to be undertaken without the need for travel, but the range of tasks that can be engaged in is greater. For example remote shared rehearsal of procedures before they are undertaken at the remote site.

Fundamental limitations in networking technology introduce insurmountable delays in communication between geographically distant sites. In many distributed systems, small delays can be tolerated, however in virtual reality where the intention is to represent a credible 'reality' with all its realtime constraints, network delays can cause significant discontinuities in experience: permitting two users located at different poles of the planet to share a VE in which they rehearse moving a ladder though a tight passageway presents fundamental challenges.

We envisage that future VR systems will be called upon to provide the above facilities in large order. This is what we refer to as "Large Scale VR".

Some of these issues are actively being investigated in the MAVERIK VR micro-kernel, which aims to deal with complex rendering and spatial management for virtual environments. MAVERIK is now available as Free Software and is part of the GNU project. The MASSIVE2/CVE being developed by The Communications Research Group at The University of Nottingham, UK, concentrates on issues of social interaction in virtual environments.

This report focuses on the design and development of a software infrastructure called DEVA that complements Maverik by addressing issues specifically concerning the description of behaviour, the management of multiple applications, and the geographical distribution of users. A particular focus of the Deva project is to provide a single framework that can simultaneously cope with the above demands. This work is therefore motivated by full scale "systems architecture" arguments, which are quite different to a series of dedicated prototype systems, each addressing one issue. We have adopted this strategy because we believe the central challenge is architectural - finding ways in which it can "all be put together".

1.3 Report structure

Following this introduction, the report is contains 4 chapters:

Scoping an architecture for VR in the large. describes the particular approach taken in the design of the DEVA VR system. Rather than begin with a discussion of available technology, our starting point is that of end users and their interaction with a virtual environment. In this sense the design is very much 'top down'.

Server, Clients, Infrastructure, Content. There are four major components of DEVA, and these are described and the relationship between them elaborated.

Implementation. A detailed examination of the techniques for implementing the DEVA.

Summary and Conclusion. An overview of the features of the system in its current state, and examples of the system in use.

Chapter 2

Requirements of a large scale VR system

This chapter defines the requirements for a VR system aimed at dealing with issues of 'scale'. 'Large-scale' issues in this context include graphical complexity, number of users and applications, geographical distribution of users and the complexity and richness of the resulting environments.

From the each of the preceding chapters, an observation can be drawn. First, that for a field with apparently so much potential, contemporary VR demonstrations tend to be of the nature of a 'walk-through'. Though a user is able to browse an often (apparently) complex three-dimensional model, little other interaction with the world is possible. Second, that in spite of the diverse architectures of the VR systems used to build such demonstrations, there is little architectural support for behaviour, dynamics or inter-working between applications. There is a clear link between these factors: lack of support for world models within the systems leads to virtual environments with limited behaviour. Most of the systems surveyed have reasonable support for the interfacing of existing applications and peripherals, and extensive support for the presentation of virtual worlds. The problem is however that such support tends to be built around 'programming models' rather than 'world models' *per se*, and that, while these programming models satisfy such technological requirements as supporting parallel processing, these are not convenient mechanisms for expressing the higher-level issues that are important for the description of interesting virtual worlds. In a sense, the programming models are too low-level, and using them the effort required to describe a 'fully-functional' world where all objects are available for inspection and interaction is great. The tendency is therefore to code for the minimum behaviour necessary to demonstrate a particular point.

This chapter outlines the requirements placed upon a VR system arising from the need to induce the experience of presence. These requirements are relating directly to implications for software and hardware architectures. From this will be developed,in the following chapters, a new VR system designed specifically with these requirements in mind.

2.1 The world model

Of the three components of a VR system outlined in the previous chapter, it is on the development of a suitable world model that attention must be focussed first, since it can be seen as the weakest element of most existing systems. Issues concerning the other two components of interfacing and presentation are generally better understood, and their exact nature will in any case be affected by the world model that they are to support. Thus they will be examined in later sections.

The quality of a world model is affected by two independent factors: first, the degree to which it provides a mechanism for coherently, succinctly and elegantly describing a virtual world and its contents, and second the pragmatic requirement of executing such a world efficiently on the available computer hardware. Existing systems could be said to give favour to the latter, requiring application programmers to describe their conceptual model of a virtual environment in terms that are appropriate for efficient execution rather than eloquent expression. The model developed here puts the needs of the 'world builder' first, bearing in mind however that the final effectiveness of a VR system ultimately involves the satisfaction of the technological requirements too.

2.1.1 The Deva world model

The model is based on a conceptual framework and language called 'Deva' described in [Pet95], which was in turn influenced by the ideas developed during the AVIARY project [Sno96] . The model was, from its inception, intended as forming the basis for such a system as will be developed in this thesis, and hence its inclusion as a starting point here is not accidental. Covered in detail in [Pet95], the key points and arguments will be summarised here for completeness and their significance to the implementation of a supporting system will be highlighted.

Rather than being addressed directly at technological issues, the Deva world model is based on a particular interpretation of our experience of the everyday world. Though not necessarily a breakdown that would stand up to too much in-depth philosophical scrutiny, the model attempts to provide a consistent narrative and working vocabulary with which worlds of various natures can be described.

The model divides our surroundings into 'environments' within which there are 'entities': in everyday terms 'places' in which there are 'things'. Though in some ways a rather artificial distinction, this splitting of a world into environments and entities has certain benefits that will be described in the following sections. Before doing so however, the programming model around which the world model is built should be outlined since the two are tightly interwoven.

VR and object-orientated methods

The language and programming paradigm used to describe virtual environments using Deva is object-orientated, and is similar in many ways to other general purpose object-orientated languages. The paradigm supports the conventional class based object descriptions, instantiation, encapsulation and multiple inheritance found in classic object-orientated languages. In particular, its syntax is closely related to that of SmallTalk80 [GR83], though such features as multiple inheritance are drawn from C++ [Str86]. In some respects the particular syntax chosen for the language is almost arbitrary, and other appropriate syntactic styles for describing the concepts of the Deva structures could no-doubt be devised. The choice of the object-orientated paradigm however, is significant, and can be supported thus.

The argument is simply that much of object-orientated technique, whether applied to analysis, design or programming, has stemmed from a desire to formalise our everyday model of reality in such a way it can be used as a mechanism for describing conceptual issues, and in particular those related to computer programming. For example, consider the phrase 'drive the car'. Without requiring a detailed knowledge of the internal structures of the vehicle, someone who understood the meaning of 'car' and what 'drive' meant in relation to it would be able to comprehend the instruction. In a similar way, object-orientated techniques allow the encapsulation of arbitrary concepts inside 'objects' that hide the internal structure and provide instead a clean interface describing only the available functionality of that object. Hence an object representing a linked list could respond to the instruction 'add item to list' and would reject instructions to 'calculate the square of' or 'draw a window'. Object- orientated techniques reflect our understanding that the world consists of 'things' each which afford different features, i.e. one can drive a car, read a book, iterate through a list or increment a counter.

Applied to virtual worlds, the object-orientated technique comes full circle. Rather than being used as a mechanism for describing programming concepts using 'day-to-day' metaphors (i.e. the encapsulation of state and function is seen as being 'something that does things'), it is using these ideas to represent 'real things' once more. It is not by accident that most guides to object-orientated programming begin by using real objects such as cars and wrist-watches to introduce the reader to the more abstract applications of the technique.

Extending the object-orientated paradigm

Although innately suitable for describing many of the aspects of objects abstract, real and virtual, the Deva model includes a novel extension to the standard object-orientated paradigm that increases its effectiveness as a mechanism for describing the contents of virtual space. In summary, the language embodies the idea that many of the properties of an object in our real world can be seen as being the result of an interaction between the object and its surroundings, between, in the terminology of the Deva framework, entity and environment. For example, wristwatches and cars on the whole have different specific or intrinsic functions, but both are subject to the effects of gravity and Newtonian mechanics 'because they exist in our world'. This is of course, 'philosophically' a dubious distinction, since it can be argued that there can be no fundamental partitioning between 'our world' and 'the things in it' simply because 'our world' can be seen as being the 'sum of its parts'. Thus, while no suggestion is being made that this entity/environment split mirrors an absolute or fundamental division of reality, its resonance with an 'everyday' vocabulary for describing our environments recommends it as a model.

In standard object-orientated languages, such a concept could be expressed by giving both wristwatch and car objects common 'parents' from which they inherit the properties required to implement gravity and Newtonian mechanics. The disadvantage of this approach is that the objects then intrinsically have associated with them all the code and state necessary to implement these features, even in circumstances where they are redundant (for example, if the watch was to be taken out into space, the effects of gravity would not be felt, and the code for implementing the effect would have to recognise the change in environment, or more abstractly if some properties of the watch were to be represented in a financial ledger, almost all of its physical properties would be extraneous). For a single, on the whole coherent model of physics such as the one that is used to describe our real world, this might not pose a problem: in theory, given a powerful enough computer, objects could be written that inherit properties describing all the different aspects of physics, and would then behave sensibly under all conditions. For virtual reality however, the luxury of a single consistent 'physical model' does not exist, and the practical difficulty of finding a processor powerful enough to manage all the possible rules in a single monolithic structure is overshadowed by the more fundamental problem of conflicting application requirements. Since individual applications have the ability to, in effect, describe different sets of rules that apply to objects under different circumstances, if multiple applications are to be supported, there is the possibility of them requiring different and even conflicting world models. The search for a single set of rules that describe the properties of all possible applications would appear to be a futile one. Furthermore the kind of rules that affect the behaviour of objects in the real world are not just the laws of physics: objects have such attributes as 'monetary value', and people have 'credit ratings' and 'social status'. Such 'social laws' often have significant effect in our real world, and yet change value or relevance depending on the physical location of those objects.

The Deva model circumvents these problems by allowing environments to *imbue* entities with properties and to enforce rules that govern their behaviour. Hence entities are described in terms of their *intrinsic* properties, i.e. those attributes and functions which are always available, and that the entity has regardless of its situation or environment. Though described independently, entities can only be instantiated within environments, at which point further functions and attributes defined by their environment are attached. Although in some ways less in line with the reality of our world (i.e. there is no 'law of gravity' that acts independent or outside of the fact that objects have mass, rather what we call gravity is instead an intrinsic property of massive objects), the abstraction is nevertheless a useful one. The separation of intrinsic entity properties and environment imbued ones allows entities to describe their own unique functions whilst being imbued with useful properties common to all objects in their environment. For example this means that a car would define its vehicular properties, a watch would define its time-keeping functions and both would gain properties that made them behave according to the laws of gravity and Newtonian mechanics by virtue of having been created within a particular environment. Figures 2.1 and 2.2 illustrate this. Either object being moved from their initial environment to a new one would retain their intrinsic properties, lose their old imbued properties and be given a new set of imbued properties to bring them into line with the requirements of their new environment.

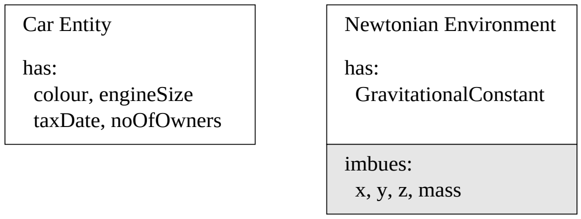

Figure 2.1: Entities define their own attributes and methods. These will always be available to the entity. Environments similarly define their own properties, but in addition may specify further attributes or methods that are to be associated with any entity that they contain.

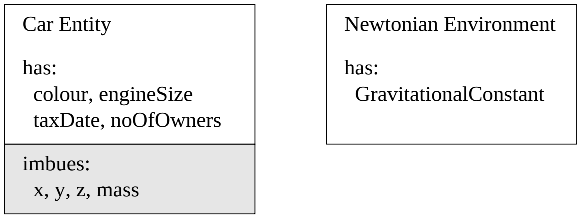

Figure 2.2: At runtime, when an entity is created within an environment, the environment grafts its 'imbued' properties onto the instance of an entity, ensuring that it has the methods and attributes necessary to participate in that environment

The makeup of an entity

With a few exceptions, the Deva model attempts to define as little about entities and environments as possible, and instead provides a framework in which the application programmer can create the kind of virtual world they require. The model provides entities with the notion of *attributes* to manage their state and *methods* to manipulate that state. Methods in general can be seen as representing the way in which that entity responds to external stimuli (e.g. a car can be asked to 'start engine').

The model defines three kinds of method which have particular purpose:

Prologue: If the entity has a method called *prologue* then this will be used to initialise the internal state before any further processing proceeds. This is similar to a constructor in C++.

Epilogue: Similarly, a method called *epilogue* will be called prior to an objects final removal from the system. This parallels the destructor in C++.

Behaviour: An entity's *behaviour* method represents its autonomous action. Any code described in such a method is to be executed continuously during the lifetime of that entity.

Example of an entity

The following fragment of code generates a new entity of class 'Ball' that inherits the properties of an abstract sphere, changes its representation to be that of a beach-ball, and adds an elastic constant to represent its 'bounce'.

First inform the 'Sphere' class that it has a new sibling:

```
(Sphere subClass: #Ball)
     hasMethod: 'collisionWith:';
     hasAttribute: 'elasticConstant'.
```

Next define the 'collisionWith' method for the Ball, just in case it ever hits anything (here it just prints "Pop!" and causes the object to vanish).

```
Ball defMethod: #collisionWith
for: 'Making the ball go pop and then vanish'
as:
[   collidedObject |
    'Pop!' printNl.
     self delete.
     ^ self.
].
```

Finally, define what happens when one of these objects is created:

```
Ball defMethod: #init as:
[
    super init.
    elasticConstant:= 10.
].
```

This is all that is required to create a new class of object that can be created and managed in any virtual environment within the Deva model.

The makeup of an environment

Being a specialisation of an entity, an environment has all the above features. In addition however, the definition of an environment includes attributes and methods which are to be imbued upon contained entities. Further, an environment can define *forces* which partially govern the behaviour of entities by manipulation of their imbued attributes (a force can be thought of as being a behaviour that is imposed on an entity by its environment). Hence an environment being a kind of entity contains definitions of its own intrinsic properties, and also those pertaining to its contents.

Example of an environment

A new environment is created by informing its parent that it has a new sibling:

```
(Holospace subEnvironment: #SolidWorld)
    hasMethod: 'intersection:with:';
    imbuesMethod: 'collisionWith:';
    imbuesAttribute: 'safeX';
    imbuesAttribute: 'safeY';
    imbuesAttribute: 'safeZ';
    imbuesAttribute: 'mass'.
```

The the environment defines what happens when its parent environment (i.e. Holospace) informs it via the 'intersection:with:' method that two objects have intersected. Since this is an environment modelling solid bodies, the environment causes the objects to stop, and to move into their last 'safe' position prior to the collision. It also then informs both objects that they have collided with one another, in case either should want to take a particular action.

```
SolidWorld defMethod: #intersection:with:
    for: 'causing objects that have collided to halt'
    as:
    [ object1, object2 |
         object1 x: (object1 safeX).
         object1 y: (object1 safeY).
         object1 z: (object1 safeZ).
         object2 x: (object2 safeX).
```

```
        object2 y: (object2 safeY).
        object2 z: (object2 safeZ).

        object1 collisionWith: object2.
        object2 collisionWith: object1.
        ^ self.
    ].
```

2.1.2 Virtual worlds from entities and environments

The model describes two tree-like inheritance structures. The first is rooted at the 'simplest possible kind of abstract virtual environment' and represents a gradual reification of that concept towards more and more complex environments along with their laws and properties. This structure is called the Metaverse Graph. The second is rooted at 'the simplest kind of abstract object' and represents the more common object-orientated progression from 'abstract base class' to 'concrete entity'. With the exceptions described above, the model prescribes little about either entities or environments, being instead a framework for further development. Rather than forcing every implementor using the framework to start from scratch however, the model populates the roots of both trees with useful entities and environments, the properties of which can be extended as required.

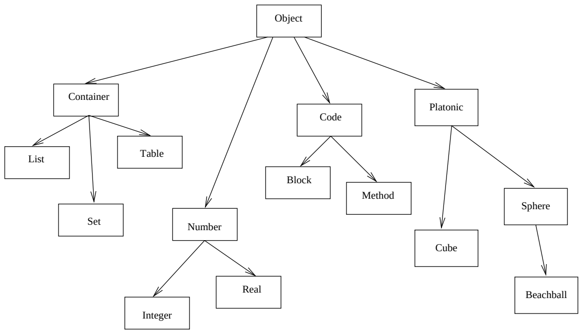

Figure 2.3: The Deva models supports the traditional type of 'class hierarchy' seen in the majority of object-orientated languages.

The more philosophical reasons for the particular content and structuring of both graphs are described in [Pet95]. However the practical considerations of this mechanism, which are but hinted at in that thesis, are also important and will be examined in more detail in chapter 4. A virtual world built within this framework consists of a single instance of a node from the environment graph, in which instance an arbitrary number of entities from the entity graph may be created. The intrinsic properties of the entities are grafted together at the point of instantiation with the imbued properties from the environment to create the final entity which then attempts to behave according to its own rules but within any constraints and according to any forces defined by its environment. Environments can be created within one another, with the same rules as other entities applying to their construction and behaviour.

There is no explicit mention of either user or application within the framework: in a similar vein to the AVIARY philosophy [Sno96] a user is simply considered to be an entity whose behaviour is controlled by input devices rather than algorithms, and an application is represented by an entity, environment or collection thereof that affects the behaviour of other entities or environments. The model is in this sense implicitly multi-user and multi-application. The specifics of interfacing these objects to external peripherals and applications is considered briefly in chapter 5.

To summarise, the Deva model of virtual worlds allows the definition in object-orientated terms of both the different kinds of virtual environment that may be required, as well as the different kinds of entities that may populate them. Although little is prescribed about either, a set of useful ('perception

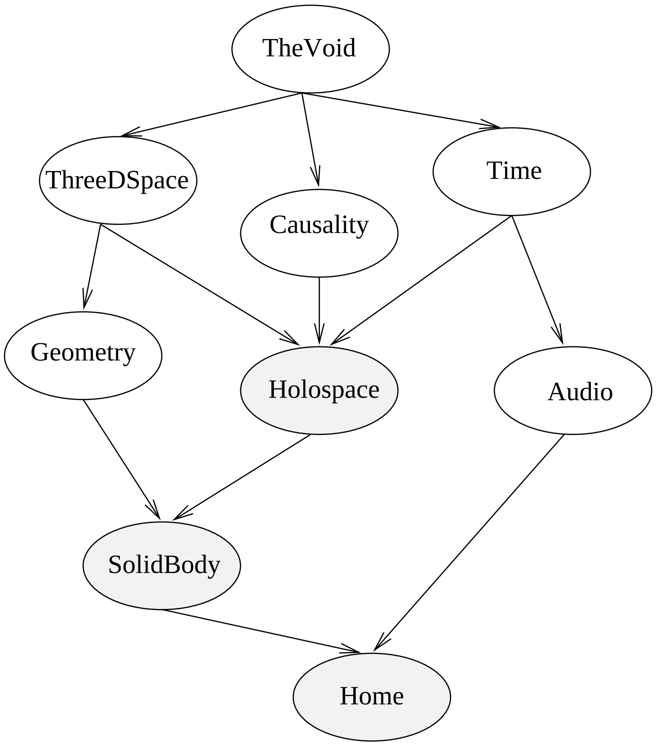

Figure 2.4: In addition to the traditional entity class hierarchy, the model supports a second environment class hierarchy, defining properties of the various environments in which entities may be created. Unshaded environments represent ‘abstract’ classes: that is environments where there are insufficient properties for a user to experience the existence of an entity.

friendly') abstract entity and environment classes are provided to act as a foundation. Entities may define autonomous behaviour, specific responses to particular stimuli, and then are imbued with further dynamics by their containing environment, providing both a richness of behaviour and the ability to adapt that behaviour in a consistent way to different environments.

2.1.3 The Metaphysical principle and system independence

There are two important aspects of this model that should be highlighted at this point:

- It is but one of many possible expressions of the metaphysical principle outlined in the previous chapter.
- It is system independent: the model is a conceptual one, and although it is described in object-orientated terminology, it could in theory be implemented on any platform (e.g. in principle, a layer could be added to the DIVE system to give it the functionality described in the model).

2.1.4 Practical considerations

Although the model is primarily a conceptual one, and as mentioned above could be used as an 'interface layer' to existing VR systems, it lends itself particularly well to a certain kind of architecture, aspects of which will be examined in the following sections.

Autonomy and multiple loci of control

Although the notions of entity, force and behaviour etc. could be translated into queries and updates to a relational database structure or manipulations of VEOS's 'grouplespace' [Coc92] and so on, there is a much more straightforward mapping from the idea of multiple autonomous objects to a 'virtual machine' that provides an object-orientated 'multi-threaded' programming environment. In this scheme, each object is represented by a separate 'thread' or 'locus of execution' that functions independently of the rest. A number of such systems already exist: SmallTalk80 [GR83], Java [HC97], Eiffel [Mey92] and C++ [Str86] are all object-orientated languages which to some degree or other have support for such threads, either via built-in constructs capabilities or by the use of libraries. Before deciding on a platform however there are a number of other practical considerations to take into account.

Spreading the processing load

It is not surprising that many existing VR systems exploit some form of parallelism: the combination of the management of world models, however simple, and the presentational requirements (which inevitably include some element of graphics processing which is usually a heavy task in itself) form a formidable workload. Add to this the need to support multiple applications and/or users and the task is more complex still.

There is as yet no panacea for general parallel processing. Although there are numerous different techniques, both software and hardware, which attempt to simplify the problems of writing programs that execute efficiently in parallel, with the exception of a few applications that are inherently parallel anyway the task is still complex. Two categories of techniques are popular:

Express the problem non-serially. Many functional languages such as Prolog, Haskell [Pet96] or UFO [SHK95] express problems in a way that does not imply any particular processing order, or at least breaks problems down into subsections between which there is no implied sequence. Such programs can relatively straightforwardly be executed on parallel hardware given the correct algorithms (for example, in the case of a Prolog process, traversing down independent branches of a search-space can be done cleanly in parallel). Another method of this kind is a database search which has logical rather than imperative expressions (e.g. some elements of SQL), and such techniques are used by most contemporary large scale databases such as Oracle and Ingres.

Use an abstraction. A common way of representing parallel processing hardware to the programmer is to treat it as being a uni-processor machine with a single virtual address space; this can be done in hardware (e.g. Silicon Graphics multi-processors running Irix 6.4) or in software (as in the 'parallel virtual machine' system available for most architectures [GBD+94]). Where the machine is to execute a number of independent processes (e.g. separate heavyweight Unix-style processes, or applications in a WindowsNT environment), such distribution is effective. However, finer grain distribution on this kind of architecture is complex and automated mechanisms for spreading the workload of a single process across multiple processors tend to lead to poor scaling caused either by inter-processor communications or memory-bus clashes in the case of non-shared and shared memory processors respectively.

A novel means of abstracting a parallel processor as being a uni-processor is to use a technique known as 'virtual shared memory', whereby each processor treats its allocated memory as being an effective memory 'cache'. Whether this is implemented on a shared memory, loosely or tightly coupled system the mechanism is logically identical and only the inter-processor communication layer differs. A processor accessing a particular address in the virtual memory causes that location to be cached locally with the coherence of the distributed caches being managed by either hardware or software. The technique allows to some degree the automatic extraction of parallelism from a running process. The (late) Kendell Square Research machine (KSR1) [FBR93] is an example of a large multi-processor that uses this technique (implemented in hard and firmware).

The problem nevertheless remains that efficient parallel execution is generally hard to achieve: in the case of non-serial execution, the problem must first be expressed in an appropriate form (which for mathematical or logical expressions is often tractable, but for many 'real-life' problems is complex). In the case of the uni-processor abstraction, the fact is that in order to execute a program efficiently, it must be broken down by hand into functionally independent processes. Although techniques such as shared and virtual shared memory mean that monolithic processes will execute on parallel hardware, there will either be so much inter-processor communication—whether in the form of cache relocations, clashes on memory buses or network traffic—or sub-processes stalled waiting for resources to become free that the performance will be well below the scaled ideal. The problem then remains that in order to achieve anything close to efficient execution in a parallel environment a degree of functional analysis and 'manual' partitioning is required.

Of the systems reviewed in the previous chapter, those that exploit parallel techniques use either non serial methods such as the event driven distributed database in dVS and the functional language and 'grouplespace' used in VEOS, or by partitioning the functionality of the system in such a way that it can be represented by independent processes and located where possible on different processors (e.g. AVIARY, MR, and to a lesser extent DIVE). Other systems such as Superscape and WAVES have no explicit parallel programming support, instead providing library functions for inter-process communications and relying on the application programmer to organise for any parallelism.

In terms of parallel distribution, the Deva world model presents both a difficulty and an opportunity. The difficulty is that, being based on an interpretation of our perception of reality, it contains elements that are inherently sequential, i.e. we experience the passing of time and tend to express concepts in terms such as "this happens, then that. If such-and-such occurs, then such-and-such will happen". Inherent sequence is of course the anathema of parallelism and is often cumbersome to express in functional languages. The opportunity for parallelism is that we also experience our reality as consisting of on the whole independent 'things' which only interact under certain circumstances. These are represented by entities in the model, and as such form convenient, mostly autonomous, packages of state and function that can be distributed. This object-based distribution is exploited in AVIARY's 'artefacts' and to a degree in the WAVES framework's 'objoids'.

2.1.5 Summary of world model requirements

To summarise, an appropriate world model for virtual environments satisfies both the requirements of being a coherent framework for describing worlds and their contents as well as being able to cleanly exploit parallel hardware to manage the large processing loads that virtual environments often demand. The object-orientated paradigm appears suitable for both, being at the same time both a convenient way of partitioning our conceptual model of the a world so that it can be expressed formally and also a guideline for the distribution of workload around a multi-processor system. The Deva world model is an object-orientated framework aimed at addressing both these issues.

2.2 Interfacing

Many applications may benefit from representation in a virtual world. At the very least, even those applications that are not ideally suited, or have no immediate benefit from being made 'virtual' may well need some manifestation in a virtual world in order to be support tools for those other applications that do make the transition well. For example, there may be little point building a virtual calculator for its own sake, but as a tool to use in a more complex three-dimensional CAD package it could prove very useful. The variety of architectures and structures sported by applications however is vast, and an effective interface to a VR system needs to provide for this variety.

A coarse categorisation of applications could be formulated as consisting of those which already to some extent represent a virtual space, and those which do not. In the former category fall many CAD applications, data visualisation systems, some simulators and games, and in the latter perhaps come databases and mathematical packages[1]. The distinction is somewhat artificial, but useful nevertheless: some applications already have some notion of managing 'objects' in a 'virtual space', others have no such concepts and this will influence their preferred mechanisms for interaction with a VR system.

Another useful distinction is between interactive applications and those which are predominantly 'batch processors': the former will be built in such a way that they update their data structures frequently, and may be 'intimately' connected to any representation of those in a virtual world, the latter is likely to influence its representation irregularly and can not be expected to interact with the VR system very frequently.

These categories are orthogonal, and there are examples of applications of all four resulting combinations:

	Interactive	Non-Interactive
Spatially orientated	Flight Simulator	Offline Ray-tracer
not spatially orientated	Calculator	Payroll system

Figure 2.5: Four different classifications of application

There are two implications that this simple classification holds for a VR system's interface.

1. The interface can not expect the application to return to a particular function regularly. Though this may be straightforward in an interactive program which is likely to have a 'control loop' or

[1] It could be argued that all applications represent some kind of conceptual space, but the distinction that is intended here is again based on *presence*; the player of a video game, the participant in a flight simulator or even the designer of a CAD object will to some degree interact with the computer as though the objects 'really exists' in some 'physical sense'; the interaction with a relational database or mathematical equation plot is likely to be of a much more abstract and cognitive nature.

'event based' structure, batch and non-realtime programs where complex functions expect to run their course without interruption will find this difficult.

2. The interface can neither assume that all applications represent in themselves a form of virtual world, nor can it assume that they do not. Instead it must allow an application that already represents some form of virtual environment to pass on its knowledge of its specific contents and their behaviour whilst at the same time providing functions that support non-VR type applications in expressing their functionality in terms appropriate for representation in a virtual world.

2.3 Presentation issues

The final requirement upon a VR system, having established some means of interfacing with an application and representing its semantics as a virtual environment, is to present that environment to the user. The majority of existing systems (e.g. DIVE, dVS, NPSNet) have bespoke rendering engines to perform this task. The geometry of the virtual environment is either generated by calls to an interface or read in from a static file, and converted into some internal representation–typically scene-graph based–that is then rendered.

Although this technique makes for easy specification of scene geometry, it can also be argued that the importing of application data into a system-specific renderer is one of the significant factors contributing towards limited richness of behaviour in virtual environments. The argument is this. In order to represent itself as a virtual environment using a 'built-in' renderer, a conversion from application-specific data structures into renderer-specific data structures must take place. Typically, a system's rendering module will support 'generic' low-level graphical primitives, such as polygons, triangular-strips, meshes and so on. This disparity between application domain and renderer domain structures leads to two problems:

1. There are now at least two representations of the objects from the application's domain: one canonical within the application, another replicated in some form within the renderer. Routines to convert between the two domains must be provided, in order for changes within the application's structures to be reflected by changes in the rendered scene. The application becomes responsible not only for maintaining its own semantics (e.g. the imposition of physical constraints upon objects in a simulation) but for translating its data structures and semantics into forms that enable the renderer to also reflect these changes.

2. There is a significant loss of information during such a conversion, and although the application may be able to use its knowledge of the meaning of the data it contains to optimise its own internal structures, such higher-level information is lost during the importing of the data into the renderer, which is then forced to use 'generic', or put another way, un-optimised, routines for display, level of detail, and spatial management.

   For example, in the MAVERIK virtual cityscape demonstration [GPW98], the fact that most buildings are cuboid occluders is used to optimise the culling of the scene. In most existing VR systems translation of the buildings into a generic polygonal mesh occurs at an early stage in the rendering process, resulting in the loss of much of the semantic information in the environment (e.g. in this particular case that 'this is a building, and therefore a convenient occluding cuboid' and also that the scene on the whole is a cityscape and therefore consists of a large number of such occluders).

These two factors limit the behavioural and graphical complexity of virtual environments. More recent VR systems, such as Spline, Bamboo and MAVERIK, have moved away (in different directions) from the 'built-in renderer' and allow bespoke rendering routines or modules to be used instead. Experiments with large geometric models and the MAVERIK system have shown this technique to be highly effective [CHK98].

2.4 Summary of requirements

This chapter has presented a set of features required of a large scale virtual reality system under the headings of 'world model', 'interfacing' and 'presentation'.

In the main, it has concentrated on understanding the requirements of a 'world model', since this is seen as being the area in which most existing systems are weakest. The world model must address two issues associated with scale. First, it must provide a coherent framework for describing the properties of virtual environments in such a way that encourages the coding of rich behaviour and allows meaningful interaction between environments or applications. Second, it should provide a powerful enough processing context for the necessary workload (which realistically implies some form of parallelism). The Deva metaphysical model is proposed as satisfying both these requirements.

The requirements for application interfacing are described in terms of servicing the needs of the combinations of interactive/non-interactive and spatialised/non-spatialised applications.

Finally, the requirements of presentation are addressed and it is argued that the traditional scene-graph or polygonal representations of scenes do not form a suitable basis for rendering the contents of virtual environments, and that instead a more flexible, application or environment specific approach must be taken.

In the next chapter, these requirements are used to guide the development of an architecture for large scale VR.

Chapter 3

Scoping an architecture for VR in the large

In this chapter begins the process of scoping a design appropriate for large scale virtual reality.

Such a design must take into account factors from three areas:

- Functional requirements of applications and tasks ("what needs to be done?").
- Features of the technology ("what can be done?").
- Characteristics and requirements of the participants ("How will it be used?")

Although in practice, design is an iterative process that is affected throughout by all of these issues, until recently the limitations in hardware and the need to establish the fundamental viability of VR, has meant that application requirements and available technology have dominated the practical design of most popular VR systems. As the field matures, the opportunity for alternative perspectives on systems design improves. The goal here is to approach the development a new VR system using a high level analysis of the characteristics and requirements of the participants as the starting point. That is, the design is driven by paying particular attention to the fundamental process of actually perceiving virtual environments rather than by features of technology. Aspects of this relationship between perceiver and perceived will be used to guide the architecture of the system. As will be demonstrated, this approach offers a possible solution to key technological problems that otherwise appear, *prima-facie*, insurmountable.

Developing such a user-centred approach without paying attention to the context of a mature future technology is likely to be misleading. Therefore this work is situated in the context of 'large scale' VR, in a system that will be presumed to support:

- a flexible model of worlds, environments and applications:
  1. a world model that encourages construction of virtual environments that exhibit rich dynamic behaviour.
  2. the co-existence of many such environments, and the management of the complex relationships that may emerge between them and the users who move amongst them.
  3. ready comprehension of these environments and their inter-relationship by their inhabitants.
  4. an appropriate framework for integrating applications into such a world model.
- A low-level infrastructure capable of addressing:
  1. Geographically separated users.
  2. Diverse rendering and interaction techniques.
  3. Parallel architectures that meet the processing requirements of multiple concurrent environments, applications and users.

3.0.1 Methodology

The emphasis in this design is to begin with the 'human factor', and to work towards an architecture that can be implemented using existing technology. The design will begin with the model of human perception , and move towards a realistic implementation of the system, taking into account at each stage the features and failings of hardware and software, and offsetting these against the strengths and weaknesses of the links between the human and the virtual environment.

This chapter develops the high-level architecture of the system. Major components of the are system examined in chapter 4, and appropriate detail further refined in chapter 5.

3.1 Perception and reality as the starting point

Using the vocabulary developed in previous chapters, a diagram showing human interpretation of an environment can be shown thus:

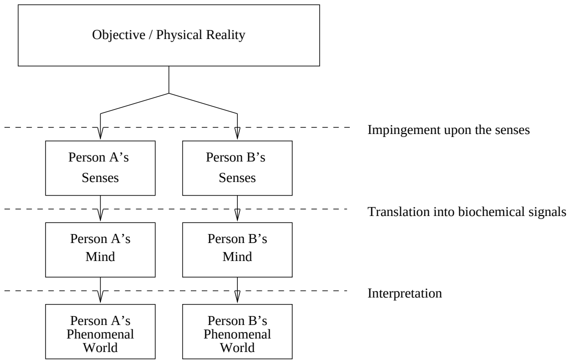

Figure 3.1: Two people's perception of the real world. A single 'underlying reality' is processed by two independent pipelines. Both the senses and the mind have the opportunity of altering the resulting phenomenal world.

An initial translation of this structure into into a concrete architecture for a virtual reality system is shown in figure 3.1. Here, a single process maps first between the represented application domain and the world model (that is, the interfacing component) and then between world model and the individual users' phenomenal worlds (that is, the presentation process). Given the particular world model defined in chapter 2, it is straightforward to see in principle how an interface may be specified that allows applications to communicate with entities in the world model. Therefore, for the time being, issues regarding the interfacing of application semantics will be put to one side, and the design will concentrate on issues concerning the world model and its presentation.

Figure 3.2 represents what could be thought of as the basis for a canonical or reference architecture for a multi-user VR system. Whatever the exact configuration, there must be a 'world engine' in which a reality is maintained, and an arbitrary number of different 'viewers' that present the world state to the users. For a system where there is a single viewer and a single environment, this single-process model may be a practical implementation. An example of this approach is the early single-user monolithic versions of Superscape. For a large-scale system a single monolithic process would immediately raise two problems requiring further consideration.

1. The issue of processing load and scalability. Even prior to addressing behavioural or semantic richness, the graphical rendering of multiple viewpoints of any non-trivial three-dimensional environment is a significant physical and organisational workload.

2. The desire to support *geographical separation* of participants. This necessitates some mechanism for transmitting the generated viewpoints to them.

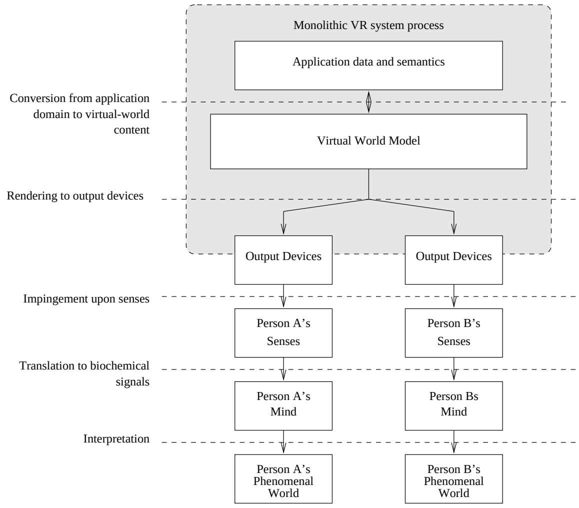

Figure 3.2: The perceptual model interpreted as a single monolithic process

3.2 Issues of distribution

These concerns require the monolithic process to be split, delegating different jobs to different processors connected via the communication medium. The spectrum of possibilities that arises can be described by its extremes. At one end is the 'pure peer to peer' model, homogeneously replicating the entire world engine for every viewer. At the other end of the spectrum is the 'pure client–server' model, maintaining a 'single world server' with client viewers sent for the local rendering of the user's viewpoint.

3.2.1 Peer to peer systems

In a pure peer-to-peer system, each process has equal status and may communicate 'directly' with any other. It is towards this end of the architectural spectrum that the majority of existing VR systems lie.

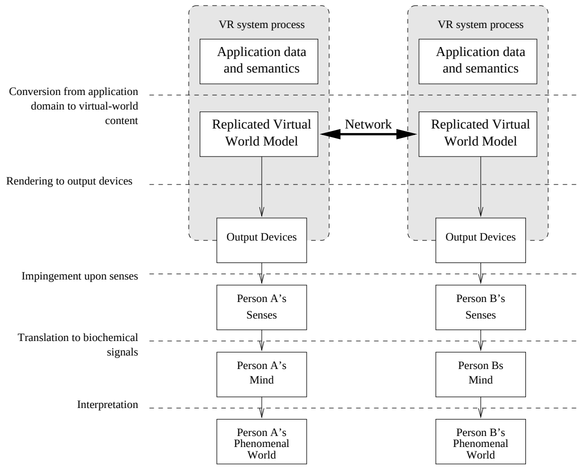

Figure 3.3: The perceptual model interpreted as a traditional peer-to-peer system

The main advantage of such an architecture is that the interrogation of the world engine is a local operation for viewer, and thus a light-weight route for obtaining a description of the environment for rendering. The disadvantages are twofold:

1. Synchronization of arbitrary events across the network is complex and treated naively will scale poorly as the number of objects and events to be synchronized increases and events need to be propagated to all participating peers. Broadcast techniques such as multicasting, and in particular hierarchical multicasting as used in the MASSIVE 2/CVE system [Gre97] alleviate much of the excesses of standard 'unicast' point-to-point communication. Maintaining a fully replicated environment of any behavioural complexity between geographically dispersed stations however remains a significant challenge.

2. Being homogeneous, there is no scope for utilising spare capacity in the system, or for optimisations that are best achieved through centralised resources. Increasing the complexity of the environment impacts equally on all users' resources, and introducing a single new processor into the system would be of no benefit.

3. The model implies either:

   (a) that each process contains a sychnronisable copy of *every environment*, whether currently inhabited or not, or

   (b) that environments not inhabited are inactive, and entities contained therein would be unable to perform their autonomous behaviour (since there is nowhere for them to be processed).

The first and second of these problems are essentially those of synchronization, replication of resources, and management of bandwidth. Though manifesting in slightly different guises, they are common to both peer-to-peer and client-server architectures; solutions to them will be examined in more detail in later sections. The third problem of maintaining the processing of environments that are unoccupied is peculiar to the peer-to-peer configuration. In the vocabulary of this report, such an architectural organisation implies that the 'reality' of an environment is somehow associated with and dependent upon a user's perception of that environment. In the limit when there are no participants to perceive the world, there is no world engine process it and the reality ceases exist. In practice, systems such as DIVE tend towards being peer-to-peer systems and either save the state of uninhabited environments to file, allowing them to be reactivated on re-entry, or maintain 'daemon' occupants (i.e. permanently running automated peers whose sole purpose is to keep environments alive when there are no human controlled inhabitants). There is no real parallel to this problem in our experience of physical reality: whether or not a tree falling in an uninhabited forest makes a sound, it is our experience that the trees will still be there next time the forest is entered, and further more that events associated with the passage of time will have occurred (falling leaves, moving animals etc.). In this sense the problem is artificial, and associated more with the architecture of the system than with the process of experiencing virtual environments as such.

At the other end of the architectural spectrum the pure Client/server model avoids this situation.

3.2.2 Client/server systems

In the case of a client/server architecture, the world-engine is a single process, with client 'viewers' connected to it by a network. The advantage of such an architecture is that it provides an obvious single location for processing multiple environments and their autonomous content, regardless of the participation of users.

Being a single centralised object resources can readily be focussed within it. Providing a single high-end parallel realisation of the server, capable of supporting multiple environments, is more cost-effective than an expensive parallel world engine replicated for each peer.

The server also provides a single 'locus of control' for each object, and cleanly supports notions of persistence and autonomy with users logging in and out of the running service while entities continue to update themselves. In this way the client/server model maps more closely to the perceptual model of figure 3.1, with the underlying reality being managed by one process, and the perception of that reality being managed by others.

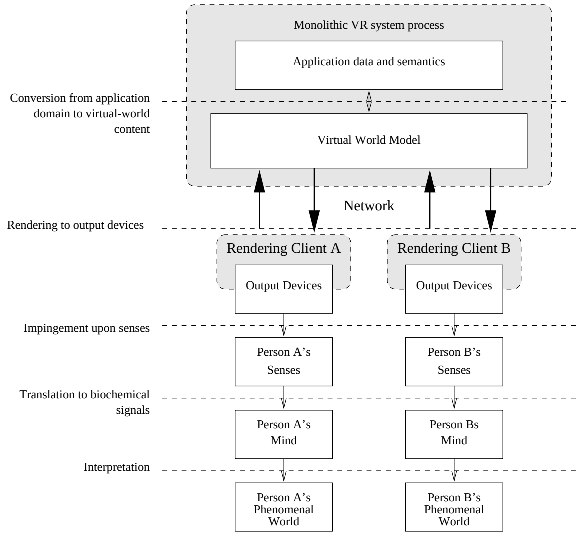

Figure 3.4: The same model interpreted in terms of a client/server structure

However, the architecture also presents significant problems related to distribution. A decision has to be made as to what degree of autonomy the viewing processes (clients) have from the world management process (server), and related to this, what level of communication occurs between them. At the extreme of 'no autonomy', the servers could broadcast frames of pre-rendered information to be displayed by each viewer. This hypothetical extreme however is far from practical, and even if graphics servers existed that were capable of computing and rendering multiple views on a virtual environment, today's network technologies do not support the massive bandwidths necessary to 'stream' such high-quality high-resolution video from to the clients, let alone deal with issues of user interaction with such a scene.[1] Communication between client and server cannot be realised at this extreme. One step up from 'raw pixel' information is the broadcasting of polygonal information to each viewer. Again, for a scene of any complexity, this too is well beyond network capabilities. Higher levels of autonomy can be conceived of, where the viewer is responsible for more and more 'caching' of state and interpretation of incoming events, for example viewers could remember the geometric or polygonal forms of objects, and receive only updates to their form or position. As the degree of autonomy increases and the requirements on the network for transmitting *immediate state* decrease, the client/server models tends once more towards being a peer-to-peer architecture, and the network requirements once more increase as the demands of synchronisation between processes increases. The trade-offs once more are between synchronization, replication and bandwidth.

3.2.3 Practical systems in the peer-peer / client-server spectrum

These issues of replication of state, synchronization, and bandwidth are therefore common to both pure extremes of architecture. The pure peer-to-peer architecture has much replication of state, and utilizes bandwidth to synchronise that state amongst participating processes. The client/server architecture has little replication, and hence less need for *explicit* synchronization, but utilises bandwidth to continuously transmit much more 'state' information (hence, it could be said, achieving synchronization implicitly).

Though none of today's VR systems are pure peer-to-peer or client/server, they sit towards one or other end of the spectrum. DIVE, for example, is a predominantly peer-to-peer style system that utilises some server processes for centralised services such as resource allocation. Each process inhabiting a DIVE world is an application that may extend the functionality of the standard Dive facilities to suit its needs.

[1] This is a different issue to the one of providing non-interactive realtime video *within* a virtual environment. Several systems already provide this facility using lossy compression techniques such as mpeg encoding. However, the transmission of 'broadcast quality' graphics in realtime is still beyond existing network technology.

However, peers representing other applications will have a different vocabulary of extensions built for the different application. Communication with other peers therefore takes place using a standard vocabulary of messages. Therefore, one application that has defined 'the force of gravity' may only communicate its effects to other participating peers as being repeated updates of the positions of all the affected objects, thus being a cause of significant network traffic. Superscape's 'Viscape' on the other hand, being a web-browser 'plug-in', is much more like a pure client/server. In this system, worlds are downloaded in their entirety into the browser (viewer) process, which then sends any user-initiated events back to the server for synchronisation and propagation to other participating users.

It is clear that issues of synchronization and replication of resources, and the associated utilization of bandwidth is a significant cause of difficulty in all existing distributed VR systems. The problem has been attacked from a number of different angles, ranging from the explicit propagation of events in dVS [Gri91] to the more implicit synchronisation proposed by VEOS's 'grouplespace' [Coc92]. These direct approaches to the problem can however be seen to place the various systems on the spectrum somewhere between the extremes of peer-to-peer and client/server, leading to combinations of the difficulties and features associated with those positions. In the remainder of this chapter an alternative solution will be sought that deals indirectly with issues of synchronization in VR by examining once more aspects of human perception and the relationship between the user and the perceived virtual environment.

3.3 Perceptual solutions to synchronization in virtual environments

It is usual to relate to the real world as though our experience of it is instantaneous, unmediated and objective, as though what we perceive in our minds *really is* what is happening in the outside world at that moment. It is also usual to assume that other inhabitants of the world experience the same environment in the same way, since it is possible to communicate coherently about shared experience.

In daily life such reflections are of academic interest, but to the perceptual psychologist, and in the field of VR which is concerned with presentation for detailed perceptual experience, these concerns have significant bearing. The extent to which a solid objective reality is often imputed upon quite flimsy evidence is quite remarkable and is well reported in the literature on perception [BG90] [Gib66]. It is evidenced in perceptual illusions (by definition), and even in such solid phenomena as the presence of the foveal blind spot, which comes as a surprise to many [Den91].

The main criticism of the naive model of perception is that the act of perceiving is itself *inherently subjective*, at least to the extent of involving 'subjects' with their own unique viewpoint on the world, and yet still enables consistent and coherent interaction with the environment and its other inhabitants. Thus human experience can not be said to be totally *objective*. Contemporary virtual reality systems (usually, implicitly) quite reasonably attempt to model an objective shared reality, and to present that reality to its users as being synchronised and definitive.

By a closer examination of the distinction between objective and subjective, it is possible to find a justifiable framework for reducing the requirements of strict synchronization in virtual environments, thereby reducing the architectural burden to achieve it.

3.3.1 Subjectivity and synchronization

There has been much recent interest in the idea of subjectivity in virtual environments, with both strong support [SM97] [Rod98] and opposition [Bow98] being expressed for it. In the existing literature the term 'subjectivity' is particularly used when discussing the advantages that may be gained by intentionally providing different ways of presenting a common shared world to each participant. An example of this idea is a CAD environment in which engineers would see more relevant engineering detail, and financiers would, at the same time, see the environment presented in a way more appropriate to their interests.

This idea of subjective views has for many years been used in the database field, where a single body of data is represented differently to users depending on the requirements of their role or their personal preference. It is only more recently that the principle has been applied to virtual environments, with applications such as Bead [CC92], Q-PIT [ACR96] and subjective version of VR-VIBE [BSG$^+$95] allowing users to 'customize' their view of a shared virtual environment.

Such 'customization' of a user's view is only one aspect of subjectivity in virtual environments, and other less studied manifestations become significant when the problems faced by larger scale distributed systems are investigated. In order to understand this further, it is necessary to disambiguate the ways in which subjectivity appears in VR, and outline a framework for the classification of subjective experiences, highlighting in particular areas where subjectivity has relevance to systems architecture.

3.4 Degrees of subjectivity

The degree to which a participants experience is 'subjective' can be described with reference to the stages of the perceptual pipeline at which the divergent experiences of two users originate. The following sections will examine these foci of subjective divergence.

3.4.1 Phenomenal subjectivity

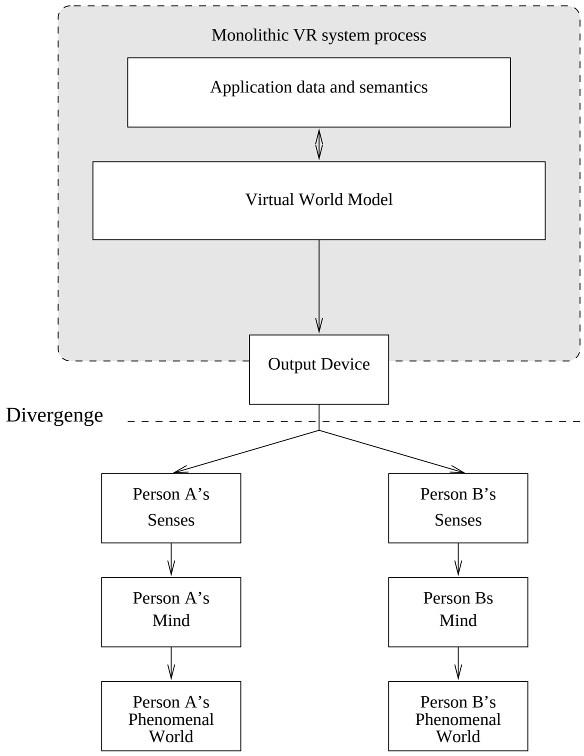

Figure 3.5: Two users view the same output device; any divergence in their experience of the virtual world is governed by factors in their own senses or minds.

Consider two users experiencing the same virtual environment from the same 'virtual' viewpoint, and presented with exactly the same rendered images, either by desktop screen, head-mounted-display or other technology. Though barely a form of multi-user environment, each user is still likely to present subtly different descriptions of their experience. Such a simple situation highlights the fact that once the information has left the realm of the electronic and has passed beyond the sense organs of the perceivers, the resulting phenomenal world of each user will have a significant subjective component. Such factors as emotional response, personal history and predisposition will affect to varying degrees the interaction of each user with that environment. Thus, even though a single, shared, and wholly objective environment is presented externally, a degree of subjectivity – called here phenomenal subjectivity – is inevitable in a participants experience of that environment.

3.4.2 Viewpoint subjectivity

From the minimalist 'multi-user' environment above, each user is now given control of their own personal viewpoint. The environment is, again from the point of view of the technology, wholly objective. This is the typical situation in a contemporary multi-user VR 'walk through', and can be illustrated in the manner of figure 3.6.

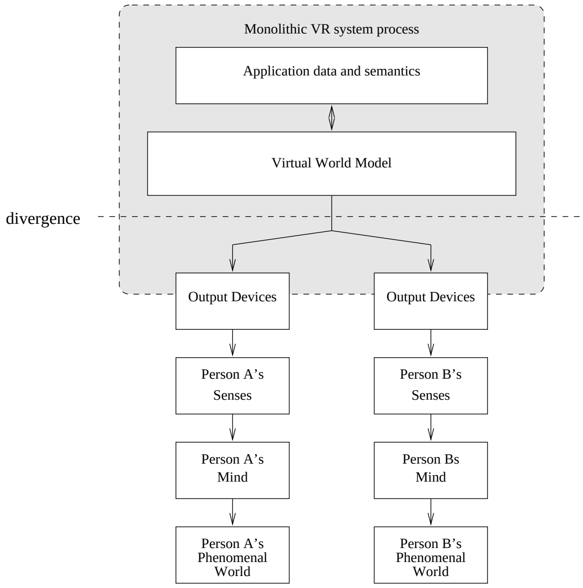

Figure 3.6: Here the users have their own output devices and are able to independently control the position from which they view the world.

In addition to phenomenal subjectivity, each user experiences the world from a different viewpoint, and hence has a more strongly subjective experience than in the previous example. Not only are they interpreting the environment through the filter of their own experience, but each user is now being presented with radically different stimuli at their sense organs. This degree of subjectivity may be called viewpoint or embodiment subjectivity, associated as it is with possessing a 'physical' locus of representation in the virtual environment.

3.4.3 Presentational subjectivity

In this form of subjectivity, the mapping between 'underlying data' and 'presented form' is adjusted for each individual user. This is the notion of subjectivity that has gained currency in VR, as described at the beginning of this section. A single piece of data within an application may be given a different representation in the view of one user to that of another. Here the notion of subjectivity is more explicitly manifest within the VR system and there is some notion of an underlying 'objective object', (or 'thing in itself') being represented. Algorithms translate between the underlying semantics and data, into a rendered subjective viewpoint, and these algorithms are associated with the display processes of the system. This form of subjectivity is, then, what is usually referred to in progressive virtual reality literature by the blanket term 'subjective'.

3.4.4 Total subjectivity

As the point of subjective departure moves towards the extreme of the application's semantics, the meaning of the term 'subjective' reaches it's limits. If the underlying 'reality' is actually different for each user then little is left that is a shared experience. Such limits are not a discussable topic in conventional reality as there is no access to objects 'beyond experience'. In VR however, we are in the privileged position of knowing what lies behind mere experience as we can, in the limit, inspect the code responsible for generating it.

The situation here then is one where the applications provide meaning to the world, and therefore in a strong sense defining the objective nature reality being experienced, are increasingly participant-dependent.

Since the term 'subjective' can only be held as meaningful in opposition to some 'objective' pole, any blurring of a single underlying objective semantic weakens the use of the notion of subjectivity too.

Progressing along this path is more akin to users experiencing completely different and distinct virtual environments than it is to them inhabiting a single common world, as we are fundamentally negating the locus of underlying objective reality. Nonetheless, this is clearly a category of subjective experience available within a VE, and could be characterised as "it looks different to me because it really is".

3.5 Relating subjectivities to architecture

Interesting parallels can be drawn between the two extremes of architecture described in section 3.2 and the degrees of subjectivity indicated above. The pure client/server model reflects the degenerate case of 'no subjectivity' almost by definition, as the world server is dominant and the viewers secondary and peripheral. Complexity is introduced into the architecture to deal with the desire for increasing degrees of subjectivity, such as separate viewpoints. Conversely, the pure peer-to-peer architecture tends towards the extreme of complete subjectivity, where each peer has its own independent environment, and complexity is introduced into the architecture in order to *reduce* the degree of subjectivity and bring synchronization to the shared experience. The Bricknet system [SSPN94] is a particularly strong example of this kind of architecture, in that the programming model is one of distinct environments that are 'shared' by virtue of explicitly exporting common components to one another.

Existing virtual reality systems and applications from both ends of the architectural spectrum either exhibit viewpoint subjectivity, or presentational subjectivity (as in the case of Bead and other generally subjective applications). Clearly 'total subjectivity' is an undesirable extreme in a shared multi-user environment, where the goal is collaboration. A useful point of departure from the conventional view of an objective experience of reality is provided by phenomenal subjectivity however, and this will now be considered in relation to systems architecture.

3.6 Phenomenal subjectivity and the relaxing of synchronization

Phenomenal subjectivity has been defined as being a differing in experience of an (assumed) objective world due to factors 'internal' to their perceiver such as personal preference, aesthetic sensitivity, cultural background and so on. One distinctive feature of this form of subjectivity is that its exact effects on the perceiver are unknowable to others: although it may be possible to don a head mounted display to see what another user's viewpoint in a virtual world is showing, there is no similar mechanism by which it is possible to *experience exactly* what the other person's phenomenal world is like. Furthermore, there are many aspects of the phenomenal world about which communication is either difficult or impossible: it is sometimes hard to explain why one likes red things, but it is impossible to convey the direct experience of 'redness'.

Though it would be easy to over emphasise these effects, we argue argue that these 'unknowable' areas of experience can usefully be exploited to reduce the constraints of total synchronization in virtual environments, whilst maintaining apparent coherence of experience for the participants.

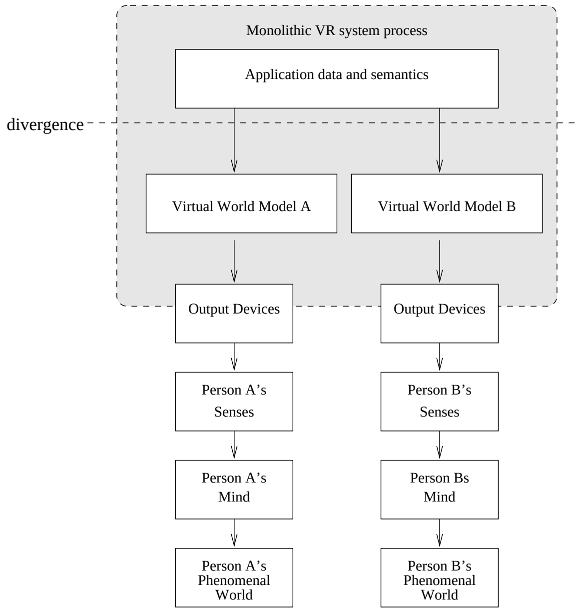

Figure 3.7: In addition to having their own output devices and viewpoint, the contents of the virtual world(s) experienced by the users are generated differently from the application data. Thus the users in one sense inhabit the same physical space, but its objective form is different.

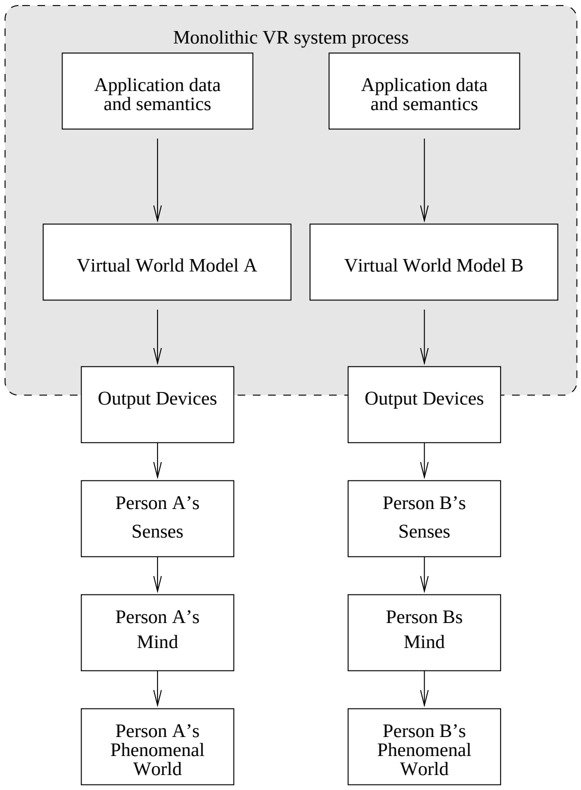

Figure 3.8: Whether or not this represents a form of subjectivity is debatable. Here the two users experience different worlds, generated from different data sets. They may 'inhabit the same physical space' but there is no meaningful link between the objective contents they perceive.

The argument is this: since there are inherently subjective areas of phenomenal experience on the basis of which objective experience is largely imputed [Den91], there are corresponding aspects of presentation that need not be fully objectified (i.e. 'synchronized') in virtual environments. In such cases we can rely instead upon the aforementioned mechanism of imputation to fill in for loss of objective constancy. That this can be exploited is easy to demonstrate; the extent to which it can be usefully exploited is a topic for further analysis.

3.6.1 An experiment

These ideas can be illustrated by a simple thought experiment. In a virtual environment, Person A holds a ball, which is dropped when Person B says "Start the experiment". When the ball reaches the floor, Person A says "End of experiment".

Consider first the experiment whereby the participants are totally immersed, and have no means of communication outside the virtual environment. After having carried out the experiment, Person A will report the same series of events as Person B given that they both experienced them in the following order:

1. Person B : "Start the experiment"
2. Person A drops the ball
3. The ball hits the floor
4. Person A : "End of experiment"

Any deviation from this perceived event ordering would result in a different objectively verifiable report of the experience (for example, Person A could experience the events in the desired order, whilst Person B experiences Person A saying "End of experiment" *before* the ball hits the floor; this is entirely possible, and even likely, with existing systems such as Massive). However, as long as perceived effect follows perceived cause, and events occur within a certain tolerance (in this case, most likely a small number of seconds), neither participant can or need be certain about exactly how long the events took, and a certain flexibility of synchronization is acceptable, or can at least be tolerated.

Consider now the same experiment, except that the participants have clocks in the *real world* that they can see at the same time as the events in the *virtual world*. With this independent metric, they would be able to detect deviations from absolute time, and communicate them to one another (e.g. Person A dropped the ball at 2 seconds, whereas Person B saw this happen at 3 seconds). Without such an external reference there is a window within which a range of deviations from common experience are accounted for by perceptual interpretation. At one common extreme special measures, such as external clocks, are required to detect otherwise imperceptible deviations. At the other extreme only an extremely good stage magician can make the apparent disappearance of an elephant convincing.

An interesting extension to this experiment is to situate the timepiece in the virtual environment. Now, within certain limits, it is again impossible for the participants to tell whether they have experienced *exactly* the same events, since the virtual clock could obviously be modified to present the flow of events as synchronous.

These experiments illustrate that under many circumstances it is unnecessary to perform absolute synchronization for *all* aspects of the virtual environment in order to maintain apparent coherence of shared experience. This observation evidently has useful implications for the task of large scale systems architecture.

3.7 Exploiting phenomenal subjectivity in a 'Kantian-hybrid' architecture

The argument of the previous section leads towards the architecture shown in figure 3.9, wherein the objective and the subjective are made explicit in the design. This configuration is called a 'Kantian-hybrid' after the 18th century philosopher, Immanuel Kant, who devised a model of reality in which there is a distinction between the *noumenal*, or 'the things in themselves' that exist in the physical world beyond human senses (i.e. the underlying objective reality), and the *phenomenal*, the 'thing as it is to me', or 'that which may be perceived' that exists in the mind of the perceiver (i.e. the subjective); the world of appearance and perception [bJM34]. This categorisation is especially apposite in VR where the underlying noumenal reality corresponds to applications that are easily conceived as distinct from the objects that are presented to perception.

In this configuration, the underlying objective reality is located in the world-engine, and partial copies necessary for the maintenance of subjective experience are distributed in the subjective viewer processors. Updates can thus be based on *perceived effect* rather than merely mechanical state-change. Consider, for example, an 'agent' in the virtual world that positions itself in space so as to point the user towards a particularly interesting part of the environment. The agent is controlled by complex search algorithms that continuously scan a database, attempting to extract 'interesting things' from it to point out to the user. The user's initial perception of the agent, however, is simply that there is an object that moves around in space, with the re-attribution of semantic significance to that position being done in the mind of the user. Hence both in the 'world-engine' and the user's mind exist notions of 'interesting information'; however at the level of the interface between VR system and user, there is simply movement in space, velocity, colour and shape that are not necessarily synchronized *exactly* with the behaviour of the search algorithms. That is, there is no way for the user to tell that the representation of the agent reaches its destination several seconds after the driving search engine returned some results. Hence there is a de-coupling between 'objective cause' and 'perceived effect' that allows a relaxation in the strict requirement of synchronization of network traffic between client and server.

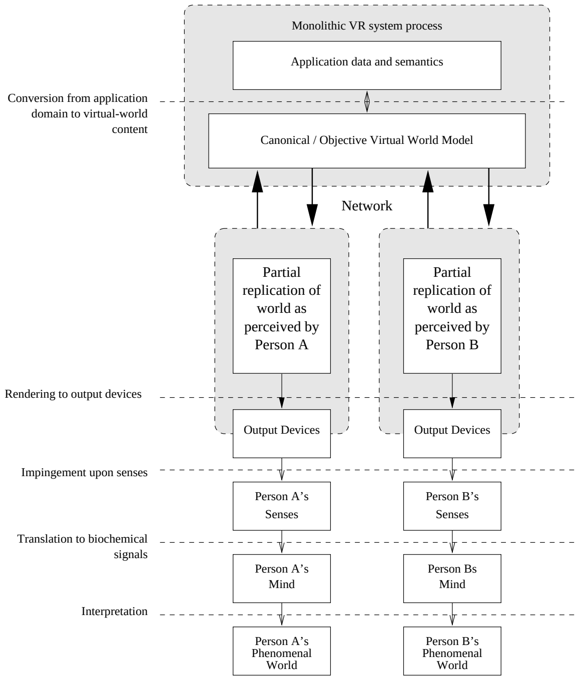

Figure 3.9: Using the objective/subjective split yields a 'hybrid' architecture sporting the best features of peer to peer and client/server systems

3.7.1 Other benefits

The Kantian-hybrid architecture explicitly addresses the issues of synchronization, replication and bandwidth from the perspective of human perceptual requirements. It can be seen as a perceptually oriented legitimation of the ideas of caching, dead-reckoning, and prediction used in existing systems as partial solutions to synchronisation.

This particular hybrid of peer-peer/client-server overcomes the individual problems of those architectural paradigms to the extent that:

- It requires only one 'heavyweight' processor/server to manage the worlds, with each client process replicating only the portion of those worlds that the user experiences at any one time. This server could be a parallel or distributed machine into which extra processing nodes could be added to meet increased world complexity.
- The world-engine provides a centralised persistent processing context for autonomous entities, and uninhabited environments. Thus a clock may continue to 'tick' even though there is no user in the environment to experience it.
- It gives the rendering processes lightweight and local access to the data necessary to draw from a user's viewpoint, reducing the need to synchronize continuously across a network (as with pure client/server), and without such gross replication of workload as is required by the pure peer-to-peer architecture.

3.7.2 Limitations of the model

Although it is expected that a large proportion of activities can be presented with a degree of 'subjective leeway', there are certain events that require more immediate synchronisation with their objective component. In particular, for single user situations, any activity that requires any kind of 'physical interaction' with an object (e.g. moving it with a mouse or input device) requires a rapid response time (often quoted as being less than 2ms [CMN83]) from the system. 'Picking up' a mug from the table, and having it snap into one's hand a second later is likely to be a disconcerting event. A method for overcoming this limitation is described in 5.6.5.

For multi-user situations the dynamics are more complex still. Though there are events where synchronization may be relaxed (such as the experiment in section 3.6.1), where more than one user is concentrating on a single entity, synchronization of some form must be maintained (consider two users trying to manouver a ladder through a confined space; inconsistencies in their 'subjective' view or delays in system response are likely to make a difficult task impossible). Dealing with such situations is an interesting field for further research.

3.7.3 Decoupling the subjective and objective

The hybrid architecture proposed here relies on the existence of an appropriate metric for distinguishing subjective phenomenal effect from semantic cause. In the example of the search agent given above, the metric used was 'visual appearance', however this is too restricted a distinction to work in practice, and other more general factors must be taken into consideration. If the separation is indistinct or too weak, then semantic cause and phenomenal result become tightly bound, thus effected within the same process, and the architecture is once more peer-to-peer like (i.e. each subjective viewing process also ends up calculating the semantics of the world). Too strong a distinction or separation, and cause and effect become dissociated, with the result that users' experiences hold little relationship to objective reality (i.e. an object's driving algorithms caused it to perform an action, but that action was ignored, delayed or misrepresented).

In conventional interfaces, semantics are explicitly conveyed using text and graphics. Such an interface is primarily cognitive with low emphasis on perceptual issues. The benefits of VEs are achieved through a primarily perceptual, less cognitively oriented interface. In this way, semantic information in VEs is largely implicit at the actual interface, and relies upon the participant to elicit meaning from the environment. In considering the communication between world engine and client visualisers, the appropriate level of discourse is thus that of phenomenal experiences, rather than of geometry and strict synchronous events which are at too low a level to take account of the arguments presented herein.

To be effective, such an approach requires a metric or method of description that expresses the relationship between semantic cause and phenomenal effect *within a VR system* to that of identifying quantifiable 'elements' of perception that contribute towards phenomenal experience.

Whatever semantics are described by an application or are exhibited within a virtual environment, they are ultimately presented to our sense organs before being re-interpreted by our minds. At this point, all information is presented purely in terms such as shape, colour, duration, volume, pitch etc. and can be formalised using a reasonably self-contained vocabulary that can be further organised within such a framework of understanding of the perceptual process and our phenomenal experience as represented by Kant's 'forms of intuition' of Space, Time and Causality[2]. Consider once more the phenomenal experience of the 'interesting things' agent from section 3.7. Within the world-engine, there will be code that manages some notion of 'user interest' and applies this to data sets. The resulting rendered scene, however, will merely display the graphical representation of the agent moving at various rates from position to position. Attribution of such notion as 'interest' is only re-applied in the mind of the user: it is not explicit in the rendered/perceived scene. Hence a protocol that allows the description of the form and movement of objects in space and through time, and their synchronization with one another (a somewhat narrow, though pragmatic interpretation of 'causality') is sufficient for describing such desired perceived effects.

The following sections will describe how the Metaverse graph introduced in chapter 2 may be used to provide the formal structures necessary for implementing such a scheme.

3.7.4 The Metaverse and elements of perception

'Beyond The Metaphysics' [Pet95] develops a technique for organising and making comprehensible the features of and relationships between different kinds of virtual environment. This structure, named the 'metaverse' graph is an object orientated hierarchy of 'environment classes'. At the base of the graph are classes that support basic components of perception (e.g. Kant's Time, Space and Causality), with more sophisticated features being incrementally added to build increasingly rich environment types. Into instances of these environments may be created 'entities' that represent the contents of the virtual environment. These entities may defined properties particular to themselves, and in addition are imbued with the basic properties of their environment (for example, an environment that implements a notion of gravity would imbue upon its contents a 'mass' attribute). Such a technique, we argue, allows sophisticated environments to be defined with a minimum of code, whilst retaining information about the relationships between different environment types and their contents.

Primarily described in terms of being a tool for organising a large number of environments, the same techniques used in the metaverse graph may be used to:

1. provide a set of concepts that may be used to form a protocol for communication between world-server and client viewer processes, and

2. form a framework for using those concepts to describe specific application requirements whilst still maintaining a formal link back to those primitives.

Using this approach, both world server and client processes contain the knowledge of the root environment classes of the metaverse graph, namely 'Time', 'Space' and 'Causality', and their multiply inherited sub-class 'Holospace' (the most basic environment that can be meaningfully inhabited and perceived). These classes describe the most basic fabric of the virtual realities, and the fundamental forms of intuition available to participants.

The exact details of the implementation of the functions provided by the four root environments will be covered in chapter 5; for now it is sufficient to assume that in their combination as Holospace, they provide sufficient functionality to deal with such notions as

[2]In fact, in [bJM34], Kant actually argues that these three forms are also products of the perceptual mechanism rather than aspects of reality itself. However in this context, the liberty is being taken of reinterpreting them as as being suitable classifications into which the 'low level primitives' of perception fall.

1. **Space:** Cartesian co-ordinate systems, position, rotation, direction, distance, geometric form, colour, transparency, scale, intersection.
2. **Time:** time-of-day, duration, 'precedes','at the same time as'
3. **Causality:** 'causes','is an effect of','is dependent upon'
4. **Holospace:** all the above, plus their higher-order combinations such as velocity and acceleration, and the essential infrastructure necessary for the above to present an experiencable environment.

As part of their functionality, each of these environments contains features for packing and unpacking descriptions of their contents in such a way that it can be transmitted across a network. Thus the definitive and objective contents represented by the root environments in the world-server can be transmitted and re-created within the client viewers.

If all VR environments to be represented were forms of pre-scripted animations with no independent semantics as such, and user interaction was limited to the control of a viewpoint, then such a protocol as this would suffice. Under these circumstances, the passage of time causes various events governing the form and movement of objects through space to be triggered (i.e. there is in principle enough functionality shared between Time, Space and Causality environment classes, and brought together in the Holospace environment class to perform the kinds of scene choreography used in Lightwave [] or 3dStudioMax [] for high-quality animation, and in VRML2 [] for real-time 3D environments).

However, the linking of arbitrary application semantics into such a system demands more than this, and implies a means of extending these primitives in line with the applications requirements (if responding to a changing database, the search agent from previous sections, for example, can not be scripted in advance of the changes in the underlying data). Such a mechanism for extension is provided in the metaverse graph model using object-oriented style multiple inheritance; in effect taking the primitives as building blocks, and combining and refining these to create application-specific functions. The world server, then, must be a full implementation of the Deva metaverse graph, allowing applications to extend the basic set of 'perceptual primitives' to suit their own needs, whilst maintaining a route from application semantics to the perceptual primitives that can be interpreted and rendered within the client viewer processes. The client viewers can be seen to contain a 'cut-down' non-extensible version of the four root environments of Deva metaverse model). In short, the world server contains code representing 'why' and 'how' things happen, while the client viewer processes can just render the 'what' without any knowledge of the originating semantics.

This scenario may be made clearer by example. Consider a ball falling under the influence of gravity, having been 'dropped' by a user. An example graph for an environment that implements such a force is shown in figure 3.10.

In figure 3.11 the environment that contains the ball has defined a 'law of gravity' such that any object that is neither attached to another nor selected by a user accelerates 'downwards' until one of those conditions is met. This example is particularly trivial to express in the terms defined by the metaverse graph, since the Holospace environment class, which is a superclass of the example environment already implements a notion of an 'acceleration vector'. The 'law of gravity' can thus be defined as simply giving the acceleration parameter of the ball a value when it is neither selected nor supported, and setting the value to zero when either condition is valid. Though trivial, the example nevertheless illustrates the desired point, which is that in the server there is code for giving an object an acceleration or not depending on specific circumstances. Between server and client process however, there is simply transmission of the fact that a particular object has suddenly acquired or lost a specific acceleration, with no detail of the semantic cause (in this case, notions of 'supported by', 'gravity' etc.) The code for implementing the effect on 'absolute position' of 'acceleration' is mirrored in the world-server and in any participating clients. The perceived effect of a ball accelerating under the influence of gravity is therefore presented in the client, faithfully reproducing the application of the objective law in the server. Communications overheads are kept to a minimum (in this case, just events signalling increase or decrease in acceleration, rather than frame-by-frame updates of an absolute position).

3.7.5 Applying 'perceptual filters'

Defining a basic set of 'perceptual primitives' for communication between server and viewers allows a certain reduction in bandwidth by enabling perceived results to be described in 'higher-order' terms (e.g. using acceleration rather than velocity or absolute position where appropriate). In addition to this, it enables aspects of human perception to be used to further optimize the inter-process communication. That is, since the communication is couched in terms of perception, general principles of perception may be used in an application independent way to reduce network traffic. For example, very small changes in velocity, regardless of their origin are unlikely to be perceived, and can be accumulated before sending to the viewers.

3.7.6 More complex primitives

In the limit, any movement through space can be described by a set of vectors defining velocity, acceleration etc. However, many situations would require a large number of such vectors (e.g. movement in a convincing circle), transmission of which could potentially negate the benefits of using higher-order attributes. This situation may be remedied by providing for the attributes to be described not in terms of linear vectors but in terms of curves. These may be subjected to the same kind of optimizations and filters as described above, however give a significant improvement in the complexity of movement that may be described using a single inter-process message.

This concludes the discussion of the development of the architectural motivation and outline for the system. In the following chapter, more detailed consideration will be given to the role of the various major components.

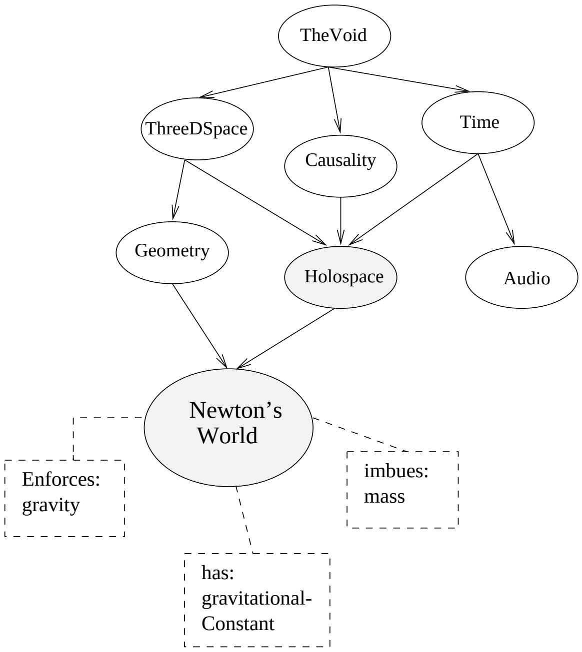

Figure 3.10: Here the default metaverse is being extended with a new environment that inherits all the properties of its parents, and in addition extends the functionality to include a new law (gravity), a new environment attribute (a gravitational constant) and a new imbued attribute (mass). Example code for this environment is given in figure 3.11

```
Holospace subEnvironment: #NewtonsWorld with: Geometry
     hasMethods: NIL
     imbuesMethods: NIL
     hasAttribute: 'gravitationalConstant'
     imbuesAttributes: NIL
     enforces: gravity.

SolidWorld defLaw: #gravity
     for: 'making things fall to the ground with a
           constant acceleration'
     as:
     [
         (((object.acceleration.y)
             < (self.gravitationalConstant))
          and:
          ((object.intersectionWith) == NIL)
          and:
          ((object.selected) == False))
          ifTrue:
          [
             object.acceleration.y:
                       (self.gravitationalConstant).
          ]
          ifFalse:
          [
             object.acceleration.y: 0.
          ]
     ].
```

Figure 3.11: An example piece of Deva code defining the environment from figure 3.10

Chapter 4

Server, clients, infrastructure and content

This chapter describes the key functions required of the two major components of the VR system: the world server and the client viewers. Their roles will be examined in turn, outlining for each their sub-components, and highlighting any practical implications that the implementation of those functions present.

Being by far the more complex of the two components, the description of the world-server occupies the greater part of this chapter. The description of the client, which to a degree is a 'cut down' version of the server, focuses on its distinguishing features.

4.1 The server

The server provides the context for processing and managing the 'objective reality' created by the representation of applications as virtual environments. This role logically has two components:

- The management of the *infrastructures* required by the Deva model, including support for message passing, traditional class inheritance mechanisms, and the various extensions to the object-orientated metaphor such as the dynamic run-time grafting of entity and environment properties.
- The provision of the functions required to implement environments on top of those structures. That is, the *content* of the metaverse graph.

These are requirements arising from the Deva model and will be examined in the following sections. As well as providing this functionality, however, a the pragmatic issues of realistic performance must also be addressed, and a later section of this chapter deals with this issue.

4.1.1 Server infrastructure: run-time support of entities and environments

The server must provide features that support the structural components of the programming paradigm developed in this report. In the same way that traditional languages support such notions as variables and function calls independently of what they may be used for, the Deva model implies support for certain novel constructs: in particular the entity and the environment. This has certain implications for the structure of the server process, which is the execution environment for these constructs.

In this report, virtual environments are described in terms of entities, forces, behaviours and so on: notions that exist at a much higher level than can directly be supported by any existing hardware, or indeed directly by any existing software platform. Thus there is a need to map between the high-level concepts in the paradigm, and the much lower level instructions of the hardware. A useful model for this process is to consider two main components: a 'virtual machine' that provides appropriate 'virtual instructions' representing the common notions of the higher level language and mapping them onto the simpler hardware instructions, and a translator that transforms the user's source code into these virtual instructions. Although more explicit in some programming paradigms than in others, this model is universally applicable: for a run-time interpreted language such as Tcl, one can consider the process as having a 'thin' translator, with a very 'thick' virtual machine whose virtual instructions are the tokenized source; for a completely compiled language such as C the situation is reversed, with the compiler performing a 'thick' translation from source into machine instructions and the 'virtual machine' layer being 'thin' almost to the point of being non existent (though the libraries necessary to support anything but the most trivial C program can be thought of as forming a virtual machine of sorts).

The task at hand is therefore to identify an appropriate level for the 'virtual machine' instructions. This involves identifying key aspects of the programming paradigm that are specific enough to warrant representation by a 'virtual instruction' and yet are general enough to form, in combination, a useful 'target instruction set'. An in-depth examination of the factors that influence instruction set design (be it real or virtual) for general purpose programming languages is beyond the scope of this report. However there are two specific features of the Deva programming model that have a significant influence and place specific requirements on the virtual machine that is to execute its programs. First there is the fact that the model is of an object-orientated nature. Being, compared to machine code and assembly language, a high-level abstraction, this paradigm in itself holds implications for the intermediate virtual instruction set. Second, the Deva model extends the traditional notion of object orientation in two directions: the dynamic grafting of entities with their environments, and the notion of 'behaviour' or multiple independent loci of control. Both these issues require further examination.

Supporting the object orientated approach

The object-orientated approach of the programming paradigm developed here places certain requirements on its run-time environment or virtual machine. These include the management of the creation and destruction of objects, their general processing and the support of notions such as inheritance. Traditionally, these have been dealt with in a number of ways, loosely dependent on the 'granularity' or 'purity' of the specific language.

Object orientated languages can be thought of as being at one extreme 'course-grained' or 'impure', and at the other 'fine-grained' or 'pure'. Eiffel, C++ and Java are examples of the former category, wherein the notions of classes are used to encapsulate otherwise traditional (non object-orientated) programming techniques. Smalltalk on the other hand is the definitive exemplar of a fine-grained language in which every programming construct–be it a single byte or an entire database–is described in object-orientated notation. Fine-grained object-orientation represents a kind of 'ideal' for the paradigm–it is object orientation in its purest form. In practice however, such an ideal is rarely attained for reasons partly of syntactic brevity but mostly of performance. It is simply that the underlying hardware of modern computers does not directly support the notions of objects and messages, thus intervention at some point is required to map between such high-level notions and the facilities provided by the hardware. The finer the granularity of the language, the more frequent (and thus heavyweight) the intervention.

A related measure to granularity is the degree to which languages require run-time support to map between metaphor and hardware. Again at an extreme, Smalltalk is compiled into 'byte-codes' that are executed on a virtual machine (VM); thus the same resulting objects from a compilation may be executed on VMs running on different physical underlying architectures. The Smalltalk VM uses instructions such as 'send message to' or 'create object', and is thus very high-level (though in practice optmizations to make the language execute at an acceptable speed complicate this picture). Java too uses a similar technique of compilation from source into byte-codes that are executed on a VM, though in this case the instructions are at a much lower level and more closely map to the real instruction sets of modern processors. Moving away from the need for a VM is the Eiffel language, which compiles into native machine instructions, but requires its resulting object code to be linked with a run-time environment that provides such features as dynamic garbage collection. At the other extreme of this spectrum lies C++, which in principle compiles via standard 'C' into machine executable objects, and requires no run time support (though nowadays there are many libraries that provide functions for C++ programs that could be considered as run-time environments).

Factors influencing the position of the Deva language on these spectra will be considered next.

Extensions for dynamic grafting of entities and environments

The entity/environment metaphor of the Deva language has features and requirements that place it somewhere in the centre of both the above spectra. In terms of its granularity, the language presents a similar grammar to Smalltalk whereby *from the point of view of its syntax* every programming construct is treated as though it were a first class object (though there are a few deviations from this that allow more compact representation of mathematical expressions than would be allowed by the standard syntax). The decision to impose such a syntax is based more on preference than on any specific constraining factors, and in itself does not necessarily imply much about the supporting environment.

The run-time merging of the two parallel class hierarchies of entities and environments, and the resulting grafting of entity and environment does however hold specific implications for the server's structure. Primarily this is because at compile time, it is not possible to predict in which environments an entity may be instantiated (since it is envisioned that much of the inter-environment movement will be at the whim of the user moving from application to application). Thus, without generating code for all possible combinations of entity and environment, some re-attribution of methods and properties must be performed at run-time, implying the presence of a level of run-time support.

Extensions for autonomous behaviour: independent loci of control

Further justification for a run-time environment is provided by the 'autonomy' of the environments and entities in the Deva model. In many popular object-orientated languages, the paradigm is of a single locus of control that passes through the program, much as in traditional imperative programming languages. That is, the behaviour of a program may be traced at compile time by inspecting the passing of messages, and thus locus of control from object to object.

Attempting to model the real world in which many events occur at once, the Deva model provides multiple concurrent loci of control. That is, explicit synchronization aside, entities are all considered to be executing their behaviours simultaneously.

Since each entity (and by extension, environment) may define a behaviour that continues regardless of interaction with other entities, some form of context switch must be performed within the server with each active entity being scheduled for execution as appropriate. Although it is conceivable that code that explicitly passes control from one entity to another could be planted at compile-time into the resulting objects (much like co-routines in SIMULA [DN66] or BLISS [WRH71]), a more flexible situation that does not imply prior knowledge of all existing entities is to generate code that returns control periodically to some run-time environment (that is co-operative multi-tasking, such as provided by Windows 3.1^{tm}). This relies on 'well-behaved' code, and although this could be arranged in principle using compilation techniques, a far better solution is to arrange for a run-time environment to enforce such context switching amongst a set of active entities (such as the pre-emption of processes in Unix systems).

Extensions for creation and destruction: persistence and dynamic loading

The Deva model addresses issues for a 'grand vision' of multi-application, multi-participant virtual environments, rather than single, stand-alone ones. Implicit in the model is the notion of the persistence of the 'virtual universe', a context where these applications may be situated. In abstract terms, this means that the the model implies a permanent minimal objective universe that exists whether or not it is populated by environments or entities. In more practical architectural terms this means that the system itself persists and provides infrastructural support whilst its specific contents changes in an effectively ad-hoc manner. The presence of multiple users and the autonomous behaviour of entities implies that inclusion or removal of code must be possible at run time without interruption to the services being provided to existing applications. That is, it is not appropriate to 'reboot' the system in order to accommodate a new entity, application or user. Such facilities are clearly provided by the majority of contemporary operating systems which allow multiple users and processes to be created and destroyed without interruption to service (indeed the idea is now becoming more common in the application domain, with word-processors and web-browsers loading 'plugins' at runtime to provide extended functions). The requirement is however architecturally significant, and the facility of dynamically loading and rejecting code 'on-the-fly' must be provided by the virtual machine that forms the main body of the world server.

This concludes the discussion of the provision of support for *structure* within the server. The following sections now examine the functional *content* that must be provided within this structure in order to realistically support the coding of virtual environments.

4.1.2 Server content: the root environments

At the heart of the Deva programming model is the concept of a metaverse graph that describes the properties and features of the available environments. As well as defining a structure for the relationships between entities and environments, practical support of this graph implies the provision of a certain amount of functionality associated with the root environments.

TheVoid, and the basics of existence

TheVoid environment is the root of the entire metaverse graph, and thus in principle contains properties that are to be true of *every entity that may exist in the VR system.* Thus TheVoid represents such notions as "being able to exist", "being able to exist in an environment", "being able to have properties", "being able to imbue properties", "having a name", i.e. the most basic aspects of the VR system.

In practice, since creating such an environment relies on the pre-existence of the properties that it itself is intended to provide, TheVoid is more of a programming metaphor than it is a real environment: its inclusion in the graph is more aesthetic than pragmatic. Nevertheless, the features it represents are fundamental to the functioning of the system.

The features of TheVoid can be summarised thus. Every entity:

1. may be created and destroyed
2. has a unique name: it must be identifiable within the server in order that other entities may communicate with it.
3. has a number of methods: these define the entity's internal functionality as well as the 'vocabulary' by which it communicates with all other entities.
4. has a behaviour: this may be 'do nothing', but may also be independent or autonomous actions that the entity performs regardless of its interaction with the environment or its contents.
5. may inherit properties from other entities: this is more a programming convenience than a fundamental aspect of an entity's reality

Also, there is a form of entity called an environment, which in addition to the above may:

1. contain other entities and environments: indeed this is the primary feature of an environment as distinct to an entity.
2. impose laws and forces upon its contents: it is this ability to impose different laws and forces appropriate to the application that is the rationale behind the metaverse graph structure.
3. define attributes that are true of its contents: this is in order that the laws and forces of the environment may be coherently applied.

Implementing the majority of these features is a lengthy but generally straightforward task that in general does not warrant further detail. For reasons that will be explained later, the exceptions to this are the provision of a single name space and the generation of the 'object code', which will both be described further in chapter 4.

Three Dimensional Space and spatial management

The management of objects in three-dimensional space is traditionally one of the main preoccupations of VR systems. Although this report emphasises many other important factors, manipulation of form in three dimensions must be supported, and in the case of the Deva model, it is the role of the Three Dimensional Space environment to provide such functionality.

Although the server process need not provide any support for actual rendering of its contents (which is delegated to the viewer processes and will be detailed in section 4.3), it nevertheless requires routines for performing transformations in space in order to provide such notions as 'distance between', 'entity intersects with' or 'move entity to position (x,y,z)'. Such notions are concerned with the objective management of space, and do not rely on the user's perception of events: thus they are to this extent a concern of the server. Such routines are non-trivial, though in themselves have little architectural impact, being complex in terms of their mathematics rather than in terms of their structure. However, the perceived results of performing such manipulations in space form the basis for the client/server communication protocol as described in 3.

Time

An effective VR system is subject to a number of 'soft' real-time constraints ('soft' in the sense that failure to meet them is not catastrophic to the system, but results in a potential degradation in presence for the user). The issues of maintaining appropriate frame-rates and reducing device input lag are fundamental to the operation of any VR system. Techniques such as rendering with 'level of detail' [Wil83] and the use of predictive algorithms to reduce apparent input device lag [FSP93a] go some way towards addressing these problems. Putting these most essential requirements aside, however, there are other higher level temporal issues that must also be considered:

Temporal consistency . Although the subjective experience of the passing of time over long periods seems to be affected by numerous external events (e.g. "The weeks just flew by", "the day was really dragging"), on a moment-by-moment basis the experience of the passing of time is constant. In virtual environments, however, it is common to find that one's forward velocity (for example) is determined by the graphical complexity of the surroundings, the number of entities in an environment and so on. The effect is usually disturbing, and a more coherent approach to the passing of time must be sought [HMWH95].

The Time environment provides for this notion, relating events in the virtual world (including the rendering of graphics) to a single consistent notion of the passing of time (specifically, to the real time from the computer's internal clock). This allows entities moving at a certain velocity to guarantee that they arrive at their destination in time, regardless of framerate or system performance.

'Local' time . In a virtual environment, it may be appropriate for the simulated 'local time' to be different to the 'system time'. That is, the environment still needs to be rendered at a certain framerate, and the system still operates at a certain frequency, however the environment is by design presented in 'slow motion'. The Time environment, by default, presents to its entities a 'local time' that is closely related to 'system time' (see 'causality' below), however the environment's behaviour may be modified to introduce arbitrary modifier functions.

Causality

The notion of causality is difficult to discuss without reference to time: the two notions are conceptually very closely related. Nevertheless there are a number of pragmatic issues that are more clearly discussed under the heading of 'causality', and are therefore provided by that environment.

Synchronization Events occurring in the virtual environment may be triggered from many different sources. All take finite and varying times to process, whereas in reality their effect is 'instantaneous'. One of the roles of the Causality environment is to provide for the notion of synchronization, or more generally of 'cause and effect' (i.e. "these events happen together", "this event follows on from and never happens before this one"). One technique for dealing with this issue, similar to that used in [FSP93b] is to allow the 'local time' to explicitly lag behind 'system time' by a small and perceptually insignificant amount (less than 25ms [CMN83]) in order to allow for a certain events to 'catch up' with others. This is, however, far from being a complete solution, and there is much scope for research area of better synchronization in virtual environments.

Processing of events The main pragmatic purpose of the Causality environment is to provide the context for the processing of behaviours, laws and forces. The current implementation of this is simply that all active behavioural methods are executed as frequently as possible. Future implementations of the system will investigate more intelligent constraint-based solutions where laws and forces are activated by certain conditions, rather than being continuously polled.

4.2 Server performance and scalability

For simplicity, the description of the server components so far has assumed that all virtual environments and their contents are managed within a single monolithic process. For the kind of virtual environment common today – that is in effect a single application represented within a single environment, perhaps inhabited by multiple users – a single server like this may be realistic. Under these conditions, the server would be hosting a similar workload to each of the participating visualising processes; perhaps less because it is not responsible for any graphical rendering, perhaps a little more because it is responsible for co-ordinating synchronisation between the clients. However, for the 'grand vision' of virtual reality systems espoused earlier it is insufficient. With many users moving between numerous simultaneously active environments, the workload of the server becomes significant, raising issues of performance. These can either be addressed by waiting for a uni-processor platform of suitable performance to be manufactured, or by employing parallel processing techniques. Opting for the first of these paths has little effect on the architecture but does rely on the unlikely existence of a single computer that is able to continuously outstrip the processing demands made upon it. The latter option, though having major implications on the system architecture, provides a much clearer path towards scaling the processing capability of the server, and will be examined in detail in the following sections.

4.2.1 Parallelism for virtual environments

Parallel processing technology is unlikely to provide a general-purpose technique for optimally parallelizing arbitrary linear code in the near future. However, the world-server contains many specific structures that may be arranged so as to capitalize upon parallel implementation.

Before examining them further, an important distinction must be made between two different motivations for multi-processor design that may both be termed as distribution or parallelism. The first of these (which tends to be described as distributed processing rather than parallel processing) is to provide for geographically separated users to experience a common virtual environment. The second motivation is the provision of platforms that employ multiple processor units in order to address issues of performance and scalability. Having dealt with issues of geographic distribution in chapter 3, it is this second form that is of concern here.

4.2.2 Parallelism in existing VR systems

With some existing VR system architectures it is difficult to distinguish between these issues. In the peer-to-peer architecture of the DIVE system, for example, both geographical distribution and some degree of parallelism are achieved using the single mechanism of a partially shared/replicated database. To the degree that DIVE peers share a common environment, they replicate both data and workload, and incur the associated costs as well as those of synchronization. A network of DIVE processes running different applications, however, can be seen to form a single 'parallel virtual machine' with its workload distributed amongst clusters of processes.

Other systems such as AVIARY , and to some extent The MR Toolkit have a more explicit parallel architecture that is independent from their techniques for achieving distribution. Both these systems provide functional units such as 'collision detectors' or 'object managers' that may be distributed among different processors.

The subjective/objective split between world server and visualiser clients in the architecture developed here addresses specifically the issues of geographic distributability by minimizing the required network bandwidth and providing for some tolerance of latency in communication. This section addresses instead issues of parallelization within the world server in order that it may manage large numbers of environments and entities.

Existing systems that exploit some degree of parallelism spread their workload based on different criteria. For DIVE, parallelism is based the effectively on the individual application, whereas in AVIARY or MR the split is based on function. The explicit object orientation of the Deva model provides for an alternative approach.

In the case of the model developed here, the environments and entities represent the kind of independent threads of execution that lend themselves well to allocation to processors of parallel machines. The model of a general 'processor farm' appears immediately attractive. Here the work to be done is characterised as being a 'workpool' of (predominantly) independent threads requiring execution, and these are farmed out to an arbitrary number of 'worker' processors, any processor being capable of executing any thread. In principle, such a technique scales well according to the ratio of processors to threads. In practice however, in the architecture of the thread-farming system, centralised functions and interdependencies between threads limit the performance and scalability. The structure of the metaphysical model in use here gives some indications as to the best arrangement of the threads and their functions for reducing the number of interdependencies and centralised functions and hence ease distribution.

4.2.3 Extracting parallelism from entities and environments

Per-environment

Figure 4.1 represents three virtual environments (with different driving applications or semantics). Environment A provides some means of access to environments B and C, and contains three entities. Environments B and C each have access back to A, and contain a further three entities each. For the purposes of this discussion, it will be assumed that movement between the environments is *explicit* and *perceivable*; i.e. the user triggers some specific event indicating the desire to move from one environment to the other, and then is aware that such a change of environment has taken place. Where the notion of multiple environments and transportation between them is supported by existing systems, it tends to be in this form, and although other mechanisms are possible, and perhaps even desirable, such 'discontinuous' environments (see [PW98]) provide a clear starting point for the task of identifying interdependencies between loci of execution.

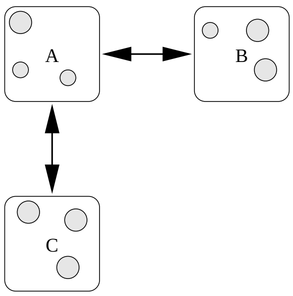

Figure 4.1: Three environments, each containing three entities, with bi-directional routes from A to B, and from A to C. Each environment maintains a degree of independence from the others, and therefore is a candidate for distribution.

The first, and most obvious, zoning of dependencies lies around each environment and its entities.

That is there is little relationship between entities in environment A and those in environment B (indeed, they may not even be aware at all of one another's existence). Within an environment, there is:

1. some relationship between the individual entities, and
2. some relationship between the entities and their environment

From this first approximation, then, environments and their contents can be considered as independent 'packets' of threads with internal dependencies. This is somewhat trivial, and is in any-case rather a course-grain distribution mechanism. Closer examination of the nature of the relationships between environments and their entities, however, reveals subtler and more interesting properties.

Per-entity

Recall that environments are themselves active entities rather than merely being passive containers. They may define their own 'behaviours' for themselves (for example, have a counter that marks the passage of time), and also define properties that are grafted onto any entities that they contain as well as 'laws' and 'forces' that affect their contents in terms of the properties that have been defined for them. It would therefore appear that there is a significant relationship between entities and their environments; perhaps with interdependencies that prohibit useful distribution of workload. Fortunately this is not the case, and the grafting mechanism that may appear to tie environments and entities into a single mass actually provides for a possible finer grain of distribution.

There are in fact three distinct layers of execution in the relationship between entity and environment.

First there is the environment itself, which defines arbitrary functions, behaviours and attributes that it requires itself; although these could explicitly be made visible to entities that enter the environment, they are primarily 'private' to the environment, and entities by default do not know of all the possible functions of all possible environments. These functions and attributes can be described as *pure environmental*.

Second, there is the section of an environment that defines its relationship with all entities contained within. This section contains two parts: functions and attributes that all entities must have; and functions within the environment that operate on those functions and attributes. Although defined by the environment, the former properties are actually given to entities and may be described as 'imbuable' or 'imbued' properties, whereas the latter remain part of the environment, and define its relationship to its contents. These may be termed *impure environmental*.

Third, there is the entity itself, which defines only the properties that distinguish it from other entities, in the knowledge that on creation it will be given any other properties necessary for it to function in a particular environment. These per-entity properties may be termed *pure entity properties*.

Exceptions to these rules are examined in 4.2.3, however taking these as a basis gives the following structure:

Thus an environment may be considered as being a single locus of control, with the contained entities represented by two other loci, one that represents the pure entity and is predominantly free-running, and another that represents the relationship between entity and environment and is bound variously to the pure environment or the pure entity.

The most suitable distribution policy for these three threads will depend on their relative workloads and interdependencies. Whether the impure thread binds most tightly to its pure environment or to its pure entity depends on the nature of the three threads. Two forces can be identified that affect this balance:

1. The influence of the programming paradigm on the style of code generated, and
2. The specific context of virtual environments, which will also influence the style of code.

Both of these influences can be illustrated with a single example:

The most primitive environment in the metaverse graph that allows objects to be perceivable in a three-dimensional virtual environment is Holospace; thus every useful environment will have at least

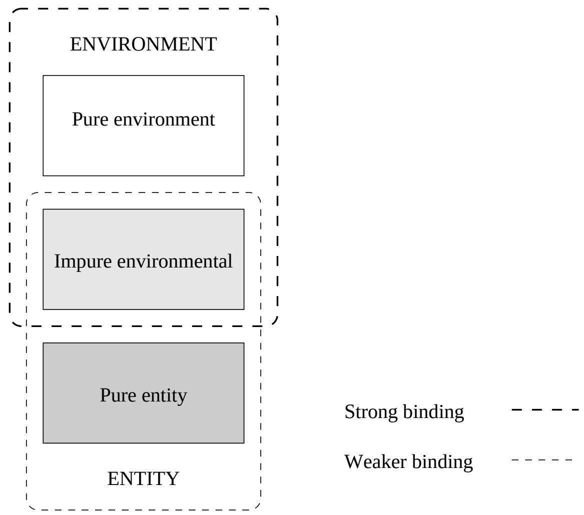

Figure 4.2: At runtime, an environment imbues a set of attributes and code to any contained entities. However, although the memory and resources are associated with the entity, the logical binding between environment (pure environment) and its imbued attributes (impure environmental attributes) is strong, since they are defined in terms of one another.

the properties defined by this, and every entity will have an impure component that also derives either directly or indirectly from the impure component of the same Holospace environment class. Thus the minimum binding and interdependence between the pure environment component (of an environment) and the impure environment component (of an entity) is known; it is as defined by the Holospace environment.

Now the Holospace class combines the functions of Time, Space and Causality as a programming convenience. Though a clean programming model, the object orientated inheritance mechanism used here hides much of the underlying complexity. In particular, the code associated with the management of three dimensional space, and embodied by the Space environment class is mathematically complex and computationally expensive. The relationships between environment and entities from this point of view can be highlighted with a simple example.

The contribution of the Space super-environment to Holospace can be illustrated in figure 4.3.

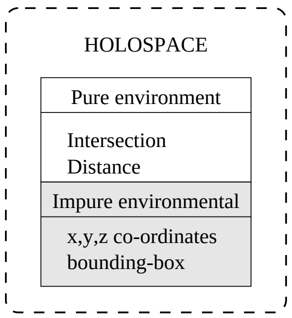

Figure 4.3: The partial definition of Holospace includes functions for calculating the distance between two entities as well as determining the intersection of two spatial extents. These functions are written in terms of the (x,y,z) co-ordinates of the entities, and their bounding extent, which are defined as being mandatory imbued components of the environment.

Creating an instance of a simple object in an instance of the Holospace environment causes grafting of impure properties as in figure 4.4.

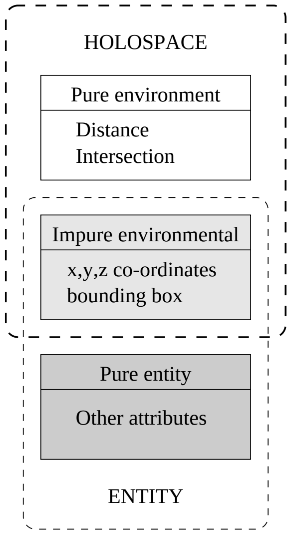

Figure 4.4: At runtime, the environment imbues its 'impure environmental' components upon all entities, which carry the attributes around with them. However, the code is written to marry with its pure-environmental component, and retains a close link with its environment.

This leads to the entity being imbued with the properties x,y and z representing its position in space, as well as a 'bounding box' structure representing its extent.

The influence of context

The main function of the Space environment class is to manage the interactions between entities in three dimensional space. To do this, the 'laws' of Space (seen to be 'part of the environment') must interact with the entities, and specifically with the components of the entities that have been imbued by the environment. Hence in order for the environment to check whether two extents have intersected, it must query the part of those entities that it imbued to see what their state is. Detection of spatial intersection alone is a heavyweight process involving frequent (ideally per frame) interrogation of every entity's position; it is not only mathematically quite complex, it is a centralised service requiring access to all entities' three-dimensional-space related properties. This suggests, by virtue of having some form of intersection detection, that the pure environment and imbued impure environment component of an entity are highly interdependent, and would benefit relatively little from distribution.

The influence of paradigm

Further, since the entity did not know in advance what properties it was to receive, it has no 'pure' functions that refer to them. Similarly, since the environment had no prior knowledge of what kind of entity was to be instanced, it has no functions referring to the entity's pure properties. The environment did, however, know that it was to imbue the entity with x,y, z and bounding-box properties, and has functions associated with these in order to manage the entity's movement through space.

The programming model thus tends to favour a binding between pure environment component and imbued impure environment component, rather than between pure entity component and the imbued impure environment component.

A caveat

The above argument uses the Holospace environment class as an example, for the sake of simplicity. However, this particular choice of class does introduce two more subtle issues that could give a misleading picture if ignored:

1. Since all entities must be created in environments that are ultimately inherited from Holospace, the programming paradigm allows for the pure component of an entity to *assume* it has at least the properties that Holospace imbues. This is vitally important, since the pure component of an entity can only find perceivable expression through the facilities of Holospace. That is, although it may have its own unique state; it can only communicate this to the user by virtue of movement, change of colour, size etc., which are governed by the imbued properties of Holospace. This tends to suggest that the binding between pure entity component and impure imbued environment component is stronger than suggested above (although in practice it has been found still to be the weaker of the two bindings).

2. Although environments do not *by default* know about the pure properties of entities, and vice-versa, the programming model does allow for such forward referencing to occur. Functions may be attached to environments and entities such that when particular pure attributes are encountered at run-time they trigger particular functions. Thus an entity can define a function that says "whenever I am created in an environment that has the notion of gravity, I will behave like so". Such functions are detailed in chapter 4 of 'Beyond The Metaphysics' [Pet95]. These functions, however, are the exception rather than the rule.

With the above caveats however, the argument still stands that there is always a strong binding between pure environment and imbued impure environment components caused by frequent interrogation by the essential and predominantly centralized spatial management functions.

4.2.4 Distribution strategy for environments and entities

This implies that there are two levels of distribution available within the environment/entity model:

1. Course grain, on the basis of individual environments.

2. Fine grain, on the basis of loose coupling between pure entity component and the imbued impure environmental component.

Assuming that some load balancing metric is available to select the least heavily loaded processor, an appropriate distribution policy based on the above distinction is therefore to:

1. allocate each pure environment entity to the a processor.

2. allocate impure environment component of each contained entity to the same processor.

3. distribute the pure entity component of each contained entity amongst the other available processors.

A typical distribution pattern according to this scheme is shown in figure 4.5.

4.3 The client viewers

The client processes are structurally much simpler than the server. There are two reasons for this:

1. They each contain only the entities and environments associated with their particular user's current viewpoint, and hence have a far smaller workload to process, at least in terms of the number of objects they individually contain.

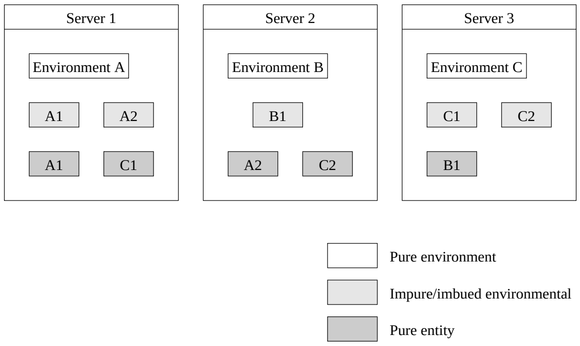

Figure 4.5: A typical distribution of entities and environments. Each environment is allocated to a single processor in the cluster, and it attracts the 'impure environmental' components of any entities created within it. The 'pure entity' components are evenly distributed amongst the available processors, since they have little interaction with their impure components (the exact distribution of pure components in the figure is not significant).

2. They deal only with the *perceivable representation* of those entities and environments, and thus have none of the overheads associated with processing the forces and behaviours that govern of those entities.

The clients may therefore be viewed as denuded forms of the server, executing much of the same code for governing time, space and causality but receiving instructions from the server describing the perceived effects of forces and behaviours rather than processing them themselves. In effect, they maintain copies of the impure environmental components of entities that are imbued by Holospace (i.e. all the attributes associated with the movement and representation of those entities). The relationship is shown in figure 4.6 below.

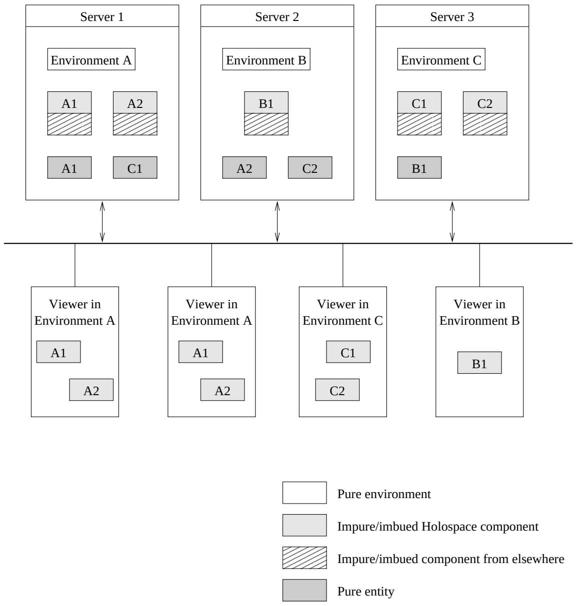

Figure 4.6: The client viewers manage 'ghost' copies of the components of each entity that have been imbued upon them by Holospace. The 'pure entity' component, and anything that has been imbued from user-defined environments beyond Holospace remain in the server.

Their workload is not trivial however. Although they are spared much of the processing load associated with behaviours and so on, and process only the content of a small subset of all possible environments (i.e. those that may be perceived by the user from their current viewpoint, rather than all environments that exist), unlike the server they are responsible for the final rendering of the resulting scenes: this in itself presents a significant workload.

The description of techniques for optimizing display management and rendering are beyond the scope of this report. However the integration of such techniques into the client/server structure is important, and will be examined in detail in the following chapter.

Chapter 5

Implementation

Having developed an overall design for a 'large-scale' VR system, this chapter presents the details of a specific implementation of the architecture. There is therefore a shift of emphasis from consideration of the theoretical consequences of various design decisions to much more practical issues such as selecting an appropriate platform for construction.

The techniques, structures and algorithms presented here represent a specific implementation of the system rather than the only possible implementation. Although the functionality of many of the components described here is fundamental to the architecture, and justification will be presented for the particular implementation, their execution as code may be approached from other angles. Further, this is the implementation of a prototype system, and the emphasis has been placed on exploring the viability of the high-level design presented in previous chapters, rather than providing a 'shrink-wrapped' product that has been optimised in all respects.

The implementation issues are divided into five categories. First those associated with the topology and networking of the system are considered. This forms the basic framework upon which the world server, its clients and any application programs are built. The next three sections examine components of the world server: the implementation of the infrastructure of the Deva model; the compilation and execution of programs within that infrastructure; and the 'support routines' that make it practical to build real environments using the system. The client viewer processes and their relationship to the structures of the world sever are described next, and the chapter concludes with an overview of the two mechanisms available for interfacing applications to the system.

5.1 Selecting a platform

Although supporting the Deva model in practice requires provision of a number of novel facilities, it also relies on many features such as memory, process or file management that are provided by 'traditional' operating systems. Since the emphasis here is on an investigation of the novel features of the Deva environment, there is little point in re-inventing an already well-researched wheel, and a decision was made early on in the implementation process to base the development of the Deva system on an existing, well supported and stable environment.

In practice there are only two 'flavours' of operating system that are generally available: Unix-style, and Microsoft Windows style. Although there are a number of more sophisticated systems such as Arena, Andrew, Mach or Amoeba in development, these tend to be more of the nature of research platforms and do not provide the stability, availability or support for hardware necessary for a realistic virtual reality system. Of the two common platforms, it was decided that Unix , being the more mature, formed a more suitable basis for development.

5.2 Topology and Networking

The system's architecture, being essentially that of a server with multiple clients, is inherently distributed. Whether or not issues of parallelism are considered, the split between the world-server and the client viewers alone implies provision of a protocol for inter-process communication. Examined more closely, the architecture reveals two 'levels' of communication:

1. between processes, in order to form initial connections and perform low level administrative tasks, and
2. between entities/environments in order to provide the functionality of the Deva model.

The requirements of the inter-process protocol will be examined first, since it is the simpler of the two and may then be used to form the basis of the higher level inter-entity message passing routines.

5.2.1 Inter-process communication

The architecture as developed in the previous chapters is that of a single 'virtual server' consisting of an arbitrary number of interconnected processing nodes to which client viewers connect across network links.

Underlying network protocol

A number of different networking protocols exist that may be used to form such a connectivity. These can broadly be categorised as being either 'unreliable' or 'reliable' transport mechanisms. Reliable links either deliver data in the order it was sent or will provide notification of the failure to deliver, whereas unreliable links provide no such guarantees, and may present data out of order or lose it all together. Reliable links such as the commonly used TCP/IP [Com88], being in effect a general administrative layer on top of unreliable protocols, incur a performance penalty for all transmitted data, whereas unreliable protocols delegate the cost of the management of lost or unordered data to the application which may be able to use application-specific knowledge to provide a more efficient 'reliable network layer' (if indeed, such reliable communication is required).

In the specific case of this VR system, as the communications layer forms the basis of the implementation of the programming model (that is, it will be used to form the inter-entity message passing) it must be reliable, since out-of-order or lost messages will result in undefined behaviour of the laws and forces coded in the server. With this in mind, the decision was taken to implement the communications layer on top of the reliable TCP/IP protocol stack. This was further influenced by two factors:

1. Much of the overhead associated with using TCP/IP rather than the Unreliable Datagram Protocol (UDP) is incurred at the time connections are first formed. Within the server, connections are made at start-up and persist for the life of the system. Thus the extra cost of establishing reliable links is absorbed into the general server-start-up sequence.
2. More significantly, the intention of the prototype is to demonstrate the viability of the higher level design features such as the objective/subjective split described in chapter 4. Thus the more generally well understood issues of optimising network traffic are of secondary importance and may be addressed once the overall architecture is established.

Forming the connections: bootstrapping the server

Assuming, therefore, the ability to form reliable connections between processes, a procedure for establishing the point-to-point communications links, and a format for encoding and decoding the data transmitted via these links must be defined. As with many issues concerning the 'bootstrapping' of systems, these factors are tightly intertwined: data may not be transmitted without links, however without a format for decoding that data it is not possible to locate the hosts and ports to which links are required.

In practice, the interconnection of the servers is a process relying on, at first, hardwired communication addresses and message formats, and leading gradually towards more sophisticated and flexible formats for both addresses and messages.

The interconnection process is mediated in its early stages by a nominated 'master' server, which is listening for attempts to communicate on a nominated port address (with such details being provided by 'environment variables' or as 'command-line arguments' to the server's executable). A fixed format message establishes communication between the incoming process and the master server, and details of all other participating processes are sent to the new process. The new process is allocated a unique server identifier (called it's 'peer-id', or PID), and then establishes for itself communication links with all the other existing processes, becoming a fully-fledged peer in the server cluster, and no longer reliant no the master server for information. The process is more-or-less the same whether the new process be a server component or a client viewer, with the exception that viewers only form client-to-client connections if they require non-server mediated communication. From this point onwards, communication between peers is addressed via their unique peer-id which is mapped internally to each process to the ports and addresses of the underlying network protocol.

5.2.2 Message formats

At this layer of the architecture's communications mechanism, a reasonably simple message format is sufficient. The messages sent from peer to peer during start-up contain straightforward structures such as integers, floating point numbers and strings representing port numbers, host-names, peer-ids and so on. There is no need for 'intelligent' decoding of such messages since their format and function will be known in advance (the first message from a peer will be its host name, the first message sent to it from the master will be its unique Peer-id etc.). However, once the initial connections between servers and clients have been established, a more sophisticated message format is required, as messages from servers begin to arrive in less predictable order carrying higher and higher level information.

To address the basic requirements of message passing between servers, the communications layer provides functions for building and decoding messages, and for sending and receiving them on the basis of peer-ids. Each primitive that is appended to a message is prefixed by a type identifier, and is converted if necessary into the 'network' byte-ordering (which overcomes the big-endian, little-endian problems of communicating between different processor architectures). Since TCP/IP provides the metaphor of a single continuous stream of data along communication lines, the packaging of messages into discreet blocks must be done by the communications layer of the VR system. Thus a 'length of message' component is prepended to each message prior to its transmission, and is the first piece of data to be decoded when it is received.

For these 'internal' or 'administrative' messages a single 'type' tag early on in the message is enough to define the message's purpose since the encoding and decoding of such messages is then hardwired into the functionality of the server.

A typical message consists thus of a short 'header', followed by the main body of the message. An example is shown by figure 5.1.

This type of message is sufficient for all the administrative tasks and low-level communications between servers. It is in itself, however, not enough to support the more complex requirements of inter-entity communication implied by the implementation of the Deva programming model.

5.3 Infrastructural support for the Deva model

Once communication routes are established between the processes of the server cluster, it is possible to begin the implementation of functions that support the infrastructural requirements of the Deva programming model. This amounts to the construction of a virtual machine that, as well as supporting the traditional object orientated paradigm, implements the Deva model's entity/environment extensions and runs over a number of connected processors. Though a lengthy task, the implementation of a Deva virtual machine draws from the fields of object-orientated languages and parallel virtual machines, and

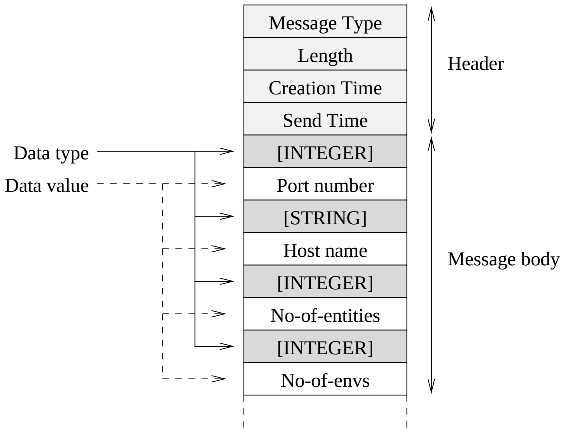

Figure 5.1: Administrative messages have a fixed-format header followed by an arbitrary number of 'type' and 'value' data pairs.

as such contains much functionality that is already commonly understood and well documented. The remainder of this section concentrates therefore on areas of specific functionality that are unique to the Deva virtual machine.

5.3.1 Inter-entity communication

The administrative messages used to establish and maintain the server cluster can be implemented using the simple message format described in section 5.2.2. This is made possible by two factors:

1. The messages are addressed to the servers themselves, which are uniquely identified by their peer-id (i.e. a simple mapping from peer-id to network address will determine the underlying communication link via which to send the message)
2. Once it has been determined what the type of the message is, it may be decoded by hard-wired routines within the servers since it is known to have originated from corresponding hard-wired routines in another server.

For messages passed between entities and environments during the execution of their behaviours, laws and forces, the situation is much more complex. First, the messages are addressed to entities rather than to physical processes. Therefore the mapping between an entity's identifier and the network connection that will ultimately carry the message to another processor is more involved. Second, the content of the message contains arbitrary data structures that are meaningful only to the receiving entity and not to the associated server.

There are therefore two issues to be examined: the routing of messages between entities via their supporting servers; and the encoding and decoding of the arbitrary data structures that may be transmitted between entities.

5.3.2 Routing message: Names and Distributed Name Serving

A significant task in any distributed system is the allocation and resolution of 'addresses' to 'objects'. Whether this be at the low level of finding addresses for bytes of data in virtual shared memory, or the higher level of assigning domain names to computers on the Internet, many of the difficulties such as ensuring consistency and uniqueness are common to both.

There are two extreme solutions:

- Have a central source of definitive addresses, through which every request for assignment or resolution is passed. This guarantees uniqueness but introduces a significant bottleneck.

- Distribute the allocation of resources. This decentralised approach alleviates the bottleneck of the former solution at the expense of the necessity for some protocol to ensure uniqueness or to arbitrate in clashes of allocation.

Practical solutions are rarely of these extreme forms, though 'low level' memory management type tasks often tend toward being 'centralised' (since the appropriate functions may be built into hardware and therefore made relatively lightweight), with 'higher level' tasks such as the mapping of logical domain names onto physical network addresses being more de-centralised (in this particular example, being mediated by a hierarchy of increasingly authoritative name servers).

In the VR system described here, the allocation of addresses to resources is at neither extreme, with the unit of distribution in the architecture being the entity (rather than 'the byte' or 'the computer'). Thus an appropriate scheme for address allocation at this level must be sought.

There are two aspects to be considered:

- Initial allocation of unique address to object
- Resolution or 'dereferencing' an address in order to access an object.

In the context of this system, there are two factors that influence the design of an address allocation scheme. First, the creation of an entity is a *relatively* heavy-weight task in itself (compared, that is to the general processing of that entity and its behaviour once it has been created). Thus the initial allocation of a name to an entity may be 'absorbed' within the time taken to perform other functions necessary for entity creation. Second, dereferencing of an entity's name is a frequent occurrence, being in fact fundamental to inter-entity communication, the enforcement of laws and the processing of behaviours etc. Thus it is an important part of the general processing of an entity and hence is a significant factor in the overall performance of the system.

The following scheme takes into account the degree of leeway available at entity creation and trades this cost against rapid de-referencing of the resulting addresses.

5.3.3 The distributed name serving mechanism

The following sections describe a mechanism that may be used to route communications between entities within the VR system.

Names and Addresses: some terminology

The Deva programming model is based around a single construct: the calling of a method upon an entity. Within the programming language, the method is defined by a textual name, e.g. 'initialise', and the entity is also given a unique textual identifier, known as it's 'name'[1]. For example:

```
myCar: startEngine.
```

This reads as "call the method 'startEngine' of the entity named 'myCar'." In this example 'myCar' is formally the *name* of the entity. In this discussion the term *address* will be used specifically to denote whatever information is necessary to locate the named entity within the VR architecture. Thus the name of an entity is given to it by the programmer; the address is allocated by the system.

The basic mechanism

The mechanism described here may be called a 'distributed name server' (DNS), since it alleviates potential name-lookup bottlenecks by spreading the allocation and de-referencing of names across a number of processing elements.

[1] In fact there are scoping rules in the language that allow the same textual identifiers to be used more than once while referring to different objects where the context is unambiguous. However for the purposes of this discussion, it will be assumed that all identifiers are textually unique.

The DNS mechanism consists of an arbitrary number of processors that form the distributed world-server described in chapter 4. Each processor maintains its own pool of entities (in the Entity Manager), and has reliable 'point to point' communication with all other servers in the cluster (this communication may by physically or virtually point-to-point; either way the exact details of the underlying transport mechanism has no effect on the logic of the name-serving mechanism and will not be considered further here).

Each server is allocated a natural number 'Physical Server Identifier' (PSI) that is unique within the cluster. The pool of entities within each server is indexed (also by a natural number, its 'entity index' EI). In combination, the tuple (PSI, EI) uniquely identifies an entity's physical location within the server cluster.

Recall from section 4.2.3 in the previous chapter that entities are allocated to processors based in part on the environment to which they belong, and in part on the current loading of the various server components. Although particular patterns of entity distribution are more likely than others given the dynamics of the allocation algorithm, in principle any entity may be allocated to any processor. Again, though certain entities are most likely to communicate with other entities in their environment, it is also possible for them to communicate with any other entity in the system. Thus there are certain common patterns of allocation, the mechanism must be able to locate entities anywhere in the system.

The proposed technique is this. Each server is configured with a 'local directory manager' (LDM) component, which maps between an entity's name and its (PSI,EI) tuple. Each name is allocated *definitively* to one of the system's LDMs, which is selected purely on the basis of a hash-function applied to its textual content. That is, the management of names is spread across the distributed components of the server evenly, but with no regard for the allocation of the entities, and thus the definitive LDM for a name may be a completely different physical processor to any of the entities processable components.

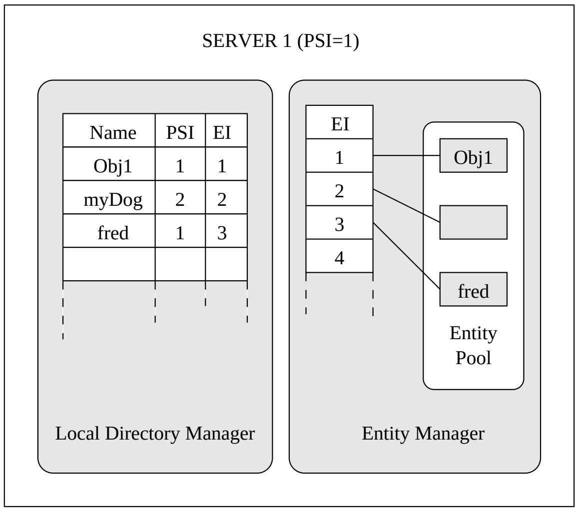

Figure 5.2: A single server, showing the Entity Manager and Local Directory Manager. Names in the LDM are accessed via a hash function applied to their textual content. Once their entry in the LDM is found, the (PSI,EI) tuple that uniquely identifies the position of the entity in the cluster can be determined.

Each name thus has a single definitive point within the system where the mapping from textual name to (PSI,EI) pair may be made. Since every server has access to the same hashing function that was used to distribute the names, it is possible for each to calculate the PSI of and subsequently query the definitive LDM and thus find the physical location of an entity.

Although this technique removes the single-name server bottleneck, it does imply that a number of messages must be sent from server to server in order to find the physical location of an entity. The pathological case is when the message originator, the LDM and the entity all occupy different processors, in which case 3 messages are generated:

- Originator calculates PSI of LDM, and sends request for name lookup.
- LDM replies with (PSI,EI) pair.
- Originator sends message to entity EI at processor PSI.

Even this worst-case scenario however is an improvement over having a single centralised name server, since it requires the same number of messages as a normal name lookup, but spreads the workload over a number of processors.

Name caching extension

This situation, however, may be greatly improved by using caching techniques. In addition to being host to a number of definitive names, each server maintains a copy of any name lookups that have been made. These are flagged as being non-authoritative, but are used as a first-guess when a name lookup is requested.

The mechanism guarantees that changes to an entity's status (for example it's deletion, or migration to another physical processor) are always reflected by the name in the definitive LDM. However, the non-definitive copies are not necessarily updated, and thus de-referencing these may lead to messages being delivered initially to the wrong addresses.

The name resolution mechanism therefore attempts to determine the (PSI,EI) pair of an entity from its name by examining it's own LDM first. If it finds a definitive name, it may safely send the message to the appropriate server. If if finds a non-definitive name, it attempts to send the message to that server, but waits for a reply of either 'successful delivery' or 'return to sender, address unknown'. When the latter occurs, the server determines the definitive LDM for that name, requests the updated (PSI,EI) pair, and resends the message which now is guaranteed to arrive at the correct entity.

Such an algorithm has the following desirable properties:

1. The workload associated with name resolution and allocation is distributed evenly across all participating processors.
2. The replication of names is kept to a minimum: there is a single definitive copy, with up to one additional non-definitive copy being made by each server only if entities on that server require it.
3. The initial cost of name resolution is *at most* three message (see above pathological case), and at best requires no actual messages (when name, LDM and entity are co-existent on the same physical processor).
4. The majority of name lookups result in a single message being sent to the correct address first time, with invalid addresses being 'automatically' corrected using three further messages.

'Virtual server' extension

The algorithm as described in the previous sections provides a degree of independence between the physical location of an entity and its name. However, all the internal tables (such as those in the LDMs) are indexed in terms of PSIs, and thus the migration of an entity from one server to another (for example, to achieve load balancing) requires all references to the PSI of that entity to be updated. This need may be alleviated by introducing an extra layer of indirection between name and physical server. That is, rather than referring directly to a physical server, the location of an entity is identified by a Virtual Server Identifier and an Entity Index pair, (VSI, EI).

A large number (in the order of thousands) of VSIs are mapped 'round-robin' onto the available Physical Servers, so that each Physical Server is represented by an equal proportion of the Virtual Servers, and that each Virtual Server maps to one Physical Server. The Entity Manager's lookup-table is also segmented into a number of 'virtual tables', forming 'virtual address spaces' associated with

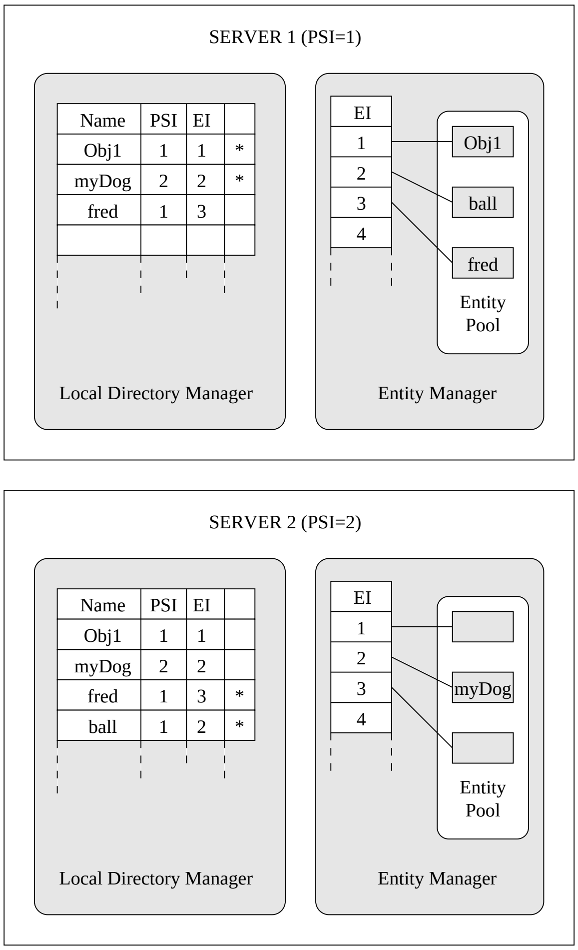

Figure 5.3: Extending the LDM to have a 'definitive' tag (shown by an asterisk in the figure) allows the LDM to accumulate non-authoritative information about other entities in the server cluster. `Obj1` has both its Entity and Name hosted on Server 1. The `myDog` entity is hosted on Server 2, but has a definitive name in the LDM on Server 1. The `fred` entity is hosted on Server 1, but has a definitive Name in the LDM on Server 2: the reference to `fred` in the LDM in Server 1 is a cached, non-authoritative name to speed up access.

each of the virtual servers. The mapping between VSI and PSI is replicated and synchronised across the participating processes using low-level administrative messages to gain exclusive write access to the map.

The benefit of such a map is that all entities associated with a particular Virtual Server may be migrated *en masse* from one Physical Server to another in order to redress load imbalance without the necessity of updating the cache entries in all processes. Their references to the entities will still contain valid Virtual Server values, however these will now refer to a different Physical Server.

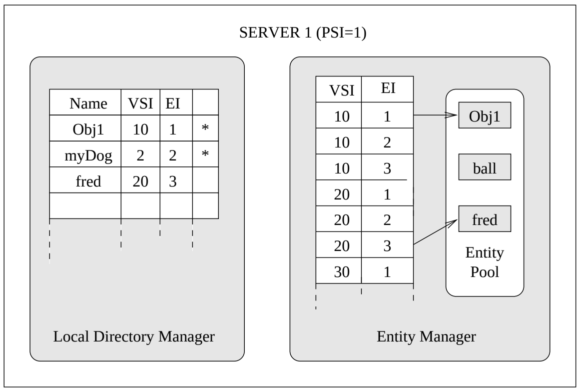

Figure 5.4: The 'virtual server' extension to the name-lookup algorithm. Referring to entities by their Virtual rather than Physical Server allows blocks of entities to be moved from one physical server to another with minimal impact on the cached name-lookup values that will have accumulated in the applications.

Encoding and Decoding messages

With the distributed name serving layer providing rapid addressing of messages to entities throughout the system, it remains to describe a mechanism for encoding and decoding the information contained within those messages.

At first this would appear to be a somewhat complex task, since unlike the administrative messages described previously, inter-entity messages contain the arguments and return values of the methods defined by the entities and environments. Thus they are the carriers of arbitrarily complex data structures rather than a small number of fixed format primitives as is the case with the inter-server administrational communication.

A similar situation exists within the AVIARY VR system [], which uses Sun Microsystem's XDR (eXternal Data Representation) functions for packaging up its equivalent of inter-server messages, but has a more complex 'Self Describing Data' (SDD) mechanism for transforming complex application data structures into linear streams of bytes that may be transmitted between processes. This mechanism involves each data structure containing a field that describes the format and contents of the structure. This is then used at run-time by AVIARY's message-passing layer to convert between data structures and messages. Though AVIARY's SDD is a powerful tool, allowing the 'flattening' of complex nested structures, its run-time cost is significant, and it also relies on the programmer to supply correct data-structure definition strings.

An alternative approach to such run-time re-structuring of data structures is to arrange for both senders and recipients of messages to be given bespoke code for packing and unpacking their contents. That is, rather than hand over a parcel of data and its associated meta-data to third party code for 'intelligent' packing and unpacking, both parties involved in a transaction agree beforehand on a format for the data. This has the disadvantage that the resulting data-streams are only meaningful to sender and recipient, since they contain no meta-data describing their format, but has the significant advantage of allowing very rapid construction and deconstruction of messages and data structures. Such a technique was proposed by Snowdon in [Sno96] though was never implemented within the AVIARY system. However initial trials of a prototype suggested a thirty-fold improvement over the SDD approach.

Though recommending itself on the basis of speed, the generation of code for packing and unpacking such messages relies on a higher-level description of the contents and purpose of the data. In the Deva model, such meta data is provided by — and indeed is fundamental to — the high-level object-orientated language that is used to describe entities and environments. By examining the signatures of the arguments and return values of the methods in the language, code can be automatically generated that packages up data into an appropriate structure to call a method, and unpacks it within the body of the method once the message is received.

For example, an entity defined as:

```
entity Sphere
begin
  methods
     NULL x: Float y: Float z: Float.
end
```

will always have its method 'x:y:z:' called in the form

```
mySphere x: 10.5 y: 1.5 z: 3.2.
```

since any other arguments that do not reduce to three floating point numbers would cause compilation errors. Thus both message sender (i.e. the code calling the method) and message recipient (i.e. the code defining the method) know the exact number and type of the arguments that are provided, and so the compiler can plant appropriate code for generating and decoding the message.

Although in this prototype implementation of the Deva virtual machine, no other message packing mechanisms have been attempted, using the high-level information available in the Deva language, and planting code at compile-time rather than interpreting structures at run-time would appear to give an optimal solution: other than allocating a buffer for the message in memory there is little else that needs to be done at run-time.

5.4 Translation and execution

Unlike contemporary VR systems, where objects are primarily graphical representations to which behavioural scripts may be attached, the Deva model considers objects (that is, entities) to be described in the main by their behaviour and attributes (one of which 'happens to be' be their graphical form). This shift of emphasis implies comprehensive support for the translation and execution of the descriptive and behavioural code associated with an entity.

Execution of Deva code implies the provision of a number of functions:

1. Run-time grafting of methods from environments to their contained entities
2. Dynamic loading and unloading of code as new entities are defined.
3. Multiple loci of control to support the autonomous behaviours and forces.

The gap between these high-level 'application specific' concepts and the general-purpose low-level instruction sets of modern processors must be bridged in order to execute programs written within this framework. This requires two developments: first a 'virtual machine' that executes on existing platforms and provides an appropriate 'virtual instruction set' for the Deva entities and environments; and second a compiler that translates from source code into this instruction set.

The selected implementation platform – that is Posix compliant Unix – already forms a layer of 'virtual machine' on top of whatever underlying hardware is used. It provides an environment with such notions as 'processes', 'priorities' and 'system calls' and is thus much richer and more complex than any of the hardware platforms upon which it may execute. However, being a general purpose environment, it does not provide any of the more specialised features required by the Deva model. The following sections examine how such functions may be implemented on top of a Unix system.

5.4.1 Compilers, interpreters and virtual machines

As discussed briefly in the previous chapter, there are two extreme forms of language translation: interpretation and compilation. The former, as historically exemplified by BASIC, and more recently by scripting languages such as Tcl involves the run time association of tokens in the source code with 'high-level' instructions in a virtual machine. It can be seen as consisting of a 'thin' translation layer (since its tokens are close to the original source code) and a 'thick' virtual machine (which has high-level instructions of significant complexity). The latter, as used in C, Pascal and the majority of traditional imperative languages involves the compilation of the source into low-level machine instructions. This can be seen as having a 'thick' translation layer (since it translates from high-level concepts into very low level instructions) and a 'thin' virtual machine (which may just provide 'system calls' rather than any further interpretation of instructions). In between these two extremes are languages such as Java, Perl and SmallTalk wherein a 'partial' compilation into 'byte-codes' is performed, which are then interpreted and executed at run-time by a virtual machine. A full investigation of the pros and cons of the various techniques is beyond the scope of this report. However the broad spectrum of languages still in common usage today is testimony to the fact that no clear winner has emerged in this field: the best that can be said is that the different techniques are appropriate in different application domains. A decision, nevertheless, needs to be made as to an appropriate translation technique for the virtual machine of the world-server being developed here.

The extreme of interpretation may be set aside easily: it is envisioned that computationally complex behaviours will be specified, and that overhead of their runtime interpretation would be unacceptably high. The extreme of total compilation into machine instructions is also problematic, since the execution of the resulting machine-code would require hardware or kernel level support for the novel features of the Deva model making porting the code to heterogeneous architectures difficult. This leaves the option of partial compilation into instructions suitable for execution on a virtual machine.

Contemporary languages that utilise this technique do so by defining a definite virtual instruction set that their compiler targets, and providing the virtual machine to execute this on a number of platforms. Java's instruction set, for example, resembles that of a realistic processor (which, indeed has now been constructed in hardware as Sun's 'Javastation'), Smalltalk's is of a higher level and is defined in terms of objects and messages, and Perl's is higher-level yet, providing instructions such as 'match pattern in associative array'.

Each of these approaches involves, in effect, the interpretation of byte-codes that represent virtual instructions of various levels of complexity. The fact that these byte-codes are being interpreted by a virtual machine, rather than executed directly on some underlying hardware, gives a significant degree of control over the execution environment. This must, however, be weighed against the cost of the instruction-by-instruction interpretation.

An alternative approach to this was taken by Bjarne Stroustrup during the initial design of compilers for the C++ language []. Instead of defining an instruction set *per-se*, the C++ language was designed to compile into C source code: thus its 'virtual instruction set' may be considered to be 'all the functionality of C', and its virtual machine a combination of a C compiler and its operating system. Although contemporary compilers for the C++ language now compile directly into machine code rather than indirectly via C, all the constructs of C++ can nevertheless be represented (albeit in a convoluted and entirely human unreadable form) in the simpler C language linked with a run-time support library.

A similar technique was applied to the Deva language, targeting it not at C but at its object-orientated variant, C++. Compared with the alternative 'byte-code' techniques, this has three main advantages:

1. It avoids the run-time overhead associated with interpretation (which is important for an application area with so many real-time constraints as Virtual Reality).

2. It provides a degree of platform independence, since compilers for C and C++ are available on almost all architectures.

3. It generates efficient code without the necessity of writing a new compiler, since compilers for C and C++ have already been the subject of much optimisation research.

Compared to byte-code compilation where the virtual machine may be made as sophisticated as necessary, this 'piggy-backing' of compilers has, however, potential disadvantages arising from the fact that the executable code, once generated, is handed over to a operating system (e.g. Unix) to be run on a host processor (say, an Intel or MIPS chip). However, as mentioned above, no contemporary operating system/processor combination explicitly supports the Deva model, and thus the success of this partial compilation technique depends upon the degree to which the standard features of a contemporary operating system may be made to support the specific requirements of the Deva VM.

5.4.2 Making Unix into a Deva VM

The Unix programming model provides the notion of a 'process': an independent locus of control representing a program and all its associated resources. A process is represented by pre-compiled executable instructions assembled in a file on disc. On execution, the file is loaded by the operating system and executed to completion as a single locus of control. If the executable completes, then any resources such as memory it has used are reclaimed by the operating system for future processes. A scheduler in the operating system allows an arbitrary number of such processes to run on a single processor, giving the illusion of 'more than one thing going on at once'.

At first glance, such dynamics appear to be similar to the creation and destruction of entities within the Deva VM. However, for Unix the creation, destruction and execution of a new process is a heavyweight task, with each process being allocated virtual resources and a virtual address space, and claiming a significant overhead of management during its lifetime. A typical Unix system can be seen to execute a few tens of processes at once, whereas it is intended for a Deva VM to be managing hundreds if not thousands of entities.

Therefore, although the Deva compiler could be coded in such a way as to generate fully-fledged Unix processes that communicate with one another and an overseeing 'Deva' process, in practice this would be far too heavyweight an option. For efficiency, therefore, the Deva VM must be constructed within a single Unix process, rather than using Unix processes to provide its entity control. However, if the 'piggybacked compiler' option is still to be taken, this generates three problems:

1. First, there is the issue of dynamically loading the compiled code into an already existing process. If the code was a file containing byte-codes rather than an executable, the opening and reading of the file and the interpretation of its contents would be a trivial matter. However at the level of control provided by contemporary operating systems, executing a compiled program is an 'all or nothing' matter.

2. Second, within a single process there is traditionally a single locus of control. Thus arranging for the multiple loci required by autonomous behaviours becomes more complex (though again, if this were byte-codes, then swapping the interpretation from file to file would not be a problem).

3. Third, within a single process there is traditionally no operating system support for memory protection (aside from the code segment/data segment distinction), and a single process is free to corrupt much of its own virtual address space. Once more, if the program was executed by interpretation of byte-codes, these could be inspected at run-time for potential memory or security violations (as is the case with Java).

There are, however, straightforward solutions to these problems. An operating system is, after all, a kind of virtual machine itself, presenting a more sophisticated programming model to the application than is actually supported in hardware by the processor. To an operating system then, executable program files are little more than files of 'byte-codes', and processes are just a programming convenience rather than something inherently supported by the underlying hardware (that is, hardware may provide interrupts and multiple program counters, but the notion of a process or a scheduler is a construct of the operating system rather than the hardware). Modern implementations of the Unix operating system, for example, support two standard mechanisms – ELF libraries and Posix-Threads [10096] – that overcome the apparent restrictions of the process construct, and allow multiple loci of control and dynamic loading of executable code to be performed, allowing functions previously available only to the kernel programmer to be made visible at application level.

5.4.3 Dynamic Loading: entities on demand

Dynamic loading is the name given to the introduction of executable code into an already executing program. The technique has been used for some time in operating systems to reduce the size of the binary executables by providing 'dynamic libraries' of common functions which are linked at run-time, rather than being included wholesale at compile-time. With the stabilization of the ELF [] format on Unix platforms, the ability to perform run-time loading and linking of code has been made available to the application programmer, resulting in the idea of 'plugins' as a means of extending the functionality of applications by the addition of new or third-party code.

The Deva virtual machine relies heavily on dynamic loading, since it considers itself to be a provider primarily of form (i.e. that is the metaverse graph paradigm) rather than content (i.e. the environments and entities). Thus the virtual machine contains a minimum of functionality necessary to maintain the inheritance structures of the Deva model, and then loads in 'on the fly' all the code associated with the system-defined root environments and any user-defined entities and extended environments.

ELF-style dynamic loading

The current prototype is based on the ELF standard for dynamic loading of code. Three functions are provided by the operating system: 'load code', 'call named function' and 'unload code'. Combined with the meta-level information about the structure of entities contained in the Deva language, these three functions are a sufficient basis on which to build the virtual machine's dynamic entity loader.

The loading of an entity's code into the VM occurs 'on demand'. That is, when the execution of a method involves interaction with an entity type that the VM has not encountered before, it searches a well-known place for new code to describe that entity. If an appropriate file is found, the ELF function to load and link that code is called. Assuming that the loading is succesfull (and since the code was generated by the compiler rather than by hand, it should contain no unresolved references), the VM searches for a specifically named function within that code that forms its 'entry point' (that is, a function called _deva_main that has the same purpose as the _main function in a traditional C program). If this entry point is found (and again, it certainly will be since the code was generated automatically), it is executed, and this then informs the VM of all the other meta-information regarding the introduction of the new entity. This information includes:

- For an entity:
  - It's name.
  - The names of any entities it inherits from.
  - The number and names of its attributes and their types.
  - The number and names of its methods and their arguments.
  - Whether or not it has a behviour.

  and, in addition for an environment:

  - The number and names of any attributes it imbues upon its contents.
  - The number and names of any methods it imbues upon its contents.
  - The number and names of any forces or laws that are enforced upon its contents.

Support for entity inheritance

The object-orientated inheritance mechanism means that not all the functionality of an entity is described within the textual body of its code. That is, some of its methods and attributes may be inherited from a parent or super-class. Although it is conceivable for the compiler to aggregate all functions from super-classes into their child classes, so that code representing all inherited functions appears in every inheriting sub-class, this is wasteful of resources and also forces the re-compilation of all inheriting classes should a method in a parent class be altered.

A more appropriate solution is to force the loading of parent class code into the virtual machine whenever an object from a dependent child class is created. This is conveniently, though indirectly, supported by the ELF-dynamic loading functions. Although designed for the loading of dynamic shared libraries, rather than object-orientated entities into a virtual machine, the ELF standard allows libraries to define at compile-time other libraries that they depend upon for functionality (for example, the X-Windows library requires the c-function library). This list of other libraries is stored within the ELF-format binary, and is parsed at load time by the dynamic loading functions, forcing other libraries and in turn any code they depend upon to be loaded into the executing program. Thus at compile time, the Deva language's compiler need only plant a list of 'required libraries', or more specifically 'parent classes' within the binary, and the ELF functions will automatically handle the resolution of code from the super-classes.

5.4.4 Lightweight threads

The Deva programming model defines entities and environments as having some degree of autonomy, much as processes do in Unix systems. That is they execute independently within their own contexts, with communication between them being via explicit method calls. This mode of functioning requires some support from the virtual machine, since it is not, by default, the behaviour within a standard Unix process.

The ability to jump between different loci of control within a single process in this way is known as 'threading', and has historically been provided by a wide variety of different libraries: [] [] [], though it is only very recently that a definitive standard for multi-tasking – the Posix-thread, or p-thread – has been agreed.

The autonomy of entities presents a definite requirements for a multi-threaded environment. It is in addition required by another more pragmatic aspect of the virtual machine. This arises from the need to suspend the execution of a function when it makes a method call to another entity. For entities that exist on the same server in the cluster as the calling entity, such method calls may be internally resolved by a function call. For entities that reside on remote servers, such a method call results in a pair of messages carrying the method request and its result. While waiting for a reply from the remote entity, the calling entity may elect to suspend its own action, returning control to any other thread that is capable of running, since failure to do this results in the server–and therefore all behaviours and forces–'hanging' until the reply is received. Traditional, monolithic code does not have the ability to suspend its behaviour at arbitrary points, execute alternative code in the meantime, and then return to its previous task.

The threads model considers a process as consisting of two components:

1. some shared resources, such as program instructions and global data, which is thought of as *the process*, and

2. information related to the execution state, such as a program counter and a stack. This is called a *thread*, and there may be many of these within the process.

Threads represent multiple concurrent loci of control *within* a single process. Avoiding the heavy-weight context switches involved in swapping virtual memory segments and process related resources, they provide a lightweight, efficient means of performing pre-emptive multitasking at the minor expense of requiring explicit synchronization code to provide coherent access to the shared resources.

Threads in the Deva VM

The implementation of a fully blown multi-threaded server is a complex task. To avoid unnecessary message passing, entities and environments in the VM share many data structures with one another and with the VM itself, and thus a sophisticated locking mechanism would have to be introduced in order to protect the vital shared structures from damage by competing interwoven threads of execution. In principle, such locking code is straightforward, and techniques such as mutual-exclusion locks, spin-locks and semaphores are well understood and either provided by the p-threads standard or may be trivially built from its primitives. Further, since the meta-level information about which resources are required by an entity is all held within the high-level Deva language description, the automation of the generation of such locking code is again a straightforward, though rather lengthy task.

In order to avoid this difficulty, and also in order to provide an environment that can be de-bugged with relative ease (since faults in multi-threaded programs are notoriously difficult to track down), a simpler threading mechanism was implemented in the prototype server.

This involves only one mutual-exclusion lock (mutex), which is passed around like a token between the server's main body of code and the entire collection of executing environments and entities. Thus, although there are a large number of threads active at any one time within the server, all but one will be held up waiting for the single mutex to become free. Since only one thread is ever active, there is no need to lock data structures since it is known that the locus of execution will only be in one thread at any one time, and will be passed on at well defined points. The behaviour is therefore a passing back and forth of a token between the server and its entities: the server performs some work, such as reading incoming messages, then passes control over to the entities which may execute methods, perform behaviours and so on before passing control back to the server.

Ironically, this compromise reduces the pre-emptive power of the p-threads standard to being a partially co-operative threading scheme: should any piece of code be 'badly behaved' and retain the mutex longer than necessary, although the p-thread kernel has the power to schedule other threads, none would be prepared to run until the lock had been released. The compromise relies on the application level code explicitly returning control within a set time and also at some point in its execution where shared structures are left in a consistent state. However, the scheme has no real performance overheads, is simple to implement, addresses the requirements of suspending code pending method call replies, and has the added benefit of being significantly simpler to debug than a 'true' multi-threaded virtual machine. The 'true' multi-threaded version may be built upon it at some appropriate point in the future.

5.5 Content: implementing the root environments

The five root environments provide what could be considered to be the 'traditional functions' of a Virtual Reality system. That is, in combination, they provide mechanisms for MΦmangaΦΦaginΦΦing space, scheduling events, rendering objects and so on. From a purist's point of view, in terms of the Deva model, they are of secondary importance to the structures and form provided by that model. From a more pragmatic point of view, however, they are an essential part of the functionality of the system and the efficiency of their implementation governs the final usability and performance of the whole system.

It is at this point of their actual implementation that the theory and practice of environments diverges. It is foreseen that there will be times where application programmers may wish to experiment with notions of Time, Space and Causality (indeed, initial work on 'causal culling' by this author and colleagues as well as the 'spatial model' developed by [BBC$^+$93] are experiments of this fundamental nature), and therefore there is a real need for such notions to be included as first class environments in the system.

However, for ease of implementation and in order to build support for as much of the Deva model as possible, the features provided by these basic environments have been merged, and implemented as a monolithic 'Holospace'. The various aspects of this monolithic environment contributed by the components are examined in the following sections.

5.5.1 Spatial and display management: integrating with the MAVERIK micro-kernel

The MAVERIK virtual reality kernel, which has been developed alongside the Deva system, addresses none of the issues of multiple participants, multiple applications, coherence of experience or geographical distribution and so on that are the primary concern in this architecture. Its main purpose is to provide a highly configurable framework within which optimised rendering and spatial management routines may be coded. Providing as it does a set of extensible techniques for dealing with objects in three-dimensional space, but assuming little about the data-structures that will describe them, MAVERIK forms the ideal basis for the spatial management component of the server and specifically the functions provided by ThreeDSpace environment.

Central to the MAVERIK philosophy is the idea there is no single canonical format for storing the data that represents a virtual environment. Rendering techniques will suggest certain organisations or configurations of data that may be particularly efficient for their own needs, but unsuitable for others. Thus the MAVERIK micro kernel considers all data associated with spatial management or rendering to be 'outside' itself. It places no restrictions on the structuring of that data, and provides instead a framework that allows the kernel to 'hook into' the data, and to interact in its own native format.

The two interesting constructs within the MAVERIK system that are of particular interest here are the 'Object' and the 'Spatial Management Structure' (SMS), the former being 'something that is rendered' and the latter representing the space occupied by those rendered items.

MAVERIK is an object-orientated kernel, and both of these concepts are represented by classes. The system providing a number of 'predefined' classes for them, but encourages these to be adapted or replaced to suit application specific needs. In Deva terminology, a MAVERIK Object is the part of an entity that exists in ThreeDSpace, and an SMS is an algorithm for managing the ThreeDSpace itself.

MAVERIK Objects

A MAVERIK Object is an encapsulation of the data and functions required to render a component of a virtual environment. Information is registered with the MAVERIK kernel, instructing it on what to do if ever the object needs to be rendered, if ever something wishes to know its bounding volume and so on. This is performed using 'call-back' functions: that is the native functions of the object are registered with the kernel to be called at a later time, rather than converting the data structures of the object into some standard format dictated by MAVERIK. This technique provides a standard interface to diverse rendering routines without forcing a standardisation of data structures, and also avoids replication of data and its associated resource waste and synchronization cost.

Via their callback functions, MAVERIK Objects provide the functionality necessary for:

1. Rendering the Object at a given position in space.
2. Accessing and manipulating the transformation matrix that governs the position, rotation and scaling of the Object.
3. Determining a bounding box that fits around the Object.
4. Determining whether a vector intersects with the Object.

MAVERIK Spatial Management Structures

Spatial management is a core function of a VR system, and is required in order to perform such tasks as culling, object selection and collision detection. A MAVERIK Spatial Management Structure is an abstraction of a three-dimensional space that may contain objects. For many applications involving spatial searching, a hierarchy of bounding volumes may be used, and this generally works well because of its logarithmic complexity. However, as with rendering routines, it is possible to find alternative structures that capitalize upon particular application specific features. Therefore, as with Objects, the MAVERIK micro-kernel predicates little about the way in which space is managed, and instead requests that each different spatial management algorithm registers call backs providing such functions as finding an object within its structures, finding the first object intersected by a certain vector.

MAVERIK supporting modules

In the same way that the Deva model defines relationships between entities and environments but provides little actual functionality, the MAVERIK micro-kernel itself provides only the minimal functions necessary to manage the relationships between Objects and SMSs. Although in principle a virtual environment could be built from kernel-level upwards, much of the tedium of doing so in MAVERIK is alleviated by the *supporting modules*, which provide standard rendering techniques and spatial management algorithms which may be used 'out of the box' or as templates for customized routines.

5.5.2 Integration with Deva

Within the Deva system, the SMSs and Objects are combined into a single module called the Spatial Manager (SM), one of which is associated with each instance of an environment: this represents the component of the environment that has been inherited from ThreeDSpace.

The SM combines the features of the MAVERIK SMSs and Objects, and in addition provides dynamic loading functionality (much as described for entities in section 5.4.3). It supplements the MAVERIK transformation matrix that represents an Object's position, rotation and scale with higher-order attributes such as velocity and acceleration.

Each environment may request different MAVERIK SMS functionality. For example, one environment may use a simple linear list of objects, another may use more complex hierarchical bounding boxes, yet another may use a policy customised for the behaviour a specific application. This allows the high-level object-orientated description of an environment to capitalize upon low-level optimized application-specific techniques for managing the spatial relationships between the entities that will exist within it. At the same time, the individual entities may associate themselves with different MAVERIK Objects (and hence optimised rendering routines) that suit their individual needs. This is illustrated in figure 5.5.

The interface to the SM provides functions for:

- Associating a specific SMS algorithm with an environment,
- Associating a specific Object rendering routine with an entity, and
- Creating an instance of an object within a specific SMS.

The SM links whatever representation the MAVERIK SMS and Objects use to represent spatial dimension with the Deva VM's internal representation, so that changes caused by behaviours etc. are reflected by changes in the SMS and vice versa.

Furthermore, the SM module provides routines for replicating and synchronizing its contents using the standard communications layer. These routines are used to generate the scenes rendered by the remote visualisers.

5.6 The client Viewers

The viewer processes provide a 'window' into the virtual environments managed within the world server cluster. As described in chapter 4, the system developed here considers the view through that window to be a subjective representation of the objective environment that exists in the server. The practical implications of this are that the scenes generated by the viewers are representative of though not identical to the scenes held within the world server; thus what is held in each viewer is a 'loosely synchronised' copy of the canonical data structure held by the server.

In practice this means that each viewer process contains its own Spatial Management module as described above (see also figure 5.5). The viewers' modules are, however, slaved to the appropriate SM within the server, and receive instructions to modify their contents from their controlling SM rather than directly from an application.

With the exception of maintaining their status either as ‘master’ or ‘slave’, the various SMs execute identical code, and function independently of one another. Thus the instruction by an application to change an object’s velocity results in the change of velocity being initially sent to the master SM, and then reflected by further instructions to all the slave SMs. The calculation of that object’s absolute position according to its acceleration and velocity is then performed simultaneously by all participating SMs, rather than being sent as repeated ‘absolute position’ updates from master to slave SMs.

The slave SMs therefore perform a kind of ‘dead-reckoning’ on the behaviour of their objects. That is, they base future behaviour on the last set of instructions received from their master SM, and continue to apply such behaviour until informed otherwise.

5.6.1 Linear dead-reckoning

The current implementation is based on a simple form of linear dead-reckoning, where the position and orientation of an entity is calculated per frame on the basis of the last set of updated vectors received from the server. Currently defined vectors control:

Linear velocity according to the environment’s co-ordinate system.

Angular velocity according to the environment’s co-ordinate system.

Linear acceleration according to the environment’s co-ordinate system.

‘Local’ Linear velocity according to the entity’s own co-ordinate system (this may be used to make an entity move ‘forwards’)

Errors in predicted positions are currently corrected by simply ‘jumping’ an entity into its correct position and orientation when an update is received. For the tiny errors that occur when the system is not overloaded, the effect is usually imperceptible, however under simulated heavy loading the jittering of entities caused by errors in dead reckoning becomes a significant perceptual issue.

5.6.2 Spline-based dead-reckoning

Although not currently implemented, it is intended to replace the vector based update method with spline-based updates, allowing arbitrary relationships to be set up between entity attributes. This allows for a much greater degree of expression in more concise terms (i.e. less network traffic), and also improves the ability to dead-reckon the position of an entity (by including more sophisticated information about its history). It is also envisioned that splines may be used to correct for errors in the predictive mechanism less obtrusively.

5.6.3 Controlling the viewer

In representing the user’s presence in a virtual environment, the viewer processes play two roles. This first of these is as consumers of information from the server cluster. That is, the server downloads information about the scene that the viewer can see, and updates that appropriately as applications or behaviours modify that scene. Their second role is as agents representing the user’s interactions with the environment, and in this role they act as producers of messages that modify the content of the environment.

When a viewer process is activated, it creates for itself a ‘camera’ entity that represents its presence in the environment. Whatever the viewer creates as its representations is a fully-fledged, first class entity, and thus experiences the same laws and forces, and has the same affordances as any other entity in the same environment.

The only significant features about this entity is that *by default* the viewer will send any instructions generated by its input devices to it, and will use its position and orientation in space to generate the rendered graphical scene. Thus interacting with whatever navigation features are activated within the viewer will cause that entity to move around within the environment (subject to its laws etc.), and will cause the viewer's display to be updated accordingly.

By virtue of using the standard communications layer, the viewer process itself (not the entity it created to represent itself!) is also a first class entity, and thus may respond to messages sent by applications or entities. In particular, it may be caused at run-time to change the 'focus' entity to any other valid entity; thus the viewpoint and results of navigation may be transferred elsewhere.

The viewer uses standard MAVERIK navigation classes, and attempts if possible to extract higher-order information from these in order to reduce the network traffic between viewer and server. For example, the standard MAVERIK navigators are written such as to cause the viewpoint to be set to an absolute position each frame. The viewer code that surrounds the navigation routines calculates the time since the previous frame and uses this to determine a 'velocity' which is then sent to the server if it has varied significantly since the last update. Thus any configuration of input devices that are available for MAVERIK applications may also be used straightforwardly by the Deva viewers.

5.6.4 Multicasting

When triggered by an update from an external source such as an application, a master SM may send copies (or at least, modified versions of) the update to all the slave SMs that represent viewers in the same environment. The current implementation iterates through a list of participating SMs, and sends an identical update message to each. The is clearly an inefficient technique, especially given that on local-area-networks such as ethernet the underlying hardware will actually send every message to every station which will then selectively ignore or receive their designated data.

Recent networking protocols include support for 'multicasting', that is the 'broadcasting' of a single message to a specific set of recipients. Where hardware allows (such as local ethernet connections) this technique significantly reduces network traffic (in this case sending 1 message for n clients rather than n messages). Where broadcast facilities are not natively supported by hardware (for example, over wide area networks connected by modem or ISDN lines), the simulation of multicast using multiple 'unicast' is no more costly than performing many unicasts in the first instance. Multicast techniques are used by both DIVE [CH93] and MASSIVE2 [Gre97] to reduce network traffic. In particular, it is envisioned that the hierarchical multicast groups developed for the MASSIVE2 system will map well onto the hierarchical environment structures of the Deva model[2].

5.6.5 Migration of the locus of control

The reduction in network traffic that results from the use of higher-order object attributes and dead-reckoning relies on the fact that a plausible environment may be experienced in spite of certain types of deviation from the objective reality. Although this prototype implementation contains no *generally applicable* code for determining whether one kind of event must be synchronized at the expense of another, it does use one technique for dealing with the most important case of this kind.

A general guideline that may be applied is that the more directly linked to a user's own perceptual apparatus an event or object, the more important it is to maintain causal relationships. Thus the user's viewpoint in a world or objects that are being directly manipulated by a user must be made to behave correctly in order to maintain the experience of presence. Conversely, items that are further removed from the user's perception (e.g. are in the distance) may be given less attention by the system.

In this prototype, this situation is dealt with by 'migrating' an entity's 'locus of control' from the server cluster into the viewer to which that entity is most closely bound. That is, for the majority of entities, their objective content exists in the server cluster, with each viewer maintaining a subjective (and possible 'out-of-sync') copy of that data. Where it is important to maintain a tight relationship

[2]It is worth noting that the combination of dead-reckoning with higher-order attributes (e.g. velocity, acceleration and splines) and perceptual filters to communicate between clients and server has reduced the network requirements of the prototype implementation below the threshold where multicast techniques are necessary. However, as more complex environments with larger numbers of users are created, it is considered that multicasting will become essential.

between cause and effect, the system allows the locus of control for an entity to be moved from the server's SM to that of a viewer. This means that requests to update that entity are redirected via the new migrated master SM before being broadcast to other participating SMs. The effect of this is that for one specific viewer, manipulations of migrated items within its environment are performed as local updates, rather than remotely via the server.

It is only the aspects of an entity related to the properties imbued by Holospace that are migrated. That is those properties that are associated with the minimal perceivable attributes are migrated whilst any other attributes, behaviours, laws or forces are maintained by the server. Much of the code associated with the user's perception of an entity is therefore moved to the process that requires access to information the most directly. Since the migration is only partial, however, it is still necessary to inform the server of any changes that have been made to the perceivable components of the entity in case any laws or forces need be applied. However, these updates may be made asynchronously, removing the need to wait for the server to authorise the changes. Once migration has occurred, the server need only intervene in situations where the viewer processes are unable to resolve conflicts between instructions from input devices and restrictions or behaviours imposed by laws or forces.

The technique represents a compromise between the extremes of migrating every item and all it's code into each viewer (which would essentially be a more traditional peer to peer system with the associated problems described in chapter 3) and the alternative of keeping every entity and all its associated code in the server (in which case there is a small overhead associated with remotely updating entities that may be unacceptable for entities closely related to the user). Using this technique, the majority of entities are managed in the parallel server cluster, whilst those that need to be updated rapidly for perceptual consistency are 'invisibly' moved close to where they are needed. The cost associated with this is that accessing migrated entities from other applications/entities is slightly slower, since an extra 'hop' is required to route messages from the server's SM to wherever the entity is temporarily hosted. In practice, however, the objective/subjective distinction hides this extra cost, and users experience smooth response from entities that they are interested in, and are unaware of the delay in updating those that are being manipulated by others.

5.7 Interfacing to the Deva system

There are three programming interfaces to the Deva system. The first of these, designed primarily for specifying new virtual environments, is the high-level Deva language. It is intrinsic to the design of the entire system and supports the description of laws and forces governing the behaviour of entities within a virtual environment. The second interface is the 'application programmer's interface' (API) that forms a link between the Deva virtual machine and existing application code. This provides a means for applications to represent their semantics in the form of a virtual environment, whether or not the applications themselves are couched in those terms (see discussion on application classification in chapter 2). The third and final interface is the low-level rendering and spatial management link to the MAVERIK system. This allows applications that have specific or optimized display management techniques extend the functionality of the Deva server and its rendering clients. The theory and practice of the high-level language have been described previously, and the main principles of the much lower level spatial management/rendering interface have been discussed in the preceding sections (more details on using MAVERIK may be found in [CHH+98]). The following sections present a brief overview of the remaining Application Programmer's Interface.

5.7.1 The Application Programmer's Interface

The Deva API consists of a library of C functions that may be linked with existing applications to allow them to communicate with entities and environments in the Server. Thus applications that are not written specifically as virtual environments may control the behaviour of entities within a VE.

The interface allows an application to behave as though it were a true Deva entity. That is, the application itself may communicate via messages to other entities and environments, and may itself respond to messages/method calls. A typical application will use the interface in 4 stages:

1. The application begins by connecting itself to the server cluster. The low level implications of this are that the standard communications layer is used to connect the application process to all other appropriate processes in the cluster (this is a very similar procedure to the connection of a server or client to the cluster). The application identifies itself by a name, just as any other entity, and declares in which environment it is to be created (by default this is TheVoid, since applications usually do not wish themselves to be rendered in an environment).

2. The application then registers the functions it wishes to use in response to method calls sent to it. These methods are registered in addition to any that have been imbued by its environment and are effectively the application's 'pure' method (for example, the application may wish to respond in a particular way to a 'quit' request).

3. The application either enters its control loop and ensures that `deva_poll()` is called frequently, or registers a signal managing function that will call the `deva_poll()` function in its behalf when messages are received. The former approach is convenient for applications with a 'main-loop', such as interactive programs, and the latter is appropriate where no single point in the application is frequently called.

4. The application disconnects from the server cluster.

A simple example application is given in figure 5.6.

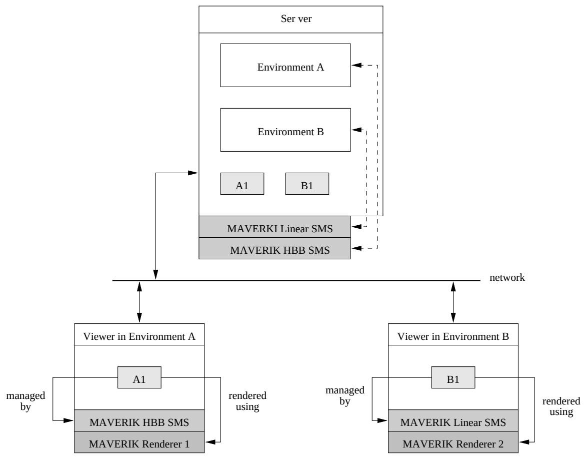

Figure 5.5: Integration of MAVERIK modules with server and clients. The modules are dynamically loaded on demand, and are associated with either Environments (to manage their space using an SMS) or entities in the viewers (in order to render them). Identical SMS code is used in both clients and servers.

```
#include <stdio.h>
#include <deva.h>

int quit= 0;

DEVA_message quit_handler(DEVA_message m, DEVA_name to, DEVA_name from)
{
   printf("Request to quit received from [%s]\n", from);
   quit= 1;
   return NULL;
}

int main()
{
    deva_opt_name="testApplication";
    deva_init();
    deva_registerNamedMessageHandler("quit",quit_handler);
    deva_system("myEnvironment: create(\"myObject\",\"Box\"));
    while (!quit)
    {
        deva_poll();
    }
}
```

Figure 5.6: A simple example application using the Deva API. It connects to the server cluster as an entity called "testApplication", creates a single `Box` in an environment called `myEnvironment` and then waits until it receives a `quit` method from anywhere before exiting. Checks for the functions failing have been omitted for brevity.

Bibliography

[10096] IEEE POSIX 1003.1c. Posix threads. ANSI/IEEE Base document, 1996.

[ACR96] J. A. Mariani A. Colebourne and T. Rodden. Q-PIT: A populated information terrain. In G. G. Grinstein and R. F. Erbacher, editors, *Visual Data Exploration and Analysis III, SPIE Proceedings Series*, volume 2656, pages 12–22, 1996.

[BBC+93] S. Benford, A. Bullock, N. Cook, P. Harvey, R. Ingram, and O.K. Lee. A spatial model of cooperation for virtual worlds. In *Informatique'93: Interface to Real & Virtual Worlds*, pages 445–456. University of Nottingham, March 1993.

[BG90] V. Bruce and R. G. Green. *Visual Perception. Physiology, Psychology and Ecology.* Lawrence Erlbaum Associates, Hove and London (UK), 2 edition, 1990.

[bJM34] Immanual Kant (Translation by J.M.D. Meiklejohn). *Critique of Pure Reason.* Everyman series. Charles E. Tuttle & Co., Rutland, Vermont, USA, 1934.

[Bow98] J. M. Bowers. An ethnography of ambulance control to guide the development of information visualisations for electronic landscapes. In J. A. Mariani, M. Rouncefield, J. O'Brien, and T. Rodden, editors, *Visualisation of Structure and Population within Electronic Landscapes*, chapter 2, pages 35–57. Lancaster University Press, 1998.

[BSG+95] S. D. Benford, D. N. Snowdon, C. M. Greenhalgh, R. J. Ingram, I. Knox, and C. C. Brown. VR-VIBE: A virtual environment for co-operative information retreival. *Computer Graphics Forum*, 14(3):349–360, 1995.

[CC92] M. Chalmers and P. Chitson. Bead: Explorations in information visualisation. In *Proceedings of SIGIR'92*, pages 330–337. ACM Press, June 1992.

[CH93] C. Carlsson and O. Hagsand. DIVE - a multi-user virtual reality system. In *Proceedings of the 1993 IEEE Virtual Reality Annual Internaltion Symposium*, pages 394–400, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[CHH+98] J. M. Cook, T. Howard, R. Hubbold, M. Keates, S. Gibson, A. Murta, S. Pettifer, and A. West. Maverik programmer's guide. Technical report, Advanced Interfaces Group, The University of Manchester, October 1998.

[CHK98] J. Cook, R. Hubbold, and M. Keates. Virtual reality for large-scale industrial applications. *Future Generation Computer Systems*, 14:157–166, 1998.

[CMN83] S.K. Card, T.P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1983.

[Coc92] Geoffrey P. Coco. Veos 2.0 tool builders manual. Available via anonymous ftp from ftp.u.washington.edu, /public/virtual-worlds/veos, 1992.

[Com88] D. Comer. *Internetworking with TCP/IP: Principles, protocols and architectures.* Prentice Hall International Editions, 1988.

[Den91] D. C. Dennett. *Consciousness explained.* Penguin Books, 1991.

[DN66] O. Dahl and K. Nygaard. SIMULA - an ALGOL-based simultion languge. *Communications of the ACM*, 9:671–678, 1966.

[FBR93] Steven Frank, Henry Burkhardt, and James Rothnie. The KSR1: bridging the gap between shared memory and MPPs. In *Proceedings of Compcon93*, pages 285–294, February 1993.

[FSP93a] Martin Friedmann, Thad Starner, and Alex Pentland. Device synchronization using an optimal linear filter. In R.A. Earnshaw, M.A. Gigante, and H. Jones, editors, *Virtual Reality Systems*, chapter 9, pages 119–132. Academic Press, March 1993.

[FSP93b] Martin Friedmann, Thad Starner, and Alex Pentland. Device synchronization using an optimal linear filter. In R.A. Earnshaw, M.A. Gigante, and H. Jones, editors, *Virtual Reality Systems*, chapter 9, pages 119–132. Academic Press, March 1993.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallal Virtual Machine, A User's Guide and Tutorial for networked parallel computing.* MIT Press, 1994.

[Gib66] J. J. Gibson. *The senses considered as perceptual systems.* Houghton-Mifflin, Boston, MA, 1966.

[GPW98] S. Gibson, S. Pettifer, and A. West. Visualising the virtual cityscape. In J. A. Mariani, M. Rouncefield, J. O'Brien, and T. Rodden, editors, *Visualisation of Structure and Population within Electronic Landscapes*, chapter 6, pages 117–138. Lancaster University Press, 1998.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[Gre97] C. M. Greenhalgh. *Large Scale Collaborative Virtual Environments.* PhD thesis, The University of Nottingham, 1997.

[Gri91] Charles Grimsdale. dVS – Distributed Virtual environment System. Division Ltd, Bristol, UK, 1991.

[HC97] C. S. Horstmann and G. Cornell. *Core Java 1.1: Fundamentals.* Prentice Hall Computer Books, 1997.

[HMWH95] Roger Hubbold, Alan Murta, Adrian West, and Toby Howard. Design Issues for Virtual Reality Systems. In *Virtual Environments '95*, pages 224–236. Springer-Verlag, 1995. ISBN 3-211-82737-4.

[Mey92] B. Meyer. *Eiffel the language.* Prentice Hall, 1992.

[Pet95] S. R. Pettifer. Beyond the metaphysics of virtual reality. Master's thesis, Department of Computer Science, The University of Manchester, June 1995.

[Pet96] J. Peterson. The haskell library report. Technical report, Yale University, May 1996.

[PW98] S. R. Pettifer and A. J. West. Understanding the requirements for connectivity in electronic landscapes. In J. M. Bowers, S. Pettifer, and M. Stenius, editors, *Understanding connection, transportation and participation*, chapter 2, pages 7–18. Lancaster University Press, 1998.

[Rod98] T. Rodden. A framework for visualising structure and population. In J. A. Mariani, M. Rouncefield, J. O'Brien, and T. Rodden, editors, *Visualisation of Structure and Population within Electronic Landscapes*, chapter 9, pages 181–200. Lancaster University Press, 1998.

[SHK95] J. Sargeant, S. J. Hooton, and C. C. Kirkham. UFO: Language evolution and consequences of state. In A. P. W. Bohm and J. T. Feo, editors, *Proceedings of High Performance Functional Computing 1995*, pages 48–62, March 1995.

[SM97] G. Smith and J. Mariani. Using subjective views to enhance 3d applications. *ACM Symposium on Virtual Reality Software and Technology*, pages 139–146, 1997.

[Sno96] D. N. Snowdon. *AVIARY: A model for a general purpose virtual environment.* PhD thesis, The University of Manchester, 1996.

[SSPN94] G. Singh, L. Serra, W. Png, and H. Ng. Bricknet: A software toolkit for network-based virtual worlds. *Presence*, 3(1):19–34, 1994.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Wil83] L. Williams. Pyramidalparametrics. *ACM Computer Graphics*, 17(3):1–11, July 1983.

[WRH71] W. Wulf, D. B. Russell, and A. N. Habermann. BLISS - a language for systems programming. *Communications of the ACM*, 14:780–790, 1971.