DEVA3: Architecture for a Large-Scale Distributed Virtual Reality System

Steve Pettifer, Jon Cook, James Marsh and Adrian West
Advanced Interfaces Group
University of Manchester
Oxford Road
Manchester, M13 9PL
UK.
{srp,cookj,marshj,ajw}@cs.man.ac.uk

ABSTRACT
In this paper we present work undertaken by the Advanced Interfaces Group at the University of Manchester on the design and development of a system to support large numbers of geographically distributed users in complex, large-scale virtual environments (VEs).

We shown how the problem of synchronisation in the face of network limitations is being addressed by the Deva system through the exploitation of subjectivity. Further, we present a model for flexibly describing object behaviours in the VEs.

Applications of the system in use are described.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/server; C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications; D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.11 [Software Engineering]: Software Architectures—Languages; J.4 [Social and Behavioral Sciences]: [Sociology]; I.3.8 [Computer Graphics]: Applications

Keywords
Virtual Environments, System architecture, Distribution, Subjectivity, Programming model, Object behavior

1. INTRODUCTION
Virtual Reality (VR) promises a lot; it is easy to envision complex shared Virtual Environments (VEs) in which people at different geographical locations communicate and work together within CAD, Engineering or Social contexts. This vision is tantalising and must surely bring some benefit if it can be realised. Yet, building VR applications of any sophistication is hard work, as evidenced by the lack of compelling examples of industrial strength demonstrations of this technology in serious use.

Until recently technological limitations were a clear brake on progress for VEs (the graphics challenge is perhaps the clearest example). Beyond the technology the key task is that of writing software. The scale and complexity of this task is often underestimated, and it is here where we believe that the major problems lie. We do not at this time have adequate frameworks to simplify the task of implementing virtual worlds. Today, a person wishing to implement a challenging VE application has two broad options, as evidenced in current demonstrations. The most sophisticated VEs are usually bespoke applications constructed from the graphics layer upwards; this is a substantial undertaking. The alternative is to use an off-the-shelf VE builder. These fall into different categories, (for example, VRML [1] builders at the lower end, up to VE systems such as DIVE [2], Ubi-et-Orbi [7] and MASSIVE [8] at the more ambitious extremity). Excellent though such offerings are, they inevitably impose a particular view of how a VE should be structured. Perhaps it is too early in the state of the art to know how best to do this without unduly limiting what can be achieved with them.

The software support challenge to facilitate large-scale VE applications is, we feel, twofold. Firstly to find techniques and algorithms to address specific needs in VEs. Collision detection, parallelism, distribution, synchronisation, navigation and so forth all require work of this kind. Secondly and perhaps rather harder, is to find frameworks that allow all the parts to be put together in ‘flexible yet powerful’ ways. The rather trite nature of such a statement belies the difficulty of quantifying that task.

Finding the ‘right’ framework is particularly difficult in the case of VR, since it brings together a number of complex technical issues, and binds them with real time constraints at the social/perceptual interface. A desirable approach for the necessary flexibility to experiment is to build as little into the system as possible, so the system provides a set of mechanisms and policies and default behaviours that can be unplugged and tailored at each level. Several technologies exist (e.g. [5, 4]) for providing such general purpose brokering/connection between applications, however in our experience and that of others ([7, 10]) they have proven to be too heavyweight and can neither meet the real time constraints of VR nor provide the scalability required to move beyond trivial VEs.

The issue of scale is an important one; simple small-scale VEs that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VRST 2000 Seoul, Korea
Copyright 2000 ACM 1-58113-316-2/00/0010...$5.00
do not challenge today’s hardware can be constructed by any number of means. Building large-scale complex applications raises a number of challenges. For shared VEs these relate, broadly speaking, to the following areas: number of objects; complexity of behaviour required of these objects; complexity of individual rendering techniques; number and geographical distribution of simultaneous users; and number of co-existing and interacting applications.

One technique for addressing the issue of scale arises from the observation that for a shared VE there is a natural distinction between the individual perceptions of the VE and what is actually ‘going on’ within it [15] [13]. That is, between the distribution and simulation of application behaviours, and the task of presenting a coherent perception of the VE to each user. This is similar to the traditional philosophical distinction between subjective perception of an underlying objective reality. The reason that this is useful is that in VR it separates out the challenging graphics/interaction tasks from the semantics of the underlying world simulation. The distinction also legitimises efforts to make a perceptually smooth presentation of the world in the light of fundamental networking limitations. Architecturally we have addressed these two aspects of the VE (perception/reality) separately. First, rendering and spatial management seem to need special treatment that is different from current approaches to graphics toolkits. Described elsewhere [9] the MAVERIK system aims to address these issues. The reason, hinted at earlier, is that no one structure or representation is appropriate for the range of interactive applications envisaged, so no one graphical representation can be selected as the ‘internal’ structure of the VE system. This is as much a performance issue as programming elegance, for key graphics optimisations are highly application specific, and are generally unavailable when the application must export its representation into the VR system.

The more challenging task, namely the simulation of the underlying reality regarded separately from its graphical representation, comprises two core services: 1) An execution environment: a means by which the application and user interactions can be integrated in a single run-time environment. Operating systems history suggests that one should design for a number of applications to either run simultaneously in the same environment, or to allow the user move from one environment to another seamlessly 2) Distribute a means of distributing the actions of the user and the behaviour the application across a wide area network.

Integrating multiple applications may be simple where each application effectively conforms to a particular ‘world view’. This to say for example that a certain event model is used throughout that interaction takes place in one of a set number of ways, and that objects are one of a set number of basic ‘types’ with essentially only superficial differences. However our experience of building non-trivial VEs suggests that this is too inflexible – the one-size-fits-all approach seems too impoverished for real tasks, and the MAVERIK system has proven to be an effective way of avoiding this problem.

Providing a mechanism that allows any disparate set of navigational techniques, rendering routines and so on to be present in a single environment is desirable, but the conflicts, inconsistencies etc. that this would present to the user make this unrealistic if it is not moderated. For example, different navigation metaphors are used for data visualisation, CAD design review, and interactive art work. A framework therefore needs to provide a way of managing these different kinds of techniques, which as well as enabling integration at a technical level provides some means for users to comprehend what is occurring in the environment, and what possibilities it offers as they move from environment to environment, application to application.

We have partitioned the VR architecture task into three parts. The first, graphics and spatial management – the users perception of the environment and its rendering using the MAVERIK system, deal with elsewhere though the integration of that component concern us here too. The second and third parts, namely the execution and distribution facilities, we have argued are the central concerns for managing the underlying world model. In this paper we describe the concrete implementation of these ideas in a
system called Deva.

2. DISTRIBUTION AND COMMUNICATIONS

The programming model employed by the Deva system is one of communicating 'entities' which can represent objects in the Virtual Environment, the properties of the environment itself, or alternatively abstract programming concepts that have no direct representation to the inhabitant of the environment. These entities are coarse-grained programming objects, exporting a number of 'methods' that can be called by other objects, and implementing these internally using optimized imperative code (currently written in C++). In this section we describe the distribution architecture that is employed by Deva that enables transparent and lightweight communication between entities distributed around the system. A detailed description of the makeup of an entity is presented in a later section; for now we concentrate on the techniques used to rapidly route messages between entities around the system.

2.1 The distribution architecture

Deva is logically a client/server architecture, which to a first approximation provides a single definitive locus of control for the Virtual Environment using its server component, with 'mirrors' of the entities being maintained in each client process. Behind the scenes, however, Deva pragmatically manages the delegation of control dynamically to the most appropriate parts of the system, thereby achieving the highest fidelity of perceptual and causal coherence attainable for the application at hand.

A diagram of the distribution architecture is given in Figure 1. The 'server' is in fact a cluster of processors running identical processes called 'server nodes' that together form a single multi-threaded parallel virtual machine capable of processing large numbers of entities. The intention is that the server provides a computing resource for multiple virtual environments, and maintains a far heavier processing load than any one user's client could manage at any one time. A networking layer provides lightweight position independent messaging between entities. Entities are created in and managed by the server node processes, and client processes - such as a visualiser or user application - connect to the server to interact with and obtain state information about the entities.

The server is persistent: it remain alive processing entities regardless of whether or not any clients are connected. Administrative tools exist to simplify the startup, monitoring and shutdown of the parallel server.

2.2 Creating and addressing entities

Each server consists of a (large) fixed number, M, of virtual servers. The M virtual servers are trivially mapped using a lookup table to the N server node processes which comprise the server. This conversion takes place at a low-level within the system and essentially hides the configuration of the server.

An arbitrary virtual server is chosen to create and manage each entity (currently a random virtual server is chosen but this selection process could take into account loading factors).

Each server node is designed to manage multiple entities. Each entity is assigned a unique 'pool ID' - an offset into the list of entities managed by a given server node.

The location of an entity in the server is uniquely defined by the virtual server and pool IDs. This pool ID is not strictly necessary and is provided for efficiency only since each entity has a unique name that can be searched for in the list of entities managed by a given server node.

When an entity is created a hash function is applied to the entity's name to obtain a second virtual server. This virtual server manages the name of that entity, i.e. it definitively knows the virtual server on which the entity is actually located and its pool ID. The name of the entity and the entity itself are managed by separate virtual servers.

The same hash function is used throughout the system allowing any Deva process to obtain the location of a named entity. This data is cached for future use.

Entity name to location lookup, while a lightweight process, is a central and frequent task in a distributed system. A scheme where this load is spread equally across all servers nodes is therefore advantageous.

Figure 2 shows an example distribution of entities and name management across server nodes.

2.3 Entity migration
The main advantage of the addressing mechanism is that it allows entities to dynamically migrate across server nodes to help balance processing load. When an entity moves it only needs to inform its name manager of its new location. Deva processes – which now contains out of date cached data – will receive an error the next time they try to communicate with the entity at its old location. This error is trapped internally and the new location of the entity is obtained from the name manager; the originator of the communication with the entity is oblivious to the migration. The name manager can always be relied upon to know the correct information and its location is trivially obtained.

2.4 Server reconfiguration
It is also possible to migrate the names managed by a given virtual server onto a different server node by updating the virtual server to server node lookup tables in every client and server process.

The migration of both entities and name management allows server nodes to be dynamically added and removed from a running server.

2.5 Networking protocol
Currently standard TCP/IP point to point socket communication is employed although some work has been undertaken investigating multi-cast, since for local area networks at least it promises improvements in performance for our application.

Although not a strict requirement, the communication strategy is based upon the assumption that inter server node communication is fast compared to server/client communication. For example, the server nodes are connected via a dedicated network or protected to some extent from superfluous traffic by a bridge; while clients connect to the servers via a high-traffic shared LAN or modem connection.

Although not a strict requirement, the communication strategy is fast compared to server/client communication. For example, the server nodes are connected via a dedicated network or protected to some extent from superfluous traffic by a bridge; while clients connect to the servers via a high-traffic shared LAN or modem connection.

3. SYNCHRONISING THE BEHAVIOUR OF ENTITIES

The distribution of semantics or ‘behaviour’ in shared VEs is a particularly difficult issue. The ‘ideal’ solution would be to describe the required behaviour in a single location and to have it instantaneously sent to all participants (a pure ‘client/server’ approach). Current network technology is far too limited to make this solution feasible. An alternative is to replicate behaviour locally for each participant (a pure ‘peer-to-peer’ approach). Though the favoured solution in many VE systems this however introduces problems of synchronisation.

The approach taken in Deva is a hybrid solution, part way between client/server and peer-to-peer. The model is one of an objective and subjective reality; the former being located on the server, the latter being represented on each client. Each user interacts with the objective reality via their own subjective view. This introduces the idea that each entity in the virtual environment has two definitions, which may differ significantly in their semantics. Our usage of the term ‘subjective’ is not intended to imply that each user experiences an entirely different VE (this would hardly count as a shared experience) [13]. Rather, we argue that an amount of the user’s experience may be decoupled from the ‘objective’ behaviour of the world without disturbing consequences. Given that absolute synchronisation is in any case impossible, it is our contention that so long as the length of the delay is not too large, and causal events occur in the correct order, it is possible to accept a degree of subjectivity without it affecting the users’ understanding of the semantics of the application. Under these circumstances, given a other frame of reference the users are unlikely to even realise the are not receiving the same view of the world as each other. In general it would therefore be useful to differentiate between what is ‘actually’ happening in a VE and what the users perceive. In this way it is possible to optimise the manner in which information is transferred to minimise causal discrepancies.

In the Deva system this notion is implemented by describing an entity as comprising of a single ‘object’ and a number of ‘subjects’. The former part is the ‘objective non perceived’ aspect of the entity, while the latter are the ‘subjective perceivable’ components (e.g. visible, audible, haptic parts). An alternative view of this distinction is that the ‘object’ represents what an entity does, whilst the ‘subjects’ represent how the entity looks, sounds and feels.

Communication between an object and its subjects can be implemented using whatever high-level ‘vocabulary’ is appropriate. In this way the system minimises the need for strict synchronisation whilst maximising the accuracy of causally important events. For example, the system would output how a Twine [11]. Twines are lightweight parametric curves that are used to smooth out spatially and temporally the discrete updates to the visible aspects (e.g. position and rotation) of appropriate entities by interpolation thus increasing their ‘plausibility’ by eliminating disconcerting jumps. We are also currently investigating an infrastructure for providing ‘quality of service’ based on balancing such smoothing against frequency of updates depending on the perceived importance of various events.

The flexible nature of the communications is also advantageous for object to object communications; for example when a user ‘grabs and moves an entity. Such manipulation with a classic client/server architecture would involve a round trip: the event is sent from the client to the server to be processed and then the effects of the even are then transmitted back to the client to be visualised. With the commonly available network technologies we are targeting for the client/server connections such a round trip would introduce an unacceptable lag. It is perceptually important for the cause and effect of the manipulation to be as tightly coupled as possible.

With the separation of subject and object, changes caused by the manipulation can be immediately perceived by the subject, with a user-definable policy function being used to determine when the object is updated. For example, an arbitrary fraction of the changes (say, one in five of a second) are transmitted back from the subject to the object and thus onwards to other visualisers connected to the server. Alternatively, a change in position is only updated to the object when the subject has moved a certain distance from its previously synchronised location.

The goal is to ensure that the entity behaves ‘correctly/optimally’ on the visualiser performing the manipulation and also that it behaves ‘plausibly/acceptably’ on any other visualiser connected to the cluster (while accepting that latencies and bandwidth issues rule out strict and absolute synchronisation in such a distributed system).

4. DEFINING THE BEHAVIOUR OF ENTITIES

Commonly. VR systems define the behaviour of an entity by attaching pieces of code, often written in Java, Tcl [12] or some other scripting language. With little specific architectural support however, it is often laborious to code and difficult to ensure consistency.
Enforced
~ method: delete
   Deletes the object

Innate
   method: setColour
      Sets the sphere's colour
   method: setRadius
      Sets the sphere's radius
   method: collide
      Causes the ball to bounce
   method: setElasticity
      Sets the ball's "springiness"

Imbued
   method: setXYZ
      Sets the XYZ position
   method: setOrientation
      Sets the orientation
   method: applyGravity
      Applies gravity to the object
   method: collide
      Stops object after a collision

To some extent this situation may be improved by taking an object-oriented (OO) approach; objects with similar characteristics can inherit from common base classes. There are however no viable OO languages with the necessary support for dynamically changing the inheritance graph at runtime when, for example, an entity moves between worlds (such as from one with the concept of gravity to one without).

We identify at least four conceptual sources of behaviour for each individual entity within a VE:

1. **Behaviour innate to an entity.** Entities each have their own particular role within a VE, for example the ability of a stopwatch to record elapsed time.

2. **Behaviour common to a range of entities.** Often many entities share some aspects of their behaviour through being of a similar type. For example each pawn on a chessboard is subject to the same detailed rules governing movement; these rules do not apply in the same way to other pieces which have their own restrictions. However, all the pieces have in common the rules determining whether they may be 'captured'; a notion that is specific to chess pieces but not necessarily to other objects in the same environment. Thus a hierarchy of common groups can be determined.

3. **Behaviour common to all entities in a particular world.** We generally consider gravity to be a phenomenon associated with everything found in the world around us rather than as a property of each individual object. This also applies to social constructs such as 'monetary' value. An earlier version of Deva first introduced the notion of environments imbuing behaviour onto the entities it contains.

4. **Behaviour that is dynamically required at runtime.** If an entity becomes inhabited by a user for example, it will behave differently—being controlled by a navigation device and so forth—to when uninhabited. Similarly an entity that is 'set alight' will suddenly have properties previously unavailable to it (for example, being able to set fire to neighbouring entities or raising the temperature of an environment).

### 4.1 Components

Our solution to the problem of merging the various sources of behaviour which comprise an entity is to use 'components'. In Deva a component is a collection of methods and attributes relating to a single concept that can be attached or detached from an entity at runtime.

The methods and attributes which comprise the component are divided into a single 'object' and multiple 'subjects' as outlined in the previous section. While typically a component will contain both parts, some components are entirely abstract and have no directly perceivable representation in the virtual environment, i.e. they have no 'subject' part.

A researcher trying out new low-level ideas is free to write components directly in C or C++ that interface to the VR kernel at whatever level is appropriate. More general users are free to use the library of existing components to construct entities, without concerning themselves with details of the implementation.
In order to facilitate efficiently moving them between worlds with different behaviours, entities contain two lists of components, one inherited from the environment, and the other containing their own innate behaviours.

When a method is called to act upon an entity, the two lists are searched in strict order. First of all the list of components given to the entity by the environment is scanned for methods marked as being 'enforced'. If one with the correct name is found, then this is called. Enforcing methods allow an environment to ensure all entities contain a particular method that cannot be overridden. Next, an entity's innate components are searched. If the method is still not found then the environment components are searched again for methods marked as 'imbued'. These are methods given to the entity by the environment but which can then be overridden by the entity itself. For example, an environment may enforce the notion of 'solid objects' upon all its entities ('you can't pass through walls'). It may also 'imbue' all entities with a concept of mass with a default value estimated based on its volume, but which the entity is free to over-ride should it prefer.

4.2 Methods and Filters

The strict order in which components get searched for methods leads to another useful concept, that of the filter. As well as methods just returning results, they can be allowed to return a new method call as their result. These types of methods are called filters, and their result continues to propagate along the component lists.

Filters are particularly useful in two cases. The first is to extend functionality already provided by another component, without needing access to the original component’s definition. For example if one wished to display some effect when a new object gets created in a particular environment, a component could be added to the environment entity with a high precedence that defines a create filter that shadows the create method of the ThreeDspace environment. The filter version gets called first and performs a graphical effect, before returning the same message that continues on to ThreeDspace where it is processed as normal.

The second common use of a filter is to help define behaviour that operates across an entire entity, but without components having to have access to other components' internals. One example would be storing an entity's state prior to migrating the entity to another node process or prior to a complete shutdown of the system. Each component defines a snapshot filter that adds any state variables it possesses to the input message and returns a new snapshot message. The final component then returns a restore message containing all the variables that is sent to the new entity. The new entity contains a restore filter in each component that originally stored variables which then unpacks the message.

Figure 3 shows how components are combined to form a logical entity.

5. THE SYSTEM IN USE

A number of applications have been successfully constructed using the Deva system. There follows a brief description of some applications and how they benefit from the features presented in previous sections.

5.1 The Distributed Legible City and The Kahun Senet Game

The Distributed Legible City (DLC) [14] and The Kahun Senet Game [6] are both shared virtual environments aimed at providing a context for studying social interaction in virtual spaces.

They have in common a strong artistic input that places definite requirements on the rendering component of the VR system. In the case of the DLC this originates from a previous single-user multimedia arts installation by Jeffrey Shaw called the Legible City in which a participant cycles round a virtual cityscape in which buildings have been replaced by fragments of three-dimensional text originating in the real-world locale represented by the VE.

The DLC extends this original work to include multiple participants represented by cycling avatars to explore the world. Users are seated on modified exercise bicycles, wear Head Mounted Displays, and can communicate with one another using a boom mounted microphone and headphones (a photograph of the DLC installation can be seen in figure 4).

The rendering requirements of the Senet Game are driven by the need to provide an aesthetically pleasing environment that will engage children’s interest and encourage them to learn to play the Egyptian game of Senet within the virtual world. As part of the Kahun City Project, the boardgame, its surroundings and ‘realistic’ avatars representing the child-players and the teacher were designed by a graphic artist. Figure 5 shows a screen shot of the virtual world. The application required a gameboard with a number of movable pieces and a dice, as well as animated avatars able to walk around the environment and to point accurately at parts of the environment (such as a board containing the rules of the game) and the game pieces they are interacting with. A bespoke text based ‘chat room’ type facility was provided and integrated with the viewer as a ‘component’ to enable communication between participants.

In both environments, the animation of the cycling or game-playing avatars was performed locally by each viewing client, which interpolated between 'keyframes' distributed by the server. The networking requirements of the two environments were marshalled using different techniques. In the DLC, whose purpose is to enable ‘exploration’ or ‘browsing’ of the environment, smoothing routines were used to ensure that updates to the positions of the avatars always appeared as continuous movement within the environment re-
regardless of the network traffic. Since neither the exact position of an avatar, nor the exact moment at which an avatar achieves a particular orientation are significant in the application, an animation routine that favoured 'continuous behaviour' over 'precise updates' was employed. In the Senet game however, the interaction between users via the pieces on the game board required a more precise kind of network utilisation (since a piece appearing to settle on the wrong game-board square could cause significant difficulties to the players). Here interpolation routines that favoured accuracy over smoothness were used.

For both the DLC and the Senet Game, the majority of the processing in the VE is performed in the client viewers, which use the dynamic loading of appropriate components to customise their behaviour and rendering capabilities to the needs of the applications. The server cluster here acts mostly as a transparent distribution mechanism, and performs little independent processing of its own. A more substantial use of the server's distributed computing platform and the component model is provided by the following application.

### 5.2 Q-PIT

The Q-PIT [3] is a Populated Information Terrain capable of taking the contents of a flat database and generating an abstract three-dimensional environment representing the relationships between various attributes of that information. A typical Q-PIT visualisation is shown in figure 6.

The generation of the Q-PIT's layout is a heavyweight computation task involving the generation of a similarity matrix for all the participating objects, followed by a number of algorithmic operations on the matrix to extract 'clusters' of objects that have 'similar' properties. A force-placement technique is then used to position the objects at optimum locations in space, and convex hull algorithms employed to surround sub-groups of similar objects to graphically highlight their relationships. This process has two distinct phases, the first pertaining to finding and grouping 'similar' objects, and the second to the three-dimensional layout of the objects in space. The first algorithmically intense phase yields no meaningful results for several seconds. The force placement algorithm is iterative and returns increasingly good results on each iteration, and the convex hull calculation is comparatively lightweight. It is not desirable to 'stall' the VE whilst time consuming calculations are performed, but at the same time it is also important to give users 'direct' access to the underlying algorithms by manipulating the objects in the virtual space. In our implementation of this PIT the heavyweight processing is performed on the server cluster, with the lighter processing performed on the individual viewers. Since the force placement and hull algorithms are deterministic, updates to the 'similarities' of objects from the central server object give consistent graphical changes to the layout of the space for all users whilst allowing dynamic and un-interrupted interaction with the objects of the Q-PIT. A server consisting of multiple processors is capable of executing the similarity calculations for multiple Q-PIT artifacts, exploiting well the parallel programming environment provided by the server cluster.

### 6. CONCLUSIONS

Although the Deva system is still in development, its flexible approach to dealing with the issue of synchronisation has already been successfully exploited in a number of public demonstrations.

It is the intention of the authors to release Deva under the GNU Public License (GPL) when the system reaches a more mature state.

### 7. ACKNOWLEDGMENTS

The authors wish to thank our colleagues in the Advanced Interfaces Group and those at ZKM, Lancaster and Manchester Metropolitan Universities.

This work is funded by EPSRC grant number GR/M66448.
8. REFERENCES


