In *Proc ACM UIST '96 (Symp. on User Interface Software and Technology), Seattle, WA, November 6–8, 1996, pp. 83–94.*

Language-Level Support for Exploratory Programming of Distributed Virtual Environments

Blair MacIntyre and Steven Feiner
Department of Computer Science, Columbia University, New York, NY, 10027
{bm,feiner}@cs.columbia.edu
http://www.cs.columbia.edu/~{bm,feiner}

Abstract

We describe COTERIE, a toolkit that provides language-level support for building distributed virtual environments. COTERIE is based on the distributed data-object paradigm for distributed shared memory. Any data object in COTERIE can be declared to be a Shared Object that is replicated fully in any process that is interested in it. These Shared Objects support asynchronous data propagation with atomic serializable updates, and asynchronous notification of updates. COTERIE is built in Modula-3 and uses existing Modula-3 packages that support an integrated interpreted language, multithreading, and 3D animation. Unlike other VE toolkits, COTERIE is based on a set of general-purpose parallel and distributed language concepts designed with the needs of virtual environments in mind. We summarize the requirements that we identified for COTERIE, describe its implementation, compare it with other toolkits, and provide examples that show COTERIE's advantages.

Keywords: distributed virtual environments, distributed shared memory, shared-data object model, virtual reality.

1 Introduction

Over the past several years, our group has built a number of single-user, distributed, virtual environment (VE) prototypes [e.g., 12, 13, 15]. These were constructed using a multi-process client-server architecture, allowing us to easily incorporate several monolithic applications that we had developed previously. However, the next steps in extending most of these projects proved to be much more difficult.

We wanted to support multiple users interacting in shared environments using many different kinds of devices, including displays that are head-worn, hand-held, desk-top, and wall-mounted. Attempting to extend our single-user prototypes in these originally unforeseen ways was becoming increasingly infeasible with each modification. Adding new processes often involved the need to share previously unshared data, effectively turning clients into servers. This resulted in an unmanageable welter of client-server relationships, with each of a dozen or more processes needing to create and maintain explicit connections to each other and to handle inevitable crashes.

We spent a sufficiently large portion of our time reengineering client-server code that it became clear to us that our implementation of the client-server model was unsuitable for exploratory programming of distributed research prototypes. The heart of the problem, as we saw it, was a lack of support for data sharing that was both efficient and easy for programmers to use in the face of frequent and unanticipated changes. Our solution was to build a toolkit based on a set of general-purpose language extensions. These extensions provide high-level mechanisms for distributed VE programming, coupled with an assortment of building blocks common to typical VE systems.

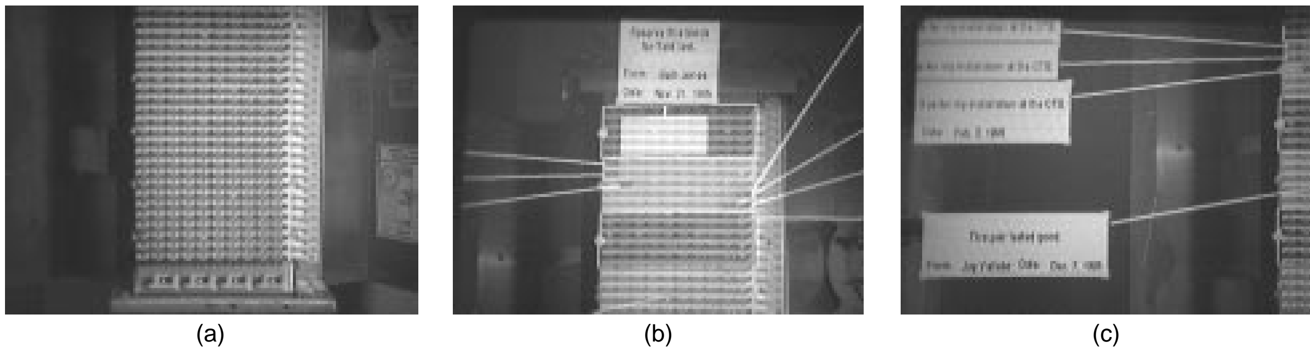
The infrastructure that we developed, COTERIE (Columbia Object-oriented Testbed for Exploratory Research in Interactive Environments), is the topic of this paper. In Section 2, we present our design requirements for COTERIE. Then, we survey related work in Section 3 and describe how well it addresses these requirements. In Section 4, we introduce COTERIE's design and implementation, showing how it was influenced by and builds on the Modula-3 environment in which it is implemented. The key component of the system, the Shared Object package, is discussed in Section 5. In Section 6, we provide examples of high-level support for VEs that we are developing in COTERIE. This is followed by a discussion of our conclusions and future work in Section 7.

2 Design Requirements

The requirements we set for COTERIE reflect our desire to experiment with distributed, multi-user VEs that combine a variety of paradigms, including immersive, "fishtank" [32], see-through, and handheld worlds. For example, Figure 1 shows a prototype COTERIE application that uses a see-through display to overlay a combination of 2D and 3D information on a telephone "crossbox."

We identified a set of core requirements to make it possible to create robust prototypes quickly and easily to allow us to explore different design alternatives. Our requirements fall into three main categories, related to distributed systems, rapid prototyping, and practical application needs.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
UIST 96 Seattle WA USA
© 1996 ACM -89791-798 7/96/11 ...$3.50



(a) (b) (c)

Figure 1: A prototype augmented-reality application created using COTERIE. Images are photographed through an optical see-through, head-worn display. (a) The bottom of a phone company "crossbox" that connects customer phone lines to company wiring. (b) The top of the crossbox with a graphical overlay designed to be presented to the field serviceperson on the head-worn display. The overlay highlights major blocks of the crossbox and a number of user-defined groups of connection posts. It also contains 2D information windows connected to the post groups by stretchable leader lines that allow selected windows to be pulled into and out of view. (c) The view when the user looks down and to the left from (b).

2.1 Distributed Systems Requirements

The key to building a flexible toolkit for prototyping VE applications is efficient and easy-to-use support for data sharing. Therefore, it was central to our design that our infrastructure exhibit satisfactory distributed system characteristics [11], especially network data transparency, scalability, openness and fault tolerance. Furthermore, while client-server data sharing is common, VE systems require that certain key pieces of data be replicated. The replicated data should be updated asynchronously for efficiency, and asynchronous notification of these updates should be provided for ease of programming.

Network data transparency. To build a distributed system, some data-sharing mechanism is needed. Our experience has demonstrated that creating a distributed system that provides facilities for distributing only "virtual environment data," such as tracker readings or graphical objects, is far too restrictive. By treating some kinds of data differently than others, we have occasionally found ourselves in the situation where one piece of data we needed, such as a tracker record, could be easily distributed to a new piece of the system, but another piece of data, such as the layout of a user's information space, could not. Therefore, we require that data types seen by programmers have a high degree of *network transparency*: the programmer should need to be aware that a data value is not local to their machine only when absolutely necessary, and should be able to use remote and local data objects interchangeably whenever possible.

Other distributed system characteristics. While an initial implementation need not be optimized for high performance, many measures of a distributed system are a function of design rather than implementation. It is vital, therefore, that our system should scale well as more users or processors per user are added. It should also be open so that it can be extended in new and interesting directions. Basic fault tolerance is an absolute requirement: as the number of machines and processes per machine increases, so does the likelihood that one of them will crash, especially during development. At the very least, we wanted to be able to easily construct applications that can recover from a single failure.

Data replication. Many of the objects in a VE system must be replicated, rather than merely shared, because the programs using them cannot afford to pay the price of remote access. A good example is the description of a graphical scene. The programs that update the displays must redraw their scenes as often as possible. Programs that do collision detection must likewise access their scenes on a continual basis and are often themselves distributed over multiple machines. However, for maximum flexibility, we wanted to be able to replicate *any* user-defined data, not just typical VE data such as tracker reports or scene objects.

Fast, asynchronous data propagation. Remote procedure or method calls are unsatisfactory for propagating rapidly changing information because they are synchronous, and are therefore too slow when used with a large number of clients. We wanted a method of asynchronous data propagation that would scale well as the number of distributed processes increased.

Asynchronous update notification. When many threads distributed over many processes share data, it is unacceptable for them to have to poll the data to check for changes. Instead, there must be some facility for interested threads to be notified of changes to relevant data items. For example, the thread that renders a graphical scene should be automatically notified of changes to the data structure.

2.2 Rapid Prototyping Requirements

Given a well-designed distributed substrate, two major components are needed to enable fast prototyping of distributed applications: an interpreted language, and a programming model that encourages building easily distributable applications.

Embedded interpreted language. To support rapid prototyping well, a system should have an embedded general-pur-

pose interpreted language, enabling entire applications to be developed without writing compiled code, as exemplified by several existing toolkits [5, 25, 29]. The interpreted and compiled components of the system should be tightly integrated so that interpreted code can be easily rewritten in the compiled language when efficiency is required. This implies that the programming model in both languages should be similar, and all data structures should be equally accessible from both languages. We also believe that both languages should be strongly typed, either statically or dynamically, to assist the programmers as much as possible in creating reliable and robust programs.

Object-oriented, multithreaded environment. Conceptually, VE systems are composed of many independent entities that perform tasks such as monitoring trackers, rendering to displays, and controlling the objects that populate the environment. This maps well to a programming model that has many threads of control communicating via shared objects. When combined with transparently distributable objects, this model is equally suited for programming within one process or between many, as an individual thread does not need to be aware of the location of any other thread. However, using heavy-weight processes for all threads of control is unacceptably inefficient. Furthermore, creating processes that are inherently multithreaded without programming language or operating system thread support is error prone and requires considerable work to ensure all conceptual threads are serviced fairly. Therefore, we believe that thread support should be integrated into the interpreted and compiled programming languages so threads may be used cleanly and uniformly on all operating systems and architectures.

2.3 Practical Application Requirements

We wanted to develop our applications in as platform-independent a fashion as possible, while simultaneously supporting many users and devices across a heterogeneous set of machines, ranging from UNIX workstations to portable computers running Microsoft Windows 95.

High-level, platform-independent, extensible, 3D graphics and GUI packages. We wanted to support a wide variety of hardware and operating systems without having to use a different graphics or GUI package on each. Furthermore, we wanted to be able to cleanly integrate new kinds of graphical objects, such as the workstation windows of [12].

Support for multiple users, displays, and devices. In addition to supporting multiple simultaneous users, we wanted to make it possible for any individual user to use multiple displays and interaction devices in a variety of combinations: opaque and see-through head-mounted displays; palmtop, tablet, desktop, and wall-mounted displays; speech input and output; spatialized sound; 2D and 3D mice, styli, and other hand-held controllers. Therefore, it would have to be easy to add new devices and coordinate their interactions.

3 Related Work

A large number of VE toolkits have been developed. We discuss only those that are intended to support distributed environments. In addition, we discuss some of the work that has been done using shared data for distributed groupware.

3.1 Distributed VE Systems

MR implements a simple shared virtual memory model [28]. Raw memory locations can be marked as shared and local changes explicitly “flushed” to the other copies, which must then explicitly receive the changes. MR has no facilities for handling heterogeneous architectures and provides a single, fully replicated VE, in which each process has a complete copy of the same world. DIVE [8] is built on top of the ISIS [3] fault-tolerant distributed system, and is similar to MR in supporting only fully replicated VEs. VEOS [5] is an extensible environment for prototyping distributed VE applications. MR, VEOS, and DIVE all use point-to-point communication, with all processes directly connected to all others. This prevents these systems from scaling beyond a relatively small number of distributed processes.

SIMNET [6] is perhaps the best known large-scale distributed VE system. It uses a well-defined communication protocol that is also used by NPSNet [34] and VERN [5]. SIMNET was designed to support a single, large-scale, shared, military VE. Broadcasting is used to send messages between nodes. While this cuts down on network traffic, all processes must handle all messages, preventing SIMNET from scaling beyond a few hundred users. NPSNet has recently been extended to accommodate a significantly larger number of simultaneous users (thousands instead of hundreds) by spatially partitioning its world to reduce message traffic [23].

VR-DECK [10] allows multiple users on a set of homogeneous workstations to share a single simulation and cannot be easily extended to support heterogeneous workstations. Message traffic is reduced by sending events only to machines known to be interested in them.

WAVES [20] uses message managers to mediate communication between processes. Each message manager controls a group of clients. All messages are distributed by the message managers to interested clients. WAVES supports the ability to filter messages to a given client, reducing the type and frequency of updates sent. However, it supports only coarse parallelism, with each process performing one well-defined function. The single shared VE comprises a set of objects that encapsulate the behavior and state of the entities in the world. Each object is owned and updated by only one client, but can move freely between clients.

RING [16] and BrickNet [29] both use a communication mechanism similar to that of WAVES, with centralized servers each controlling a set of clients, and communication routed through the servers. All message traffic goes through the servers, with no provision made for direct client-client propagation for time-critical data. RING is geared toward realistic simulations and uses physical visibility to limit

message traffic. Its VE is a set of shared entities, each with a geometric description and a behavior. Each entity is owned by one client, and only that client may update it. RING can support a large number of simultaneous users. BrickNet is geared toward creating multi-user distributed VEs in which each client has its own world composed of a combination of local and shared objects. Like WAVES, its objects have behaviors as well as state and can move between clients.

dVS [17] is a commercial distributed VE system for single-user applications. Its components and message formats are fixed and not extensible, making it unsuitable for non-immersive VEs.

Spline [1] is a recent system that seems similar in intent to our system, and shares many features, but is more focused toward efficient creation of exclusively immersive VEs. It supports a single virtual world and achieves scalability by spatially partitioning this world to reduce message traffic, starting with a scheme similar to [23] and extending it by partitioning the object space based on this spatial partition.

None of the VE systems come sufficiently close to supporting enough of the features we need to justify attempting to extend them to support the rest. With the exception of DIVE, none provide true preemptive threads, but use only heavyweight UNIX processes. With the exception of BrickNet, none support more than a single shared VE. In addition, these systems are geared toward VEs in which each user has only a single (stereo) display, and interacts with an entirely virtual world composed of 3D objects. In contrast, consider our hybrid user-interface window manager [14], a prototype of the kind of application that we would like to support. It combines a flat-panel display with a see-through head-worn display to create a workspace within which one display's image is embedded in the other's. This would be difficult to implement with a conventional VE system.

3.2 Distributed Groupware

A number of groupware systems have been built using shared object techniques. Colab [30] uses a fully replicated database in which changes are broadcast to all sites without synchronization. Colab relies on social and application solutions to avoid, or recover from, inconsistencies. For example, if inconsistencies arise when multiple people are working on the same area of a document, they will quickly become obvious because of the nature of these applications. The users can then decide how best to deal with them.

GroupKit [27] applications run the same program at all sites and communicate by using *multicast remote procedure calls* to execute procedures at all sites. Data is shared via shared data directories called *environments*. While supporting notification of the addition, deletion or modification of items in an environment, there is no support for concurrency control. As with Colab, social solutions are relied upon to solve this problem.

Object World [31] implements shared objects in LISP, and defines shared operations by allowing programmers to define broadcast methods. These methods are executed at any site that has a copy of *any* of the object parameters, with all additional parameters automatically copied to that site. Object World does not provide any consistency guarantees, but accomplishes consistency detection by requiring that all broadcast methods operate on the same version of their object parameters at all sites. Correction of inconsistencies is performed at an application level, possibly with the assistance of the user.

DistView [26] allows window and application objects to be replicated. When an object is replicated, it is wrapped in a proxy that implements the replication semantics, such as sending method invocation messages to remote copies for execution. There is no distinction between read and write methods, and consistency is guaranteed by requiring locks to be acquired for all object accesses. While DistView has a fairly intelligent scheme to minimize the cost of acquiring global locks, the system would not scale well and would not perform well in the face of continuous access from multiple sites.

The techniques for object sharing implemented in the newer groupware toolkits share some of our goals, particularly automatic replication of data to ease construction of distributed applications. However, none have integrated the distribution of data into the object model of their respective programming languages as tightly as we desire. As a result, they do not provide sufficiently strong consistency guarantees. In groupware applications, inconsistencies tend to arise from multiple users attempting to perform conflicting actions: the results are usually obvious to the users and can be corrected using social protocols. This is not an acceptable solution for VE applications. Finally, none of these object systems provide any support for asynchronous update notification, nor are they designed to support the kind of large scale distribution we have in mind.

4 The Modula-3 Environment

COTERIE is written in the Modula-3 programming language [18]. We decided to use Modula-3 because of the language itself and the availability of a set of packages that provide a solid foundation for our infrastructure. The key to COTERIE is the Shared Object package that we built and which we discuss in Section 5. It provides language-level support for data replication and fast update propagation.

Modula-3 is a descendent of Pascal that corrects many of its deficiencies. In particular, Modula-3 retains strong type safety, while adding facilities for exception handling, concurrency, object-oriented programming, and automatic garbage collection[1]. One of its most important features for our work is that it gives us uniform access to these facilities

1. The Modula-3 compiler is written and distributed by DEC's Systems Research Center. A commercially supported version of the SRC compiler is available from Critical Mass, Inc. as part of the Reactor programming environment. The SRC compiler, and thus our system, runs on all the operating systems we use: Solaris, IRIX, HP-UX, Linux, and Windows NT and 95.

across all architectures. We depend on three key packages: Network Objects, Obliq, and Obliq-3D.

The Network Object package [4] supports a client-server model of distributed data sharing through remote method calls that are virtually transparent to the programmer. These include distributed garbage collection, exception propagation back to the calling site, and automatic marshalling and unmarshalling of method arguments and return values of virtually any data type. We enhanced this package to provide automatic data conversion between heterogeneous machines. Obliq [7] is a lexically-scoped untyped language for distributed object-oriented computation that is tightly integrated with Modula-3. Like Modula-3, it supports multiple threads within a single process. Obliq's distributed computation mechanism is based on Network Objects, allowing transparent support for multiple processes on heterogeneous machines. Objects are local to a site, while computation can roam over the network.

Obliq-3D [24] is a high-level 3D animation system that consists of two parts: a Modula-3 library that provides a set of graphical objects and animation primitives, and the same primitives embedded in Obliq. Obliq-3D programs can be written in any combination of Modula-3 and Obliq, because all data structures are simultaneously available from both languages. Obliq-3D's structure and interface also make it relatively easy to extend.

Together, the language and packages provide us with closely matched compiled and interpreted languages that support object-oriented, multi-threaded programming. They also provide a clean basis for reliable distributed programming via transparent remote method calls, and a high-level, extensible 3D graphics system with animation support. What we needed to do was add support for replicated objects with asynchronous update propagation and notification, build a library of objects to support the various VE devices we use, and build an application framework that would allow us to create prototype applications with the distributed system characteristics we desired.

5 The Shared Object Package

To provide COTERIE's replicated object support, we developed a Shared Object package that, when combined with the Network Object package, provides a distributed language infrastructure geared toward the needs of VEs. The remainder of COTERIE is implemented using this package.

The Shared Object Package consists of about 15,000 lines of Modula-3 code and supports general purpose, asynchronously updated, replicated objects. It is based on two concepts: Distributed Shared Memory via the Shared Data-Object Model, and Callback Objects.

5.1 Distributed Shared Memory

Distributed Shared Memory (DSM)[22] allows a network of computers to be programmed much like a multiprocessor, since the programmer is presented with the familiar paradigm of a common shared memory. DSM mechanisms use message-passing protocols between machines to implement some model of shared memory access that is used by the programmer. The Shared Data-Object Model of DSM is particularly well suited to our needs since it is a high-level approach that can be implemented efficiently at the application layer. In this model, shared data is encapsulated in user-defined objects and can only be accessed through the objects' method calls. The DSM address space is partitioned implicitly by the application programmer, with an object being the smallest unit of sharing. The shared data is replicated in each process that is interested in the object. Each process can call any method of an object it shares, just as it can with a non-shared object. The model follows two principles:

1. All operations on an instance of an object are *atomic* and *serializable*. All operations are performed in the same order on all copies of the object. If two methods are invoked simultaneously, the order of invocation is nondeterministic.
2. Property 1 applies to operations on single objects. Making sequences of operations atomic is up to the programmer.

The advantages of this model over techniques that expose the shared memory at a lower layer are discussed in [21]. For our purposes, the programming aspects are especially important:

- The model directly supports Modula-3 type safety, making accidental incorrect usage of shared memory unlikely. This is helpful because debugging parallel programs is difficult.
- The implicit atomicity of method calls makes the model easy to understand and program because no explicit locks are required.
- A high-level, object-oriented approach to DSM means the programmer does not need to know the details of the implementation in order to use the objects efficiently.

5.2 Callback Objects

Callback Objects allow the programmer to receive notification of changes to a Shared Object in an object-oriented fashion. For any Shared Object, an associated Callback Object exists with a similar set of methods. An instance *CO* of a Callback Object is associated with an instance *SO* of a Shared Object by passing *SO* to the constructor of *CO*. When a method of *SO* is invoked and changes the state of *SO*, the corresponding method of *CO* is called. Since the internal representation of a Shared Object is hidden, the details of the change are indicated to *CO* by passing the arguments of the original method call on *SO* to the corresponding method of *CO*. A programmer simply overrides the methods corresponding to the changes for which notification is desired with methods that react appropriately to those changes. Callback Objects remove the need for object polling and enable a "data-driven" flow of control.

A Callback Object contains two callback methods for each update method in the corresponding Shared Object. These callback methods can be overridden to receive notification before or after an update to the Shared Object. An additional pair of "catch-all" callback methods can be overridden to receive notification before or after an update of any changes not handled by overriding the specific update callback methods.

5.3 Implementation

To create a Shared Object type *OBJ*, the programmer creates a Modula-3 object with no publicly accessible data fields that inherits from the *SharedObj* type. Modula-3 source code implementing the shared data-object semantics is generated automatically, and includes:

- a subtype of *OBJ* that overrides the methods of *OBJ* to implement the shared object semantics, and
- a Callback Object for *OBJ*.

5.3.1 Relationship to Network Objects

Semantically, Shared Objects are fully replicated in all processes that are interested in them. This contrasts with Modula-3 Network Objects, in which one copy of an object is maintained in the process in which the object was created, and a proxy that performs remote method calls is maintained in all other interested processes. All the information used to perform remote method calls is encapsulated in the Network Object's proxy object.

Network and Shared Objects share a number of properties that facilitate distributed programming. As with the subtypes used to implement the Shared Object semantics, the Network Object proxies are subtypes of the original objects, allowing them to be used exactly like the original objects. Automatically-generated code takes care of marshalling and unmarshalling arbitrarily complex method arguments and return values, such as large recursive data structures, between heterogeneous machines across the network. Both runtime systems automatically establish and monitor connections for liveness (to a reasonable degree), maintain threads to service incoming remote method calls and performs such tasks as distributed garbage collection. The end result is that, from the programmer's point of view, they are virtually indistinguishable from local objects. Shared Objects are passed between processes using Network Objects. Our implementation ensures that there is one and only one copy of any Shared Object in any process.

5.3.2 Replication Approach

We based our implementation of the Shared-Data Object Model on the fully replicated approach used in Orca [2]. A full replication scheme, where a single object is either fully replicated in a process or not, is significantly simpler than a partial replication scheme, in which an object may or may not be replicated in a process that accesses it, and satisfies our primary rationale for replication: fast read-access to shared data. Furthermore, when used in conjunction with the Network Object package, partial replication can be mimicked, with a slight loss of data transparency, using a combination of the two types of objects: a Network Object can be "wrapped around" a Shared Object and used to access the Shared Object remotely without replicating it in the local process. When a Shared Object is embedded in the Obliq language, this dual behavior comes for free since all Obliq objects (aside from basic types such as integers or booleans) are automatically treated as Network Objects. Thus, the added complexity of partially replicated objects gives us very little benefit.

To maintain replication consistency we chose an update scheme (where updates are applied to all copies) instead of an object invalidation scheme (where changes to one copy of an object invalidate all other copies, requiring a new copy be obtained). This choice was primarily motivated by our needs for timely asynchronous data propagation and asynchronous update notification. First, invalidation requires a extra refresh step, which would increase the lag for distribution of time-sensitive data such as tracker changes. Second, the update messages used to implement the update scheme contain exactly the information needed to provide asynchronous update notification. The final factor in choosing an update scheme is that, in our applications, large objects such as graphical models will be updated frequently, but the updates will affect only a small part of the object, so refetching the object will be wasteful. It is not surprising that we know of no existing distributed VE systems that uses an invalidation-based replication scheme.

5.3.3 Communication architecture

To provide the Shared-Data Object property of serializable updates, all updates to a given object result in an event being sent to one distinguished (heavy-weight) process called the *sequencer*. Each process is statically associated with a particular sequencer at run-time and all objects created in that process have their updates sequenced by that sequencer. A set of processes associated with the same sequencer is called a *cluster*. The sequencer for an object simply assigns a sequence number to each update event it receives and forwards them to all processes with a copy of the Shared Object. Updates are applied only after they are received from the sequencer. Any process with the Shared Object runtime system compiled into it may serve as a sequencer, in addition to its other tasks.

The sequencer currently sends direct updates only to processes in its cluster. If a process in another cluster has a copy of this Shared Object, the update is sent to its sequencer, which will then forward the update to the processes in its cluster that have a copy of the object. Thus, all update messages between clusters are currently sent through the sequencers.

Consider what happens when a thread initiates an update by calling a method that changes the object state. The thread is blocked until the update event is received back from the sequencer, at which time it applies the update to the object and returns. Figure 2 shows the control and data flow of a

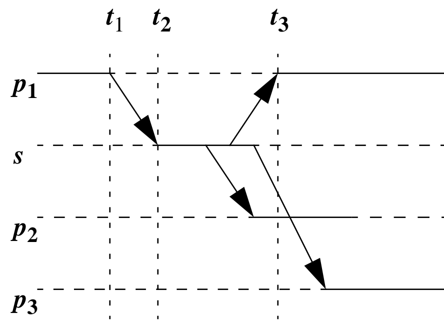


Figure 2: Control and data flow for a typical Shared Object update method call. Process p_1 calls an update method at time t_1. It blocks after a message is sent to the sequencer s. The message arrives at time t_2 and is sent to all interested clients (p_1, p_2 and p_3). p_1 receives the message back at time t_3, is unblocked, executes the original method call and continues.

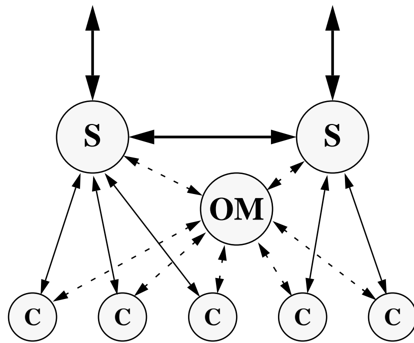


Figure 3: The partitioning of clients (C) among sequencers (S) and object managers (OM). The partitioning of sequencers can be based on efficiency considerations, whereas the partitioning of object managers can be based on the semantic grouping of processes, which are generally not the same.

typical update method call. A similar connection topology is used by many VR systems (WAVES, BrickNet, Ring) to reduce network traffic, but none have isolated it in this fashion. For example, in BrickNet and Ring, the low-level functionality of the sequencer is combined with other high-level functions such as object lookup and management. In WAVES, these are assigned to separate modules, but these modules exist in a one-to-one relationship with each other.

It is undesirable to overlap high- and low-level functionality like this. For example, all these systems have a one-to-one relationship between the number of sequencers and object management servers. In contrast, we can partition object management based on conceptual process groupings, but partition sequencer duties based on physical network characteristics. These two partitions are not always identical, as shown in Figure 3. We are experimenting with different approaches to object management, and discuss one possible approach in Section 6.4.

To allow for efficient read access, we distinguish between operations that do not change the object data (*read* operations) and those that do (*update* operations). Read operations access the local copy directly. Update operations are handled as described above. Access to an object is controlled by locks, so that all update operations are executed atomically and multiple reads may be performed simultaneously.

We modified the Orca semantics in minor ways to suit our needs and the Modula-3 environment. For example, we added the ability to globally lock an object, so multiple methods could be accessed atomically.

5.3.4 Restrictions

As with Network Objects, there are restrictions on what Modula-3 types are valid for use as arguments to method calls. For Shared Objects, the restrictions arise from the need to package arguments to update methods into the update events. Therefore, no data value can be used that is specifically associated with one process (e.g., a thread or a condition variable) or that has state that cannot be accessed repeatedly with consistent results (e.g., a file reader or writer).

6 COTERIE Examples

We are using COTERIE to develop a library of standard objects that form a framework for building VE applications, along with a number of prototype applications. Although we do not discuss the specifics of our actual library here, the simple examples in this section are designed to emphasize the ease of creating such a framework.

6.1 Shared Tracker Object

Figure 4 shows the code for *TrackerPosition*, a simplified position tracker. It is based on the first Shared Object we developed, which was a general-purpose shared tracker object. Here, we have simplified the type definition for *Data* and have reduced the number of methods for clarity. The *init* method is used to initialize a new object, the *set* method changes the value of the shared tracker object (a 3D position), and the *get* method returns the current value. Since Modula-3 supports garbage collection, the *get* returns to each caller a newly allocated copy of the data, which can be freely modified without worrying about affecting other threads.

In Modula-3, when a type in an interface is used, it must be qualified by the interface name. Thus, the type *Data* in Figure 4(a) is used elsewhere as *TrackerPosition.Data*. The notation “*T <: S*” is read as “T is a subtype of S.” The programmer REVEALs the details of *S :< Public* in Figure 4(b). The Shared Object semantics are implemented in the generated code (not shown) by REVEALing the implementation of the *T <: S* relationship in an analogous fashion. By convention, the primary type for an interface is *T*, so when programmers use *TrackerPosition.T*, they will use the Shared Object type.

Figure 5 shows part of the generated code associated with the tracker object: the interface to the Callback Object. As

```
INTERFACE TrackerPosition;
IMPORT SharedObj, Thread, Point3;

TYPE
   Data = REF Point3.T;
   T <: S;
   S <: Public;
   Public = SharedObj.T OBJECT METHODS
      set (READONLY val: Data);
      get (): Data;
      <* SHARED UPDATE METHODS set *>
   END;
END TrackerPosition.
```

(a) The *TrackerPosition* interface

```
MODULE TrackerPosition;
REVEAL
   S = Public OBJECT
      data: Data := NIL;
   OVERRIDES
      init := Init;   (* Defined below *)
      set := Set;     (* Defined below *)
      get := Get;     (* Defined below *)
   END;

PROCEDURE Init(self: S) : SharedObj.T =
   BEGIN
      self := SharedObj.T.init(self);
      self.data := NEW(Data);
      RETURN self;
   END Init;

PROCEDURE Set(self: S; READONLY val: Data) =
   BEGIN
      self.data^ := val^;
   END Set;

PROCEDURE Get(self: S): Data =
   VAR ret := NEW(Data);
   BEGIN
      ret^ := self.data^;
      RETURN ret;
   END Get;

BEGIN
END TrackerPosition.
```

(b) The *TrackerPosition* implementation

Figure 4: Modula-3 code for a *TrackerPosition* Shared Object. There are two parts to the module: (a) the external interface and (b) the internal implementation. Much of Modula-3's syntax is borrowed from Pascal. For example, pointers are defined using the REF keyword and dereferenced with ^.

```
INTERFACE TrackerPositionCB;
IMPORT TrackerPosition, SharedObj;
FROM TrackerPosition IMPORT Data;

TYPE
   T <: Public;
   Public = SharedObj.Callback OBJECT METHODS
      init(obj: TrackerPosition.T): T;
      pre_set (READONLY obj: TrackerPosition.T;
         READONLY val: Data): BOOLEAN;
      pre_anyChange(READONLY obj:
                     TrackerPosition.T);
      post_set (READONLY obj: TrackerPosition.T;
         READONLY val: Data): BOOLEAN;
      post_anyChange(READONLY obj:
                     TrackerPosition.T);
   END;

END TrackerPositionCB.
```

Figure 5: The generated tracker Callback Object interface

```
TYPE LowPassTracker = TrackerPositionCB.T OBJECT
      lpObj: TrackerPosition.T;
      distance: REAL;
      oldVal: Point3.T;
   OVERRIDES
      post_set := LPSet;(* Defined below *)
   END;

PROCEDURE LPSet(self: T;
               READONLY obj: TrackerPosition.T;
               READONLY val: Data): BOOLEAN =
   BEGIN
      IF Point3.Distance(self.oldVal, val^) >=
               self.distance THEN
         self.lpObj.set(val);
         self.oldVal := val^;
      END;
      RETURN TRUE;
   END LPSet;
...
   (* assume tobj is the normal tracker object *)
   lp := NEW(TrackerPosition.T).init();
   lpCB :=
      NEW( LowPassTracker,
           lpObj := lp, distance := 2.0,
           oldVal := lp.get()^).init(tobj);
...
```

Figure 6: A simple use of Callback Objects to create a variation of a tracker that is updated only when the tracker has been moved more than a specified distance since the last report.

described in Section 5.2, *pre_set* and *post_set* are the callback methods for *set*, and *pre_anyChange* and *post_anyChange* are the catch-all callback methods. Next, we show how this Callback Object can be used.

6.2 Filtered Tracker Object

More advanced tracker objects can be implemented using the object in Figure 4. For example, suppose a client wants to receive a tracker report only when the tracker has moved more than a specified distance since the last report. A second tracker object, *lp* (for "lowpass"), can be created in the process handing the tracker. The *lp* object is updated by using a simple Callback Object, *LowPassTracker*, whose *post_set* method is overridden by *LPSet*, as shown in Figure 6. Notice that this approach allows great flexibility. For example, the process in which *lp* is created determines where the filtering is done.

6.3 Prototype Application

Figure 1 shows a prototype of one kind of application we built COTERIE to support. It is an augmented reality system that adds both 2D and 3D information to the user's view of the world. It consists of about 500 lines of commented Obliq code, and is an experiment in how 2D windows can be attached to 3D positions. Therefore, these 500 lines include the code used to attach 2D windows to 3D points.

The program makes use of an enhancement to Obliq-3D that we added, called *Projection Callback Objects*. By attaching a projection callback node to the Obliq-3D scene graph, each time the node is traversed in the graphical scene, the *invoke* method of the supplied Callback Object is called with the 3D point corresponding to the origin of the current nested coordinate system and its current 2D projection. To

attach a window to a 3D point, we simply place an instance of a projection callback at that point. The object will contain a handle to its 2D window, and its *invoke* method moves the window to the projected 2D point.

COTERIE makes it easy to distribute this application. Instead of having the *invoke* method move the window, we associate a shared *TrackerPosition* object with each window and have the *invoke* method update this shared object with a 2D tracker report corresponding to the window's projected position. A straightforward *TrackerPosition* Callback Object (Figure 5) could be used to move the windows. However, this callback could do more interesting things, such as sending the information to the process that created the window to have it move or reshape itself. Note that this will work even if the windows are controlled by other processes.

6.4 Hierarchical Object Directories

COTERIE makes it easy to experiment with different kinds of infrastructure. In this section, we outline a nontrivial object, a hierarchical *object directory* (HOD), similar to the hierarchical environments in dVS [17], that illustrates how the three different kinds of objects in our system (Network Objects, Shared Objects and regular objects) can work together.

Design. The purpose of a HOD is to handle object lookup and management. We will design the HOD analogously to a file-system, with a single object directory (OD) containing a set of key-value pairs that associate objects with textual names. An OD can contain references to other ODs, allowing arbitrary hierarchies to be created. References to virtually any kind of object can be stored in an OD. The OD can contain the actual objects or references to entries in other remote ODs (the equivalent to symbolic links in a file-system.) The number of ODs, and their location, is independent of the number of distributed processes and sequencers used to implement a VE application.

In addition to providing a general solution to the *naming problem* (how to meaningfully assign names to services and resolve those names to computer addresses) [11], the HOD can serve as the primary structuring metaphor for a family of distributed applications. By allowing an OD to contain references to other, perhaps distantly located, ODs, we can organize the HODs into a single global name space that allows applications to communicate with each other in a meaningful way. Within this global hierarchy, data can be organized in well defined subhierarchies so that applications know where to look for particular kinds of data and services. Furthermore, by allowing clients to watch one or more ODs for changes, such as the addition or deletion of entries, clients can react to changes in the world without the need for direct communication with the instigator of those changes. Such shared data-space techniques are widely used in AI blackboard systems and programming languages such as Linda [9].

Object Structure of the OD. To build a simple OD, each of the three types of objects are used, as shown in Figure 7.

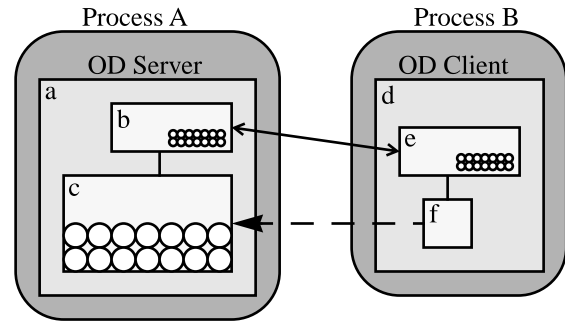


Figure 7: A single Object Directory (OD). The server on the left consists of (a) a local object wrapper, (b) a Shared Object implementing the notifier directory and (c) a Network Object implementing the storage directory. The client on the right has (d) its own copy of the object wrapper, (e) a shared copy of the notifier directory, and (f) a proxy handle for the storage directory.

The OD itself is implemented using a regular *wrapper* object (Figure 7a and d). This object has data fields and methods to implement the OD functionality. For example, there would be methods to add elements to, or delete elements from, the OD. In addition to any other incidental data, the OD contains references to two important objects in its data fields, a *storage directory* and a *notifier directory*. The *storage directory* is a Network Object that implements a centralized object store using key-value pairs (Figure 7c). It would contain the actual data objects stored in the OD. The *notifier directory* is a Shared Object that contains a small, constant-size piece of information for each entry in the directory, such as the type of the object, and is also used to receive notification of changes to the directory (Figure 7b and e).

Because the storage directory is implemented with a Network Object, it is not replicated. The single copy is accessed via remote method calls from any process that receives a copy of the OD. Conversely, since the notifier directory is implemented with a Shared Object, it is fully replicated in all processes that receive a copy of the OD.

Putting It All Together. There are three things to understand about why the OD is designed this way: what happens when an OD is passed to a remote process, how OD methods are implemented, and how are Callback Objects used by the OD.

First, consider what happens when an OD is passed from one process to another, as a parameter or return value of a Network or Shared Object method call. Since the OD is a regular object, the run-time system automatically creates a new, independent copy of the object in the second process. As part of creating that copy, it attempts to copy the data fields of the object. The semantics of copying Shared Objects from one process to another result in the new process containing a local replica of the original notifier directory. The semantics of copying Network Objects result in the new process containing a remote proxy for the original storage directory (Figure 7f). All of this happens automati-

cally when a reference to the OD is passed to the second process.

Given the structure shown in Figure 7, how are OD methods implemented? Consider adding an element to an OD with a simple *put* method, "*put(name, object)*," which stores *object* in the OD under the key *name*. The *put* method of the OD would perform the following actions:

- store the object in the storage directory using the corresponding storage directory *put* method, and
- store the type of the object in the notifier directory using the corresponding notifier directory *put* method, which has been designated as a shared update method.

No matter which process performs the *put* operation, the outcome is the same:

- a remote procedure call is performed to store the object in the central storage directory, and
- the type of the object is stored in the shared notifier directory in all replicas, causing any Callback Objects registered for these replicas to have their *put* methods invoked.

The wrapper object would use the Callback Objects associated with the notifier directory to monitor an object directory for changes on behalf of local clients that have requested notification when the OD changes.

7 Conclusions and Future Work

We have presented our requirements for, and our design and implementation of, COTERIE, a toolkit for building and experimenting with distributed virtual environments. Our guiding belief has been that object-oriented data distribution and multi-threading should be supported at the language level.

Our initial experiences building prototype applications with COTERIE, such as that of Figure 1, have been very promising. We have implemented half a dozen prototypes during the past few months (e.g., an augmented reality system for building construction assistance [33]), the initial versions of which took less than a day each. By following a programming model in which many threads do simple tasks and communicate via Shared Objects, we have found it very easy to reconfigure applications to run on different combinations of processes across a wide variety of platforms.

Our observations and plans for future work can be grouped under the categories software engineering, efficiency, reliability, and scalability.

7.1 Software Engineering

It is quite important that shared data is encapsulated in an object and only accessed via programmer-defined method calls. The programmer has great flexibility in partitioning the work into parts executed once at the calling site and parts executed at all sites, because only update methods are broadcast and executed at all sites. The work is partitioned by having a read method call an update method after performing some work locally. This technique can be used to lessen the impact of the restrictions on update method argument types (e.g., by having a read method manipulate the restricted argument locally and use the results as arguments to an update method call).

Because shared object update method calls are performed synchronously on the local copy, but asynchronously on all other copies, return values and exceptions are ignored in all copies except the local one, with one exception: a certain class of exceptions is used to signal isolated method failures (e.g., exhaustion of local resources). Raising one of these exceptions invalidates the local copy, and any further attempt to access the object raises an exception. The programmer can decide how to handle this exception as appropriate for the specific situation. For example, a new copy could be retrieved, or the threads using the object could be distributed to other processes.

When we designed the Shared Object package, we chose a design that allowed us to ensure consistency of individual replicated objects in the face of reasonably large-scale distribution, and to avoid full replication of the database. To achieve this, we sacrificed ease of dealing with the coordination of simultaneous updates to multiple objects. Systems such as ISIS [3] can ensure *causal* or even *total message ordering* between processes in a distributed system*,* which has the effect of ensuring that all messages are seen in the same order in all processes (or, in the case of causal ordering, all the messages that should matter are seen in the same order.) Unfortunately, providing these more rigid guarantees incurs overheads that are higher than we are willing to pay. In particular, standard message-ordering techniques require that all updates are sent to all processes, and that threads block when they attempt to read local data, not when they perform updates. The latter restriction is much more serious, from our point of view, as the primary goal of replication is fast read access. We are investigating possible modifications to our current object semantics to add stricter guarantees without seriously affecting efficiency.

One way to update multiple objects as a unit is to require that a designated object be treated as a semaphore and be locked prior to updating any of the objects. The problem then reduces to ensuring that all the updates are delivered to a site atomically, so that readers see the changes as one unit, if this is also required. If speed is not a concern, readers can also be required to lock the semaphore object. However, this solution will not scale well. We have found that techniques similar to those used to deal with updating display-list-based graphics terminals can provide insight into how to design objects and applications to deal with this problem. For example, data structures and libraries can provide facilities for "checking out" copies of data and then atomically updating the data structure via a single update method. Objects that handle groups of data can provide methods to swap an existing data element with a new one, instead of only providing methods to add data elements or remove them. We are using these techniques to create a distributed

version of Anim3D that avoids the need for explicit global locks when updating the graphical scene.

Currently, the integration of Shared Objects into Obliq is not as simple as we desire. Programmers must generate Modula-3 Shared Objects and then make each object available as a new data type in Obliq. While relatively straightforward, it goes against our goal of enabling programs to use Obliq until they decide to recode bits of their applications in Modula-3 for efficiency. Therefore, we will be modifying the Obliq language to directly support Shared and Callback Objects as native data types.

7.2 Efficiency

The run-time structure of a distributed system created with COTERIE is similar to that used in RING [16], both in terms of the communication topology and the requirement that all updates to a single object pass through one distinguished process. Their experimental results show that this approach is suitable for large-scale distributed VEs.

Our design includes the ability, currently being implemented, to designate that a specific copy of a Shared Object requires updates in as timely a fashion as possible and to designate a copy as the primary updater. The system can then arrange for update events to be sent directly from the primary updater to that copy. This bypasses the sequencer and decreases the typical network hops from two to one, by having the primary updater handle the sequencing for this object. (Of course, updates by any process other than the primary updater will now take longer, having their network hops increased from two to four because the sequencer must now route update events through the primary updater.) This facility is only needed when it is critical to minimize lag (e.g., when a head tracker is connected to a different machine than the graphics display). In this case, only the primary updater will update the object, so the increased number of network hops for other updaters is not an issue.

Communication between processes is currently implemented using TCP. We will be implementing a version that uses UDP in the near future. The change will be invisible to the programmer, aside from anticipated performance gains.

Another possible efficiency concern is the use of an interpreted language. Since the match between the two languages allows code to be migrated easily from Obliq to Modula-3, the interpreted language has not caused any efficiency problems. The only problem we have encountered thus far is that if a programmer carelessly allocates, and subsequently drops references to, significant amounts of memory when handling frequently occurring events (e.g., tracker movement), the garbage collector takes a noticeable amount of time. However, this problem can be eliminated by using techniques such as caching data structures that will be reused.

7.3 Reliability

As discussed previously, we do not intend to provide the level of reliability of systems such as ISIS [3], but do require robustness in the face of simple faults. For example, it should be possible to restart a crashed process without restarting the entire system. This is extremely important as programs become increasingly distributed and complex, especially during development.

At the object level, crashes affect Network and Shared Objects in different ways. With Network Objects, the surrogate copies can be reobtained when the process is restarted. If the process with the original object crashes, however, the object is no longer valid and all clients will be automatically notified via the Network Object runtime system. When the process restarts and recreates the lost object, higher-level protocols, such as the object directories discussed in Section 6.4, can be used to notify interested clients.

For Shared Objects, we only need to differentiate between the sequencer process and any other process with a copy. If a non-sequencer process with a copy crashes, it can reobtain a copy of the Shared Object when it restarts. As with RING, if a sequencer crashes, we cannot recover. (We will eventually fix this by selecting another process as the sequencer, although we have not yet decided the best way to do this.)

Fault tolerance and crash recovery in VE systems is hindered by the fact that many services cannot be replicated because they are tied to specific hardware. However, the system should handle the disappearance of these services in a reasonable fashion. For example, if the process controlling a head tracker crashes, it should be possible to revert to a stationary display model for that user. For certain data objects, such as trackers, only their current value is meaningful, and will have changed between the time the process crashes and restarts. Furthermore, the state of many other objects in the system is partially based on these volatile objects. It is not clear how to apply typical crash recovery concepts, such as message logging and playback, to these objects.

7.4 Scalability

Scalability was an important motivating factor for our design. The biggest threat to scalability we foresee is the distribution of Shared Object updates, since Shared Objects are used to encapsulate rapidly changing data. Funkhouser demonstrates the scalability of a similar update propagation topology in [16].

Another factor affecting scalability is the mental burden that complex systems place on programmers. We believe that the highly modular style of programming encouraged by a multithreaded, object-oriented system significantly reduces this burden by allowing programmers to create small, self-contained components. COTERIE encourages this style of programming for building distributed systems.

Acknowledgments

We would like to that the reviewers for their comments and suggestions. We would also like to thank Xinshi Sha and Chris Wang for porting and enhancing Obliq-3D, and Tobias Höllerer for initial work on the Shared Object code generator. Rory Stuart of NYNEX Science & Technology provided

us with the crossbox and assistance in designing Figure 1's application. The folks at Digital Equipment Corporation's System Research Center and at Critical Mass, Inc., helped during development, especially Marc Najork and Bill Kalsow. Sun Microsystems provided early access to OpenGL on the Ultra1. This work was supported by the Office of Naval Research under Contract N00014-94-1-0564, the National Science Foundation under Grant CDA-92-23009 and ECD-88-11111, the New York State Center for High Performance Computing and Communications in Healthcare (supported by the New York State Science and Technology Foundation), and gifts from NYNEX Science & Technology and Microsoft.

References

[1] D. B. Anderson, J. W. Barrus, J. H. Howard, C. Rich, C. Shen, and R. C. Waters. Building multi-user interactive multimedia environments at MERL. Technical Report Research Report TR95-17, Mitsubishi Electric Research Laboratory, November 1995.

[2] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[3] K. P. Birman. The process group approach to reliable distributed computing. *CACM*, 36(12):36–53, Dec 1993.

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proc. 14th ACM Symp. on Operating Systems Principles*, 1993.

[5] W. Bricken and G. Coco. The VEOS project. *Presence: Teleoperators and Virtual Environments*, 3(2):111–129, 1994.

[6] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The SIMNET virtual world architecture. In *Proc. IEEE VRAIS '93*, pages 450–455, Sept 1993.

[7] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan 1995.

[8] C. Carlsson and O. Hagsand. DIVE—a multi-user virtual reality system. In *Proc. IEEE VRAIS '93*, pages 394–400, Sept 1993.

[9] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1992.

[10] C. F. Codella, R. Jalili, L. Koved, and J. B. Lewis. A toolkit for developing multi-user, distributed virtual environments. In *Proc. IEEE VRAIS '93*, pages 401–407, Sept 1993.

[11] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 1994.

[12] S. Feiner, B. MacIntyre, M. Haupt, and E. Solomon. Windows on the world: 2D windows for 3D augmented reality. In *Proc. ACM UIST '93*, pages 145–155, 1993.

[13] S. Feiner, B. MacIntyre, and D. Seligmann. Knowledge-based augmented reality. *CACM*, 36(7):52–63, July 1993.

[14] S. Feiner and A. Shamash. Hybrid user interfaces: Breeding virtually bigger interfaces for physically smaller computers. In *Proc. ACM UIST '91*, pages 9–17, Hilton Head, SC, November 11–13 1991.

[15] S. Feiner, A. Webster, T. Krueger, B. MacIntyre, and E. Keller. Architectural anatomy. *Presence: Teleoperators and Virtual Environments*, 4(3):318–325, Summer 1995.

[16] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Proc. 1995 ACM Symp. on Interactive 3D Graphics*, pages 85–92, March 1995.

[17] G. Grimsdale. dVS—distributed virtual environment system. In *Proc. Computer Graphics '91 Conference*, 1991.

[18] S. P. Harbison. *Modula-3*. Prentice-Hall, 1992.

[19] R. Holloway. *Trackerlib User's Maunal*. UNC Chapel Hill Computer Science Department, 1991.

[20] R. Kazman. Making WAVES: On the design of architectures for low-end distributed virtual environments. In *Proc. IEEE VRAIS '93*, pages 443–449, Sept 1993.

[21] W. Levelt, M. Kaashoek, H. Bal, and A. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22(11):985–1010, Nov 1992.

[22] K. Li. Shared virtual memory on loosely coupled multiprocessors. Technical Report Research Report 492 (Ph.D. dissertation), Yale University, 1992.

[23] M. R. Macedonia, M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham. Exploiting reality with multicast groups. *IEEE Computer Graphics and Applications*, 15(5):38–45, September 1995.

[24] M. A. Najork and M. H. Brown. Obliq-3D: A high-level, fast-turnaround 3D animation system. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):175–145, June 1995.

[25] R. Pausch, T. Burnette, A. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. Alice: A rapid prototyping system for 3D graphics. *IEEE Computer Graphics and Applications*, 15(3):8–11, May 1995.

[26] A. Prakash and H. S. Shim. DistView: Support for building efficient collaborative applications using replicated objects. In *Proc. ACM CSCW '94*, pages 153–162, October 1994.

[27] M. Roseman and S. Greenberg. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[28] C. Shaw and M. Green. The MR toolkit peers package and experiment. In *Proc. IEEE VRAIS '93*, pages 18–22, Sept 1993.

[29] G. Singh, L. Serra, W. Png, A. Wong, and H. Ng. BrickNet: Sharing object behaviors on the net. In *Proc. IEEE VRAIS '95*, pages 19–25, 1995.

[30] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *CACM*, 30(1):32–47, January 1987.

[31] I. Tou, S. Berson, G. Estrin, Y. Eterovic, and E. Wu. Prototyping synchronous group applications. *IEEE Computer*, 27(5):48–56, May 1994.

[32] C. Ware, K. Arthur, and S. B. Kellogg. Fish tank virtual reality. In *Proc. ACM INTERCHI '93*, pages 31–37, 1993.

[33] A. Webster, S. Feiner, B. MacIntyre, B. Massie, and T. Krueger. Augmented reality in architectural construction, inspection and renovation. In *Proc. ASCE Third Congress on Computing in Civil Engineering*, pages 913–919, Anaheim, CA, June 17–19 1996.

[34] M. J. Zyda, D. R. Pratt, J. G. Monahan, and K. P. Wilson. NPSNET: Constructing a 3D virtual world. In *Proc. 1992 ACM Symp. on Interactive 3D Graphics*, pages 147–156, Mar. 1992.