# 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

## 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 constructs 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 an example of an application implemented in COTERIE that shows its advantages.

## 1 Introduction

Over the past several years, our group has built a number of single-user, distributed, virtual environment (VE) prototypes [e.g., 10, 11, 13]. 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 Testbed for Exploratory Research in Interactive Environments), is the topic of this paper. In Section 2, we present the design requirements that we identified for COTERIE. Then, we survey related work in VE support software in Section 3 and describe how well it addresses these requirements. Next, we introduce COTERIE's design and implementation in Section 4, showing how it was influenced by and builds on the Modula-3 environment in which it is implemented. In Section 5, we provide examples of higher-level support for VEs that we are developing in COTERIE. This is followed by a discussion of design and implementation issues and our conclusions in Section 6.

## 2 Design Requirements

The requirements we set for our system reflect our desire to experiment with distributed, multi-user VEs that combine a variety of paradigms, including immersive, "fishtank" [26], see-through, and handheld worlds. This means that it is essential that we be able to create robust prototypes quickly and easily to explore different design alternatives. We identified the following core requirements to make this possible.

**Uniform treatment of data.** To build a distributed system, some data sharing mechanism, such as remote procedure or method calling, 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 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 general-purpose facilities for sharing data.

**Embedded interpreted language.** To support rapid prototyping well, a system should have an embedded general-purpose interpreted language, enabling entire applications to be developed without writing compiled code, as exemplified by several existing toolkits [4, 22, 24]. 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, allowing the programming language to assist the programmers as much as possible in creating reliable and robust programs.

**Object-oriented and multithreaded environment.** Conceptually, VE systems are composed of many independent objects that perform tasks such as monitoring trackers, rendering to displays, and controlling the entities that populate the environment. This maps well to an object-oriented, multithreaded environment. However, using heavy-weight processes for all threads of control is unacceptable because of communication and context switching overhead. 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. Furthermore, adding new conceptual threads in this fashion can be extremely difficult. 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.

**High-level, platform-independent, extensible, 3D graphics package.** We wanted to support a wide variety of hardware and operating systems without having to use a different graphics package on each. Furthermore, we wanted to be able to cleanly integrate new kinds of graphical objects, such as the workstation windows of [10].

**Distributed system characteristics.** It was important that the VE systems we build exhibit satisfactory distributed performance [9], with respect to scalability, openness, fault tolerance, and network data transparency. The system should scale well as more users or processors per user are added. It should 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. Finally, the data types seen by programmers should 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.

**Support for multiple users, displays, and devices.** In addition to supporting multiple simultaneous users, we want 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.

**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, we want all user-defined data to be capable of being replicated, not just typical VE data such as tracker reports or scene objects.

**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.

## 3 Related Work

A large number of VE toolkits have been created. We discuss only those that are intended to support distributed environments.

MR implements a simple shared virtual memory model [23]. 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 [7] is built on top of the ISIS [2] fault-tolerant distributed system, and is similar to MR in supporting only fully replicated VEs. VEOS [4] 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 [5] is perhaps the best known large-scale distributed VE system. It uses a well-defined communication protocol that is also used by NPSNet [27] 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 [20].

VR-DECK [8] 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 [17] 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 [14] and BrickNet [24] 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 [15] 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.

None of these 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 window manager [12], a simple example 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 typical VE system.

# 4 Architecture

COTERIE is written in the Modula-3 programming language [16]. Our decision to use Modula-3 was based on the language itself and the availability of a set of packages that provide a solid foundation on which to build our infrastructure.

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 across all architectures. We also depend on three key packages:

- **Network Objects.** The Network Object package [3] supports distributed data sharing by providing remote method call facilities 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 between heterogeneous machines.
- **Obliq.** Obliq [6] 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 of control 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.** Obliq-3D [21] 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 those same primitives embedded in Obliq. Obliq-3D programs can be written in Modula-3, Obliq, or any combination of the two because all data structures are simultaneously available from both languages. The package's structure and interface also make it relatively easy to extend.

1. The Modula-3 compiler is written and distributed by DEC's Systems Research Center. The compiler, and thus our system, runs on all the operating systems we use: OSF/1, SunOS, Solaris, IRIX, HP-UX, Linux, and Windows NT.

Together, the language and packages provide us with closely matched compiled and embedded, 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.

To provide COTERIE's replicated object support, we decided to implement a Shared Object package that, when combined with the Network Object package, would provide us with a distributed language infrastructure geared toward the needs of VE applications. The remainder of COTERIE is implemented using this package.

## 4.1 The Shared Object 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.

### 4.1.1 Distributed Shared Memory

Distributed Shared Memory (DSM)[19] 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 non-deterministic.
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 [18]. 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.

#### 4.1.2 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.

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

#### 4.1.3 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. The Modula-3 source code implementing the shared data-object semantics consists of[1]:

- a subtype of *OBJ* that overrides the methods of *OBJ* to implement the shared object semantics,
- a Callback Object for *OBJ*, and
- an Obliq module allowing objects of type *OBJ* to be manipulated in Obliq.

**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 methods 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 as the original object. 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, 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.

To make a Network Object available on the network, a process advertises it via a network object daemon, which need not be on the same machine. Thus, processes that wish to acquire this object are only required to know the location of the network object daemon. Network Objects are used to make Shared Objects available. Both are passed between processes as arguments to, or return values of, Network Object method calls. Our implementation ensures that there is one and only one copy of any Shared Object in any process.

**Replication Approach.** We based our implementation of the Shared-Data Object Model on the fully replicated approach used in the implementation of Orca described in [1]. A full replication scheme is significantly simpler than a partial replication scheme and satisfies our primary rational for replication: fast read-access to shared data. Furthermore, when used in conjunction with the Network Object package, partial replication can be mimicked using a combination of the two types of objects. When a Shared Object is used from 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 will 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 no surprise that we know of no existing distributed VE systems that uses an invalidation-based replication scheme.)

**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 process called the *sequencer*. Each process is associated with a particular sequencer 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 may serve as a sequencer, in addition to its other tasks.

The sequencer 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 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

1. Note to reviewers: This code is currently hand-generated from a sample template. Later this year, after gaining more experience using the shared objects we have created thus far, we will install an automated code generator similar to that used for Network Objects, which is run automatically from the makefile prior to compilation.

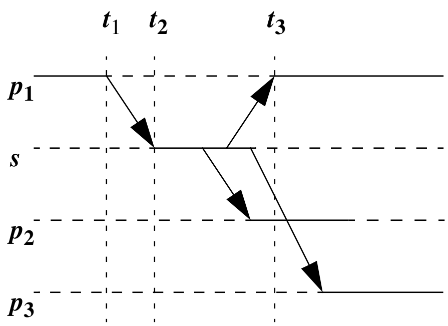


Figure 1: Control and data flow for a typical Shared Object update method call. Process $p_1$ calls an update method at time $t_1$. It blocks after a message is sent to the sequencer $s$. The message arrives at time $t_2$ and is sent to all interested clients ($p_1$, $p_2$ and $p_3$). $p_1$ receives the message back at time $t_3$, is unblocked, executes the original method call and continues.

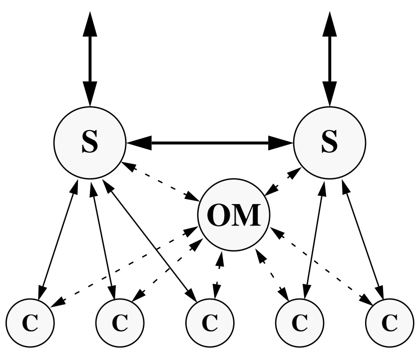


Figure 2: The partitioning of clients (C) among sequencers (S) and object managers (OM). The partitioning of sequencers can be based on efficiency considerations, whereas the partitioning of object managers can be based on the semantic grouping of processes, which are generally not the same.

until the update event is received back from the sequencer, at which time it applies the update to the object and returns. Figure 1 shows the control and data flow of a 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 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.

Overlapping high- and low-level functionality like this is undesirable. 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 2.

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. We provided the ability to designate a copy of a Shared Object as requiring 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 in cases where minimizing lag is critical (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.

**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 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) can be used. Furthermore, since the special mechanisms of Network Object method calls are currently required to pass Network and Shared Objects between processes, neither can be used as arguments to update methods. (We do not feel this is a serious restriction, and will remove it if the need arises.)

**Implications.** The fact that shared data is encapsulated in an object and only accessed via programmer-defined method calls on that object is quite important. First, the programmer of an object 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 same technique can be used to lessen the impact of the restrictions on update method argument types, for example, 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 (such as exhaustion of local resources). Raising one of these exceptions results in the local copy being invalidated. Any further attempt to access this object will raise an exception. The programmer can opportunistically decide how to handle this exception, typically basing the decision on the cause of the problem. For example, a new copy could be retrieved, or the threads using the object could be distributed to other processes.

```
INTERFACE TrackerPosition;
IMPORT SharedObj, Thread;
TYPE
   Data = REF RECORD x,y,z: REAL END;
   T <: S;
   S <: Public;
   Public = SharedObj.T OBJECT METHODS
            set (READONLY val: Data;
            get (): Data;
            <* SHARED UPDATE METHODS set *>
         END;
END TrackerPosition.
```

(a) The interface, *TrackerPosition.i3*

```
MODULE TrackerPosition;
REVEAL
   S = Public BRANDED Brand OBJECT
         data: Data := NIL;
      OVERRIDES
         init := Init;
         set := Set;
         get := Get;
      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 implementation, *TrackerPosition.m3*

Figure 3: An example Shared Object. There are two parts to a module, the interface and the implementation.

#### 4.1.4 Shared Object Example

Figure 3 is the code for a simple shared tracker object. (We have simplified the *Data* type, and reduced the number of methods for clarity.)

In Modula-3, when a type in an interface is used, it must be qualified by the interface name. Thus, the *Data* type in Figure 3(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 3(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* in their programs, they will use the Shared Object type.

Figure 4 shows part of the generated code associated with the tracker object. Part (a) contains the interface to the Callback Object. As described in Section 4.1.2, *pre_set* and *post_set* are the notification methods for *set*, and *pre_anyChange* and *post_anyChange* are the catch-all notification methods. Parts (b) and (c) contain the help files describing the Obliq interfaces to the tracker and Callback Objects, which we include because the

```
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.
```

(a) The generated Callback Object interface, *TrackerPositionCB.i3*

```
   TrackerPosition_New(): T;
WHERE
   T <: SharedObj_T & { get: () => Data,
                         set: (Data) => Ok}
```

(b) The generated helpfile for the Obliq interface, *TrackerPosition.hlp*

```
   TrackerPositionCB_New(obj: TrackerPosition_T,
      overrides: Obj): T;
WHERE
   T <: SharedObj_T & overrides;
   overrides contains one or more of these
   callback methods:
      pre`set(obj: TrackerPosition_T,
         val: Data_T): bool;
      post`set(obj: TrackerPosition_T,
         val: Data_T): bool;
      pre`anyChange(obj: TrackerPosition_T);
      post`anyChange(obj: TrackerPosition_T);
```

(c) The generated helpfile for the Obliq interface to the TrackerPositionCB object, *TrackerPositionCB.hlp*

Figure 4: The generated tracker Callback and Obliq code.

Modula-3 code implementing the Obliq interface to the objects is fairly complex. (The syntax of the Obliq help files is well defined.)

## 5 VE Objects and Application Framework

We are using COTERIE to develop a library of standard objects that provide a framework for building VE applications, along with a number of prototype applications. The examples we provide here are intended to emphasize the ease of creating such a framework, rather than the prototypes themselves.

**Tracker Object.** The first object we developed was a generic shared tracker object, on which the simple 3D position tracker object in Figure 3 is based. Aside from the addition of data fields and methods, the basic object very similar to the one presented here. The aim is to encapsulate support for all our tracking systems into one common object, as done by other systems, such as the UNC Tracker Library [25].

More advanced tracker objects can be implemented using this object. For example, suppose a client wants to receive at most one tracker report per second. A second tracker object, *slowobj*, can be created in the process handing the tracker. It is updated less often by using a simple callback object, *LowFreqTracker*, whose *post_set* method is overridden by *LFSet*, as shown in

```
TYPE LowFreqTracker = TrackerPositionCB.T OBJECT
      slow: TrackerPosition.T;
      interval, next: REAL := 0.0;
   OVERRIDES
      post_set := LFSet;
   END;

PROCEDURE LFSet(READONLY obj: TrackerPosition.T;
                READONLY val: Data): BOOLEAN =
   BEGIN
      IF self.next < Time.Now() THEN
         self.slow.set(val);
         INC(self.next, interval);
      END;
   END LFSet;
...
   (* assume tobj is the normal tracker object *)
   slowobj := NEW(TrackerPosition.T).init();
   slowcb := NEW(LowFreqTracker, slow := slowobj,
                   interval := 1.0).init(tobj);
...
```

Figure 5: A simple use of Callback Objects to create a variation of a tracker that is updated at most once a second.

Figure 5. *slowobj* can now be given to clients who want to have updates sent to them less frequently. Notice that this approach allows great flexibility. For example, the process in which *slowobj* is created determines where the filtering is done.

**Hierarchical Object Space.** Our current approach to object management uses hierarchical object spaces, similar to the hierarchical environments of dVS. Each user would have one object space, regardless of the number of distributed processes and sequencers that implement their VE. Each object space consists of two objects, an *storage space* Network Object that implements a centralized object store using simple key-value pairs, and a *notifier space* Shared Object that can be used by clients to receive notification of changes to the space. Virtually any object can be stored in an object space, including other object spaces.

By implementing the storage space with a Network Object, it is not replicated, but can be accessed remotely from any process. It supports the standard operations of an object space, such as storing and retrieving values by key. It also supports an atomic *put* method that stores a object into the space if the key does not exist, and raises an exception otherwise. This allows simple distributed locking protocols to be implemented.

A notifier space exactly parallels a storage space, except the value stored with a key is the type of the corresponding data field in the storage space. The notifier spaces are used by creating Callback Objects to monitor the corresponding storage space for changes. The callback objects can decide if the change to the space is interesting based on the name and/or type of the entry. For example, consider one approach to organizing how information is displayed. One subspace is used to contain the graphics objects in a user's VE. Another set of subspaces contain the specific objects presented on each display. The first space is monitored by one or more threads that knows about the specific display devices in use. When an object is added or removed from the user's VE, the subspace corresponding to one or more of the displays is updated to contain the object(s) that should be displayed. Finally, the thread(s) controlling the physical displays simply monitor their designated subspaces for changes and update the displays accordingly. For example, a 3D graphics display is updated by modifying the Obliq-3D scene graph to reflect the changes.

Notice the separation of concerns here. Programs that want something displayed simply add it to the single space representing the user's VE without worrying about the number and kind of displays in use or which processes control them. Alternatively, a process that is updating the display need not concern itself with which threads are deciding what should be displayed. If the thread has been given access to the space, it can change it, none of which should concern the process updating the display. Finally, with the exception of the threads controlling physical displays, any thread can be run in any processor.

By forcing all data sharing to go through object spaces, we remove the tangle of explicit connections that hampered our previous prototypes. Furthermore, since the data values that can be stored in the object spaces (and thus shared between process) are virtually indistinguishable from non-shared data values, the network data-transparency of the system is significantly improved. Furthermore, advanced features such as support for persistent storage and journaling, can be added in a straightforward manner.

## 6 Conclusions

Our initial experiences building and using COTERIE are promising. Our observations and plans for future work can be grouped in three general categories: efficiency, reliability, and scalability.

**Efficiency.** Since the match between the two languages allows code to be migrated easily from Obliq to Modula-3, we do not foresee the interpreted language causing 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, techniques such as caching data structures that will be reused eliminate this problem.

**Reliability.** As discussed previously, we do not intend to provide the level of reliability of systems such as ISIS [2], 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 spaces discussed in Section 5, are 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.

We expect that we can recover from most crashes by providing the ability to restart the crashed process, reconnect it to its object space, and update or recreate the appropriate objects. Certain values will still need to be retained, of course, but this data can be stored in the object space. The reliability of the system then reduces to the reliability of the object spaces. This is a much simpler problem and can be largely satisfied, if desired, by using redundancy and replication. If a process containing an object space crashes, secondary object spaces can be automatically used.

**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 will be used to encapsulate rapidly changing data. Funkhouser demonstrates the scalability of a similar update propagation topology in [14].

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 multi-threaded, object-oriented system significantly reduces this burden by allowing programmers to create small, self-contained components. COTERIE allows this style of programming to be used for building distributed systems.

The final scalability issue is how multiple users are supported. Our current approach is to have each user's system export an object space (possibly different than their internal object space) for access by other users' systems. If a different version of an object is desired, an approach similar to that used in the slow tracker example of Figure 5 can be applied. Connecting to, or disconnecting from, another user's VE is simply a matter of adding their external object space to, or deleting it from, the local hierarchy.

## Acknowledgments

We would like to thank the folks at Digital Equipment Corporation's System Research Center for their help during the development of the system, especially Marc Najork and Bill Kalsow. 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, and by the Columbia University CAT in High Performance Computing and Communications in Healthcare, a New York State Center for Advanced Technology supported by the New York State Science and Technology Foundation.

## References

[1] 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.

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

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

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] Personal communication regarding University of North Carolina at Chapel Hill Computer Science Department TrackerLib, 1995.

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

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