A Distributed 3D Graphics Library

Blair MacIntyre and Steven Feiner[1]
Department of Computer Science
Columbia University

Abstract

We present Repo-3D, a general-purpose, object-oriented library for developing distributed, interactive 3D graphics applications across a range of heterogeneous workstations. Repo-3D is designed to make it easy for programmers to rapidly build prototypes using a familiar multi-threaded, object-oriented programming paradigm. All data sharing of both graphical and non-graphical data is done via general-purpose remote and replicated objects, presenting the illusion of a single distributed shared memory. Graphical objects are directly distributed, circumventing the "duplicate database" problem and allowing programmers to focus on the application details.

Repo-3D is embedded in Repo, an interpreted, lexically-scoped, distributed programming language, allowing entire applications to be rapidly prototyped. We discuss Repo-3D's design, and introduce the notion of *local variations* to the graphical objects, which allow local changes to be applied to shared graphical structures. Local variations are needed to support transient local changes, such as highlighting, and responsive local editing operations. Finally, we discuss how our approach could be applied using other programming languages, such as Java.

CR Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; H.4.1 [**Information Systems Applications**]: Office Automation—*Groupware*; I.3.2 [**Computer Graphics**]: Graphics Systems—*Distributed/network graphics*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Graphics data structures and data types*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Virtual reality.*

Additional Keywords and Phrases: object-oriented graphics, distributed shared memory, distributed virtual environments, shared-data object model.

1 INTRODUCTION

Traditionally, *distributed graphics* has referred to the architecture of a single graphical application whose components are distributed over multiple machines [14, 15, 19, 27] (Figure 1a). By taking advantage of the combined power of multiple machines, and the particular features of individual machines, otherwise impractical applications became feasible. However, as machines have grown more powerful and application domains such as Computer Supported Cooperative Work (CSCW) and Distributed Virtual Environments (DVEs) have been making the transition from research labs to commercial products, the term *distributed graphics* is increasingly used to refer to systems for distributing the shared graphical state of multi-display/multi-person, distributed, interactive applications (Figure 1b). This is the definition that we use here.

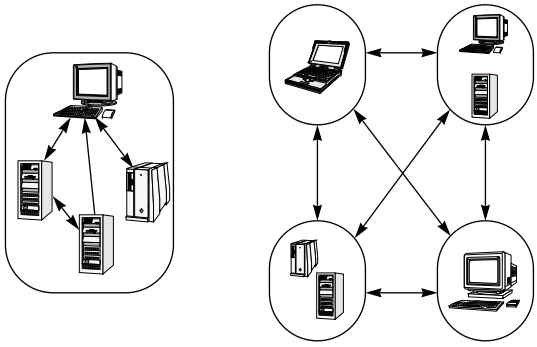

Figure 1: Two meanings of *distributed graphics*: (a) a single logical graphics system with distributed components, and (b) multiple distributed logical graphics systems. We use the second definition here.

While many excellent, high-level programming libraries are available for building stand-alone 3D applications (e.g. Inventor [35], Performer [29], Java 3D [33]), there are no similarly powerful and general libraries for building distributed 3D graphics applications. All CSCW and DVE systems with which we are familiar (e.g., [1, 7, 11, 12, 16, 28, 30, 31, 32, 34, 37, 41]) use the following approach: A mechanism is provided for distributing application state (either a custom solution or one based on a general-purpose distributed programming environment, such as ISIS [4] or Obliq [8]), and the state of the graphical display is maintained separately in the local graphics library. Keeping these "dual databases" synchronized is a complex, tedious, and error-prone endeavor. In contrast, some non-distributed libraries, such as Inventor [35], allow programmers to avoid this problem by using the graphical scene description to encode application state. Extending this "single database" model to a distributed 3D graphics library is the goal of our work on Repo-3D.

Repo-3D is an object-oriented, high-level graphics package, derived from Obliq-3D [25]. Its 3D graphics facilities are similar to those of other modern high-level graphics libraries. However, the objects used to create the graphical scenes are directly distributable—from the programmer's viewpoint, the objects reside in one large distributed shared memory (DSM) instead of in a single process. The underlying system replicates any of the fine-grained objects across as many processes as needed, with no additional effort on the part of the programmer. Updates to objects are automatically reflected in all replicas, with any required objects automatically distributed as needed. By integrating the replicated objects into the programming languages we use, distributed applications may be built using Repo-3D with little more difficulty than building applications in a single process.

1. {bm,feiner}@cs.columbia.edu, http://www.cs.columbia.edu/graphics

No matter how simple the construction of a distributed application may be, a number of differences between distributed and monolithic applications must be addressed. These include:

- *Distributed control*. In a monolithic application, a single component can oversee the application and coordinate activities among the separate components by notifying them of changes to the application state. This is not possible in a non-trivial distributed application. Therefore, we must provide mechanisms for different components to be notified of changes to the distributed state.
- *Interactivity.* Updates to distributed state will be slower than updates to local state, and the amount of data that can be distributed is limited by network bandwidth. If we do not want to sacrifice interactive speed, we must be able to perform some operations locally. For example, an object could be dragged locally with the mouse, with only a subset of the changes applied to the replicated state.
- *Local variations.* There are times when a shared graphical scene may need to be modified locally. For example, a programmer may want to highlight the object under one user's mouse pointer without affecting the scene graph viewed by other users.

Repo-3D addresses these problems in two ways. First, a programmer can associate a *notification* object with any replicated object. The notification object's methods will be invoked when the replicated object is updated. This allows reactive programs to be built in a straightforward manner. To deal with the second and third problems, we introduce the notion of *local variations* to graphical objects. That is, we allow the properties of a graphical object to be modified locally, and parts of the scene graph to be locally added, removed, or replaced.

In Section 2 we describe how we arrived at the solution presented here. Section 3 discusses related work, and Section 4 offers a detailed description of the underlying infrastructure that was used. The design of Repo-3D is presented in Section 5, followed by some examples and concluding remarks in Sections 6 and 7.

2 BACKGROUND

Repo-3D was created as part of a project to support rapid prototyping of distributed, interactive 3D graphical applications, with a particular focus on DVEs. Our fundamental belief is that by providing uniform high-level support for distributed programming in the languages and toolkits we use, prototyping and experimenting with distributed interactive applications can be (almost) as simple as multi-threaded programming in a single process. While care must be taken to deal with network delays and bandwidth limitations at some stage of the program design (the languages and toolkits ought to facilitate this), it should be possible to ignore such issues until they become a problem. Our view can be summarized by a quote attributed to Alan Kay, "Simple things should be simple; complex things should be possible."

This is especially true during the exploration and prototyping phase of application programming. If programmers are forced to expend significant effort building the data-distribution components of the application at an early stage, not only will less time be spent exploring different prototypes, but radical changes in direction will become difficult, and thus unlikely. For example, the implementation effort could cause programs to get locked into using a communication scheme that may eventually prove less than ideal, or even detrimental, to the program's final design.

Since we are using object-oriented languages, we also believe that data distribution should be tightly integrated with the language's general-purpose objects. This lets the language's type system and programming constructs reduce or eliminate errors in the use of the data-distribution system. Language-level integration also allows the system to exhibit a high degree of *network data transparency*, or the ability for the programmer to use remote and local data in a uniform manner. Without pervasive, structured, high-level data-distribution support integrated into our programming languages and libraries, there are applications that will never be built or explored, either because there is too much programming overhead to justify trying simple things ("simple things are not simple"), or because the added complexity of using relatively primitive tools causes the application to become intractable ("complex things are not possible").

Of the tools available for integrating distributed objects into programming languages, client-server data sharing is by far the most common approach, as exemplified by CORBA [26], Modula-3 Network Objects [5], and Java RMI [39]. Unfortunately, interactive graphical applications, such as virtual reality, require that the data used to refresh the display be local to the process doing the rendering or acceptable frame refresh rates will not be achieved. Therefore, pure client-server approaches are inappropriate because at least some of the shared data must be replicated. Furthermore, since the time delay of synchronous remote method calls is unsuitable for rapidly changing graphical applications, shared data should be updated asynchronously. Finally, when data is replicated, local access must still be fast.

The most widely used protocols for replicated data consistency, and thus many of the toolkits (e.g., ISIS [4] and Visual-Obliq [3]), allow data updates to proceed unimpeded, but block threads reading local data until necessary updates arrive. The same reason we need replicated data in the first place—fast local read access to the data—makes these protocols unsuitable for *direct* replication of the graphical data. Of course, these protocols are fine for replicating application state that will then be synchronized with a parallel graphical scene description, but that is what we are explicitly trying to avoid. Fortunately, there are replicated data systems (e.g., Orca [2] or COTERIE [24]) that provide replicated objects that are well suited to interactive applications, and it is upon the second of these systems that Repo-3D is built.

3 RELATED WORK

There has been a significant amount of work that falls under the first, older definition of distributed graphics. A large number of systems, ranging from established commercial products (e.g., IBM Visualization Data Explorer [21]) to research systems (e.g., PARADISE [19] and ATLAS [14]), have been created to distribute interactive graphical applications over a set of machines. However, the goal of these systems is to facilitate sharing of application data between processes, with one process doing the rendering. While some of these systems can be used to display graphics on more than one display, they were not designed to support high-level sharing of graphical scenes.

Most high-level graphics libraries, such as UGA [40], Inventor [35] and Java 3D [33], do not provide any support for distribution. Others, such as Performer [29], provide support for distributing components of the 3D graphics rendering system across multiple processors, but do not support distribution across multiple machines. One notable exception is TBAG [13], a high-level constraint-based, declarative 3D graphics framework. Scenes in TBAG are defined using constrained relationships between time-varying functions. TBAG allows a set of processes to share a single, replicated constraint graph. When any process asserts or retracts a constraint, it is asserted or retracted in all processes. However, this means that all processes share the same scene, and that the system's scalability is limited because all processes have a copy of (and must evaluate) all constraints, whether or not they are interested in them. There is also no support for local variations of the scene in different processes.

Machiraju [22] investigated an approach similar in flavor to ours, but it was not aimed at the same fine-grained level of interactivity and was ultimately limited by the constraints of the implementation platform (CORBA and C++). For example, CORBA objects are heavyweight and do not support replication, so much of their effort was spent developing techniques to support object migration and "fine-grained" object sharing. However, their fine-grained objects are coarser than ours, and, more importantly, they do not support the kind of lightweight, transparent replication we desire. A programmer must explicitly choose whether to replicate, move, or copy an object between processes when the action is to occur (as opposed to at object creation time). Replicated objects are independent new copies that can be modified and used to replace the original—simultaneous editing of objects, or real-time distribution of changes as they are made is not supported.

Of greater significance is the growing interest for this sort of system in the Java and VRML communities. Java, like Modula-3, is much more suitable as an implementation language than C or C++ because of its cross-platform compatibility and support for threads and garbage collection: Without the latter two language features, implementing complex, large-scale distributed applications is extremely difficult. Most of the current effort has been focused on using Java as a mechanism to facilitate multi-user VRML worlds (e.g., Open Communities [38]). Unfortunately, these efforts concentrate on the particulars of implementing shared virtual environments and fall short of providing a general-purpose shared graphics library. For example, the Open Communities work is being done on top of SPLINE [1], which supports only a single top-level world in the local scene database.

Most DVEs [11, 12, 16, 31, 32] provide support for creating shared virtual environments, not general purpose interactive 3D graphics applications. They implement a higher level of abstraction, providing support for rooms, objects, avatars, collision detection, and other things needed in single, shared, immersive virtual environments. These systems provide neither general-purpose programming facilities nor the ability to work with 3D scenes at a level provided by libraries such as Obliq-3D or Inventor. Some use communication schemes that prevent them from scaling beyond a relatively small number of distributed processes, but for most the focus is explicitly on efficient communication. SIMNET [7], and the later NPSNet [41], are perhaps the best known large-scale distributed virtual-environment systems. They use a fixed, well-defined communication protocol designed to support a single, large-scale, shared, military virtual environment.

The techniques for object sharing implemented in recent CSCW toolkits [28, 30, 34, 37] provide some of the features we need, particularly automatic replication of data to ease construction of distributed applications. However, none of these toolkits has integrated the distribution of data into its programming language's object model as tightly as we desire. As a result, they do not provide a high enough level of network data transparency or sufficiently strong consistency guarantees. In groupware applications, inconsistencies tend to arise when multiple users attempt 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 a general-purpose, distributed 3D graphics toolkit. Furthermore, none of these CSCW systems provides any support for asynchronous update notification, or is designed to support the kind of large-scale distribution we have in mind.

Finally, while distributed games, such as Quake, have become very popular, they only distribute the minimum amount of application state necessary. They do not use (or provide) an abstract, high-level distributed 3D graphics system.

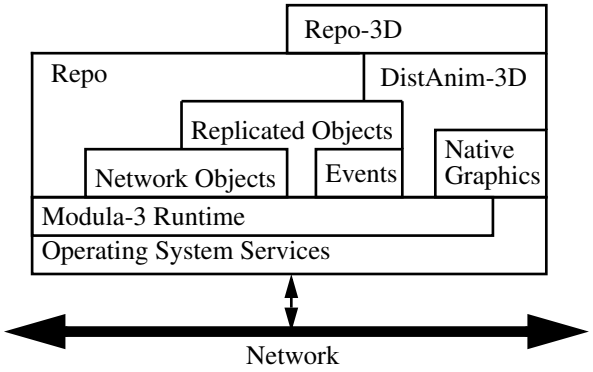

Figure 2: The architecture of Repo-3D. Aside from native graphics libraries (X, Win32, OpenGL, Renderware) the Modula-3 runtime shields most of the application from the OS. The Replicated Object package uses an Event communication package and the Network Object package. DistAnim-3D is implemented on top of a variety of native graphics libraries and Replicated Objects. Repo exposes most of the useful Modula-3 packages, as well as using Network Objects and Replicated Objects to present a distributed shared memory model to the programmer.

4 UNDERLYING INFRASTRUCTURE

Our work was done 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. Modula-3 is a descendant of Pascal that corrects many of its deficiencies, and heavily influenced the design of Java. In particular, Modula-3 retains strong type safety, while adding facilities for exception handling, concurrency, object-oriented programming, and automatic garbage collection[2]. One of its most important features for our work is that it gives us uniform access to these facilities across all architectures.

Repo-3D relies on a number of Modula-3 libraries, as illustrated in Figure 2. Distributed data sharing is provided by two packages, the Network Object client-server object package [5], and the Replicated Object shared object package [24] (see Section 4.1). DistAnim-3D is derived from Anim-3D [25], a powerful, non-distributed, general-purpose 3D library originally designed for 3D algorithm animation (see Section 4.2). Finally, Repo itself is a direct descendant of Obliq [8], and uses the Replicated Object package to add replicated data to Obliq (see Section 4.3).

4.1 Distributed Shared Memory

Repo-3D's data sharing mechanism is based on the Shared Data-Object Model of Distributed Shared Memory (DSM) [20]. DSM 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. 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 level. In this model, shared data is encapsulated in user-defined objects and can only be accessed through those objects' method calls. The DSM address space is partitioned implicitly by the application programmer, with an object being the smallest unit of sharing. All shared data is fully network transpar-

2. The Modula-3 compiler we used is available from Critical Mass, Inc. as part of the Reactor programming environment. The compiler, and thus our system, runs on all the operating systems we have available (plus others): Solaris, IRIX, HP-UX, Linux, and Windows NT and 95.

ent because it is encapsulated within the programming language objects.

Distribution of new objects between the processes is as simple as passing them back and forth as parameters to, or return values from, method calls—the underlying systems take care of the rest.[3] Objects are only distributed to new processes as necessary, and (in our system) are removed by the garbage collector when they are no longer referenced. Furthermore, distributed garbage collection is supported, so objects that are no longer referenced in any process are removed completely.

There are three kinds of distributed object semantics in our DSM:

- *Simple* objects correspond to normal data objects, and have no special distributed semantics. When a simple object is copied between processes, a new copy is created in the destination process that has no implied relationship to the object in the source process.
- *Remote* objects have client-server distribution semantics. When a remote object is copied between processes, all processes except the one in which the object was created end up with a proxy object that forwards method invocations across the network to the original object.
- *Replicated* objects have replicated distribution semantics. When a replicated object is passed between processes, a new replica is created in the destination process. If any replica is changed, the change is reflected in all replicas.

The Network Object package provides support for remote objects. It implements 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 machine architectures. The package is similar to other remote method invocation (RMI) packages developed later, such as the Java RMI library [39]. All method invocations are forwarded to the original object, where they are executed in the order they are received.

The Replicated Object package supports replicated objects. Each process can call any method of an object it shares, just as it can with a simple or remote object. We will describe the Replicated Object package in more detail, as Repo-3D relies heavily on its design, and the design of a replicated object system is less straightforward than a remote one. The model supported by the Replicated Object package follows two principles:

- 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, just as if two threads attempted to access the same memory location simultaneously in a single process.
- The above principle applies to operations on single objects. Making sequences of operations atomic is up to the programmer.

The implementation of the Replicated Object package is based on the approach used in the Orca distributed programming language [2]. A full replication scheme is used, where a single object is either fully replicated in a process or not present at all. Avoiding partial replication significantly simplifies the implementation and the object model, and satisfies the primary rationale for replication: fast read-access to shared data. To maintain replication consistency an update scheme is used, where *updates* to the object are applied to all copies.

The method of deciding what is and is not an update is what makes the Orca approach particularly interesting and easy to implement. All methods are marked as either *read* or *update* methods by the programmer who creates the object type. Read methods are assumed to not change the state of the object and are therefore applied immediately to the local object without violating consistency. Update methods are assumed to change the state. To distribute updates, arguments to the update method are marshalled into a message and sent to all replicas. To ensure all updates are applied in the same order, the current implementation of the Replicated Object package designates a *sequencer* process for each object. There may be more than one sequencer in the system to avoid overloading one process with all the objects (in this case, each object has its updates managed by exactly one of the sequencers.) The sequencer is responsible for assigning a sequence number to each message before it is sent to all object replicas. The replicas then execute the incoming update messages in sequence. The process that initiated the update does not execute the update until it receives a message back from the sequencer and all updates with earlier sequence numbers have been executed.

There are three very important reasons for choosing this approach. First, it is easy to implement on top of virtually any object-oriented language, using automatically generated object subtypes and method wrappers that communicate with a simple runtime system. We do this in our Modula-3 implementation, and it would be equally applicable to an implementation in C++ or Java. For example, the JSDT [36] data-sharing package in Java uses a similar approach.

Second, the Replicated Object package does not pay attention to (or even care) when the internal data fields of an object change. This allows the programmer great flexibility in deciding exactly what constitutes an update or not, and what constitutes the shared state[4]. For example, objects could have a combination of global and local state, and the methods that change the local state could be classified as *read* methods since they do not modify the global state. Alternatively, *read* methods could do some work locally and then call an *update* method to propagate the results, allowing time-consuming computation to be done once and the result distributed in a clean way. We took advantage of both of these techniques in implementing Repo-3D.

Finally, the immediate distribution of update methods ensures that changes are distributed in a timely fashion, and suggests a straightforward solution to the asynchronous notification problem. The Replicated Object package generates a *Notification Object* type for each Replicated Object type. These new objects have methods corresponding to the *update* methods of their associated Replicated Object. The arguments to these methods are the same as the corresponding Replicated Object methods, plus an extra argument to hold the Replicated Object instance. These notifiers can be used by a programmer to receive notification of changes to a Replicated Object in a structured fashion. To react to updates to a Replicated Object instance, a programmer simply overrides the methods of the corresponding Notification Object with methods that react appropriately to those updates, and associates an instance

3. An important detail is how the communication is bootstrapped. In the case of the Network and Replicated Object packages, to pass a first object between processes, one of them exports the object to a special *network object demon* under some known name on some known machine. The second process then retrieves the object.

4. Of course, it falls squarely on the shoulders of the programmer to ensure that the methods provided always leave the object in a consistent state. This is not significantly different than what needs to be done when building a complex object that is simultaneously accessed by multiple threads in a non-distributed system. For example, if a programmer reads an array of numbers from inside the object and then uses an update method to write a computed average back into the object, the internal array may have changed before the average is written, resulting in a classic inconsistency problem. In general, methods that perform computations based on internal state (rather than on the method arguments) are potentially problematic and need to be considered carefully.

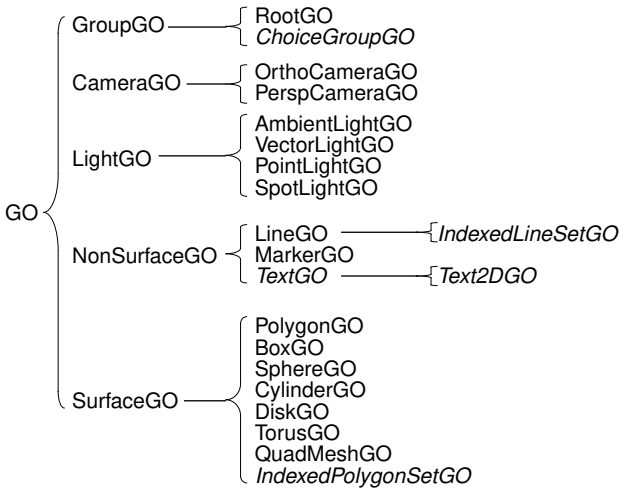

Figure 3: The Repo-3D GO class hierarchy. Most of the classes are also in Obliq-3D; the italicized ones were added to Repo-3D.

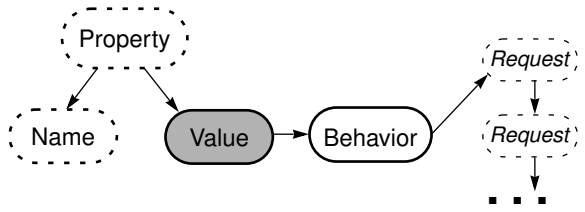

Figure 4: The relationship between properties, names, values, and behaviors. Each oval represents an object and arrows show containment.

of it with the Replicated Object instance. Each time an update method of the Replicated Object is invoked, the corresponding method of the Notifier Object is also invoked. Notification Objects eliminate the need for object polling and enable a "data-driven" flow of control.

4.2 Obliq-3D

Obliq-3D is composed of Anim-3D, a 3D animation package written in Modula-3, and a set of wrappers that expose Anim-3D to the Obliq programming language (see Section 4.3). Anim-3D is based on three simple and powerful concepts: *graphical objects* for building graphical scenes, *properties* for specifying the behavior of the graphical objects, and input event *callbacks* to support interactive behavior. Anim-3D uses the *damage-repair* model: whenever a graphical object or property changes (is damaged), the image is repaired without programmer intervention.

Graphical objects (GOs) represent all the logical entities in the graphical scene: geometry (e.g., lines, polygons, spheres, polygon sets, and text), lights and cameras of various sorts, and groups of other GOs. One special type of group, the `RootGO`, represents a window into which graphics are rendered. GOs can be grouped together in any valid directed acyclic graph (DAG). The GO class hierarchy is shown in Figure 3.

A *property* is a defined by a *name* and a *value*. The name determines which attribute is affected by the property, such as "Texture Mode" or "Box Corner1". The value specifies how it is affected and is determined by its *behavior*, a time-variant function that takes the current animation time and returns a value. Properties, property values, and behaviors are all objects, and their relationships are shown in Figure 4. When a property is created, its name and value are fixed. However, values are mutable and their behavior may be changed at any time. There are four kinds of behaviors for each type of properties: *constant* (do not vary over time), *synchronous* (follow a programmed set of *requests*, such as "move from A to B starting at time t=1 and taking 2 seconds"), *asynchronous* (execute an arbitrary time-dependent function to compute the value) and *dependent* (asynchronous properties that depend on other properties). Synchronous properties are linked to *animation handles* and do not start satisfying their requests until the animation handle is signalled. By linking multiple properties to the same handle, a set of property value changes can be synchronized.

Associated with each GO *g* is a partial mapping of property names to values determined by the properties that have been associated with *g*. A property associated with *g* affects not only *g* but all the descendants of *g* that do not override the property. A single property may be associated with any number of GOs. It is perfectly legal to associate a property with a GO that is not affected by it; for example, attaching a "Surface Color" property to a GroupGO does not affect the group node itself, but could potentially affect the surface color of any GO contained in that group. A RootGO sets an initial default value for each named property.

There are three types of input event callbacks in Anim-3D, corresponding to the three kinds of interactive events they handle: *mouse* callbacks (triggered by mouse button events), *motion* callbacks (triggered by mouse motion events) and *keyboard* callbacks (triggered by key press events). Each object has three callback stacks, and the interactive behavior of an object can be redefined by pushing a new callback onto the appropriate stack. Any event that occurs within a root window associated with a RootGO *r* will be delivered to the top handler on *r*'s callback stack. The handler could delegate the event to one of *r*'s children, or it may handle it itself, perhaps changing the graphical scene in some way.

DistAnim-3D is a direct descendant of Anim-3D. In addition to the objects being distributed, it has many additional facilities that are needed for general-purpose 3D graphical applications, such as texture mapping, indexed line and polygon sets, choice groups, projection and transformation callbacks, and picking. Since DistAnim-3D is embedded in Repo instead of Obliq (see Section 4.3), the resulting library is called Repo-3D.

4.3 Obliq and Repo

Obliq [8] is a lexically-scoped, untyped, interpreted language for distributed object-oriented computation. It is implemented in, and tightly integrated with, Modula-3. An Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet. Obliq uses, and supports, the Modula-3 thread, exception, and garbage-collection facilities. Its distributed-computation mechanism is based on Network Objects, allowing transparent support for multiple processes on heterogeneous machines. Objects are local to a site, while computations can roam over the network. Repo [23] is a descendant of Obliq that extends the Obliq object model to include replicated objects. Therefore, Repo objects have state that may be local to a site (as in Obliq) or replicated across multiple sites.

5 DESIGN OF REPO-3D

Repo-3D's design has two logical parts: the *basic design* and *local variations*. The *basic design* encompasses the changes to Obliq-3D to carry it into a distributed context, and additional enhancements that are not particular to distributed graphics (and are therefore not discussed here). *Local variations* are introduced to handle two issues mentioned in Section 1: transient local changes and responsive local editing.

5.1 Basic Repo-3D Design

The Anim-3D scene-graph model is well suited for adaptation to a distributed environment. First, in Anim-3D, properties are attached to nodes, not inserted into the graph, and the property and child lists are unordered (i.e., the order in which properties are assigned to a node, or children are added to a group, does not affect the final result). In libraries that insert properties and nodes in the graph and execute the graph in a well-defined order (such as Inventor), the *siblings* of a node (or subtree) can affect the attributes of that node (or subtree). In Anim-3D, and similar libraries (such as Java 3D), properties are only inherited *down* the graph, so a node's properties are a function of the node itself and its ancestors—its siblings do not affect it. Therefore, subtrees can be added to different scene graphs, perhaps in different processes, with predictable results.

Second, the interface (both compiled Anim-3D and interpreted Obliq-3D) is programmatical and declarative. There is no "graphical scene" file format per se: graphical scenes are created as the side effect of executing programs that explicitly create objects and manipulate them via the object methods. Thus, all graphical objects are stored as the Repo-3D programs that are executed to create them. This is significant, because by using the Replicated Object library described in Section 4.1 to make the graphical objects distributed, the "file format" (i.e., a Repo-3D program) is updated for free.

Converting Anim-3D objects to Replicated Objects involved three choices: what objects to replicate, what methods update the object state, and what the global, replicated state of each object is. Since replicated objects have more overhead (e.g., method execution time, memory usage, and latency when passed between processes), not every category of object in Repo-3D is replicated. We will consider each of the object categories described in Figure 4.2 in turn: graphical objects (GOs), properties (values, names, behaviors, animation handles) and callbacks. For each of these objects, the obvious methods are designated as update methods, and, as discussed in Section 4.1, the global state of each object is implicitly determined by those update methods. Therefore, we will not go into excessive detail about either the methods or the state. Finally, Repo-3D's support for change notification will be discussed.

5.1.1 Graphical Objects

GOs are the most straightforward. There are currently twenty-one different types of GOs, and all but the RootGOs are replicated. Since RootGOs are associated with an onscreen window, they are not replicated—window creation remains an active decision of the local process. Furthermore, if replicated windows are needed, the general-purpose programming facilities of Repo can be used to support this in a relatively straightforward manner, outside the scope of Repo-3D. A GO's state is comprised of the properties attached to the object, its name, and some other non-inherited property attributes.[5] The methods that modify the property list are update methods. Group GOs also contain a set of child nodes, and have update methods that modify that set.

5.1.2 Properties

Properties are more complex. There are far more properties in a graphical scene than there are graphical objects, they change much more rapidly, and each property is constructed from a set of Modula-3 objects. There are currently 101 different properties of seventeen different types in Repo-3D, and any of them can be attached to any GO. A typical GO would have anywhere from two or three (e.g., a BoxGO would have at least two properties to define its corners) to a dozen or more. And, each of these properties could be complex: in the example in Section 6, a single synchronous property for a long animation could have hundreds of requests enqueued within it.

Consider again the object structure illustrated in Figure 4. A property is defined by a name and a value, with the value being a container for a behavior. Only one of the Modula-3 objects is replicated, the property *value.* Property values serve as the replicated containers for property behaviors. To change a property, a new behavior is assigned to its value. The state of the value is the current behavior.

Animation handles are also replicated. They tie groups of related synchronous properties together, and are the basis for the interaction in the example in Section 6. In Anim-3D, handles have one `animate` method, which starts an animation and blocks until it finishes. Since update methods are executed everywhere, and block access to the object while they are being executed, they should not take an extended period of time. In creating Repo-3D, the `animate` method was changed to call two new methods: an update method that starts the animation, and a non-update method that waits for the animation to finish. We also added methods to pause and resume an animation, to retrieve and change the current relative time of an animation handle, and to stop an animation early. The state of an Animation handle is a boolean value that says if it is active or not, plus the start, end, and current time (if the handle is paused).

Most of the Modula-3 objects that comprise a property are not replicated, for a variety of reasons:

- *Properties* represent a permanent binding between a property value and a name. Since they are immutable, they have no synchronization requirements and can simply be copied between processes.
- *Names* represent simple constant identifiers, and are therefore not replicated either.
- *Behaviors* and *requests* are not replicated. While they can be modified after being created, they are treated as immutable data types for two reasons. First, the vast majority of behaviors, even complex synchronous ones, are not changed once they have been created and initialized. Thus, there is some justification for classifying the method calls that modify them as part of their initialization process. The second reason is practical and much more significant. Once a scene has been created and is being "used" by the application, the bulk of the time-critical changes to it tend to be assignments of new behaviors to the existing property values. For example, an object is moved by assigning a new (often constant) behavior to its `GO_Transform` property value. Therefore, the overall performance of the system depends heavily on the performance of property value behavior changes. By treating behaviors as immutable objects, they can simply be copied between processes without incurring the overhead of the replicated object system.

5.1.3 Input Callbacks

In Repo-3D, input event callbacks are not replicated. As discussed in Section 4.2, input events are delivered to the callback stacks of a RootGO. Callbacks attached to any other object receive input events only if they are delivered to that object by the programmer, perhaps recursively from another input event callback (such as the one attached to the RootGO). Therefore, the interactive behavior of a root window is defined not only by the callbacks attached to its RootGO, but also by the set of callbacks associated with the graph rooted at that RootGO. Since the RootGOs are not replicated, the

5. Some attributes of a GO, such as the arrays of Point3D properties that define the vertices of a polygon set, are not attached to the object, but are manipulated through method calls.

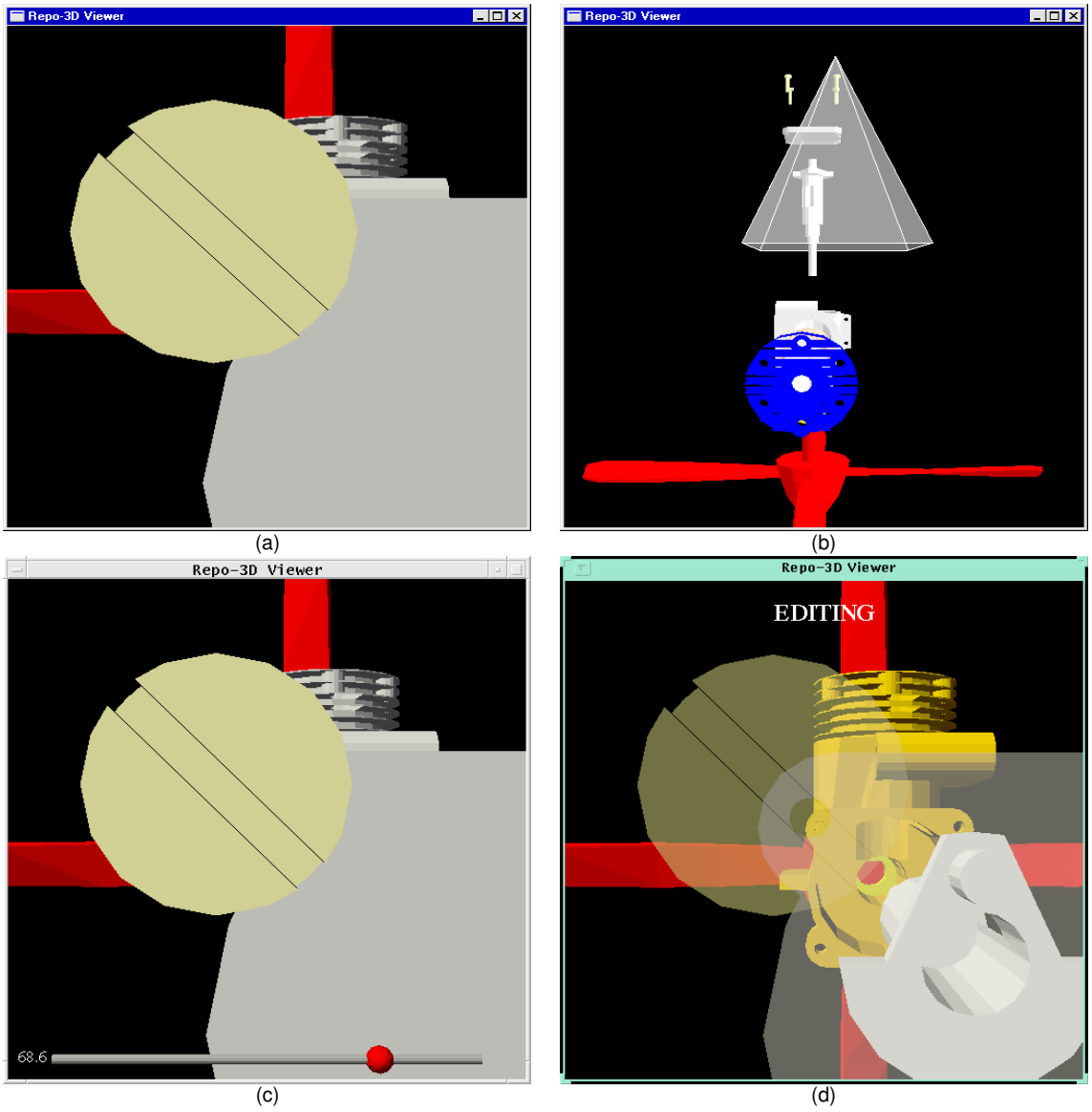

Figure 5: Simultaneous images from a session with the distributed CATHI animation viewer, running on four machines, showing an animation of an engine. (a) Plain animation viewer, running on Windows NT. (b) Overview window, running on Windows 95. (c) Animation viewer with local animation meter, running on IRIX. (d) Animation viewer with local transparency to expose hidden parts, running on Solaris.

callbacks that they delegate event handling to are not replicated either. If a programmer wants to associate callbacks with objects as they travel between processes, Repo's general-purpose programming facilities can be used to accomplish this in a straightforward manner.

5.1.4 Change Notification

The final component of the basic design is support for notification of changes to distributed objects. For example, when an object's position changes or a new child is added to a group, some of the processes containing replicas may wish to react in some way. Fortunately, as discussed in Section 4.1, the Replicated Object package automatically generates Notification Object types for all replicated object types, which provide exactly the required behavior. The Notification Objects for property values allow a programmer to be notified of changes to the behavior of a property, and the Notification Objects for the various GOs likewise allow notification of updates to them.

5.2 Local Variations

Repo-3D's *local variations* solve a set of problems particular to the distributed context in which Repo-3D lives: maintaining interactivity and supporting local modifications to the shared scene graph.

If the graphical objects and their properties were always strictly replicated, programmers would have to create local variations by copying the objects to be modified, creating a set of Notification Objects on the original objects, the copies of those objects, and all their properties (to be notified when either change), and reflecting the appropriate changes between the instances. Unfortunately, while this process could be automated somewhat, it would still be extremely tedious and error prone. More seriously, the overhead of creating this vast array of objects and links between them would

make this approach impractical for short transient changes, such as highlighting an object under the mouse.

To overcome this problem, Repo-3D allows the two major elements of the shared state of the graphical object scene—the properties attached to a GO and the children of a group—to have *local variations* applied to them. (Local variations on property values or animation handles are not supported, although we are considering adding support for the latter.)

Conceptually, local state is the state added to each object (the additions, deletions, and replacements to the properties or children) that is only accessible to the local copies and is not passed to remote processes when the object is copied to create a new replica. The existence of local state is possible because, as discussed in Section 4.1, the *shared state* of a replicated object is implicitly defined by the methods that update it[6]. Therefore, the new methods that manipulate the local variations are added to the GOs as *non-update* methods. Repo-3D combines both the global and local state when creating the graphical scene using the underlying graphics package.

As mentioned above, local variations come in two flavors:

- *Property variations.* There are three methods to set, unset, and get the global property list attached to a GO. We added the following methods to manipulate local variations: add or remove local properties (overriding the value normally used for the object), hide or reveal properties (causing the property value of the parent node to be inherited), and flush the set of local variations (removing them in one step) or atomically apply them to the global state of the object.
- *Child variations.* There are five methods to add, remove, replace, retrieve, and flush the set of children contained in a group node. We added the following ones: add a local node, remove a global node locally, replace a global node with some other node locally, remove each of these local variations, flush the local variations (remove them all in one step), and atomically apply the local variations to the global state.

This set of local operations supports the problems local variations were designed to solve, although some possible enhancements are discussed in Section 7.

6 EXAMPLE: AN ANIMATION EXAMINER

As an example of the ease of prototyping distributed applications with Repo-3D, we created a distributed animation examiner for the CATHI [6] animation generation system. CATHI generates short informational animation clips to explain the operation of technical devices. It generates full-featured animation scripts, including camera and object motion, color and opacity effects, and lighting setup.

It was reasonably straightforward to modify CATHI to generate Repo-3D program files, in addition to the GeomView and RenderMan script files it already generated. The resulting output is a Repo-3D program that creates two scene DAGs: a camera graph and a scene graph. The objects in these DAGs have *synchronous behaviors* specified for their surface and transformation properties. An entire animation is enqueued in the requests of these behaviors, lasting anywhere from a few seconds to a few minutes.

We built a distributed, multi-user examiner over the course of a weekend. The examiner allows multiple users to view the same animation while discussing it (e.g., via electronic chat or on the phone). Figure 5 shows images of the examiner running on four machines, each with a different view of the scene. The first step was to build a simple "loader" that reads the animation file, creates a window, adds the animation scene and camera to it, and exports the animation on the network, requiring less than a dozen lines of Repo-3D code. A "network" version, that imports the animation from the network instead of reading it from disk, replaced the lines of code to read and export the animation with a single line to import it. Figure 5(a) shows an animation being viewed by one of these clients.

The examiner program is loaded by both these simple clients, and is about 450 lines long. The examiner supports:

- Pausing and continuing the animation, and changing the current animation time using the mouse. Since this is done by operating on the shared animation handle, changes performed by any viewer are seen by all. Because of the consistency guarantees, all users can freely attempt to change the time, and the system will maintain all views consistently.
- A second "overview" window (Figure 5(b)), where a new camera watches the animation scene and camera from a distant view. A local graphical child (representing a portion of the animation camera's frustum) was added to the shared animation camera group to let the attributes of the animation camera be seen in the overview window.
- A local animation meter (bottom of Figure 5(c)), that can be added to any window by pressing a key, and which shows the current time offset into the animation both graphically and numerically. It was added in front of the camera in the animation viewer window, as a local child of a GO in the camera graph, so that it would be fixed to the screen in the animation viewer.
- Local editing (Figure 5(d)), so that users can select objects and make them transparent (to better see what was happening in the animation) or hide them completely (useful on slow machines, to speed up rendering). Assorted local feedback (highlighting the object under the mouse and flashing the selected object) was done with local property changes to the shared GOs in the scene graph.

Given the attention paid to the design of Repo-3D, it was not necessary to be overly concerned with the distributed behavior of the application (we spent no more than an hour or so). Most of that time was spent deciding if a given operation should be global or a local variation. The bulk of programming and debugging time was spent implementing application code. For example, in the overview window, the representation of the camera moves dynamically, based on the bounding values of the animation's scene and camera graphs. In editing mode, the property that flashes the selected node bases its local color on the current global color (allowing a user who is editing while an animation is in progress to see any color changes to the selected node.)

7 CONCLUSIONS AND FUTURE WORK

We have presented the rationale for, and design of, Repo-3D, a general-purpose, object-oriented library for developing distributed, interactive 3D graphics applications across a range of heterogeneous workstations. By presenting the programmer with the illusion of a large shared memory, using the Shared Data-Object model of DSM, Repo-3D makes it easy for programmers to rapidly prototype distributed 3D graphics applications using a familiar object-oriented programming paradigm. Both graphical and general-purpose, non-graphical data can be shared, since Repo-3D is embedded in Repo, a general-purpose, lexically-scoped, distributed programming language.

Repo-3D is designed to directly support the distribution of graphical objects, circumventing the "duplicate database" problem and allowing programmers to concentrate on the application function-

6. The local state is not copied when a replicated object is first passed to a new process because the Repo-3D objects have custom *serialization* routines (or Picklers, in Modula-3 parlance). These routines only pass the global state, and initialize the local state on the receiving side to reasonable default values corresponding to the empty local state.

ality of a system, rather than its communication or synchronization components. We have introduced a number of issues that must be considered when building a distributed 3D graphics library, especially concerning efficient and clean support for data distribution and local variations of shared graphical scenes, and discussed how Repo-3D addresses them.

There are a number of ways in which Repo-3D could be improved. The most important is the way the library deals with time. By default, the library assumes all machines are running a time-synchronization protocol, such as NTP, and uses an internal animation time offset[7] (instead of the system-specific time offset) because different OSs (e.g., NT vs. UNIX) start counting time at different dates. Hooks have been provided to allow a programmer to specify their own function to compute the "current" animation time offset within a process. Using this facility, it is possible to build inter-process time synchronization protocols (which we do), but this approach is not entirely satisfactory given our stated goal of relieving the programmer of such tedious chores. Future systems should integrate more advanced solutions, such as adjusting time values as they travel between machines, so that users of computers with unsynchronized clocks can collaborate[8]. This will become more important as mobile computers increase in popularity, as it may not be practical to keep their clocks synchronized.

The specification of local variations in Repo-3D could benefit from adopting the notion of *paths* (as used in Java 3D and Inventor, for example). A path is an array of objects leading from the root of the graph to an object; when an object occurs in multiple places in one or more scene graphs, paths allow these instances to be differentiated. By specifying local variations using paths, nodes in the shared scene graphs could have variations *within* a process as well as *between* processes. One other limitation of Repo-3D, arising from our use of the Replicated Object package, is that there is no way to be notified when local variations are applied to an object. Recall that the methods of an automatically generated Notification Object correspond to the update methods of the corresponding Replicated Object. Since the methods that manipulate the local variations are non-update methods (i.e., they do not modify the replicated state), there are no corresponding methods for them in the Notification Objects. Of course, it would be relatively straightforward to modify the Replicated Object package to support this, but we have not yet found a need for these notifiers.

A more advanced replicated object system would also improve the library. Most importantly, support for different consistency semantics would be extremely useful. If we could specify semantics such as "all updates completely define the state of an object, and only the last update is of interest," the efficiency of the distribution of property values would improve significantly; in this case, updates could be applied (or discarded) when they arrive, without waiting for all previous updates to be applied, and could be applied locally without waiting for the round trip to the sequencer. There are also times when it would be useful to have support for consistency across multiple objects, either using *causal ordering* (as provided by systems such as ISIS and Visual-Obliq), or some kind of transaction protocol to allow large groups of changes to be applied either as a unit, or not at all. It is not clear how one would provide these features with a replicated object system such as the one used here.

While a library such as Repo-3D could be built using a variety of underlying platforms, the most likely one for future work is Java. Java shares many of the advantages of Modula-3 (e.g., threads and garbage collection are common across all architectures) and the packages needed to create a Repo-3D-like toolkit are beginning to appear. While Java does not yet have a replicated object system as powerful as the Replicated Object package, a package such as JSDT [36] (which focuses more on data communication than high-level object semantics) may be a good starting point. Work is also being done on interpreted, distributed programming languages on top of Java (e.g., Ambit [9]). Finally, Java 3D is very similar to Anim-3D, even though its design leans toward efficiency instead of generality when there are trade-offs to be made. For example, the designers chose to forgo Anim-3D's general property inheritance mechanism because it imposes computational overhead. By combining packages such as Java 3D, JSDT, and Ambit, it should be possible to build a distributed graphics library such as Repo-3D in Java.

7. Computed as an offset from January 1, 1997.
8. Implementation details of the combination of Network and Replicated Objects made it difficult for us to adopt a more advanced solution.

Acknowledgments

We would like to thank the reviewers for their helpful comments, as well as the many other people who have contributed to this project. Andreas Butz ported CATHI to use Repo-3D and helped with the examples and the video. Clifford Beshers participated in many lively discussions about the gamut of issues dealing with language-level support for 3D graphics. Tobias Höllerer and Steven Dossick took part in many other lively discussions. Xinshi Sha implemented many of the extensions to Obliq-3D that went into Repo-3D. Luca Cardelli and Marc Najork of DEC SRC created Obliq and Obliq-3D, and provided ongoing help and encouragement over the years that Repo and Repo-3D have been evolving.

This research was funded in part by the Office of Naval Research under Contract N00014-97-1-0838 and the National Tele-Immersion Initiative, and by gifts of software from Critical Mass 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. Bharat and L. Cardelli. Migratory Applications. In *ACM UIST '95*, pages 133-142, November 1995.

[4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *CACM*, 36(12):36–53, Dec 1993.

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

[6] A Butz, Animation with CATHI, In *Proceedings of AAAI/IAAI '97*, pages 957–962, 1997.

[7] 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.

[8] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, Jan 1995.

[9] L. Cardelli and A. Gordon. Mobile Ambients. In *Foundations of Software Science and Computational Structures*, Maurice Nivat (Ed.), LNCE 1378, Springer, 140–155. 1998.

[10] R. Carey and G. Bell. The Annotated VRML 2.0 Reference Manual. Addison-Wesley, Reading, MA, 1997.

[11] C. Carlsson and O. Hagsand. DIVE—A Multi-User Virtual Reality System. In *Proc. IEEE VRAIS '93*, pages 394–400, Sept 1993.

[12] 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.

[13] C. Elliott, G. Schechter, R. Yeung and S. Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications, In *Proc. ACM SIGGRAPH 94*, pages 421–434, August, 1994.

[14] M. Fairen and A. Vinacua, ATLAS, A Platform for Distributed Graphics Applications, In *Proc. VI Eurographics Workshop on Programming Paradigms in Graphics*, pages 91–102, September, 1997.

[15] 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.

[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] H.W. Holbrook, S.K. Singhal and D.R. Cheriton, Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation, *Proc. ACM SIGCOMM '95*, pages 328–341, 1995.

[20] 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.

[21] B. Lucas. A Scientific Visualization Renderer. In *Proc. IEEE Visualization '92*, pp. 227-233, October 1992.

[22] V. Machiraju, A Framework for Migrating Objects in Distributed Graphics Applications, Masters Thesis, University of Utah, Department of Computer Science, Salt Lake City, UT, June, 1997.

[23] B. MacIntyre. Repo: Obliq with Replicated Objects. Programmers Guide and Reference Manual. Columbia University Computer Science Department Research Report CUCS-023-97, 1997.}

[24] B. MacIntyre, and S. Feiner. Language-level Support for Exploratory Programming of Distributed Virtual Environments. In *Proc. ACM UIST '96*, pages 83–94, Seattle, WA, November 6–8, 1996.

[25] 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.

[26] R. Ben-Natan. CORBA: A Guide to the Common Object Request Broker Architecture, McGraw Hill, 1995.

[27] D. Phillips, M. Pique, C. Moler, J. Torborg, D. Greenberg. Distributed Graphics: Where to Draw the Lines? Panel Transcript, SIGGRAPH 89, available at: http://www.siggraph.org:443/publications/panels/siggraphi89/

[28] 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.

[29] J. Rohlf and J. Helman, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time {3D} Graphics, In *Proc. ACM SIGGRAPH 94*, pages 381–394, 1994.

[30] 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.

[31] C. Shaw and M. Green. The MR Toolkit Peers Package and Experiment. In *Proc. IEEE VRAIS '93*, pages 18–22, Sept 1993.

[32] 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.

[33] H. Sowizral, K. Rushforth, and M. Deering. The Java 3D API Specification, Addison-Wesley, Reading, MA, 1998.

[34] 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.

[35] P. S. Strauss and R. Carey, An Object-Oriented 3D Graphics Toolkit, In *Computer Graphics (Proc. ACM SIGGRAPH 92)*, pages 341–349, Aug, 1992.

[36] Sun Microsystems, Inc. The Java Shared Data Toolkit, 1998. Unsupported software, available at: http://developer.javasoft.com/developer/earlyAccess/jsdt/

[37] I. Tou, S. Berson, G. Estrin, Y. Eterovic, and E. Wu. Prototyping Synchronous Group Applications. *IEEE Computer*, 27(5):48–56, May 1994.

[38] R. Waters and D. Anderson. The Java Open Community Version 0.9 Application Program Interface. Feb, 1997. Available online at: http://www.merl.com/opencom/opencom-java-api.html

[39] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System, In *Proc. USENIX COOTS '96*, pages 219–231, July 1996.

[40] R. Zeleznik, D. Conner, M. Wloka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes, and A. van Dam. An Object-oriented Framework for the Integration of Interactive Animation Techniques. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 105–112, July, 1991.

[41] 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.