The Architecture of a Distributed Virtual Worlds System

Manny Vellon, Kirk Marple, Don Mitchell, Steven Drucker
Virtual Worlds Group
Microsoft Research
Microsoft Corporation

Abstract

We have developed an object model that facilitates the development of shared virtual environments. We have implemented our object model on top of COM and OLE Automation and facilitated access from Active Scripting enabled languages. This paper provides a brief description of the work we have done on the *V-Worlds* project.

1. Introduction

Virtual reality is a user-interface paradigm in which the user feels immersed in a computer-generated space. Two aspects of this feeling of immersion have been discussed: simulating the sensory experience of being in a space, and the non-sensory aspect of presenting the user with consistent structure and action [Mitchell94]. The sensory aspect has been pursued by research and development in 3D computer graphics and display technology [Sutherland65, Sutherland68, Brooks86, Brooks88]. Some currently popular computer games have demonstrated immersive 3D graphical interfaces on personal computers available to the general public [DOOM93, QUAKE97].

We use the term *virtual world* for virtual reality systems that allow multiple users to interact in the same space. Adding multiple users to VR creates a number of interesting new problems. Networking with multiple clients is obviously necessary, and in some cases the technology of distributed databases may be required to support a multi-user VR system. The interface now requires social functionality for talking and gesturing. The graphical presentation of the space must allow a group of people to interact socially and see one another's actions and responses. Security is an issue, especially if the system allows users to build and program within the world.

Current work on multi-user VR systems can be divided into graphical chat systems, characterized by static spaces and transient user identity and persistent worlds with dynamic spaces, movable objects and permanent user identities. Two very different kinds of systems have pioneered the development of large-scale virtual worlds: military simulation networks [Thorpe87, Zyda92], and text-based multi-user worlds known as MUDs [Reid94]. SIMNET is based on vehicle and flight simulators that generate real-time 3D images of a virtual world. A peer-to-peer network protocol allows these simulators to display other users' vehicles and projectiles during virtual battle simulations.

MUDs maintain long-term persistent worlds in a central object server; these worlds are accessed via clients similar in appearance to the old text adventure computer games. Having existed for almost twenty years, MUDs are a rich source of experience about the structural aspects of virtual worlds. Some MUDs have been in continuous operation for ten years and have on the order of 10,000 subscribed users [FurryMUCK, LambdaMOO], so there is also considerable experience about the sociology of on-line worlds. We've drawn more extensively from the technology of MUDs than from graphical VR systems and standards, because we are explicitly interested in supporting the structural and social mechanisms found in MUDs.

Elizabeth Reid's thesis gives a well-researched history and analysis of MUDs [Reid94]. Early multi-user combat/adventure games appeared in the late 1970s, and by the mid 1980s, some of them had abandoned actual game play and enhanced user communication and self-expression, becoming what are now called social MUDs. Jim Aspnes' *TinyMUD* and the *Habitat* system by Farmer and Morningstar [Morningstar91] were good examples of purely social MUDs. Habitat was distinguished by a 2D graphical interface, and TinyMUD was the first system to give users extensive abilities to build new places and objects in the world. Stephen White developed the TinyMUCK and the MOO systems, extensions to TinyMUD that allowed users to write scripts controlling objects. The MOO was developed further by researchers at Xerox PARC [Curtis92]. In the meantime, combat/adventure MUDs have also evolved, and servers like the LPMUD have essentially the same technical capabilities as the most advanced social MUDs.

An object-oriented MUD, like White and Curtis' MOO, is a network database server which stores objects having properties and methods. The topology of the space is defined by "room" objects, representing discrete locations, interconnected by portal objects. Each room has descriptive text which users read to situate themselves in the location. Portals with names like "north", "climb", "trapdoor", connect one location to another

and may print text to embellish the user's experience of movement and/or announce someone's entrance or exit to others. MUDs are *non-Cartesian*, meaning they are not limited by any geometric constraint on the spatial arrangement of rooms. For example, a portal named "sleep" could connect a bedroom to a collection of dream-world locations.

Objects in a MOO can also represent things located in a room, and objects called "players" or "avatars" represent the user's character in the world. Users in the same room are able to talk by typing text and reading the text that others type. Each MUD room is superficially similar to an Internet chat room or IRC channel, but the description of structure and actions repeatedly suggest to the user that they are the avatar, acting in a virtual space.

Our most fundamental departure from MUDs is the support of a graphical view of the virtual world. The medium of text is certainly not inferior to graphics, and in fact many types of MUD experiences would be difficult to reproduce visually. However, the textual descriptions of MUDs limit the speed with which a large amount of interesting information about the world's structure can be conveyed. A graphical world is simply a different user experience, in the same way that a movie is a different way of seeing a story than reading a book. We believe it will be a more accessible experience, and we want to explore the possibilities of this new medium.

Our basic requirements posed several technical challenges:

- A distributed architecture needs to be supported.
- Objects need to persist over time.
- End users should be able to easily extend the system.
- End users should be able to make changes to the system while the system in running.
- Finally, since end users are modifying the system, security is of great concern.

These are explored in further detail in the sections that follow.

2. Ease of Development

V-Worlds is a platform for developing shared virtual environments. It is intended that content developers create specific environments with their own artwork and programmed behavior. It is our objective that, ultimately, V-Worlds allow even end-users to be able to create interesting content. Some of V-Worlds' design is influenced by ideas from text MUDs – especially from LambdaMOO. LambdaMOO is notable for its features that allow end-users to create Artifacts, Rooms and other objects with programmed behavior.

Programming behavior in V-Worlds is accomplished by defining methods on objects in the environment. These methods can respond to activity in the environments (e.g. users talking or moving) and can be exposed through the user-interface (through context menus).

In order to facilitate the development of new types of objects, V-Worlds implements object *inheritance*. A V-Worlds object has a property that references its *exemplar*. The object's exemplar is similar (but not identical) to the *class* of a C++ or Smalltalk object. When V-Worlds accesses a property or a method of an object, it first looks in the object itself for that property or method. If it does not find it there, it then looks in the object's exemplar. The search continues up the exemplar hierarchy until the property or method is found or the top of the hierarchy is reached (in which case an error results). This mechanism differs from C++'s in several ways:

- The search is done by *name*, at run-time (i.e. *late-bound*)
- An object instance can have methods and properties attached to it (beyond those introduced by its exemplar)
- An object's exemplar is, itself, another object instance
- An object's exemplar can be changed at run-time
- V-Worlds does not support multiple inheritance

Inheritance facilitates development because it allows content authors to create new objects by specializing existing ones. Having created a new object, the author can allow others to further specialize by declaring his object an exemplar and allowing others to instantiate it or create additional exemplars that inherit from it.

V-Worlds does not support multiple inheritance, mostly, to keep the programming model simple. Supporting multiple inheritance requires that users be prepared to handle unintentional name collisions and classes encountered multiple times through different base classes (the C++ "virtual base class" problem).

3. Basic Object Model

Similar to MOO's, a few basic objects are provided, such as "Rooms", "Avatars", "Portals", and "Artifacts". These in fact, are all based on the single generic object "Thing". Users of the system can add properties and methods to instances of the objects, or change the inheritance chain dynamically.

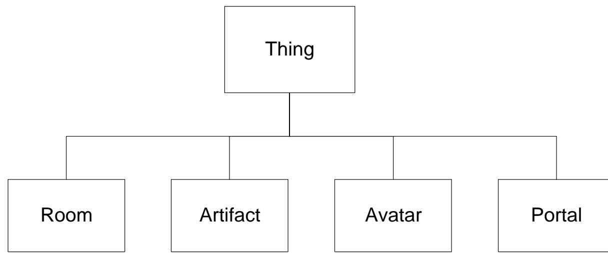

Figure 1. The core exemplars

The Thing exemplar is the parent of all objects and defines properties and methods shared by all. These include properties such as the name, a reference to the exemplar parent object, a reference to the owner (an avatar object), a text description, and the geometrical model associated with the object. It also defines a container object and a contents list of objects, defining a containment relationship that is used for a variety of purposes—the contents of an avatar is its inventory of carried objects, the artifacts and avatars in a room are contained in its contents list. This is an example of logical structure that makes the world more accessible to scripting. Thing also defines methods like MoveInto, which changes the container the object is located in.

Artifact is not much different from Thing, but its version of the MoveInto method allows users to pick up and drop objects, while objects, in general, can only be moved by their owners. There are numerous in-world security policies that define a balance between freedom of action and protecting the topological integrity of the database.

Room and Portal define the topology of the world in much the same way as rooms and portals in MUDs do. Rooms have entrances and exits properties that give a list of portals leading to or from that location. Rooms can be locked or a list of friends can be specified, allowing users to own private personal space in a world. Portals have source and destination properties referring to the rooms they connect. Each room represents a discrete 3D or 2D place with interior and exterior geometry (e.g., vehicles are subclasses of rooms which may have different graphical presentations to users inside them and users outside) and collision-detection structures, to be discussed below. Portals may also have some scripts and data to present a graphical transition to users, and to others around them, when they leave a room. Our virtual worlds are made up of these discrete rooms, within which a continuous 2D or 3D region is defined. Users move their avatars continuously in a room or make discrete transitions to other rooms.

Avatar has a variety of properties and methods to specify the object representing the user in the world. These include properties such as gender (of the avatar, not necessarily of the user), list of friends, list of users being muted, its home room, optional user information, log-in password, etc. Avatar methods include a Tell function that allows strings of text to be transmitted to the user's client, and an IsConnected property, allowing scripts to determine if the avatar is actively attached to a logged-in user.

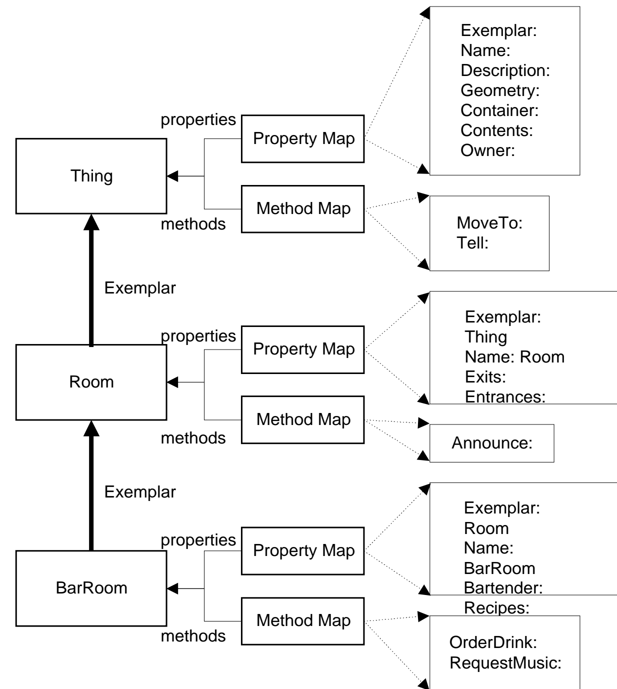

Figure 2. Property and Method Inheritance

Figure 2 illustrates the mechanism for dispatching a reference to an object's properties and methods. It is a straightforward implementation of dynamic inheritance.

In addition to providing inheritance, V-Worlds also provides an *event* mechanism that facilitates writing methods that respond to actions in the environment. V-Worlds objects support a method called *FireEvent*. This method is passed an event *name* and results in a prescribed sequence of method invocations. When `Bob.FireEvent("Foo")` is called, the following methods are invoked:

- Each of the objects in Bob's contents has its OnContainerFoo method called
- Bob's container object has its OnContentFoo method called
- Bob's OnFoo method is called

- Each of the other objects in Bob's container has its OnPeerFoo method called

Events are fired for all key V-Worlds activities: connecting and disconnecting, talking, moving, entering and exiting Rooms, crossing collision-detection boundaries, etc. This event routing mechanism allows objects to sense key activities in the environment and to respond to them. It allows 'bots, agents and other interesting artifacts to be implemented without having to make other objects in the world aware of them.

4. Distributed Architecture

V-Worlds is a multi-user multimedia system. Users can "enter" a world and interact with other users in the world. To facilitate the coordination of activity and the implementation of persistent world state, we chose a client/server architecture for V-Worlds. Although this paper will not detail the advantages and disadvantages of client/server versus peer-to-peer architectures, we note that while early work in this area used a peer-to-peer approach, most recent work has used a client/server architecture.

V-Worlds does *not* use DCOM (or RPC) for its client-server communications. There are two reasons for this.

First, DCOM is *one-to-one* oriented whereas V-Worlds needs a *one-to-many* communication solution. In the usual DCOM scenario, a client-side object invokes a server-side object by calling a client-side proxy object. The call is automatically remoted and performed on the server. In V-Worlds, however, the scenario is a bit different. Each client keeps a locally cached copy of the objects that it needs to render the virtual environment and to handle user interface operations. Changes made to the "master copy" at the server have to be reflected to all the clients that have local copies of the object. While DCOM custom marshaling provides the mechanism needed to have smarter locally cached copies of server objects (i.e. smarter local proxy objects), it provides no simple mechanism for updating all of the copies of a server object. It might be possible to have each server object keep a list of the client-side proxies and iterate through them whenever it needs to update the clients, but this would cause $O(n^2)$ object growth (every new client would require *n* additional client proxy objects to be maintained at the server. In contrast, the V-Worlds mechanism keeps a single *connection* object for each client and iterates through all these objects when it needs to inform clients of a server-side change.

Second, DCOM's benefits are best realized when using early-bound (compile-time), static interfaces whereas V-Worlds needs a very dynamic object model (one that supports the ability to add methods and properties at run-time). It would be unacceptable to have to stop V-Worlds, update IDL files, and rebuild the system every time new functionality needed to be added to V-Worlds. V-Worlds needs a mechanism that allows for late-bound remote procedure calls.

V-Worlds support for client-server programming is inherently built into its object model:

- Client-side V-Worlds objects "know" that they are proxies of server objects
- Client-side changes to object properties are automatically propagated to the server and to other clients
- Server-side changes to object properties are automatically propagated to clients
- V-Worlds object methods can be marked as "client-side" or "server-side"
- Client-side invocations of server-side methods are automatically remoted to the server
- Server-side invocations of client-side methods are automatically remoted to clients

From the V-Worlds user's perspective ("user" here referring to a content developer using the V-Worlds SDK) the client-server communication is invisible. Once the client has been connected to the server, modifications to properties are automatically replicated (to the server and other clients) and methods automatically run on the designated machine. The only awareness that is required of the user is that remoted methods are executed asynchronously (there is a way to perform synchronous client-to-server communications, but it requires explicit coding).

Because client-server communications are handled automatically, it's important that unintended and unnecessary communications be avoided. V-Worlds provides several mechanisms for this purpose. Properties, for example, can be marked as *local* indicating that changes to them should not be automatically propagated.

The most important mechanism that V-Worlds provides for limiting communication needs is its *bystander* algorithm. This algorithm determines what information needs to be provided to clients and only updates this information when necessary. The bystander algorithm relies on a hierarchy of *containment* of V-Worlds objects.

All V-Worlds objects have a *container* property that references the object that contains it and a *contents* property (a list of all the objects that it contains). This maps well to the "physical" nature of V-Worlds objects. V-Worlds has objects for Avatars (people), Rooms (a section of a shared environment) and Artifacts (miscel-

laneous things, for example, Portals to other Rooms). Room objects may contain Avatar objects that, in turn, may contain Artifact objects (the "inventory" of objects being carried by the Avatar). Artifacts can also be contained in Rooms or in other Artifacts (for example, an object within a box object). Room objects can also contain Room objects.

5. Bystander Updating

The V-Worlds bystander algorithm assumes that a client will have a typical working set of locally cached objects. These objects are:

- The user's Avatar
- The Artifact objects contained in the user's Avatar
- The Room that contains the user's Avatar
- The other objects in that Room (Avatars, Artifacts, Rooms, etc.)

The client-side object cache is established when the user (or, more precisely, the user's Avatar) "enters a Room". At that time, the client releases any old objects in its cache and receives its new working set of objects from the server.

Note that some objects are explicitly excluded from the working set (for example, the contents of the other Avatars in the Room). V-Worlds content authors have to be aware of what objects are present in the client machine and have to avoid client-side access to those objects (namely, they have to avoid client-side methods that access objects which are not available on the client).

Note also that since the objects in the cache are determined by the user's location (the Room containing the user's Avatar) that the user can only manipulate the objects associated with his location. An Avatar cannot "be in two places at once" and, thus, the user must move his Avatar into the appropriate Room before manipulating the objects inside it.

The benefit of the bystander algorithm is that it simplifies the logic that governs the updating of client caches. Given the knowledge of what objects a client has cached, the server can determine what clients need to be informed of changes. If a property is changed on an Avatar, for example, and that Avatar was in the "Game Lobby" Room, then the server must inform all of the clients associated with the Avatar objects in the Game Lobby. Clients whose associated Avatars are in other Rooms are not informed of the change. If a property is changed on an Artifact, the server informs all of the clients associated with Avatars in the container of the Artifact. Note that if the Artifact is contained by an Avatar (in other words, the Artifact is in an Avatar's inventory), then only the containing Avatar needs to be informed of the change. If the Artifact is contained by a Room, however, then the change needs to be communicated to all of the clients associated with the Avatars in that Room. In general, when a property is changed, the server determines "who the bystanders" are and informs only those machines of the change.

There are occasions when it is useful to cache additional objects on the client. For example, it is desirable to have a "closed" box Artifact in a Room that can be "opened" revealing its contents. The standard working set, however, would contain the box object, but not its contents. To support this and other scenarios, V-Worlds provides several ways of including additional objects in the client-side cache.

First, objects can be marked (with a property) as *closed* or *opened*. When a container is opened, the V-Worlds server will send local copies of its contents to all of the bystander clients. In the case of the box example, the box object is originally marked as closed and clients do not cache its contents. When the box is opened, however, the server automatically marshals its contents to all of the bystander clients so that the clients can render the contents of the box and allow users to interact with them.

Second, V-Worlds allows objects to be marked as *noticeable*. A noticeable object is "visible" even if its immediate container is closed. In the box example of the previous paragraph, if all of the box's contents were marked noticeable, then it would not have been necessary to use the opened/closed mechanism as the contents would be present in all clients even if the box remained closed.

Finally, V-Worlds provides an explicit mechanism for registering explicit *interest* in an object. If a client registers interest in an object, it is informed of changes to that object regardless of the containment hierarchy and other mechanisms. This registration technique is discouraged as it requires clients to keep track of registered objects and to remember to deregister interest when the object is no longer needed.

6. Persistence

The ability to create and change objects is a fundamental advantage of V-Worlds over graphical chat products. While both V-Worlds and chat products provide shared virtual environments, chat products do not typically provide a way to change the environment in a persistent fashion (other than, perhaps, changing Avatar characteristics). Mostly, this is because chat products are usually implemented atop simple messaging server software –

for example, IRC servers. These servers provide a mechanism for the real-time dissemination of data, but no mechanism for long-term storage of world state. V-Worlds provides its own server software that allows for persistent, changeable, world state.

V-Worlds implements persistence by allowing entire objects to be serialized and by automatically *logging* changes to object properties.

Storing the state of an entire object is relatively straightforward – V-Worlds stores the values of its properties and a record of what methods it has.

V-Worlds automatically logs changes to object properties. When a property value is changed on the server, the server automatically records the change in a log file. This file is a simple sequential file. To restore the state of an environment, V-Worlds reads this log, reapplying the property changes. If the server crashes, only the unwritten change records are lost (although expensive, the server can be told to immediately write changes out to the log file in order to provide the maximum robustness).

To avoid unnecessary logging, V-Worlds allows properties to be marked as *volatile* indicating that changes to them not be logged.

To avoid large log files, V-Worlds can write out its entire state to a new log file (by writing out complete objects) and then the old file can be deleted (or archived).

7. Run-time Editing

Another aspect of MUDs that we have adopted in V-Worlds is the ability to perform *live* editing of content. V-Worlds allows objects to be created and modified while those objects (and others in the same environment) are in use (on the server and connected clients).

Most Web content cannot be edited in such a manner. Web pages, for example, are usually authored off-line and then posted on public servers during times when users are not likely to be accessing them (in order to avoid missing pages or incorrect links during the posting process).

The live-editing capability of V-Worlds includes more than just the ability to create object instances and to modify their properties. V-Worlds allows methods and properties to be added and deleted from objects and object exemplars to be changed. As with property changes and method invocations, V-Worlds will propagate these changes to all the clients affected by the changes. (Note: in practice, these types of changes are usually made to exemplar objects and exemplar objects are typically cached by *all* client machines. Thus, changes to object structure are usually replicated in all connected clients.) In addition to replicating these changes, V-Worlds will also persist them by writing out the necessary log records.

The replication and logging of these changes occurs automatically as an object's structure is maintained in its properties. An object's exemplar is referenced by an object-valued property. An object's methods are kept in a "map-" (dictionary-) valued property. An object's properties are kept in a map-valued property. Thus, changes to an object's structure are really modifications to an object's properties. As V-Worlds automatically replicates and persists any property changes, this mechanism also replicates and persists changes in object structure.

The ability to change an object's structure at run-time is very valuable. First, it allows changes to be made to an environment without having to shut down access to it. Second, it allows a system to be extended by content providers and, ultimately, end-users without having to teach them about IDL files and recompiling a complicated system. Together, these features facilitate long-term operation, maintenance and enhancement of virtual environments by less-skilled content developers.

8. Security

End-user object creation (including the ability to author methods), clearly raises security issues. The typical scenario that we want to enable is the one that raises the most concern. We want to allow end-users to create interesting objects that can be used by others. In such a scenario, it is imperative that a security mechanism be provided to avoid "Trojan Horses" (objects that look good, but do bad things).

The V-Worlds security mechanism is similar to that used by LambdaMOO. Its basic tenet is that code should only be able to modify objects owned by the user that wrote the code. Implicit in this tenet are three requirements:

- That all methods and objects be associated with an *owner*
- That the system be able to, internally, impersonate a user for the purposes of security testing
- That all method and property access be validated

All V-Worlds objects have an *owner* property that references the Avatar object that created them. Because the user has an Avatar associated with him/her, this Avatar is used to establish a current security *context*. On the client, the security context is always associated with the user's Avatar object. On the server, each communication (socket) connection is associated with

the Avatar of the user that established the connection (by logging in to the virtual environment). When the server processes a client-side message (e.g. a remoted method invocation or property change), the connection on which the message arrives establishes the security context. Thus, on the client, all user-interface operations are treated as being initiated by the user's Avatar. On the server, all operations are treated as being initiated by the Avatar associated with the communications connection that received the message requesting the operation.

To prevent Trojan Horse objects, there is an additional, internal mechanism somewhat like the UNIX *setuid* mechanism that is used to explicitly establish a current security context. Imagine that the user clicks on an object (say a large horse) in the room and, through the event mechanism tries to invoke the Horse.OnLeftClick method. First, the system assures that the current security context (the Avatar associated with the user) has the right to access the method – assume that it does (typically, methods can be accessed by anyone). Thus, the system continues with the invocation of Horse.OnLeftClick. At this point, V-Worlds calls its internal setuid mechanism to set the security context to the Avatar that created the Horse.OnLeftClick method – assume that this was not the user's Avatar. At this point, the system is now impersonating the Avatar that authored the method. If Horse.OnLeftClick tries to damage the user (say by trying to change the User.Description property) it will fail. V-Worlds will intercept the attempt to write to User.Description property and will determine that the current security context cannot modify that property.

The V-Worlds security algorithm works by intercepting all method and property accesses and assuring that the current security context has the necessary rights to perform the access. In general, only the owner of an object can modify its properties.

There are some very subtle details here that won't be discussed in depth. One instance is illustrated by the following problem: it should be possible to create an exemplar that stores data in instances of objects owned by Avatars other than its creator. In particular, V-Worlds contains a generic state machine object exemplar. This exemplar is owned by the "root" Avatar. A user, however, needs to be able to create an instance of this exemplar (thus, owned by the user's Avatar) and have that instance store state machine information (i.e. its current node and its transition information). This would violate the rules described above, as the state machine exemplar would be unable to update the current node property.

To enable this scenario, V-Worlds allows an exemplar to access (read and write) any property that *it created*, even if that property is attached to an object with a different owner.

Because security is a difficult topic that may require some experimentation to get right, we have centralized our security policy in a single function that is called on method and property access. This allows us to easily try different policies if the existing one proves inadequate.

9. Implementation Details

Providing a comprehensive description of how V-Worlds works would exceed the objectives of this paper. There are a few implementation aspects, however, that are worth noting.

As mentioned at the top of this paper, V-Worlds is implemented on top of COM. At the heart of V-Worlds is the *IThing* interface. All V-Worlds objects (Avatars, Rooms, Artifacts, etc.), from the COM perspective, are instances of *IThing.* The *IThing* interface provides much of the key functionality of V-Worlds:

- The ability to add and delete methods and properties to an object at run-time
- The ability to access methods and properties, taking object inheritance into account
- Object-level persistence (serializing a whole object)
- The low-level properties required of all objects (*exemplar*, *owner,* etc.)
- Easy access via OLE Automation

From the C++ perspective, *IThing* is straightforward, though awkward. Methods and properties are added by calling *AddProperty* and *AddMethod.* Properties are read and written to by calling *get_Property* and *put_Property*. Methods are invoked by calling *InvokeMethod*. Note that access to properties and methods is through helper functions. In the case of methods, invoking them is further complicated by the requirement that arguments be packed into an OLE DISPPARAMS structure.

Although awkward, these helper functions are key to providing inheritance and the ability to dynamically modify objects at run-time. Rather than binding statically (during compilation), accessing properties and methods through these helper functions allows V-Worlds to perform late binding. The helper functions also enforce V-Worlds security policies and automatically perform any remoting (e.g. replicating property changes or invoking remote methods). On the server, the *put_Property* helper function is responsible for logging property changes.

From scripting languages (and anything else that uses OLE Automation), access to V-Worlds objects is much more straightforward. V-Worlds *IThing* objects implement *IDispatch* by consulting the dynamically added properties and methods in addition to the static OLE TypeLib information. Essentially, the implementation of IDispatch turns an *x.y* reference into an `x.get_Property("y")`, `x.put_Property("y")` or `x.InvokeMethod("y")` helper function call. Thus, the content developer can more naturally access added methods and properties:

```
' In VBScript

' add a new property and initial-
   ize it
foo.AddProperty "Age", 12

' access the property
DogYears = foo.Age * 7
foo.Age = DogYears

' add a new method
bServerSide = True
set method = world.CreateMethod(…,
   bServerSide, …)
foo.AddMethod "newmethod", method

' call it
foo.newmethod 7, "Bob"
```

10. Status

The V-Worlds system has been implemented and will be released to a limited developer beta-test in the second quarter of 1998. We're currently working with about a half dozen third party developers and universities to build different worlds on top of the basic platform described in this paper. Performance improvements and optimizations are still in progress. To date, we have built 4 test worlds on top of the V-Worlds platform and the system has been tested with 20 simultaneous users, and 150 simulated users.

11. Summary

The Virtual Worlds Group has implemented a platform that facilitates the development of shared virtual environments. The platform provides features that handle client/server computing, persistent state management, security and ease of development. These features are built on top of standard COM functionality.

12. References

[Brooks86] Brooks, F.P. Walkthrough – A Dynamic Graphics System for Simulating Virtual Buildings – Proceedings of 1986 Workshop on Interactive 3D Computer Graphics (Chapel Hill, North Carolina). *Computer Graphics*. pp. 9-21. 1986.

[Brooks88] Brooks, F.P. Grasping Reality Through Illusion: Interactive Graphics Serving Science. Proceedings of SIGCHI - 1988. pp 1-11. 1988.

[Curtis92] Curtis, P. Mudding: Social Phenomena in Text-Based Virtual Realities, *Intertek* Vol 3.3 1992

[DOOM93, QUAKE97] The *DOOM and Quake* PC computer games, produced by Id Software, Mesquite TX, 1993, 1997.

[FurryMUCK] http://www.furry.com/

[LambdaMOO] lambda.parc.xerox.com:8888

[Mitchell94] Mitchell, A.R., S. Rosen, W. Bricken, R. Martinez, B. Laurel, Panel on Determinants of Immersivity in Virtual Reality: Graphics vs. Action. *Computer Graphics*, p. 496. 1994.

[Morningstar91] Morningstar, C., Farmer, R., "The Lessons of Lucasfilm's Habitat" in *Cyberspace: First Steps*, Michael Benedikt (ed), pp 273-302, MIT Press, 1991.

[Reid94] Reid, E. Cultural Formations in Text-Based Virtual Realities. Masters Thesis. English Department. University of Melbourne. 1994.

[Sutherland65]. Sutherland. I.E. The Ultimate Display. Information Processing. *Proc. IFIP Congress 65*. 506-508. 1965.

[Sutherland68] Sutherland I.E. A Head Mounted Three Dimensional Display. *In Fall Joint Computer Conference, AFIPS Conference Proceedings 33*. 757-764. 1968.

[Thorpe87] Thorpe, Jack. The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting. *Proceedings of the Ninth Interservice Industry Training Systems Conference*. 1987.

[Zyda92]. Zyda, Michael J. D.R. Pratt, JG Monahan, K.P. Wilson, NPSNET: Constructing a 3D Virtual World. Proceedings of 1992 Symposium on Interactive 3D Graphics. *Computer Graphics*. 1992.