

Distributed Interoperable Virtual Environments

Michael Capps David Stotts
The University of North Carolina at Chapel Hill
Chapel Hill, North Carolina, USA

Jim Duff Jim Purtilo
The University of Maryland
College Park, Maryland, USA

Abstract

This paper exhibits the use of existing software bus technology in interconnecting Virtual-Reality Environment (VE) software. Interoperability and application construction from heterogeneous modules are well-explored topics of distributed systems. A joint project using the Polyolith software bus from the University of Maryland and VE software from the UNC graphics lab has shown the utility of composing existing applications as opposed to making extensive individual modifications. This paper claims only a unique application of these methods to a new client area. Multi-user VE walkthroughs (software navigators) are an exciting new area in graphics software but we see that with the rapid development of graphics technology, next-generation applications (including multi-user systems) are commonly redesigned from the ground up. Here we see an excellent opportunity to examine module reusability, with proven software, in a new application area. As well, we hope our experiments will likely lead to conclusions about VE programming abstractions and produce development methods for making easily interoperable next-generation VE applications.

Interconnecting VEs

A number of groups have been studying distributed Virtual-Reality Environments, and have built prototype environments and tools to help build those systems [1,2,3,4]. Each of these systems depends on a homogeneous data and control environment to ease the VE development process. While they all provide a distributed system, they are each still stand-alone in the sense that they operate in a particular language and control domain. In the long term, it is unlikely that the VE community will settle on one particular language for control structures. Thus a system such as those presented here will be necessary, should a software engineer decide to reuse VEs built using different control paradigms.

Development costs could be reduced by using software engineering techniques to build reusable virtual components. Each virtual component may operate as a stand-alone virtual reality, but when interconnected with other components, it becomes part of a larger virtual-reality environment (VE). Thus each component of the VE will need to have features that allow interconnection, while still maintaining hardware-specific software that gives high performance.

This paper presents prototype implementations of control abstractions using the Polyolith software bus, and demonstrates its use in the development of shared virtual-reality environments. As a result of this work, VE software engineers will have a tool that facilitates reuse of virtual components, and further, they will be able to interconnect components in novel and creative ways that heretofore would have required a completely new system.

Turning a VR walkthrough (a VR navigating program) into a virtual component of a larger system can make a simple design considerably more complicated. Now the virtual component must listen not only to its own events, but to events coming from other components. Furthermore, now it must generate events for other components, whose data requirements and event control systems may be quite different from its own. We conclude that any system to handle interconnection of virtual components will also have to be built using the event based paradigm. Note however, that although virtual components will continue to be event based, they will not standardize on one event control system. This is due to the multiplicity of hardware designs discussed earlier.

Based on the need to handle multiple event types, and maintain processing speed, we can conclude that there will always be a need for a software entity to provide for control transformations. That is, there must be a module somewhere in the design that is capable of listening to a variety of event classes, and handling them both simultaneously and asynchronously. Thus we chose to build an event-listener as a central intermediary, capable of doing those transformations, and remaining highly available to all the components in the virtual-reality environment.

Scenarios

Consider an architectural VE built for visualizing new ship designs; it allows an individual to construct a 3-D model of a new ship and to "walk through" the ship, experiencing the design from a ship-dweller's perspective. Consider also a second VE built to simulate a group conference; it allows a collection of individuals physically remote from one another to experience a face-to-face meeting in a virtual 3-D environment. Could we not combine these two systems (without the expense and time required to develop a separate third system) so they interoperate, allowing a group of physically separated people to virtually walk through a ship "together," communicating as a group?

This scenario raises several open technical issues. Instead of traditional (simple) data types and data structures needing to be exchanged, we have data *and behavior* from each system that must be understood by the other. For example, in a ship walk-through system, we not only have data that will define where walls are (physical coordinates in 3-space), but we have other information that will tell whether the walls are solid (realistic) or permeable (a feature of a VE that surpasses reality). In a group conferencing system, we not only have data telling what participants look like, and who is currently speaking, but we also have protocols that define the allowed interactions among participants (like Robert's Rules of Order). When combined, we have interesting interactions among these behavioral constraints. For example, in a group walk-through VE, an interaction between two group members that is permissible in the normal conferencing environment might be inadmissible "inside" the ship VE if a "solid" wall stands between the two group members.

We are studying the interoperation among virtual environments in order to discover essential principles governing their construction and effective use. Our approach focuses upon the control properties of interfaces between VR applications: existing VR applications are being examined in order to expose commonalities, and our abstractions of VR control behavior are being specified in terms of the software bus model of interconnection [5]. When our investigations are complete, we will not only have produced a framework for interconnecting existing VEs, but we will also have established guidelines to assist future VE designers to build systems that are interoperable.

In order to illustrate the issues driving our research, consider the problems of interconnecting two VR applications. For example, one useful combination would be a VR-based flight simulator and a VR-based radar operator system used for training. What are the problems to be overcome in integrating the two, so that pilot actions will be reflected upon the radar operator's "screen", and that the operator's analysis and directions can be used to guide the pilot?

Data relations: One program may have its data embedded within the code itself, but the other may be driven by a backend database system. The very manner by which important data values themselves are accessed by the two programs may be very different.

Data representation: However data values are accessed, they may have very different representation within the respective computer systems. The data structures important to the pilot system may be a polar coordinate system, geared to helping a graphics display quickly update the image of "external" features (e.g. other aircraft or missiles, and ground features such as an airstrip or radar control area) with correct perspective during flight. Yet the radar trainer system may represent data as

records in a database, indexed by a sequence number for fast retrieval--based upon the operator's actions, different sequences of simulated events may thus be portrayed on his screen. An integration of the two systems would need to resolve such conflicts in data representation.

Control representation: Less understood than issues of data representation are those of control representation. Illustrating this, the pilot system may be designed so that images are presented to the graphic display using data streams (a continuous sequence of primitive data); or the pilot system may have each event communicated to all the necessary software components using a broadcast communication paradigm. In contrast, the radar trainer may consist of a centralized computer system that accesses subsystems using a procedure call paradigm. Interconnecting these two VR applications will require existence of a 'software glue' that can accommodate translation between the two paradigms; and that in turn will require a robust control specification technology for even deciding what the interactions should be at all.

Event mapping: With each of these systems running separately, many "events" that occur are of course simulated by the computer. But in an integrated reality, events from one system must be translated for presentation as stimulus to the other system. Because most VR systems are currently built without an architectural orientation, the mere act of identifying which abstract events could be mapped to another reality can be a difficult task. How the events are named and then redirected will be an important issue to sort out.

Time mapping: The pilot system may be a true real-time system, in that the 'simulation' time and 'real' time are intended to coincide. (Clearly, a pilot in air combat should never see a message on his HUD saying "Targeting subsystem is garbage collecting...please wait.") In contrast, for training purposes the radar operator may operate in a virtual time frame, that is, long idle periods may be speeded up, or busy periods slowed down, for purposes of education or elucidation. Finding an abstract way to characterize these two extremes, and then arbitrate a common time scale between them, will be an important task, and one affecting how we set up our control specification framework.

Synchronization: In the same way that the time scales between multiple VR applications must be mated, so must the interconnection system provide for identification and ordering of important events. The radar operator should not be shown any images

of a bogey in his airspace after the pilot has successfully neutralized the threat.

The remainder of this paper explains three interconnection experiments we have completed using Polyolith and several VEs from the UNC graphics lab. We first present a brief overview of the Polyolith system we are using for building distributed systems.

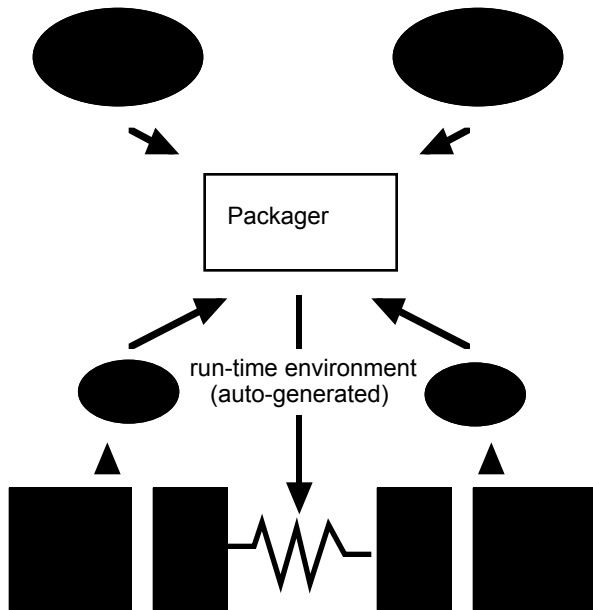


Figure 1: Polyolith Modules

Overview of Polyolith

Heterogeneity is a natural result of the diversity in problems we attempt to solve; the promise of increased performance leads us to specialize the tools we employ. But this diversity challenges us as well: it limits our ability to combine tools with one another. Programs written in different languages cannot be interfaced easily; data produced on one computer architecture may not be represented in a way that is usable by programs on another architecture; and differences in operating systems may prevent programs in a network from cooperating to solve a problem, even though there are physical connections between the hosts. Whenever we are inhibited from leveraging our tools, then we spend our time *re-solving* old problems instead of solving new ones.

The Polyolith project demonstrates that not only can programmers build applications for heterogeneous computing environments, but they can do so without giving up the apparent simplicity of homogeneous environments. Our software organization allows programmers to create applications whose components are written in different languages; distribute their programs across a network of diverse computers having different operating systems; and vary the choice of media or

protocols used for communication among the application components. Even though each of these activities has been addressed separately in the past, our research unifies the capabilities and demonstrates how all can be made available simultaneously.

The Polyolith research results are expressed in terms of both distributed systems and software engineering. Each choice of programming language, architecture and operating system defines a unique software *domain*, e.g., the ANSI-C language on a Vax running 4.3-BSD Unix fixes one particular domain. A *module* is a source program unit drawn from a single domain, e.g., a C function from the above domain could constitute a module. A *heterogeneous program* (or *configuration*) is an application built from modules drawn from different domains.

Differences between domains have been considered in previous research, and mechanisms have been developed to help overcome the barriers imposed by heterogeneity. Many previous systems provided some form of interconnection capability [6,7,8,9]. However, their foci were on the interconnection mechanisms themselves (such as data coercion operations or communication protocols) and not on the software engineering environment in which the mechanisms would be employed to solve large scale problems. There are some fundamental engineering concerns that previous systems do not (and, to be fair, were never intended to) address. Specifically, either the systems support only one form of interconnection, or they force their programmers to choose their interfacing mechanism at the same time they implement each module.

We have devised an interconnection framework that *separates* interface programming from intra-module programming, while still providing access to the mechanisms that make heterogeneous programming possible. This model is the *software bus* organization [5,10,11]. Intuitively, a software bus presents a standard interface into which modules may be "plugged." In the same way that a hardware bus presents a standard for electrical characteristics and signal protocols --- so boards consistent with that standard may be plugged together --- a software bus interconnects software components whose internal properties may remain private as long as their interfaces match the bus standard. It is easier to interface a module to the bus than to all other previously developed modules.

More specifically, a software bus is a communication facility between separately-specified modules. The *abstract bus* is a specification of the services provided by this facility, and the *bus implementation* shows how those specifications are to be realized for a particular set of programming languages, host platforms and communication media. Using a software bus, we can encapsulate decisions concerning the interfacing of modules, rather than distribute those decisions among the application modules themselves.

The bus implementation incorporates existing communication and interconnect mechanisms, providing a comprehensive approach to the construction of

heterogeneous programs where more than one form of diversity is present. A site implementor is responsible for mapping each domain into the abstract bus specification, and for showing how the domain's type model, data representation, and control mechanisms relate to the bus standard. This correspondence would only need to be established once, and thereafter programmers would be free to use modules from that domain within their configurations.

Figure 1 shows the components of the Polyolith bus and their relationship to two processes in a heterogeneous distributed system. The communications interface between the two processes is generated by the bus from libraries of modules specific to different communication methods, hardware architectures, operating systems, etc. The specs files contain information about the abstract interfaces of the modules to be connected. The system design is a specification detailing the network of processes desired.

We think this technology is especially well suited to interconnection of VEs, and that it provides benefits for engineering new VEs from existing ones. It requires development of appropriate specs files that capture the abstractions commonly found in different VEs. It also requires encoding somehow the various behaviors objects exhibit when manipulated in a VE, which will be addressed in a later section.

Xfront to Xfront

The first virtual environment application chosen for interconnection was Xfront, a joystick-navigated 3-d walkthrough with generic hi-resolution mono output and an X- interface. Xfront was chosen for the primary experiment for a host of reasons related to suitability and availability. First and foremost, the code is stable and in full release, and many members of the original programming team were still available at UNC for consultation. Second, it requires limited resources or experience to use, as it does not use the more complicated head-mounted display arrays and is intended to be a simple, easy-to-use walkthrough. Lastly, the code has a very tight event loop (approximately 15 lines) that made understanding the code structure, and modifying it, a simple task.

Our goal for the Xfront modification and interconnection was to examine the capabilities of the Polyolith bus system, so our modifications were chosen with simplicity of implementation in mind. The project decided upon was a shared-walkthrough in which two separate Xfront processes would navigate identical virtual worlds, and each user would be able to see the other user's position as marked by a stick-figure object. Running more than two processes was assumed to be a trivial exercise, excepting only that the Xfront software requires a specific configuration which UNC can only provide twice simultaneously.

This model embodies a simple communication model in which each of the two Xfront processes has a data source and sink connected to the other process. For the sake of

simplicity, an elementary blocking read was used. In Figure 2, we show the orchestration file, which starts the various processes and defines the communication sources and sinks. Each line sends or receives an array of twelve floating-point values; this is a one-dimensional representation of the current transformation matrix for the user's eyepoint in each VE. The matrix is regenerated in the opposite VE and used to transform the object which represents the opposite user in the local world.

```

service "user_h" : {
    implementation : {binary :
    "~capps/polyolith/xf/scr.h
    machine : "hugin.cs.unc.edu"}
    source "publish" : {float(12)}
    sink "news" : {float(12)}
}
service "user_j" : {
    implementation : {binary :
    "~capps/polyolith/xf/scr.j
    machine : "jason.cs.unc.edu"}
    source "publish" : {float(12)}
    sink "news" : {float(12)}
}
orchestrate "ve" : {
    tool "user1" : "user_h"
    tool "user2" : "user_j"
    bind "user1 publish" "user2 news"
    bind "user2 publish" "user1 news"
}

```

Figure 2. Xfront to Xfront orchestration file.

The shared-walkthrough model necessitated only two major changes to the code: the transformation matrix of the user needed to be calculated and sent to the other process, and the other user's eye matrix was needed to move the stick-figure object correctly in the local process. Figure 3 shows the code modifications. These modifications were simply added to the beginning of Xfront's event loop. Note that 'waitstate' is a counter that allows WAITSTATE number of graphics updates between each bus update; this was added because the communications lag is astoundingly larger (approximately four orders of magnitude) than the time required by the specialized UNC graphics hardware.

The Polyolith system helped greatly in abstracting the communications modifications, but unfortunately progress was slowed because the design of Xfront and its libraries was not particularly conducive to interconnection. It was determined that availability of certain functions is likely quite necessary for the smooth interconnection of VE applications such as Xfront. For example, a function for determining the absolute location of the user's eyepoint is needed, as is a method for transforming an object to such an absolute location by replacing its matrices rather than incrementally multiplying. The doubly-connected Xfront project progressed more slowly than expected because it was necessary for us to become intimately familiar with the code and libraries, as well as the graphics theory, to add this functionality. In Figure 3, we see that a function for

retrieving the eyepoint already exists in the libraries, but the matrix operations and object moves had to be added.

```
while ( !done )
{
  XEvent      event;
#ifdef BUS
  waitstate++;
  if (waitstate>WAITSTATE)
  {
    waitstate = 0;
    pg_inquire_view_matrix (eyeinfo);
    COLLAPSE_MATRIX (eyeinfo, I);
    mh_write ("publish", "V12F", NULL, NULL, I);
    mh_read ("news", "V12F", NULL, NULL, I);
    EXPAND_MATRIX (I, eyeinfo);
    OBJECT_MOVE (OBJECT_NAME, eyeinfo);
  }
#endif /* bus */
  /* Loop below is xfront input handling. */
  ...
}
```

Figure 3. Eyepoint retrieval function.

Xfront to Vixen

The second VE program chosen for interconnection was Vixen, an immersive walkthrough application that makes use of a head-mounted display with head and hand tracking devices. Motion is handled by buttons on a 3-d mouse, rather than by joysticks or keyboard input. Regardless of the different equipment for both input and output, Vixen is built upon many of the same libraries as Xfront, so it is similar in function and file-formats and still satisfies our goal to connect heterogeneous VE applications.

The task to create a shared walkthrough with the two disparate programs was much less difficult than the previous effort. The modified version of Xfront required absolutely no modification for this project, which showed the value of using the Polyolith system for inter-process communication. Once a walkthrough has been modified with Polyolith message-handlers, it can be easily plugged through the orchestration files to connect to any other similarly-modified application. Since Vixen and Xfront share file formats and many libraries, the specialized object-moving and position-grabbing functions were able to be re-used.

```
#Object Coordinates
#Object 1
1 0 0 0
0 1 0 0
0 0 1 100

#Object 2
1 0 0 0
0 1 0 0
0 0 1 0
```

Figure 4. Initial world correspondence file.

The major difficulty in sharing a world between different walkthroughs was scaling and relative positioning

of objects in the world at startup. This was corrected by keeping standard object starting locations in a file available to both applications, and using the absolute object-move function to adjust the initial virtual world accordingly (Figure 4). Even with heterogeneous applications, and significant differences between their design, this project took only a fourth the time of the original dual-Xfront connection. A subsequent dual-Vixen version took only the time to change the name of a process in the Bus orchestration file; the interoperability of the communication method shone here.

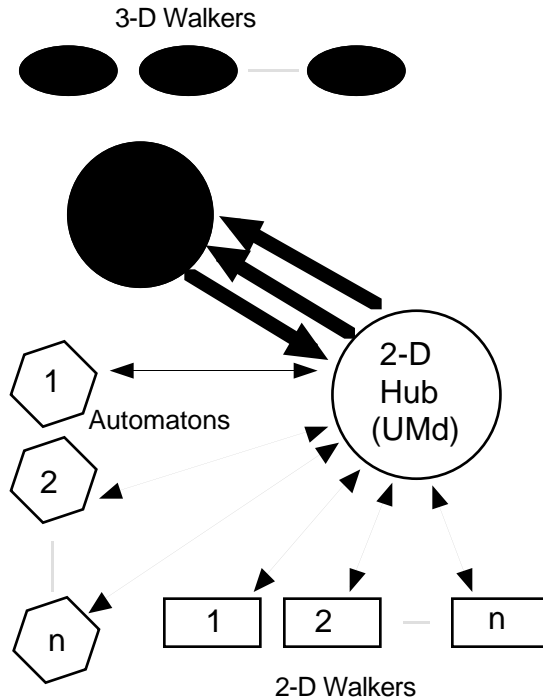


Figure 5. Sample instance of a MEMUD using GVixen.

Multi-environment multi-user dungeons

Our most recent experiment with the Polyolith system and VE walkthroughs was much more complex as it involved multiple walkthroughs and many heterogeneous non-VE processes. The project goal was to implement a Multi-Environment Multi-User Dungeon (MEMUD), in which many users share a dynamically-generated maze populated with robots. Each robot figure can be any of three types: a 2-d Walker, a 3-d Walker, or an Automaton; however, there is no way for any robot to tell the type of another robot.

The bus sends all users a textual description of the maze (Figure 6); blank spaces represent walls, non-blank the halls and rooms, and the 0 is the origin. Other data on textures, lighting, and so forth for the 3d translation is provided as well. The 3-d Walkers are users in the immersive Vixen program; a modified version called GVixen generates a three-dimensional maze complete with

stone textured walls and dirt floors (see Figure 8, at the end of this paper, for a left-eye view). All other robots, no matter the type, are represented as stick-figures moving about the corridors of the maze. The 2-d Walkers are users of an interactive two-dimensional UNIX X display; cursor movements cause a small icon to move about the same maze, which also is rendered from the text description. Other robots are represented as unique icons, but no differentiation is made between the different robot flavors. The Automaton has the same display as 2-d Walkers, but their motions are generated by a random engine.

This project was planned in order to make more use of the distributed capabilities of the Polyolith system, as well as to further cooperation between the Maryland and North Carolina sites. Therefore, the MEMUD system was developed with two main hubs, one at each of the universities. At UNC, the 3dhub process accepts the text-descriptor of the maze and distributes it to the copies of GVixen running locally (currently limited to two by hardware only). It also collects position information for the 3-d Walkers and exchanges that information with 2dhub running at Maryland. The Maryland hub holds the maze description and collects location information from the multiple 2-d Walkers and Automaton. Therefore, all position information is communicated between universities on only one channel, which proved to be the most efficient method. The positions sent are only the map x,y coordinates, and each process translates the map coordinates into 3-d coordinates, X-display pixels, etc., as appropriate.

```
structure_name: gallery_0
block_size: 2
wall_texture:  stone  0.00 0.00  1.00 0.505
floor_texture:  dirt   0.00 0.00  0.25 0.25
ceiling_texture: stone 0.00 0.00  1.00 0.50
fog:           0 0 0  0.01 15.0
```

```
012345678901234567890123
-----
0 | *****0***** |
1 | * ***** |
2 | * **      * ** |
3 | * ***** * ***** |
4 | * ***** ** ***** |
5 | * ***** ** ***** |
6 | * ***** ** * ** |
7 | * ***** ** * ** |
8 | * ***** * * ** |
9 | * ***** * ** |
0 | * ***** ** |
1 | ***** ** ** |
2 | ** ***** ** |
3 | ** * ** * ** |
4 | ** ** ** ***** |
5 | ** ** * ** ***** |
6 | ** ** * ** ***** |
7 | ***** ** ***** |
8 | * ** ** |
9 | ***** |
```

Figure 6. Map file.

Figure 5 demonstrates the configuration of the different processes and shows the communications connections and patterns between them. The maze description propagates directly outward in all directions from process 2dhub. Figure 7 is the orchestration file; notice the similarities to Figure 2; indeed, the communication path between any two modules are the same, though now all communications go through hubs at each site to reduce network costs. Figure 7 shows a model with only a single 3-d Walker and a single 2-d Walker, for sake of brevity.

MEMUD is a truly distributed and scalable application that connects Virtual Environment walkthroughs with two-dimensional graphical interfaces. It demonstrates the flexibility of the Polyolith system, especially in that the GVixen program was ready to use only after the maze-generating modifications were made. The 3dhub translates the GVixen user transformation into maze coordinates, so the generic communication already in Vixen was still usable. The scalability in number of processes is another benefit of Polyolith. Making the hubs scalable was a trivial task, and starting additional instances of identical processes with the bus is exactly what Polyolith was designed to do.

```
# The central hub located at Maryland.
# All 2-d walkers talk through this module.
service "2dhub" : {
  implementation : {binary : "hub" }
  machine : "lens.cs.umd.edu"}
  algebra: {"HUBKIND=2DHUB:NUMWALKERS=1"}
  source "write0": {integer}
  source "write1": {integer}
  source "write_remote": {integer}
  sink "read0": {integer}
  sink "readdata0": {integer}
  sink "read_remote": {integer}
}

# The central hub located at UNC
# All HMD walkers talk through this module
service "hmdhub" : {
  implementation : {binary : "hub"
  machine : "hugin.cs.unc.edu"}
  algebra: {"HUBKIND=HMDHUB:NUMWALKERS=1"}
  source "write0": {integer}
  source "write_remote": {integer}
  sink "read0": {integer}
  sink "readdata0": {integer}
  sink "read_remote": {integer}
}

# An HMD walker
service "hmd" : {
  implementation : {binary : "gv-glab.sh"
  machine : "hugin.cs.unc.edu"}
  source "write" : {integer}
  source "writedata" : {integer}
  sink "read" : {integer}
}
```

```

# A 2-D Walker
service "2dwalker" : {
  implementation : {binary : "2dwalker" }
  machine : "lens.cs.umd.edu"
  source "write" : {integer}
  source "writedata" : {integer}
  sink "read" : {integer}
}

orchestrate "ve" : {
  tool "2dhub"
  tool "hmdhub"
  tool "2dwalker0" : "2dwalker"
  tool "hmd0" : "hmd"
  bind "2dhub write0" "2dwalker0 read"
  bind "2dhub write_remote" "hmdhub
  read_remote"
  bind "hmdhub write0" "hmd0 read"
  bind "hmdhub write_remote" "2dhub
  read_remote"
  bind "hmd0 write" "hmdhub read0"
  bind "hmd0 writedata" "hmdhub readdata0"
  bind "2dwalker0 write" "2dhub read0"
  bind "2dwalker0 writedata" "2dhub readdata0"
}

```

Figure 7. Gvixen orchestration file.

Object attributes and interactions

If I pass you a sphere from my VE into yours, what will happen when you drop it? Will it bounce, like a basketball? Or will it "thud" to the floor, like a bowling ball?

Some of the walkthroughs we have used allow users to move objects within the virtual world; representation and communication of such operations is not difficult, but this begs the same collaboration issues we have seen for years in shared editors and workspace environments. Many of the same issues of concurrent editing might apply in multi-user VE's as they do in 2-d shared workspaces/windows, yet we see that many graphics researchers trying to solve these problems without the benefit of the knowledge of current computer-supported collaboration work. Handling such contention in three-dimensional environments promises a new and interesting application of established collaboration theory.

Allowing users to move simple objects is one issue, but this model breaks down when the object being moved is another user. Certain objects may resist movement, or at least respond to it differently. For instance, slick objects may not stop moving once let go; basketballs may bounce if dropped from a height; donkeys might not ever move on the first tug; and people might push back! Some very simple and static models for specifying object behaviors already exist in three-dimensional graphics toolkits. What is needed is an easily-extended object-oriented model for specifying arbitrary object interactions and behaviors. This is the second phase of our research; it is hoped a collaborative environment with user-extendable objects and

interactions might be useful not only as an interesting application but as a project management tool.

We have begun investigating the feasibility of using MOO technology to solve both of these problems. Object-oriented MUDs, or Multi-User Dungeons, were created to specify objects, their behaviors, and their interactions, and they were designed for arbitrary dynamic programming by users. In addition, the original muds were designed as collaborative spaces, and therefore concurrency controls and other issues can be easily and well defined. Recent progress has shown that we can feasibly attach LamdaMoo (which, along with LPMUD, was one of the first object-oriented MUDs) to the Polyolith software bus. We hope to use the MOO as a database to store all the information desired about objects, including actions that are defined for them, their locations, and their ownership, leaving us free to study communication schemes and behavior specification.

Conclusions

We still have considerable experimentation left to do before we can conclusively recommend specific software engineering methods and principles for constructing interoperable VEs. However, our initial experiments have led us to believe that the Polyolith system is an adequate technological framework on which to build them. Our first interconnection (the Xfront-to-Xfront system) took 6 weeks to complete; in this time, we were designing the abstractions and their Polyolith representations for the entities in the Xfront VE. When we did the second experiment (Xfront-to-Vixen), the interconnection required 1.5 weeks; many of the components designed for Xfront were completely workable for Vixen as well. Finally, the Vixen-to-Vixen interconnection took one afternoon. We conclude that Polyolith allows us to leverage the specifications and abstractions from earlier work in subsequent systems. The GVixen experiment, interconnecting 3-D VEs with traditional 2-D graphics systems, required about 3 weeks of effort. We expect to find the same leverage when we work on the next mixed-mode VE.

The experiments so far have relied on systems sharing a common data format. However, each component system (Xfront, Vixen, robots) was designed and implemented by a different individual, thus giving us components with internal data structure mismatches and control structure mismatches, as well as disparate software libraries in use. Subsequently experiments will deal with VEs using different world model data formats as well.

Bibliography

1. M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. Barham. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. *VRAIS 95*, p. 2-10.
2. Q. Wang, M. Green, and C. Shaw. EM - An Environment Manager For Building Networked Virtual Environments. *VRAIS 95*, p. 11 - 18.
3. G. Singh, L. Serra, W. Png, A. Wong, and H. Ng. BrickNet: Sharing Object Behaviors on the Net. *VRAIS 95*, p. 19 - 25.
4. C. Carlsson and O. Hagsand. DIVE --- A Platform for Multi-User Virtual Environments. *Computers and Graphics*, p. 663 - 669, 1993.
5. J. Purtilo. The Polylith Software Bus. *ACM TOPLAS*, (January 1994).
6. Barbacci, M., D. Doubleday, C. Weinstock and J. Wing. "Developing applications for heterogeneous machine networks: The Durra environment." *Computing Systems*, vol. 2, pp. 7-35.
7. D. Perry. The Inscape Environment. *Proceedings of 11th International Conference on Software Engineering*, (1989), pp. 2-12.
8. R. Snodgrass. *The Interface Description Language: Definition and Use*. Computer Science Press, (1989).
9. J. Wileden, et alia. Specification-level interoperability. *CACM*, vol. 34, (May 1991), pp. 72-87
10. J. Purtilo, D. Reed and D. Grunwald. Environments for prototyping parallel algorithms. *Journal of Parallel and Distributed Computing*, vol. 5, (1988), pp. 421-437.
11. J. Purtilo and P. Jalote. An environment for developing fault tolerant software. *IEEE Transactions on Software Engineering*, vol. 17, (1991), pp. 153-159.

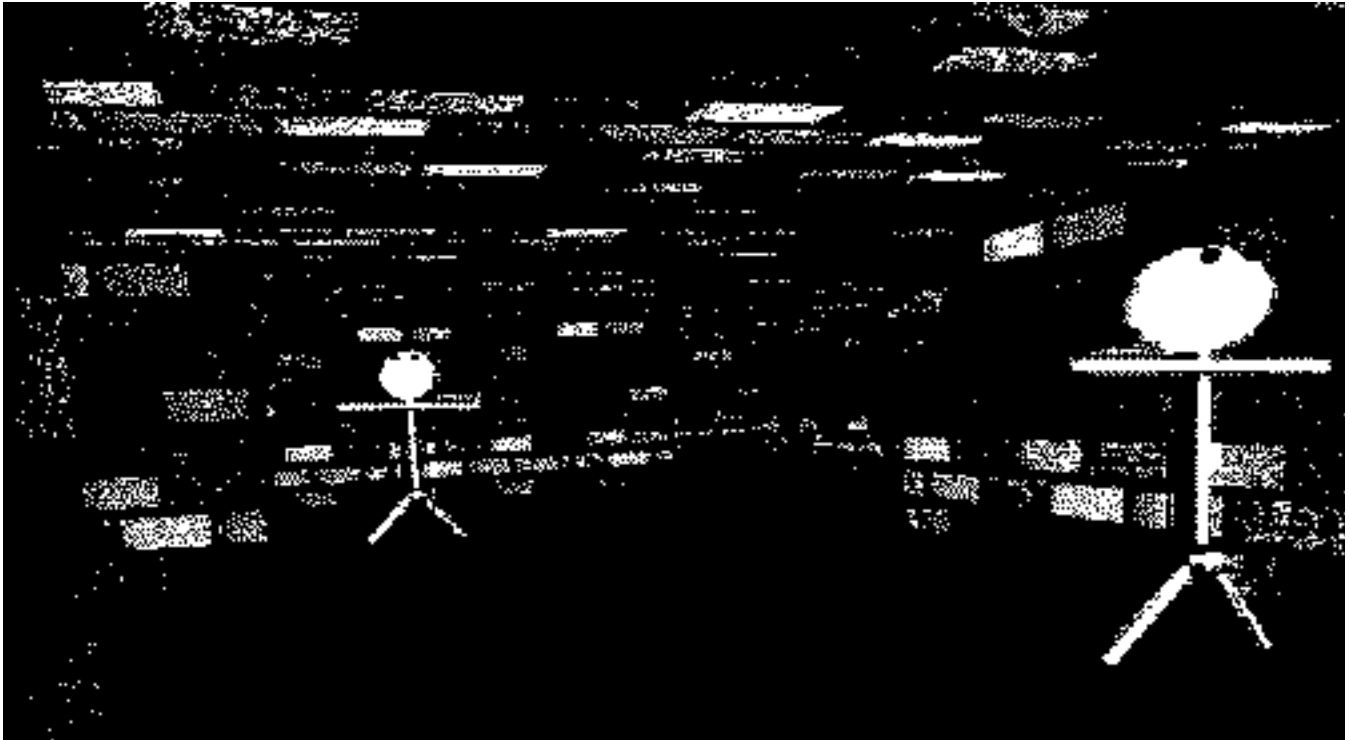


Figure 8. The view of a maze from the Head-Mounted Display, with robots nearby