VRPN: A Device-Independent, Network-Transparent VR Peripheral System

Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, Aron T. Helser
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

The Virtual-Reality Peripheral Network (VRPN) system provides a device-independent and network-transparent interface to virtual-reality peripherals. VRPN's extended methods for factoring devices by function are novel and powerful. VRPN also integrates a wide range of known advanced techniques into a publicly-available system. These techniques benefit both direct VRPN users and those who implement other applications that make use of VR peripherals.

CR Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time systems; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – *Virtual Reality*; I.6.8 [Simulation and Modeling]: Types of Simulation – *Distributed*.

Additional Keywords: interactive graphics, virtual environment, virtual world, input devices.

1. Introduction

VRPN is a set of classes within a library and a set of servers that implement a device-independent, network-transparent interface between application programs and the set of physical devices (trackers, etc.) used in a virtual-reality (VR) system. VRPN provides:

- Access to a variety of VR peripheral devices through a common, extensible interface,
- Network-transparent interface to devices,
- Time stamps for all messages to and from devices
- Clock synchronization between clients and servers on different machines,
- Multiple simultaneous connections to devices,
- Automatic reconnection to failed remote servers, and
- Storage and replay of interactive sessions.

The VRPN application-side library runs on at least PC/Win32, SGI/Irix, PC/Linux, PC/Cygwin, Sparc/Solaris, HP700/Hpux, and PowerPC/AIX. The server-side library is fully functional under SGI/Irix, PC/Win32 and PC/Linux, and functional except for serial-port code on the other systems. There are drivers for:

Trackers: Ascension Flock of birds (single or multiple serial lines), Polhemus Fastrak, Intersense IS-600 and IS-900 (including wands and styli), Origin Systems DynaSight, SensAble Technologies PHANToM, 3rdTech HiBall, Logitech Magellan, and Radamec Serial Position Interface (video/movie camera tracker).

Other devices: Logitech Magellan (analog values and buttons), B&G systems CerealBox (buttons, dials, sliders), NRL ImmersionBox serial driver (buttons), Wanda (analog, buttons), National Instruments A/D cards, Win32 sound server based on the Miles SDK, SGI button and dial boxes, and the UNC hand-held Python controller (buttons).

VRPN is public-domain, open-source software with a user community in academia, industry and the national labs. [1] Twenty-seven off-site users are currently on the project mailing list. New device drivers and bug fixes have come from the University of Illinois at Urbana-Champaign, the National Center for Supercomputing Applications, the University of North Carolina at Chapel Hill, the Naval Research Laboratory, and Brown University. Commercial support for driver development and new features has come from Walt Disney VR Studios and Schlumberger Cambridge Research. All of these improvements have been released in the public domain versions.

VRPN was developed to address the following concerns:

- Laboratories with multiple graphics display stations require access to VR peripherals from a variety of machines. It is often inconvenient to co-locate the machines with the devices, or to run interface cables from each device to each host.
- Some VR devices (especially trackers) perform more reliably when left on continuously, and require lengthy reset procedures when closed and re-opened.
- Different devices may have radically different interfaces, yet perform essentially the same function; some require specialized connections (PC joysticks) or have drivers only for certain operating systems.
- VR applications require minimum latency, and need to know at what time events occur in the system.

These criteria led us to an architecture where input/output devices at each display station are connected to one or more local device servers. These servers communicate with graphics engines through a switched Ethernet.

This paper describes VRPN version 05.04. Device factoring is described in detail, since it is the novel contribution. The other features are mostly drawn from existing systems: their combination in a publicly-available system is the sec-

ond contribution. These features are presented in the following categories: networking, separate client and server, storage and replay, and performance. Implementation details for these features are usually omitted (they are in the publicly-available code).

Related work: Many existing toolkits provide complete distributed virtual world interfaces, concentrating on flexibility and generality. There are both commercial systems (including Division's dVS [2], Sense8's WorldToolKit [3], and Panda3D [4]) and research systems (including the MR toolkit [5], GIVEN++ [6], DIVE [7], BrickNet [8], Alice/DIVER [9], AVIARY [10], Maverik/DEVA [11], VR Juggler [12], Bamboo [13], and Dragon [14]). There is also recent unpublished work by the DIVERSE [15] group.

VRPN does not aim to provide an overall VR API. It focuses on the sub-problem of providing low-latency, robust, and network-transparent access to devices; and specifically to provide a uniform interface to a wide range of devices.

Several users of existing toolkits have integrated VRPN as a device-interface layer. Users at Brown have developed a VRPN server that works with WorldToolkit, the Maverik system is being extended to use VRPN devices, NCSA uses VRPN within several of its CAVE applications, the Naval Research Laboratory has integrated VRPN into Dragon applications, VRPN is being extended to be a dynamically-loadable Bamboo module, the developers of Panda3D are using VRPN to communicate with several VR devices, and the DIVERSE group is developing a VRPN layer.

2. Device Types and Factoring

It has been very fruitful to think of VRPN not as providing drivers for a set of devices, but rather as providing interfaces to a set of functions. Particular devices are of one or more *canonical device types*. Each type specifies a consistent interface and semantics across devices implementing that function. [16] Common device types are listed below. Other device types are provided; new types can be created.

- *Tracker* reports poses (position plus orientation), pose velocities, and/or pose accelerations.
- *Button* reports press and release events for one or more buttons; individual buttons may be set by the client to be toggle or momentary.
- *Analog* reports one or more analog values.
- *Dial* reports incremental rotations.
- *ForceDevice* allows clients to specify surfaces and force fields in 3-space.

Mapping a set of devices into one canonical type requires mapping the different capabilities of each device onto one interface. There is a tension between providing a very simple interface (which does not allow access to particular advanced features) and providing a feature-rich interface (where many devices do not implement many of the features, forcing application code to deal with many cases). VRPN deals with these issues by:

- factoring devices based on their functions,
- mapping devices to connections within VRPN,
- allowing devices to export multiple interfaces,
- silently ignoring unsupported message types,
- providing application-level access to all messages.

Factoring based on function: Often a particular device's special features amount to implementing more than one function (a tracker sensor with a built-in button, for example). VRPN has the driver for the device export interfaces for multiple device types. The server for the PHANToM haptic display illustrates this: it exports Tracker, Button, and ForceDevice interfaces under the same device name. The client deals with a Phantom as if it were three separate devices, one for each of its functions.

Factoring makes it easy to move an application to different sets of input/output devices. No client-side code change is needed to move an application from an Intersense IS-900 tracker with an integrated button and analog controller to a Fastrak sensor attached to a Wanda device, then to a Phantom haptic display. An application built for a Phantom haptic display can be tested on either of the other setups, although no forces will be generated.

Mapping devices to connections: While the Tracker, Button and ForceDevice devices for a Phantom are logically separate, they are all internally mapped to the same data stream for communication efficiency. Internally, VRPN maintains a list of open connections; when a new device or connection request maps to an existing connection, a pointer to the existing connection is returned.

Exporting multiple interfaces: In some cases, the same physical device may behave as different device types at different times. For example, a freely-rotating dial might be used to either specify an orientation (the Dial interface) or to specify a value (the Analog interface). A VRPN driver can export both interfaces for the same device under different names and the client can use either.

A special case of multiple interfaces is the *layered device*. In this case, higher-level behavior is built on top of an existing device. An instance is the *AnalogFly* server, designed for flying with joysticks: the joystick driver reports analog values for each of its axes and the AnalogFly integrates these values into Tracker messages. The AnalogFly is also used with the Radamec Serial Position Interface video-camera-tracking device, integrating its pan and tilt axes into a Tracker report. Clients can connect both to the low-level analog device (to read focus and zoom from the Radamec camera tracker, for example) and to the higher-level tracker device (see figure *Layered*).

Another special case of multiple interfaces is the *multiple-behavior device*. Within DEPARTMENT, there are several groups that use joystick devices to fly the user. Each group has its own mapping of joystick axes to transformations, depending on application requirements and user prefer-

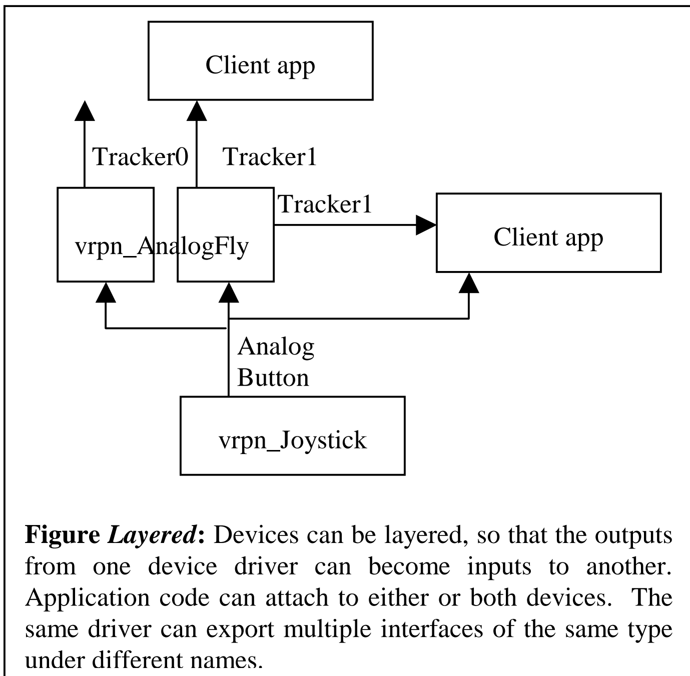

Figure *Layered*: Devices can be layered, so that the outputs from one device driver can become inputs to another. Application code can attach to either or both devices. The same driver can export multiple interfaces of the same type under different names.

ences. Using multiple instances of AnalogFly servers, each with a unique name, joystick servers export all interfaces at once. Each client connects to the interface with the desired mapping (see figure *Layered*).

Silently ignoring unknown messages: Functions applicable to several devices of a type that can be ignored by other devices can be implemented in the base device type. An example is the message that sets the report rate for tracker servers. Servers with variable update rates (like the ceiling tracker and Phantom) adjust their rates to match that requested, while other servers silently ignore these messages. This extends the common interface and enables access to special functions on some devices while not requiring all functions to be implemented on all devices.

Application-level access to messages: Some devices may have idiosyncratic functions captured neither by existing device types nor generally applicable to other devices of the same class (calibration, for example). In this case, the calibration application can directly send and receive new message types, giving access to the extended functions on the device server without requiring changes to the library classes. Such a specialized calibration application could be run at the same time as standard applications. To date, no device has required this sort of idiosyncratic interface.

Example new device: An example is helpful to show how these features work together. Let's plan a driver for Measurand *SHAPE TAPE*. [17] This tape consists of a linear chain of links embedded in flexible tape. Two orientation components of each link relative to the previous one are measured (*twist* and *nose-dive*). The most basic interface to this tape would be as an Analog device, reporting two angular values for each link.

Whereas the Analog interface gives all the necessary information needed to derive information about the tape, it is probably not the most appropriate interface for many applications. The basic function of the tape is to describe a curve in space. Although this might be shoehorned into the Tracker interface, it is more properly a new device type. To define a general class for this, we might think of oriented splines. Thus, a layered interface would be provided, possibly by implementing a general server that reads in analog orientations and exports splines.

In practice, this tape will be used to track the pose of a user's arm relative to the torso. For this purpose, three poses are desired at known distances along the tape (where it attaches to shoulder, elbow and wrist). Thus, another layered interface would be written that takes the spline as input and reports Tracker poses at some number of locations specified along the length of the tape. The client application will attach to the device using its Tracker interface, which reports the poses, while the calibration application will attach using the Analog interface. Either application, or another one entirely, could attach to the spline interface and render the curve of the tape itself.

3. Networking

Connection initialization: The connection initialization code design meets the following requirements:

- rapid start-up when connecting to a running server,
- rapid return to client code during connection set-up and reconnection even when no server is active (can't use TCP connection requests from the client to the server, which can hang indefinitely for ports in certain states),
- no dependence on opening a particular TCP port on the server (a port that has been used by a recently-exited server can remain unavailable for several minutes on some operating systems),
- no dependence on opening particular ports on the client (same reason, plus the fact that multiple clients would require the same port number),
- ability to attempt reconnection without causing long pauses in the client, and
- ability to connect or reconnect to a starting server relatively quickly (to enable restarting a failed server without restarting the client application).

Figure *Connect* shows the algorithm used. Although complicated, it runs only at connection startup – messages flowing during a session are sent directly. Server connections open a well-known UDP port for connection requests from clients. Clients open any available TCP port and send a UDP request for the server to connect on that port. The client then enters a state in which it does zero-time selects on its TCP port to see if the server has called back, returning control to the application immediately if not; it sends another request packet once per second if there is no response from the server (the client devices will not be connected to their server counterparts, see "Client/server object

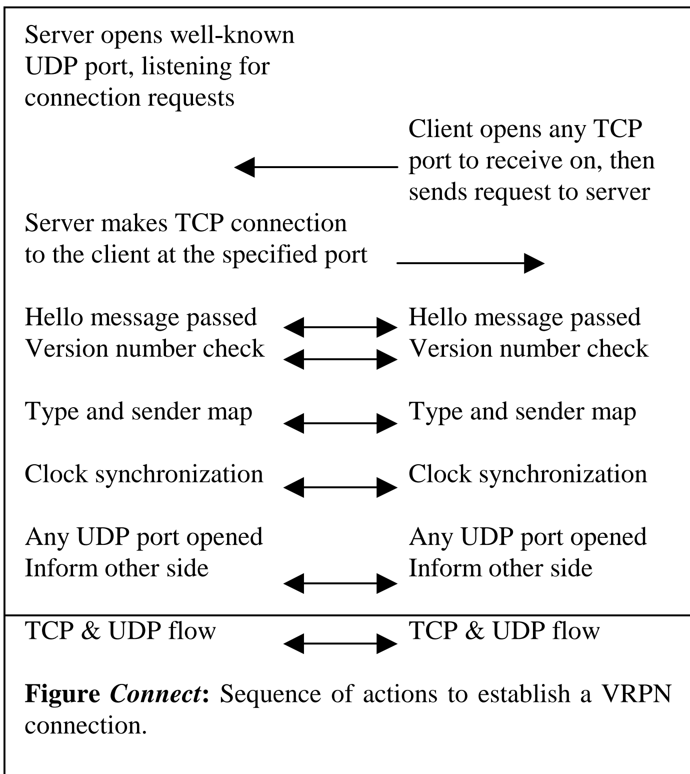

Figure *Connect*: Sequence of actions to establish a VRPN connection.

verification"). When the server receives the connection request, it opens a new communications endpoint, which calls the client back at its specified TCP port.

Type and sender map: VRPN devices and user code register human-readable type names ("Tracker Pos/Quat", "Button Change") and sender names ("Magellan0", "My_tracker") and receive integer tokens that are used to send messages. At connection, client and server exchange mappings from tokens to names; each builds a translation table to convert incoming tokens to local tokens. This enables efficient transmission and translation while providing flexibility.

Reliable vs. fast: Different device types have quite different requirements for message delivery, ranging from button presses (which must not be lost, but have relatively low sensitivity to latency) to tracker reports (which have stringent latency requirements but if one is lost another will be coming soon). Systems willing to devote an extra thread on each host to message delivery have been able to provide a wide range of delivery semantics. [18] VRPN neither requires nor provides a separate thread for delivery, and so only provides two classes of delivery: reliable (via TCP) and unreliable (via UDP).

Multiple connections to a server: Server object messages are sent to all connected clients. Messages received from one client are delivered to all server objects, but not sent out to the other clients. **Client/server object verification:** VRPN client objects emit warnings when not connected to their remote server. Callback handlers trigger initialization code when a connection to an object's counterpart is established or broken. **Starting a remote server:** VRPN enables Unix clients to start a needed remote server when run.

Synchronized time stamps: Each message has a time stamp matching either the time at which the data for the message became available, or the time at which an action should be taken by the receiver. VRPN provides clock synchronization between hosts on either side of a connection.

TCP_NODELAY: VRPN sets TCP_NODELAY on its TCP sockets, causing immediate sending of acknowledgements regardless of whether there is data waiting to be transmitted. This prevents delays on one-way information flows (like Analog data from some servers).

Network standard byte order: The htonl() family of routines provides network-standard byte ordering for all messages; VRPN includes routines to marshal and unmarshal each of its data types into this format.

Warning/error printing: VRPN enables servers to pass warning and error messages to remote clients, so device drivers and servers anywhere in the system can send human-readable warning and error reports. This has proven very useful for debugging distributed system behavior.

4. Separate Client and Server Processes

The ability to run different parts of a VR application in different processes is of widely-recognized importance. [5-7, 19-21] For the case of VR devices, client and server should be run as separate process when:

- they have very different update rates,
- server initialization takes a long time,
- message timing is critical, or
- the server requires frequent access to a device.

Local or same-process client and server: Whereas VRPN is designed to handle network connections between clients and servers, it is possible to run either in separate processes on the same machine, or within the same process. When running within the same process, messages are handed directly to the callback routines without passing through the network.

5. Storage and Replay

VRPN provides a log file mechanism, by which all messages passed over a client/server connection session can be stored to file, and then the session replayed or analyzed. This capability has been used to:

- record user motion during human-factors studies,
- provide an electronic lab notebook recording actions and responses during materials science experiments,
- store interactions between collaborating users to allow comparisons between different sharing strategies, and
- capture a series of user motions and button presses to enable debugging of new interaction techniques without repeatedly donning the VR equipment.

```
#include "vrpn_Tracker.h"

void handle_pos(void *, const vrpn_TRACKERCB t) {
            printf("Pos, sensor %d = %5.3f, %5.3f, %5.3f\n",
                   t.sensor, t.pos[0], t.pos[1], t.pos[2]);
}

main() {
      vrpn_Tracker_Remote *tkr = new vrpn_Tracker_Remote("Tracker0@myhost");
      tkr->register_change_handler(NULL, handle_pos);
      while (1) { tkr->mainloop(); }
}
```

Figure *ExampleClient*

Logging can be done at either the client side or the sender side. When client-side logging is performed and the server crashes and restarts, logging continues.

Log file replay: A client application "connects" to a stored log file and reads from its devices by specifying a file URL as the location of the device, and does not require any extra consideration if the intent is to replay the original session at its normal rate.

6. Performance

One criterion for evaluating VR device libraries and architectures is comparison of their performance with a dedicated, locally-connected device using device-specific drivers. Milliseconds of latency are the critical currency in VR systems; the value of different features is measured against their cost in time. [22] On this scale, VRPN measures up well; for some configurations the time to read a message using VRPN can be significantly *less* than that of a locally-connected device with manufacturer-supplied drivers. To explain how, we describe timing information and latency-reducing optimizations within VRPN. Developers of other libraries or stand-alone applications can use these same techniques.

Overhead added by VRPN: Network latency tests were run between an SGI and a Linux box within a switched Ethernet environment. Ping tests between the machines showed an average one-way time of 0.51ms. Application-level VRPN messages (from the client to the server, then a response message being received by the client callback handler) had average one-way times of 3.3ms. This includes all overhead from the operating system network layers, as well as from VRPN. Slightly lower times have been found from a Linux client to a Windows 98 server, and an average of 1.7ms one way is found from an SGI client to a Windows 98 server.

Three serial port accelerations: While developing the drivers for trackers that communicate over serial ports (Polhemus Fastrak, Ascension Flock of Birds, Origin DynaSight), we discovered three ways to significantly decrease latency: 1) decreasing buffering in the UARTS (3ms), 2) decreasing latency within the operating system by setting the scheduler to run at 1kHz rather than 100Hz (5ms), and 3) providing multiple serial connections to the device (one per sensor for a Flock of Birds) (3ms). Since the VRPN overhead is below the latency reductions provided by these techniques, it can actually be faster to read from a device connected to a remote, well-configured server than a device connected to the local machine.

Optimized, time-aware drivers: The driver that ships with a product is not always optimized for minimum latency, and seldom deals explicitly with time. Some wait a pessimistic amount of time before reading; VRPN drivers continually read the available characters and send a report as soon as available. A report's time is based on when the first character is received, rather than when the whole report has been collected. Since a separate server process usually polls devices at 1kHz, this provides much more accurate timing than is available using a locally-connected device within a 60-Hz polling process.

Notes: This paper does not present the end-to-end timing for trackers and other devices within VRPN, but rather addresses the incremental latency due to VRPN. VRPN messages are sent between hosts using a single UDP or TCP packet in steady state, so network latency should be minimal compared to other networking toolkits.

7. Using VRPN: User/Application Layer

VRPN is optimized to be as easy to use as possible for a client program. An example client program that reads positions from all of the sensors on a tracker is shown in figure *ExampleClient*.

This program constructs a tracker client object (vrpn_Tracker_Remote), giving a string that includes the name of the server object (Tracker0) and the location of the server program (@myhost). The name must match the name of the server device (obtained from a configuration file). The information after the @ sign is a Universal Resource Locator (URL), whose default type is to a VRPN connection at the host whose name is specified. Real applications read the device name from the command line or an environment variable.

The program next registers a callback handler to receive pose reports from the tracker. This handler is called whenever tracker pose messages are received. The callback parameter *t* holds the data passed by the server; for trackers, this is the time at which the message was sent, the sensor number, its position, and its orientation.

The *mainloop()* method must be called periodically for each client object. This method causes VRPN to send all pending messages and read all incoming messages for the connection associated with the device. (When there are multiple devices sharing the same connection, the *mainloop()* call on one device will in fact deliver the pending messages to all of them.)

Callback handlers and flow of control: The application sets up handlers for each message type. The handlers may potentially be called in three circumstances: when *mainloop()* is called on the object they are registered with; when *mainloop()* is called on another object sharing the same connection; and when an appropriate-type message is sent by an object that shares the same connection. Because the mapping of objects to connections is flexible and the effects of object methods vary, the application should operate under the assumption that the callback handlers may be invoked whenever a call is made to any VRPN object, but at no other time.

The *controlled ambiguity* of when callbacks can be invoked is an important part of the semantics of VRPN or any library that allows both local and remote servers. The ambiguity is due to the possibility of multiple devices mapping to same connection, and to the optimization of local message delivery. The ambiguity is controlled because the handlers are not called at arbitrary times, but rather only when a VRPN method is invoked. This allows the programmer to proceed as though the system were a single-threaded application.

8. Conclusion

VRPN provides a network-transparent interface to virtual-reality peripherals. Due to its flexibility and performance, it is a widely used platform, even by users of other general-purpose VR frameworks. This document describes the features of VRPN that are critical to its success, in particular its novel method of factoring a device into separate and almost independent functions. This separation allows each function to be handled in a suitable way, without the complexities of combinations. VRPN handles the mapping of functions to communications channels and device drivers transparently and efficiently. The bundling back into device groupings is (1) higher-level and (2) handled almost without the user thinking about it.

VRPN also integrates a number of more well-known features. Having these features combined into a single system is valuable both to those who use VRPN directly and as an example to those who implement libraries or applications that make use of VR peripherals.

References

1. Taylor II, R.M., *The Virtual Reality Peripheral Network (VRPN)*, . 1998: http://www.cs.unc.edu/Research/vrpn.
2. Staff, *dVS Technical Overview*. 1993, Bristol, UK: DIVISION Limited.
3. Corporation, S., *WorldToolkit Technical Overview*, . 1998.
4. Panda3D, *http://www.panda3d.com/*, .
5. Shaw, C., *et al. The decoupled simulation model for VR systems*. in *Proceedings of CHI '92*. 1992.
6. Sokolewicz, M., *et al. Using the GIVEN++ Toolkit for System Development in MuSE*. in *Proceedings of First Eurographics Workshop on Virtual Reality*. 1993. Polytechnical University of Catalonia.
7. Ståhl, O. and M. Andersson. *DIVE - a Toolkit for Distributed VR Applications*. in *Proceedings of the 6th ERCIM workshop*. 1994. Stockholm.
8. Singh, G., *et al. BrickNet: Sharing Object Behaviors on the Net*. in *Proc. IEEE Virtual Reality Annual International Symposium (VRAIS'95)*. 1995: Research Triangle Park, NC.
9. Gossweiler, R., *et al. DIVER: a DIstributed Virtual Environment Research Platform*. in *IEEE 1993 Symposium on Research Frontiers in Virtual Reality*. 1993.
10. Snowden, D.N. and A.J. West. *The AVIARY VR-system. A Prototype Implementation*. in *Proceedings of the 6th ERCIM workshop*. 1994. Stockholm.
11. Pettifer, S., *et al. DEVA3: Architecture for a Large Scale Virtual Reality System*. in *Proc. ACM Symposium in Virtual Reality Software and Technology 2000 (VRST'00)*. 2000. Seoul, Korea.
12. Just, C., *et al. VR Juggler: A Framework for Virtual Reality Development*. in *2nd Immersive Projection Technology Workshop (IPT98)*. 1998. Ames.
13. Watsen, K. and M. Zyda. *Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments*. in *1998 IEEE Virtual Reality Annual International Symposium (VRAIS'98)*. 1998. Atlanta, Georgia.
14. Julier, S., *et al. The Software Architecture of a Real-Time Battlefield Visualization Virtual Environment*. in *Proceedings IEEE Virtual Reality '99*. 1999. Houston, Texas: IEEE Computer Society Press.
15. Arsenault, L., *et al.*, *http://www.diverse.vt.edu/*, .
16. Foley, J., V.L. Wallace, and P. Chan, *The Human Factors of Computer Graphics Interaction Techniques.* IEEE Computer Graphics and Application, 1984. **4**(11): p. 13-48.
17. Measurand, *www.measurand.com*, . 2000.
18. Kessler, G.D. and L.F. Hodges. *A Network Communication Protocol for Distributed Virtual Environment Systems*. in *Proceedings of VRAIS '96*. 1996. Santa Clara: IEEE.
19. Adachi, Y., T. Kumano, and K. Ogino. *Intermediate Representation for Stiff Virtual Objects*. in *Proc. IEEE Virtual Reality Annual International Symposium (VRAIS'95)*. 1995. Research Triangle Park, NC.
20. Bryson, S.T. and S. Johan. *Time Management, Simultaneity and Time-Critical Computation in Interactive Unsteady Visualization Environments*. in *IEEE Visualization '96*. 1996: IEEE.
21. Mark, W., *et al. Adding Force Feedback to Graphics Systems: Issues and Solutions*. in *Computer Graphics: Proceedings of SIGGRAPH '96*. 1996.
22. Holloway, R., *Registration error analysis for augmented reality.* Presence, 1997. **6**(4): p. 413-432.