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
University of North Carolina at Chapel Hill
Department of Computer Science
CB #3175, Sitterson Hall, Chapel Hill, NC 27599-3175
919-962-1700

taylorr@cs.unc.edu

ABSTRACT

The Virtual-Reality Peripheral Network (VRPN) system provides a device-independent and network-transparent interface to virtual-reality peripherals. VRPN's application of factoring by function and of layering in the context of devices produces an interface that is 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.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: *Distributed applications.*

General Terms

Algorithms, Performance, Design.

Keywords

Interactive graphics, virtual environments, virtual worlds, input devices, library, peripherals.

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, buttons, etc.) used in a virtual reality (VR) system [1]. 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.

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 06.00. 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 second contribution. These features are presented in the following categories: establishing VRPN connections, dealing with distributed objects, VRPN message characterstics, separate client and server, storage and replay, and performance. Implementation details for these features are often omitted (they are in the publicly-available code).

2. RELATED WORK

Many prior and existing toolkits provide complete distributed virtual world interfaces, concentrating on flexibility and generality. There are both commercial systems (including CAVELib [2], Division's dVS [3], Sense8's WorldToolKit [4], and Panda3D [5]) and research systems (including the MR toolkit [6], GIVEN++ [7], DIVE [8], BrickNet [9], Alice/DIVER [10], AVIARY [11], Maverik/DEVA [12], VR Juggler [13], Bamboo [14], and Dragon [15]). There is also recent unpublished work by the DIVERSE [16] group.

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

VRPN is complementary to VR toolkits, as indicated by the fact that 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.

3. 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 [17]. 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;
- *Analog* reports one or more analog values.
- *Dial* reports incremental rotations.
- *ForceDevice* enables 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 enable 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,
- enabling devices to export multiple interfaces,
- silently ignoring unsupported message types, and
- providing application-level access to all messages.

Factoring based on function: A particular device's special features often amount to implementing more than one function (such as a tracker sensor with a built-in button). A VRPN driver for the device will export interfaces for multiple device types. The server for the SensAble™ Technologies 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 designed to use a Phantom™ haptic display can be tested on either of the other setups, although no forces will be generated.

Mapping devices to connections: Although the Tracker, Button and ForceDevice devices for a Phantom™ are logically separate they are all internally mapped to the same network connection for communication efficiency.

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.

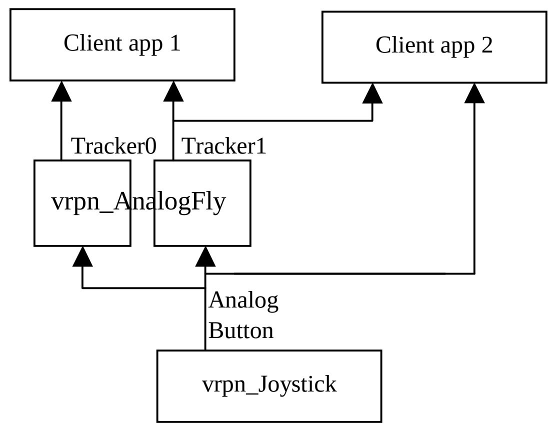

Figure 1: Devices can be layered, so that the outputs from one device driver can become inputs to another. Application code can attach to either or both devices. The vrpn_AnalogFly driver exports multiple vrpn_Tracker interfaces under different names (Tracker0 and Tracker1). Client 1 reads both trackers. Client 2 reads one of the trackers, the analogs, and buttons.

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 to enable flying using joysticks: the joystick driver reports analog values for each of its axes and the AnalogFly integrates these values into Tracker messages. Clients can connect both to the low-level analog device (to read the buttons on a joystick, for example) and to the higher-level tracker device (see figure 1).

Another special case of multiple interfaces is the *multiple-behavior device*. Within our department, there are several groups that use joystick devices to fly the user. Each group has a favorite mapping of joystick axes to transformations that depends on application requirements and user preferences. Using multiple instances of AnalogFly servers, each with a unique name, joystick servers can export all interfaces at once. Each client connects to the interface with the desired mapping (see figure 1).

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 Phantom™ and the 3rdTech HiBall™ tracker) adjust their rates to match that requested, while other servers silently ignore these messages. For messages that can be safely ignored, 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 is an example; a calibration application for a particular device 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. The server would provide handlers for the new message types, and use them to communicate with the calibration application at the same time it exported its standard interface. Such a specialized calibration application could be run at the same time as standard applications.

Example new device: An example is helpful to show how these features work together. Let's plan a driver for Measurand *SHAPE TAPE* [18]. 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 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 might be used to track 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 spline. A client application would attach to the device using its Tracker interface, which reports the poses, while a calibration application would 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.

4. ESTABLISHING VRPN CONNECTIONS

The application side (client) and server-side portions of each device driver communicate with each other over a *Connection* object. This section describes the algorithms used to establish VRPN connections between the client and server, and how reconnection works after a server shutdown.

Connection initialization: The connection initialization 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 running,
- 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).

The following connection algorithm was developed to address these needs. Although complicated, this algorithm runs only at connection startup – messages flowing during a session are sent directly.

Establish reliable channel: Each server opens a well-known UDP port for connection requests from clients (TCP connection requests from the client to the server cannot be used because they can hang indefinitely for ports in certain states). A client opens any available TCP port and sends the server a UDP request asking it to connect to that TCP port. The client then enters a state in which it polls 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 until it gets a response from the server (the client devices will not be connected to their server counterparts, see "Client/server object verification"). When the server receives the connection request, it calls the client back at its specified TCP port.

Finish connection setup: Once the reliable TCP channel is established, it is used to perform version checking between the endpoints, perform clock synchronization between the hosts, and establish a separate unreliable UDP channel between the hosts.

To support connections from clients that are behind firewalls or which are using network address translation (NAT), the server also listens for TCP connection requests at the same well-known port and routes all packets through TCP for a connection established this way. No changes to the application are required in this case; the client just connects using a different URL.

Dropped connections and reconnect: When a connection is dropped by the other side, a message is sent to any interested objects or user code indicating the drop, to trigger any needed clean-up. (A similar message is sent whenever a new connection is established to trigger required set-up.) A client connection will attempt to re-establish the link, using the same algorithm it used for the initial connection. This enables the application to be robust in the presence of server failure and re-start, and is very useful in cases where the application has a long start-up time (due to database loading, for example).

5. DISTRIBUTED OBJECTS

When an application uses servers on remote machines and there is an error condition, there is the question of how to indicate this to the user (who may not even be logged into the machine). Early versions of VRPN did not provide support for this. VRPN now handles this by enabling each object to send text messages. These messages have associated severity (normal, warning, or error), and they propagate across connections. VRPN includes a static text-printing object that prints these messages on the client. By default, it prints warnings and errors to standard output. This mechanism enables device drivers and servers anywhere in the system to send human-readable warning and error reports. This has proven to be *very* useful when debugging system behavior.

In the presence of remote servers that can exit and restart, a mechanism is needed to inform the application and/or user that a device server is no longer operating. When a client object is created, it sends a verification message to its associated server to verify that it is running. After three seconds without response, the client object begins to emit warning text messages; after 10 seconds, it begins to emit error messages. In addition to using these messages to alert the user, a client or server object can provide callback handlers for the verification messages and use them to trigger initialization code whenever a connection to its counterpart is established.

6. VRPN MESSAGE CHARACTERISTICS

Once a connection has been established and each client object is connected to its server, message flow begins. For each message, VRPN provides a selectable *class of service* and a *synchronized time stamp*.

Class of service: 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 [19]. 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). The class of delivery can be selected on a message-by-message basis.

Synchronized time stamps: Each message in VRPN has an associated time stamp, which is generated by the server-level or application-level code sending the message. This time is intended to match the time at which the data for the message became available (may be in the past), or the time at which an action should be taken by the receiver (may be in the future). For example, the time attached to messages sent by the serial tracker servers is the time at which the first character for that message was read from the serial port. In order to make timestamps from devices running on different host computers comparable, VRPN provides clock synchronization between the hosts on either side of a connection. As messages are passed between hosts, the time on each is adjusted by an offset that makes it comparable to values returned by the system function *gettimeofday()* on the local host. Time stamps, along with clock synchronization, give a global ordering to events within a VRPN system and enable predictions based on local time information.

7. SEPARATE CLIENT AND SERVER

The ability to run different parts of a VR application in different processes is of widely recognized importance [6-8, 20-22]. 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.

Stark examples of different update rates include running a haptic server (1kHz update rate) with a graphics application (30 Hz update rate), [22] or an interactive graphics application with a simulation. [21] When server device initialization takes a long time, it can be more efficient to leave a device server running and connect to it when access is needed; in this way, the application can start immediately. When accurate measurement of the timing for each device message (such as tracker position reports) is required, having a separate server allows frequent (1kHz) checks, independent of application update rate. A separate server process also provides frequent servicing of devices even in the presence of long pauses in the application, which can be important to avoid losing reports (serial trackers) or missing important events (rapid button press/release).

Local client and server: Whereas VRPN is designed to separate client and server over network connections, it is also 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.

8. 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 enable 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.

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

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. Replay does not require any extra code to replay the original session at its normal rate.

9. 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; Holloway reports about 1mm offset for each 1ms of latency in a VR system [23]. The value of different features is measured against their cost in time. On this scale, VRPN measures up well; for some configurations the time to read a message using a remote VRPN server 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 (which are applicable to specific current hardware and software).

```
#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 2: Complete program to open a tracker and print its position updates.

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 overheads 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 by changing operating-system parameters (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 VRPN server than from 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. Within VRPN, 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 VRPN 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 application loop.

This paper does not present the end-to-end timing (user motion to screen update) for trackers and other devices within VRPN, but rather the incremental latency due to VRPN. VRPN messages are sent between hosts using a single UDP or TCP packet, so network latency should be minimal compared to other network-capable toolkits. As described before, it is also possible to run both the client and server in the same thread; in this case, the communications overhead reduces to that of a few function calls.

Faster initialization: Because the VRPN server keeps the attached devices running continuously, clients avoid waiting for the initialization/reset procedure that is lengthy for many devices. This reduces what can be a several-second procedure to a several-millisecond procedure. Clients that use VRPN in the single-process mode do not get this benefit, of course.

How far can you go? We use VRPN to communicate between a graphic and haptic interface on one end of a network and an Atomic Force Microscope teleoperated on the other end. We find that the system has round-trip latency requirements of about 40ms for acceptable performance. The system has been operated successfully on:

- local, switched Ethernet,
- Internet connection to a local high school,
- Internet2 from Washington, D.C. to UNC-CH (via Atlanta),
- Internet2 from Columbus, Ohio to UNC-CS.

The system is not able to operate on a shared Ethernet segment with a large number of machines, nor over the standard Internet to California or Louisiana from UNC.

10. USING VRPN: APPLICATION LAYER

VRPN is designed to be as easy to use as possible for a client program. A complete example client program that reads and prints positions from all of the sensors on a tracker is shown in figure 2.

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 information after the @ sign is a Universal Resource Locator (URL), whose default type is a VRPN connection at the host whose name is specified. Real applications would 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 associated with the message by the server, 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 only when a VRPN method is invoked.

11. USING VRPN: SERVERS

There is a server program, *vrpn_server*, that is able to run most devices supported by VRPN: Origin Dynasight, Polhemus Fastrak, Intersense IS-600 and IS-900, Ascension Flock of Birds, Tracker example server, AnalogFly, SGI dial & button box, Pinchglove, BG Systems CerealBox, Logitech Magellan, Radamec SPI, ImmersionBox, Wanda, Dial example server, UNC joystick, and UNC buttons. The server determines which devices to start and the parameters for each by reading a configuration file (*vrpn.cfg* by default). The VRPN distribution includes instructions and example entries for each of these devices in the file *vprn.cfg.SAMPLE*, in the server_src directory. This file can be copied over the vrpn.cfg and the user can uncomment and customize the lines for the devices to be used. As new device servers are implemented, they are usually added to this file, and vrpn_server updated to be able to launch them.

Some servers really only work on one architecture or require linking with other outside libraries, so they have their own executables. Currently, the Phantom server, a Sierra video router server, a National Instruments D/A server, and the sound servers are this way. These servers are distributed with VRPN. They can be compiled separately from vrpn_server.

Also included in the server_src directory is a *client_and_server* example server that shows how to run an application that has both client objects and server objects within the same executable.

12. CONCLUSIONS AND AVAILABILITY

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 independent functions. This separation enables 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.

VRPN is public-domain, open-source software with a user community in academia, industry and the national labs. There is a mailing list for the community of people using VRPN; pointers to this plus the code are available from the project web page at www.cs.unc.edu/Research/vrpn.

Supported platforms: The VRPN application-side library runs on PC/Win32, SGI/Irix, PC/Linux, Sparc/Solaris, HP700/Hpux, and PowerPC/AIX. The server-side library is fully functional under SGI/Irix, PC/Win32, PC/Linux, and Sparc/Solaris (it is functional except for serial-port code on the other systems).

Supported Devices: 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, Phantom™, 3rdTech HiBall 3000, 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, the "Totally Neat Gadget" (TNG3) from Mindtel, and the UNC hand-held Python controller (buttons).

13. ACKNOWLEDGMENTS

VRPN was developed under funding from the National Institutes of Health National Center for Research Resources at the Research Resource on Molecular Graphics and Microscopy.

Commercial support for driver development and new features has come from Walt Disney VR Studios, Schlumberger Cambridge Research, and 3rdTech.

Many people other than those listed as authors on this paper have contributed device drivers and other code to the development of VRPN. They include several members of the UNC-CH department of computer science: Phillip Winston wrote the UNC ceiling-tracker driver and helped with the Windows port. Jun Chen helped with the PHANToM driver. Arthur Gregory added meshes to the PHANToM code and worked to speed up collision detection. Lin Cui implemented the vrpn_Text class. Jason Clark wrote the Miles-based sound server. Zak Kohn added more sound support.

Contributions have also come from individuals and groups at other institutions: Randy Heiland (NCSA) ported a version of the PHANToM driver from NT to SGI. John Stone (Beckman Institute, UIUC) cleaned the code up so that it compiles on multiple architectures, and in particular so that it will compile outside of UNC. Rob King from the Navy Research Labs write the ImmersionBox driver, the serial-port mouse-button driver, the TNG driver, and submitted several bug fixes. The AIX and Solaris patches that allow compilation on these systems come from Loring Holden and Bob Zeleznik at Brown University. The Wanda driver comes from Brown University; they also helped with the Pinchglove driver. The driver for the 5DT is provided by Philippe DAVID and Yves GAUVIN from Direction de la Recherche. The Linux Joystick drivers (Joylin) were written by Harald Barth.

14. REFERENCES

1. Taylor II, R.M., *The Virtual Reality Peripheral Network (VRPN)*, . 1998: http://www.cs.unc.edu/Research/vrpn.
2. VRco, *CAVELib*, . 2001, http://www.vrco.com/CAVE_USER/.
3. Staff, *dVS Technical Overview*. 1993, Bristol, UK: DIVISION Limited.
4. Corporation, S., *WorldToolkit Technical Overview*, . 1998.
5. Panda3D, *http://www.panda3d.com/*, .
6. Shaw, C., *et al. The decoupled simulation model for VR systems*. in *Proceedings of CHI '92*. 1992.
7. 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.
8. Ståhl, O. and M. Andersson. *DIVE - a Toolkit for Distributed VR Applications*. in *Proceedings of the 6th ERCIM workshop*. 1994. Stockholm.
9. 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.
10. Gossweiler, R., *et al. DIVER: a DIstributed Virtual Environment Research Platform*. in *IEEE 1993 Symposium on Research Frontiers in Virtual Reality*. 1993.
11. Snowden, D.N. and A.J. West. *The AVIARY VR-system. A Prototype Implementation*. in *Proceedings of the 6th ERCIM workshop*. 1994. Stockholm.
12. 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.
13. Just, C., *et al. VR Juggler: A Framework for Virtual Reality Development*. in *2nd Immersive Projection Technology Workshop (IPT98)*. 1998. Ames.
14. 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.
15. 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.
16. Arsenault, L., *et al.*, *http://www.diverse.vt.edu/*, .
17. 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.
18. Measurand, *www.measurand.com*, . 2000.
19. 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.
20. 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.
21. Bryson, S.T. and S. Johan. *Time Management, Simultaneity and Time-Critical Computation in Interactive Unsteady Visualization Environments*. in *IEEE Visualization '96*. 1996: IEEE.
22. Mark, W., *et al. Adding Force Feedback to Graphics Systems: Issues and Solutions*. in *Computer Graphics: Proceedings of SIGGRAPH '96*. 1996.
23. Holloway, R., *Registration error analysis for augmented reality*. Presence, 1997. **6**(4): p. 413-432.