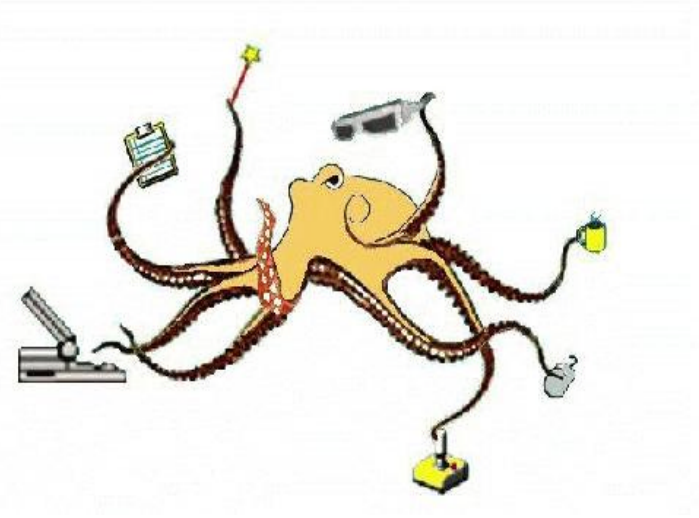
DIVERSE:

a Software Toolkit to Integrate Distributed Simulations with Heterogeneous Virtual Environments.

Lance Arsenault, John Kelso,
Ron Kriz, Fernando Das Neves

University Visualization and
Animation Group

Virginia Tech, Blacksburg, VA



diverse@vt.edu
www.diverse.vt.edu

ABSTRACT

We present DIVERSE (**D**evice **I**ndependent **V**irtual **E**nvironments- **R**econfigurable, **S**calable, **E**xtensible), which is a modular collection of complimentary software packages that we have developed to facilitate the creation of distributed operator-in-the-loop simulations. In DIVERSE we introduce a novel implementation of remote shared memory (distributed shared memory) that uses Internet Protocol (IP) networks. We also introduce a new method that automatically extends hardware drivers (not in the operating system kernel driver sense) into inter-process and Internet hardware services. Using DIVERSE, a program can display in a CAVE™, ImmersaDesk™, head mounted display (HMD), desktop or laptop without modification. We have developed a method of configuring user programs at run-time by loading dynamic shared objects (DSOs), in contrast to the more common practice of creating interpreted configuration languages. We find that by loading DSOs the development time, complexity and size of DIVERSE and DIVERSE user applications is significantly reduced. Configurations to support different I/O devices, device emulators, visual displays, and any component of a user application including interaction techniques, can be changed at run-time by loading different sets of DIVERSE DSOs. In addition, interpreted run-time configuration parsers have been implemented using DIVERSE DSOs; new ones can be created as needed.

DIVERSE is free software, licensed under the terms of the GNU General Public License (GPL) and the GNU Lesser General Public License (LGPL) licenses.

We describe the DIVERSE architecture and demonstrate how DIVERSE was used in the development of a specific application, an operator-in-the-loop Navy ship-board crane simulator, which runs unmodified on a desktop computer and/or in a CAVE with motion base motion queuing.

INTRODUCTION

MOTIVATION

Distributed operator-in-the-loop simulations have been a practical tool for quite some time [Schacter][Aviation] and with the introduction of immersive visual displays [CAVE][HMD] distributed virtual reality (VR) has emerged as an active research topic for the last decade [Lehner].

There are many software toolkits that provide a framework for building virtual reality and graphics applications, and many come with optional add-on software to aid in the building of distributed virtual reality and graphics applications in which many remote users may interact [DVR].

DIVERSE (Device Independent Virtual Environments– Reconfigurable, Scalable, Extensible) is a highly modular collection of complimentary software packages designed to integrate distributed simulations with heterogeneous virtual environments (VEs). DIVERSE currently consists of two packages:

1. The DIVERSE ToolKit (DTK) is a utility package for creating distributed operator-in-the-loop simulations. DTK contains:

   - an extensible interrupt-driven hardware peripheral server and a C++ client application programming interface (API),
   - a novel inter-process communication mechanism for asynchronously distributing simulations using Internet Protocol (IP) networks,
   - and wall-clock time process synchronization, and many other standard system library wrappers.

   DTK provides access to local and networked interaction devices, both real and simulated. It also provides support for run-time swapping of I/O devices and/or device emulators, enabling the creation of device independent applications.

2. The DIVERSE graphics interface for Performer (dgiPf) adds immersive and/or non-immersive graphics to simulations by augmenting OpenGL Performer™, a high-level scenegraph-based graphics API built using the OpenGL™ graphics API[Performer][OpenGL]. By using dgiPf the same program can display in a CAVE™, ImmersaDesk™, head mounted display (HMD), desktop monitor and laptop without modification. dgiPf uses DTK for utility functions and I/O facilities.

DTK is a separate stand-alone package, and applications that do not need graphical display output do not need to link with dgiPf or Performer.

This design architecture is different from comparable software packages in which the distributed simulation toolkit depends on a specific graphics package and/or is limited in application scope to the sharing of avatars in VEs [Hartling]. Computer generated graphical displays, immersive

or not, are output devices, and computer simulations do not necessarily require graphical output. But, the computer simulations do drive the graphical output, so we made the DTK utility package independent of the graphics package, dgiPf.

We find that a *non-graphics-centric* design leads to a much more flexible set of software tools. Unlike most other VE graphics packages, the modular design of DIVERSE does not lock the user into using a particular, or even any, graphics API. The non-graphics-centric design architecture of DIVERSE distinguishes it from similar packages and makes DIVERSE a more diverse set of software tools.

DIVERSE is designed to build simulators, and to provide the ability to augment the simulator with immersive and/or non-immersive computer graphics displays, at the user's option.

OVERVIEW

DIVERSE is written in C++. It is a collection of C++ APIs and application, utility and example programs. It currently runs on GNU/Linux and SGI IRIX systems [Linux][IRIX]. Multiple IRIX binary types are supported– o32, n32 and 64.

DIVERSE is free software. Its libraries are licensed under the LGPL (GNU Lesser General Public License), and its programs are licensed under the GPL (GNU General Public License) [GPL].

DIVERSE uses the SGI IRIX, or GNU/Linux operating system, a catchall term for all the underlying standard system software libraries, including but limited to:

- OpenGL,
- X windows,
- TCP networking,
- Math libraries, and
- Drivers to communicate with hardware interfaces such as graphics displays, serial lines and network cards.

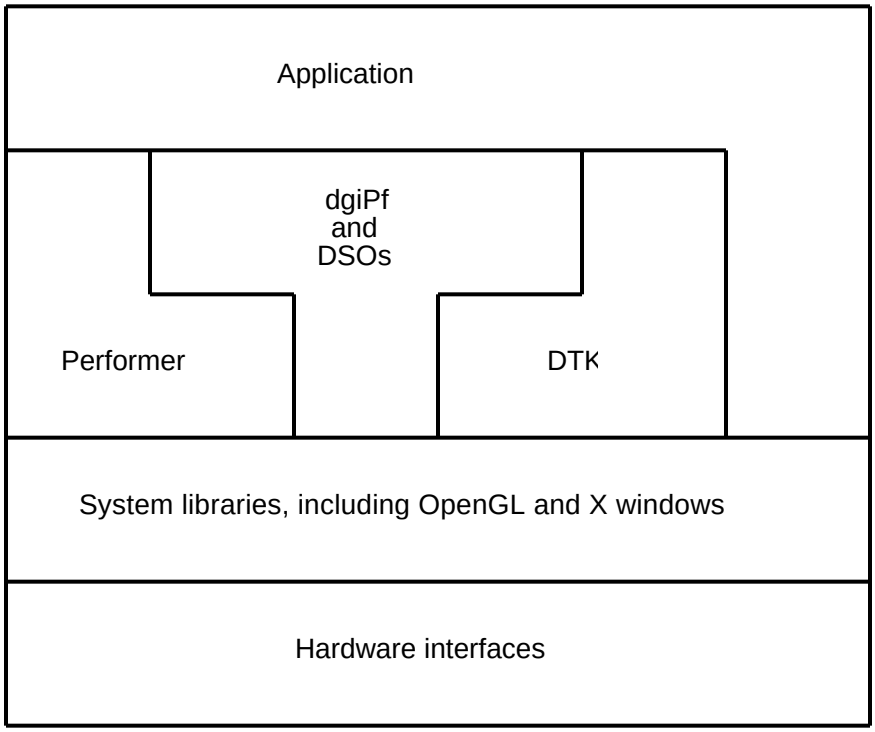


Figure 1, DIVERSE library dependencies

Figure 1 shows all the possible dependencies between the various libraries and a DIVERSE application. Not all applications need to use all of the libraries, and a specific application might use other libraries not needed by DTK or dgiPf. Dependency is illustrated by horizontal lines between components with the component above a line dependent on components below a line.

DESIGN CRITERIA

Some simple criteria and directives guide the design of DIVERSE. Our hope is that by always looking back at first principles, we would develop a package that achieves a coherent and consistent design.

Our design directives include:

1. Works by default

   Any good software package needs reasonable default behaviors. We try to make the default behavior be the case that is most used or the least damaging, if the most used case can cause harm. "Works by Default" also implies that the software should be robust and work all the time.

2. Stay out of the user's way!

   For packages like DIVERSE this also means to stay out of the application programmer's way. The DIVERSE developers have noticed that some software packages make assumptions that prevent you from doing what you want to do. They are designed to be

the central part of your system. We say that they follow the "*center of the universe*" design paradigm.

We assume that programmers know best how to design and structure what they are doing. DIVERSE is designed to augment, without imposing a particular structure. This facilitates freedom in design.

3. Easy to use

   Like the first directive this one may seem obvious, but it can also conflict with the second design directive. But, since the second directive fosters innovation and flexibility we tend to favor it over ease of use.

Our design criteria, which are driven by our design directives, are:

1. Modular design

   A module is a standardized, interchangeable component. By using a highly modular design we are able to achieve:

   · Selectivity– The programmer only needs to use the parts of DIVESRE that are relevant to the application. The end-user only needs to use the parts of DIVERSE that are relevant to the application's specific execution environment or desired modes of behavior.

   · Interchangeability– Modules interact with other modules via standard interfaces. This allows the abstraction of such things as hardware devices and navigation techniques. For example:

     o A head tracker is not tied to a particular hardware device, or even **any** hardware device, as it can be emulated via software. The application is unaware of the tracker data source, and this data source can be switched amongst the various hardware and software trackers, real or simulated, without the application having to be restarted, or it even being aware of the source being changed.

     o A navigation module allows locomotion through the virtual world. Navigation techniques that are encapsulated as modules allow these techniques to be developed and used independently of the application program. This also allows for on-the-fly switching between navigation techniques (modules).

   · Flexibility– Modules make DIVERSE easy to extend and reconfigure; this also applies to applications that use DIVERSE. For example:

     o New modules can be built on top of existing modules to add functionality, and new instances of existing modules can be written to support new implementations.

- Applications can use the new module types and instances without requiring program modification.

2. Augment, don't replace

   Whenever possible DIVERSE tries to augment an existing software library instead of creating its own implementation of the package. Similarly, if an existing library can be used by DIVERSE it will do so. This has two benefits; first, DIVERSE can take advantage of the quality software efforts of others and focus on its specific goal. Second, this often makes it easier to port existing applications into DIVERSE, as they don't have to be completely rewritten.

3. The same program works everywhere

   A DIVERSE program should be able to run on any supported hardware configuration without modification. On the same machine platform, it should be able to do so without needing to be recompiled. It will, however, require different system-specific run-time configuration files to be loaded.

DTK, THE DIVERSE TOOLKIT

REMOTE SHARED MEMORY

DTK remote shared memory [Carter] was developed by necessity. In addition to providing a general Inter-Process Communication (IPC) application-programming tool, DTK remote shared memory is the IPC method with which DTK provides the seamless distribution of DTK I/O device services. It provides a standard coding interface to I/O device data for local (same computer) and remote access, without requiring that local and remote access to be coded differently. The nodal connection topology is typically managed from outside of the applications that are using the shared memory, which in general leads to a more flexible and manageable nodal connection topology. This, in turn, makes for flexible applications and program interconnectivity, i.e. all applications that use DTK shared memory are inherently capable of being networked applications without additional coding. With respect to this feature, this is similar to the way the X window system makes window-based applications inherently capable of being networked applications.

An example of a classic DTK service is a tracker server that serves data corresponding to a user's head position and orientation in a VE display system, such as in a CAVE.

DTK provides many utilities, the most distinguishing of which is a novel implementation of remote shared memory. We prefer the term remote shared memory over the term distributed shared memory because the word *remote* will hopefully imply inter-computer or Internet connectivity, which we provide, whereas the word *distributed* seems to refer to operating system level or inter-process connectivity, which is provided implicitly by DTK remote shared memory.

DTK remote shared memory is based on the DTK shared memory arena. The DTK shared memory arena is, in turn, based on the system calls mmap() and ftruncate() as illustrated in

UNIX Network Programming [Stevens]. The DTK shared memory arena provides a name string with which to refer to each shared memory segment. This name string is called the DTK shared memory segment name, or segment name for short. Segment names are useful for describing the information that is stored in a DTK shared memory segment. They are mapped to the same name by all processes, although the virtual address of the shared memory segment is, in general, mapped to a different address for each process that accesses it.

Although the DTK shared memory arena does provide a user API, the primary API is through a shared memory segment manager class called dtkSharedMem. For each DTK shared memory segment the user gets an instance of a dtkSharedMem class. The dtkSharedMem class provides read(), write() and many other methods that access and manage a DTK shared memory segment.

DTK remote shared memory is implemented in user space. The action that causes DTK remote shared memory to be shared on an IP network is the overloaded dtkSharedMem::write() method. Each time write() is called the current process "pushes" the data to all the IP addresses that are present in its IP network list. Each time write() is called a local process counter is compared to a shared memory counter, and if the two counters differ then the IP network list is updated; then write() "pushes" the data to the IP network list.

We have optimized the DTK shared memory for interactive operator-in-the-loop simulations. We feel that using a "push write" data transfer method is the most effective for interactive simulations, because it eliminates the need for request packets. dtkSharedMem::write() directly implements the IP network writing, so operating system interrupts to trigger the "push write" aren't needed, thereby reducing system overhead. DTK shared memory is read by the dtkSharedMem::read() method; the main action of which is just a local memory copy. Both the read() and write() methods use read/write locks to guarantee a coherent data state. The DTK client class contains a shared memory factory class object that creates and manages dtkSharedMem objects.

We have built DTK shared memory queues that are similar to DTK shared memory segments:

- DTK shared memory queues may be written to and read from by any number of processes.
- The read and write operations wrap read/write locks to guarantee coherent data state.
- Instances of the dtkSharedMemQueue class are gotten from a DTK shared memory factory class object that creates and manages dtkSharedMemQueue objects, in addition to the dtkSharedMem objects.

Each dtkSharedMemQueue object has its own set of pointers into the queued data so that read and flush operations affect just that local dtkSharedMemQueue object's pointers, thus implementing shared queues. The queues are circular; writing to a full queue just overwrites the oldest entries.

Any DTK shared memory segment can be queued in a DTK shared memory queue at the request of any connected object. When a DTK shared memory segment is "queued," a DTK shared memory queue is added to the DTK shared memory segment write list, in the same manner as IP network write lists. This means that a polled data service can be queued by any connecting client, with no extra development overhead imposed on the server.

DTK-SERVER, THE DTK SERVER

DTK distinguishes between a DTK server, and the DTK services, which are provided by a DTK server. Any DTK service can be loaded at any time while a DTK server is running. Multiple instances of the same DTK service can be loaded at the same time. Command line options can specify any number of DTK services to be loaded at DTK server startup.

dtk-server, the DTK server program, runs as a single process, single-threaded, interrupt driven, select()-based server. It can run as a daemon or a regular process through the use of command line options. Multiple DTK servers can run concurrently on a single computer. In designing the DTK server it is assumed that a DTK server will utilize less than 50 percent of CPU time utilization; this is the case for all of the DTK services currently implemented. Most DTK services use less than 5 percent CPU.

Most of the dtk-server code is implemented using the C++ API of the DTK client library. The DTK server program structure can be broken into the following functional sections:

- **Startup**- parses command line arguments, creates DTK server daemon (optional), instantiates a global **Request Processing Object**, loads initial DTK services using the **Service Loader**, and returns a status to the shell (if daemon).

- **Service Loader**- loads and unloads DTK services.

- **Command String Parser**- is a function that can be called by any loaded DTK service to parse character strings representing DTK client-to-server or server-to-server command requests (the protocol is defined in the DTK client C++ library), and to call the corresponding **Request Processing Object**'s method.

- **Request Processing Object**- is the heart of the DTK server. The **Request Processing Object** is a global object whose methods are called by the **Command String Parser**, or from any loaded DTK service. It shares state information about the server. It handles all of the internal actions that affect the DTK server and any connected shared memory arena, such as: loading the unloading services by calling the **Service Loader**, modifying and propagating remote shared memory write lists to other servers and clients, reading from local shared memory and writing to remote shared memory, reading from remote shared memory and writing to local shared memory, and telling a client the name of the local shared memory arena file.

- **Main Loop** is entered after **Startup** finishes; in a loop, it: calls a blocking select(), finds the DTK service object that is associated with the selected file descriptor, and calls the serve() method of that object.

Without DTK services loaded, the DTK server does not serve any function. It is not even self-serving, like so many middle managers and computer system administrators.☺ Therefore there is a set of default DTK services that are loaded at startup.

DTK SERVICES

In addition to the general DIVERSE design criteria, the DTK Server adds the design criteria to "minimize the amount of code that needs to be written to provide interrupt driven I/O device services to multiple client programs." We do this in order to minimize the amount of code needed by DTK services. DTK services are compiled Dynamic Shared Objects (DSOs) that are written in C++ and are loaded by a running DTK server; their value is that they extend the server's functionality at run-time.

DTKSERVICE, THE DTK SERVICE CLASS

DTK service DSOs contain C++ objects that are based on the pure virtual base class dtkService, which defines the standard interface that the DTK server uses. The only dtkService methods that are required to be written into a service DSO are the constructor and the serve() method. It must also set a file descriptor to be used by the server. dtkService::serve() is called by the DTK server any time data is available through the file descriptor. Typically a destructor is used to close the file descriptor. This destructor is called when a DTK server unloads a DTK service DSO. There is additional utility built into the dtkService base class in order to decrease the amount of code that needs to be written in the service DSOs. The **Request Processing Object** and a DTK shared memory factory object are also accessible by loaded DTK service DSOs in order to provide additional utility.

A typical DTK service DSO declares a dtkService constructor and destructor that opens and closes one file; this may be an IP socket, serial port or other device. The dtkService::serve() method reads and parses the file data and then writes the state of its function to DTK shared memory. When the service writes to DTK shared memory, the DTK shared memory write() operation automatically determines if the DTK shared memory is connected to any remote DTK shared memory arenas, and writes to the remote arenas. In this way there is no difference between local and networked DTK services.

CORE DTK SERVICES

There is a list of core DTK services that are loaded whenever a DTK server starts up. These core services provide the default IPC methods that are needed by the server so that it may receive and send requests and responses with DTK clients and other DTK servers. The current core DTK services are:

- Internet Domain TCP/IP Service,
- Internet Domain UDP/IP Service, and
- UNIX Domain UDP/IP Service.

These three DTK services all do the same thing, but through the different transport protocols. They all read commands on their sockets, call the **Command String parser**, and write a response to the socket, or not, which implements the DTK server protocol.

THREE SELECTED EXAMPLES

1. The **Moog™ Motion Base Controller Service** uses a RS422 serial line to control a Moog 6DOF 2000E motion base [Moog]. Any number of DTK clients may send commands to the motion base. Motion base position commands are time-stamped and queued in a DTK shared memory queue. Positional commands are read from the queue and linearly interpolated using the time-stamp. This interpolation results in smoother motion of the motion base compared to with no interpolation. Advisory locks are used to prevent multiple simultaneous position request commands. A separate DTK shared memory segment and DTK shared memory queue pair is used to send non-positional command requests, such as engage, park, freeze, continue, and change modes of operation. Non-positional command requests go through a shared memory queue so that any process may send such requests. The user of the motion base service can distribute the monitoring and control of the motion base between any number of programs appropriate to his or her particular application.

   We have implemented two clients specifically for the Moog Motion Base Controller, one is a GUI and one has a terminal-based interface. Both clients can send any non-positional command request to the motion base. Both clients can request the positional write lock and if successful can move the motion base to requested positions. The GUI Moog Motion Base client can move the motion base based on the position of slider widgets. A "shock absorber" filter is applied to the data from the GUI-based sliders before sending positional requests to prevent "noisy" motion.

2. The **InterSense IS900 position Tracker Service** reads a RS232 serial line and writes the current state of the trackers into DTK shared memory [InterSense]. Clients can read the data from the DTK shared memory segment named "head" to find the position of a person's tracked head. Data for the IS900's wand, joystick and buttons are also available in an analogous manner.

3. The **Hello Service** is a very short example DTK service that is part of the DTK tutorial. The hello service reads from standard input from a terminal and than echoes what was read into a DTK shared memory segment named *hello*. The hello service is only about 40 lines of code, but because it writes to DTK shared memory it is automatically a network-enabled service.

DTK CLIENTS

DTK clients connect to the DTK server using the DTK C++ API library. This API provides a DTK client class that seamlessly implements all of the DTK client/server protocols. A client class object communicates to the DTK server through the **Core DTK Services**. The dtkClient class provides an object that manages the DTK clients connected to a DTK shared memory arena. DTK servers act as blocking readers for the clients, so clients do not have to spin or block to read IP networked data. The dtkClient inherits a DTK shared memory factory that client

programs can use to manage DTK shared memory segments and DTK shared memory queues. DTK comes with a suite of GUI and terminal-based DTK client programs.

THE DTK C++ API

The DTK C++ API library provides a set of small operator-in-the-loop simulation utilities in addition to its client-server and shared memory interfaces. Most of these utilities are C++ class objects; their main objective is to help the user write less code. For example:

- The **Realtime Synchronizer** wraps interval timer system utilities to provide a method for synchronizing running programs with wall clock time (real-time) without incurring unwanted CPU spinning.

- The **DTK Time Class** is a small gettimeofday() wrapper that provides a time offset and output of the system clock. It's very handy for getting wall clock time differences as well as absolute time.

- The **4th Order Runge-Kutta Integrator Class** solves ordinary differential equations with user-varying independent variable (time) steps.

- The **Distribution Function Builder Class** is handy for building one-dimensional distribution functions for performance analysis of simulations.

- The **DTK C++ DSO Loader** wraps the "dl" family of functions, like dlopen(), in order to decrease the code needed to manage a list of loaded DSOs.

- The **DTK Error Message Class** keeps stacks of errors, and sets error message verbosity (spewage) levels.

- The **DTK Sockets Wrapper** is a handy way to write less code when dealing with sockets; it wraps TCP/IP, UDP/IP, and UDP/IP multi-cast sockets.

- The **DTK Base Class** gets class types, and checks an object's validity, which is useful since DTK does not currently throw C++ exceptions.

- The **DTK Read/Write Locks** guarantees a coherent data state of the DTK shared memory. The implementation is OS dependent, so this wrapper provides a compatibility layer.

DGIPF, THE DIVERSE GRAPHICS INTERFACE TO PERFORMER

dgiPf provides the following capabilities:

- Abstractions of hardware input devices, graphic displays, and navigation (locomotion) and interaction techniques, providing a consistent programming interface.

- Automatic generation of both symmetric and asymmetric stereo viewing frusta, either based on the position of a locator device, such as a head tracker, or computed positional data.

- Arbitrary number, size, and orientation of graphic display windows.

dgiPf uses the DTK to:

- Load compiled configuration files, in the form of DSOs, which describe the graphic display, input devices, navigation techniques and interaction styles.

- Access data from input devices, either local or remote, using DTK's "remote shared memory" facility.

- Allow "hot-swapping" of input devices by reloading dtk-server DSOs while a dgiPf application is running.

Since DSOs are loaded at run-time, and their behavior is cumulative, the application itself can be reconfigured by using different sets of DSOs without needing to be recompiled. This allows the same application to run in an immersive virtual environment, such as a CAVE using stereo glasses and a head tracker, or on a desktop using a monitor and mouse, and to be navigated and controlled with different interaction techniques, without modification [CAVE].

DGIPF, THE TOP-LEVEL CLASS

A dgiPf application creates exactly one object of the dgiPf class, which can be considered to be the user's entry-point into the dgiPf system.

The dgiPf class loads and unloads DSO files containing objects based on the dgiPfAugment class. These dgiPfAugment objects describe the dgiPf configuration for graphical display, input, navigation and interaction techniques.

The dgiPf class creates a single dgiPfDisplay object that is configured by the dgiPfAugment objects, which in turn generates graphical output by invoking Performer methods.

The dgiPf class, in order to queue event data, such as from input devices, also creates a single dgiPfRecord object.

The dgiPf class unloads and deletes all objects that it creates, as appropriate.

To keep program complexity to a minimum some of dgiPf's methods are automatically invoked by some of its other methods when required, but only when appropriate. This minimizes the complexity of application programs as well as minimizes errors due to Performer and dgiPf methods not being invoked, or being invoked at the wrong time.

DGIPFAUGMENT, THE CLASS THAT ADDS FUNCTIONALITY

The dgiPfAugment class contains methods to augment dgiPf functionality by means of pre-defined callback functions. The dgiPf object is a factory of dgiPfAugment objects, and calls these objects' callbacks via dgiPf methods. A dgiPfAugment object inherits four virtual methods, any or all of which it may overwrite. The return status from the implemented methods tells the dgiPf object what actions to take with respect to the dgiPfAugment object.

The four virtual methods are:

- prePfConfig(), which is called once, just before Performer forks into multiple processes.
- postPfConfig(), which is called once, just after Performer forks into multiple processes.
- prePfApp(), which is called every frame, just before the Performer scenegraph is traversed.
- postPfApp(), which is called every frame, just after the Performer scenegraph is traversed.

DGIPFDISPLAY, THE GRAPHICAL DISPLAY CLASS

The dgiPfDisplay class contains methods to describe the size, location, orientation and other aspects of the virtual world's graphical display. The default dgiPf coordinate system is oriented so that +X is to the right, +Y is straight ahead, and +Z is up. These right-handed coordinates are referred to as *dgiPf coordinates*. In an immersive environment the boundaries of the tracked system range from -1 to 1 along each axis to form a cube with its origin at the center, referred to as the *dgiPf coordinate cube*. The application can specify a transformation to its own preferred coordinate system, called *model coordinates*, by invoking dgiPfDisplay methods. (See Figure 2.)

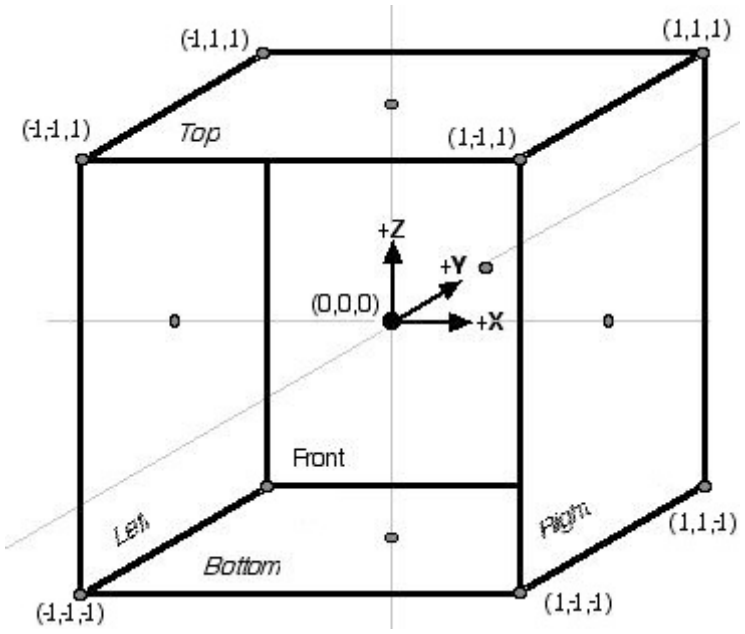


Figure 2, the dgiPf coordinate cube

The display DSO can also set the number of meters per dgiPf coordinate, if applicable to its configuration. This can be used to ensure that objects are displayed at a physically correct size in an immersive environment, an important consideration for some applications such as architectural walkthroughs.

Specifically, the following three methods allow you to orient and scale your virtual world:

- origin() specifies a location and orientation that corresponds to the origin in the dgiPf coordinate system.

- scale() specifies the ratio of dgiPf units to model units. If scale gets bigger, objects will look bigger.

- meters() specifies the number of meters per dgiPf unit. This is usually set in the dgiPfDisplay DSO that describes the display configuration.

Notice that if you are using symmetric frusta, scale() doesn't have any noticeable effect. This is because as scale changes, distance changes at the same rate as the size of the world, effectively canceling each other out. For example, if scale is set to 2, then objects will be twice as far away, but twice their size, resulting in no visual change.

In general, you use the scale() method if you want to fit your data to the display, regardless of the display's physical size, and use both the scale() and meters() methods if you want your data to be displayed at a physically correct size regardless of display size.

The dgiPfDisplay class also creates and manages one or more dgiPfPipe objects, as set by the DSOs:

- The dgiPfPipe class contains data related to Performer pfPipes, and creates and manages one or more dgiPfPwin objects.

- The dgiPfPwin class manages Performer pfPipeWindows, or mapped graphical output displays, and creates and manages one or more dgiPfScreen objects. A dgiPfPwin can be set to stereo or mono mode. The stereo separation is set in the dgiPfScreen class, and can be different for each dgiPfScreen object. (Figure 3).

- The dgiPfScreen class manages graphical display areas within Performer pfPipeWindows. A dgiPfScreen object will create one Performer pfChannel if the dgiPfScreen's dgiPfPwin is in mono, or two if in stereo.

Screens are specified by a size, position and orientation in the virtual word; in an immersive application these positions usually correspond to the locations of the physical display screens.

A screen can be set to have either a "*symmetric*" or "*asymmetric*" view frustum.

- A symmetric view frustum is typically used for desktop displays, when the frustum does not change shape as its viewpoint is moved in position or orientation. A symmetric frustum's **base is moved** to accommodate the viewpoint.

- An asymmetric view frustum has its **base in a fixed location**, and the shape of the frustum is modified to accommodate the viewpoint's changes in position. This type of

frustum is often used in immersive displays, where the viewpoint is connected to a head tracker. The viewpoint's change in orientation has no effect on the frustum's shape.

The *interoccular distance* specifies stereo separation, which is the distance between the eyes in model coordinates. Symmetric frusta also use a *convergence* value to calculate stereo separation. If convergence is set to 1.0, then the stereo images will fuse at the far clipping plane, and if set to 0.0, images will fuse at the near clipping plane. Asymmetric frusta do not need a convergence value, as their images always fuse at the frusta's base.

All of this functionality is contained in the display DSO. The application itself doesn't need to know whether it's running in a CAVE or on a desktop workstation; it generally only uses the methods in the dgiPfDisplay class. The dgiPfPipe, dgiPfPwin and dgiPfScreen classes are configured by the display DSO.

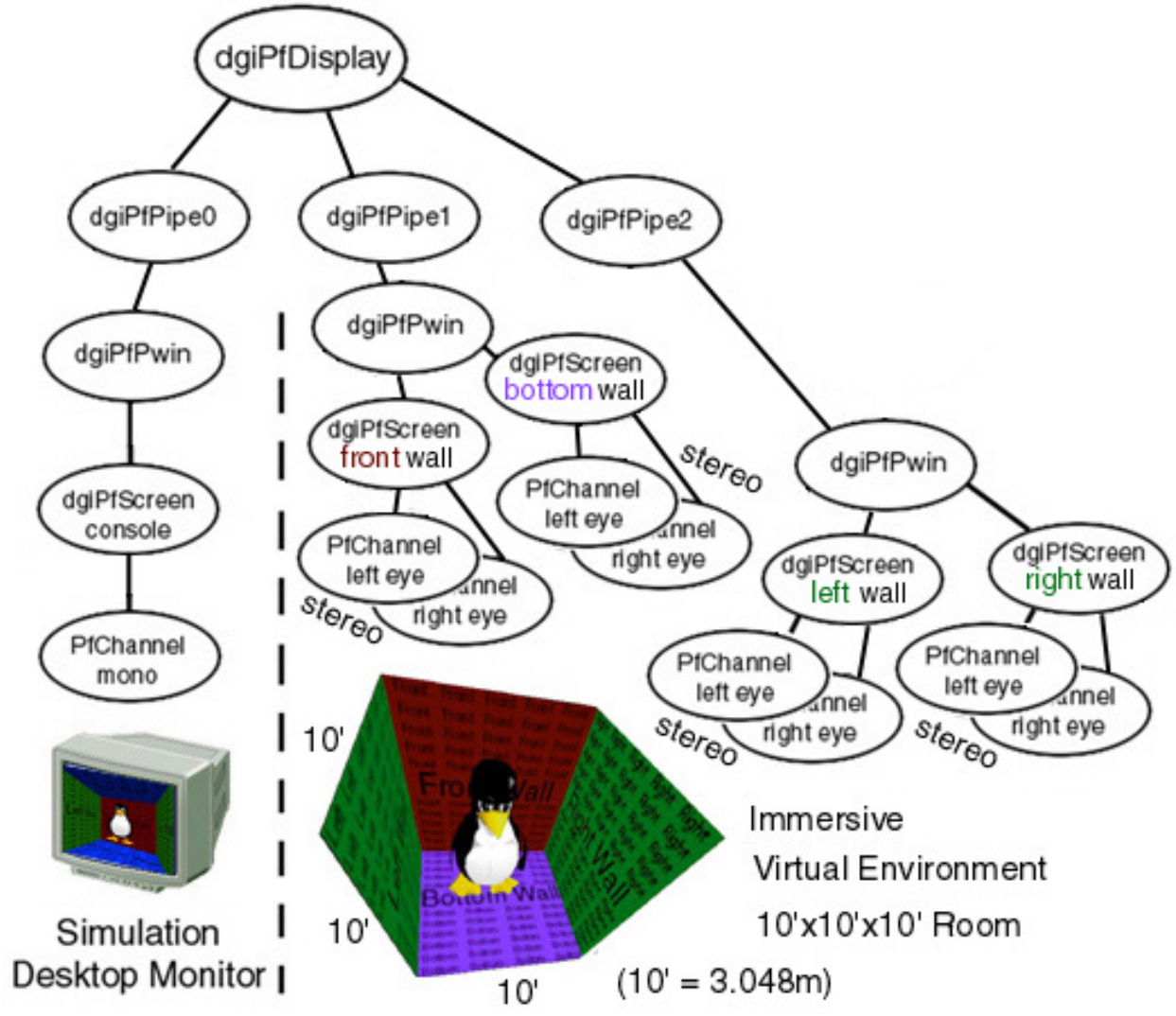


Figure 3, display class hierarchy for Virginia Tech's CAVE

Figure 3 is a graphical representation of the DSO display object used in Virginia Tech's CAVE.

The top node is a single dgiPfDisplay object, which creates and configures three dgiPfPipe objects. The first dgiPfPipe object, pipe 0, creates and configures a single dgiPfPwin object, which in turn creates and configures a single dgiPfScreen object, which ultimately creates a single Performer pfChannel, a mono console display.

The other two dgiPfPipe objects, pipes 1 and 2, each contain a single dgiPfPwin object, configured to display in stereo. Each of these dgiPfPwin objects contains two dgiPfScreen objects, each with an asymmetric frustum, and one per CAVE wall. Since each wall is in stereo, each dgiPfScreen object creates and configures two Performer pfChannel objects.

Our CAVE applications also load an input DSO and a DSO that ties the position of the head tracker to the viewing frusta and stereo parallax computations.

The dgiPfDisplay object creates a small Performer scenegraph. All nodes of the scenegraph are accessible by the application. (Figure 4)

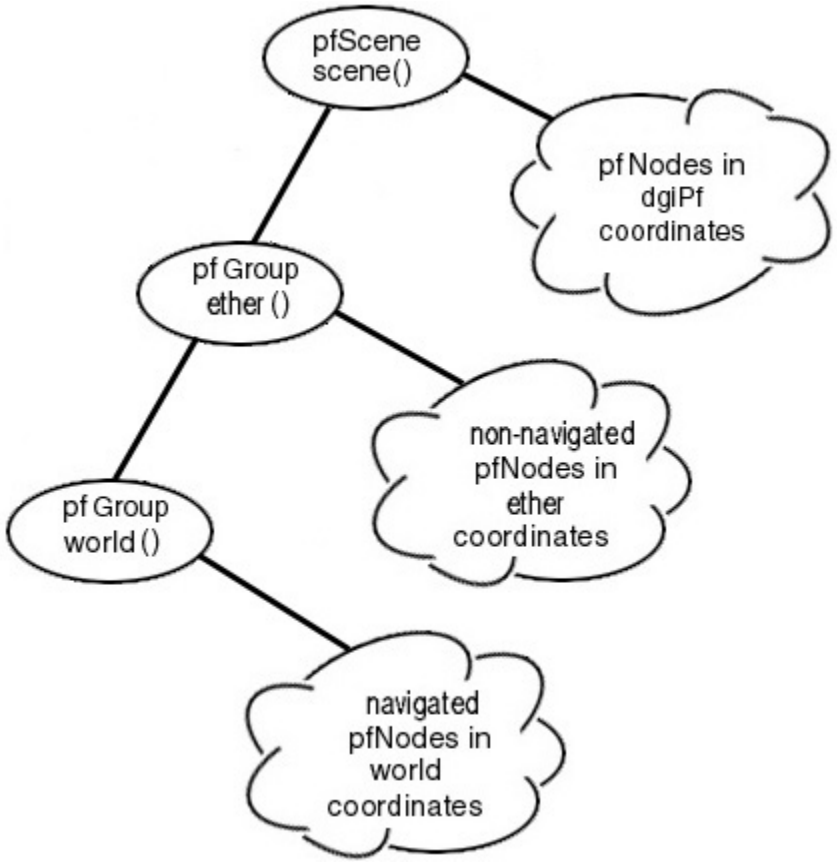


Figure 4, the dgiPf scenegraph

The application should add nodes that use dgiPf coordinates under the root node of the scenegraph. Applications and DSOs can create additional nodes under this node that will always

be displayed with the same size and orientation, regardless of the model coordinate system in use.

The dgiPf object initially creates and places two more nodes under the root node. Directly under the root node is a node containing the transformation from dgiPf coordinates to model coordinates. This transformation is fixed for the duration of the application. Graphical objects move through this fixed coordinate system, much like worlds moved through the ether of pre-relativistic 19th century physics. Applications should create child nodes of this node for objects that use model coordinates, but which will **not** be moved with the navigated world. Examples of such objects might be a sky dome, or a cylinder containing a texture map of a mountain ridge representing an unreachable horizon.

A third node contains the transformation from dgiPf coordinates to navigated model coordinates. Applications should create child nodes of this node for objects that use model coordinates and which will move with the navigated world. Having separate non-navigated and navigated nodes allows dgiPf to transparently support and switch between multiple simultaneous navigational methods.

NEW PERFORMER NODE TYPES

dgiPf supplies new Performer node types; each new type of node is implemented as a class derived from an existing Performer class. The purpose of these new node classes is to increase usability and to allow other methods to automatically modify the node's data. This functionality is accomplished by way of callback functions built into the new node's class.

An example is the dgiPfDCS class, derived from the pfDCS class. It allows a group of files to be loaded, each with its own scale, position and orientation, under a Dynamic Coordinate System (DCS) scenegraph node. The transformation contained in the DCS is updated every frame based on the data referenced by a pointer to a floating point array. This supports a data-driven programming paradigm, useful for animations and physically based simulations.

Another class that uses the data-driven paradigm is the dgiPfToggle class, derived from the pfGroup class. The node's constructor takes a pointer and a bit offset as parameters. When the indicated bit is set, the node is visible; otherwise it's invisible.

INPUT

Input devices are implemented as dgiPfAugment objects, and are loaded as DSOs. The user can implement additional devices, or device types, as dgiPfAugment objects in the same manner, allowing virtually any type of input device to be defined.

dgiPf currently implements four generic device types:

- A *button* is a device that has a value of one or zero, and is represented by a bit. Each button device can have up to 32 buttons. It is implemented in the dgiPfInButton class.

- A *locator* is a device that provides positional information, represented as a location and orientation.. It is implemented in the dgiPfInLocator class.

- A *valuator* is a device that returns an array of floating point values. It is implemented in the dgiPfInValuator class.

- A *keyboard* is a device that returns an X-windows "KeyCode", which represents a key on the keyboard. It is implemented in the dgiPfInKeyboard class.

Any device can be either queued or polled, as specified by the application. A queued device, typically a button or keyboard, queues each change of state. A polled device, typically a locator or valuator, does not queue, but merely maintains a state, which can be read at any time.

Event records contain a snapshot of the data associated with all loaded input devices, and are stored in a circular queue. Each change to the state of a polled input device pushes an event record onto the queue. In this way when an event is queued, it's possible to get the values of all other devices at the moment of the event.

NAVIGATION

A navigation technique is usually implemented as a DSO based on the dgiPfNav class, which converts data from a particular set of input sources into a locational transformation. By encapsulating navigation in a DSO, the following benefits are realized:

- An application gets navigation "for free." That is, by loading a navigation DSO an application's virtual world can be navigated without adding **any** navigation code to the application itself.

- The navigation technique can be changed without modifying the application; all that is needed is to specify a different navigation DSO at run-time.

- The user can choose a navigation technique most appropriate to the particular hardware environment or their needs. For example, a navigation based on wand, joystick, and button data can be used in the CAVE, and the same application program can be navigated with a "glass trackball" on the desktop.

- Multiple navigation techniques can be loaded and the application program, or another DSO, can switch between them at run-time. This allows navigation techniques to dynamically adapt to differing circumstances in the virtual world.

Navigation is implemented by inserting a node in the scenegraph between the non-navigated and navigated scenegraph nodes; the navigated node inherits the transformations of the new node.

CONFIGURATION

Once dgiPf has been installed the user might want to set things up so the desired settings occur by default on different systems. The user might also need to create your own configurations if the ones supplied in the distribution don't meet his or her needs.

DGIPF-CONFIG

dgiPf-config is a small application that can be used to determine information about the user's dgiPf installation. It can tell the user the current version, flags needed for compiling and linking the user's application with dgiPf, and compiled-in values, such as the default DSO directory. Use of dgiPf-config in shell-scripts and Makefiles will increase their portability and compatibility.

Installers are encouraged to put the dgiPf-config program in the default PATH for dgiPf users and developers. There are no other environment variables, other than PATH, that need to be set in order to develop and run dgiPf programs.

DYNAMICALLY SHARED OBJECTS

Instead of using complex interpreted configuration files of unique syntax and questionable generality, DIVERSE, and hence dgiPf, use small compiled C++ DSO programs to configure dgiPf at run-time. DSOs are just loadable versions of dgiPfAugment objects. These allow dgiPf DSOs to access the full functionality of the C++ programming language and Unix operating system resources to create "smart" configurations.

Although all dgiPfAugment DSOs have the same format, we find it convenient to roughly categorize them as:

- Display DSOs, which specify the basic attributes of a dgiPfDisplay configuration. These can be hardware dependent, like a DSO that describes a specific immersive system such as a CAVE, or hardware independent, such as laying out a series of windows and screens on a desktop.

- Input DSOs, which implement one or more input devices. These can be hardware dependent too, such as a DSO that provides input from an InterSense tracker, or hardware independent, such as a desktop slider widget that simulates a tracker.

- Navigation DSOs, which convert input data into navigational data, thus encapsulating a navigation technique.

- Modifier DSOs, which change the attributes of previously loaded configurations. An example is a DSO that changes the characteristics of an interaction technique, or switches between a set of navigation objects, or sets all display windows to mono.

- Group DSOs, which merely load a sequence of other DSOs (which can also be group DSOs) as a way of completely specifying a configuration in one DSO.

An application can load a sequence of DSOs that act in a cumulative manner. Some DSOs require other DSOs to be loaded- for example a navigation DSO that uses the wand and joystick needs the wand and joystick input DSOs to be loaded. Dependency conflicts can be minimized by use of group DSOs.

There are several DSOs provided in the distribution that don't fit into any obvious category. Two in particular deserve some elaboration:

- caveSim and caveSimInput implement a virtual CAVE environment. While not a good interface for desktop use, it can be very handy for debugging and developing an immersive application on a desktop. The interaction techniques are based on, but not copied from the VRCO CAVElib™ Simulator [CAVElib]. The caveSimInput DSO creates and manipulates the head, wand and joystick input devices. The caveSim DSO assumes that these input devices have already been created, and just uses their values. Both DSOs create dgiPfToggle objects. The visibility of these objects, representing the head and wand, a gnomon, axis, frame, and a coordinate cube in all three coordinate system can be toggled via keypresses. The caveSimInput DSO allows the position of the head and wand, and the values of the joystick and buttons to be modified via combinations of keypresses. A "jump" mode allows an "out of body" view of the virtual world from almost any position. (Figure 5)

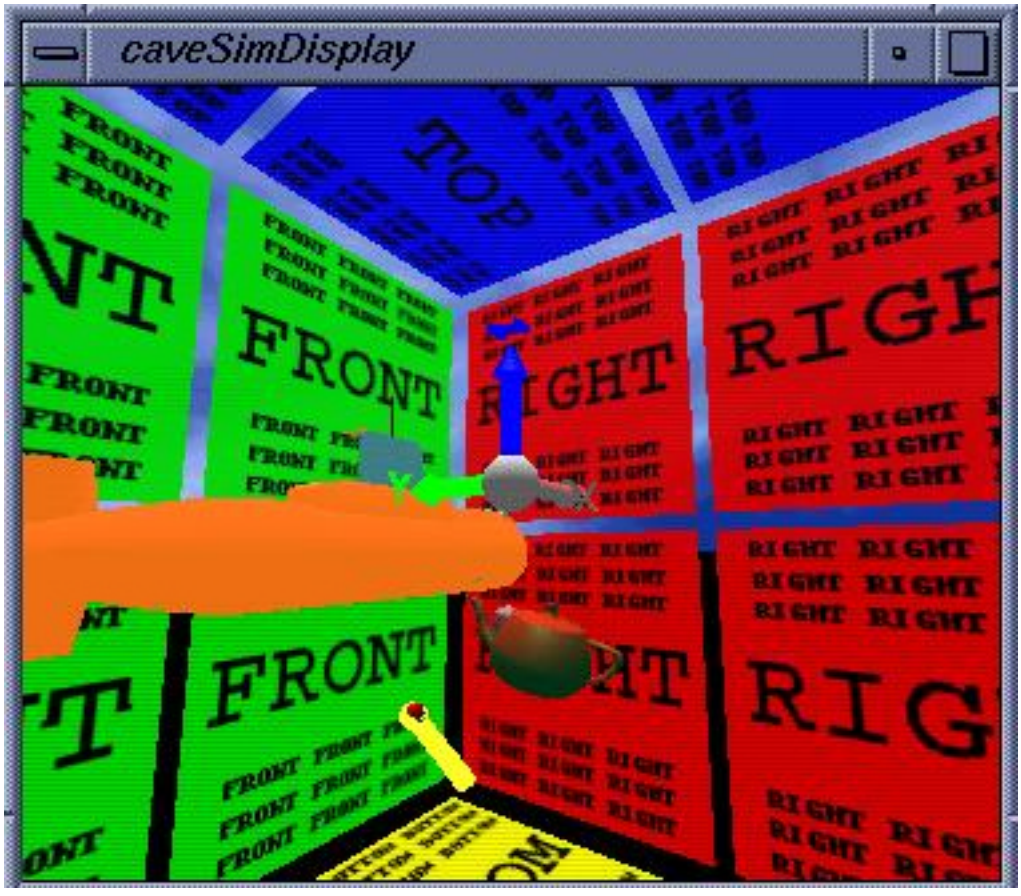


Figure 5. a "jumped" view in the CAVE simulator

- iconSleepKeyboardMod causes the application to sleep if all of its windows are unmapped (iconified, or on a different virtual desktop). The usual Performer behavior is to soak the processor in a tight loop. This can be useful when you want to run, and switch between, several concurrent graphics programs.

RUNNING AN APPLICATION

dgiPf can use environment variables to specify which DSOs to load, and the directories to find DSOs and model files. These only need to be set if the values, as set by the application or as returned by dgiPf-config --env, do not have the desired values.

Running a dgiPf application with default DSOs requires no special commands or environment variables. For example, to run the helloWorld example program just type:

```
./hello
```

A simple script like this might be handy for running an application with a specific set of DSOs and command-line options. For example:

```
#!/bin/sh
export DGIPF_DSO_FILES="desktopCaveSimGroup"
exec hello
```

would run the program hello on a desktop using the keyboard and mouse to supply input to the CAVE simulator. And this:

```
#!/bin/sh
export DGIPF_DSO_FILES="vtCaveGroup"
exec hello
```

would run the program hello in the Virginia Tech CAVE. The head and wand input would be via the DTK server, and the position of the view, used to calculate the viewing frusta, would be tied to the head input segment.

EXAMPLE PROGRAMS

dgiPf comes with dozens of programs; each one is designed to demonstrate a specific feature of dgiPf. Source code examples of both standalone programs and loadable DSOs are included. Each set of example programs is in its own directory, and has a Makefile.

Figure 6 is a small but complete program that illustrates how dgiPf augments the Performer API. dgiPf-specific code is in *italics*. This program can be run on a desktop or in a CAVE, with various navigation techniques and other accessories just by running it with different sets of DSOs.

DIVERSIFLY

diversifly is a program that allows non-programmers to load and navigate through model files with optional global transformations; it is the DIVERSE analogue of Performer's perfly. Like any other DIVERSE program, display format, navigation and interaction techniques can be selected by DSOs. diversifly has become popular as an easy way to see how a model will look in various display settings, such as CAVEs or desktops. It has also proven useful as a generic program that can be used to test new interfaces that are encapsulated as DSOs.

```
#include <dgiPf.h>

int main(void) {
        pfInit();
        dgiPf app;
        pfConfig();
        app.display()->world()->addChild(pfdLoadFile("model.pfb"));
        while(app.state & DGIPF_ISRUNNING)
                pfFrame();
        pfExit();
        return 0;
}
```

Figure 6, a simple, useable dgiPf program

CASE STUDY- SHIP-BOARD CRANE SIMULATOR

Most of today's computationally intensive computer applications or High Performance Computing (HPC) applications typically consist of one computer program that may be broken into many processes, or threads. A typical HPC application program can be parallelized by using HPC compiler tools and/or APIs, like Message Passing Interface (MPI) [MPI] [MPI/RT]. Each process runs in a predetermined synchronous and reliable fashion; for an application to be successful, none of the processes can fail. Moreover, all IPC must be reliable and all processes must synchronize at regular intervals.

We have implemented a different HPC application paradigm. It is a real-time interactive HPC application consisting of many programs running on many computers. The paradigm tolerates faults and requires neither reliable IPC methods nor regular process synchronization. The result is an increase in the performance of each of the programs and of the total system. This paradigm is general and can be used to model and simulate real-time, interactive, person-in-the-loop, dynamical systems.

The crane ship VR simulator is an operator-in-the-loop simulator in a CAVE. A motion platform is used to provide motion queuing to the user (operator) of the simulator. The user is be immersed in a virtual environment, with a four-wall CAVE (which has three vertical walls and a floor), and four-channel sound. The user is strapped into an operator seat. Attached to the operator seat is a control console with joystick crane controls, which the user can use to operate the crane. The motivation for the development of this simulator is to develop and test crane control methods for crane operation at high sea wave states, though this simulator can be used for other operator-in-the-loop virtual prototyping studies, and ship-based crane operator training. Because we built the crane ship VR simulator using DIVERSE it can run on a diverse combination of hardware platforms, hardware emulators, and software emulators, other than a CAVE with a motion base, such as a laptop with GUI sliders as operator joysticks, and no motion base.

The structure of the software system of the crane simulator is a cluster of cooperating programs. They are coupled to each other by data structures that retain current physical state information of the system in shared memory from which the programs can read and write. This design facilitates a modular structure for the system. All the programs in the simulator run asynchronously with respect to all other programs. Some programs are simulating physical dynamical models whose cyclic solver is synchronized to real-time; others use the current shared physical state information from the shared memory. Consequently, the need for process synchronization is replaced indirectly by the sampling of physical state values that are being simulated (modeled) in real-time. If the coupling between two processes is large, then the shared data is queued and the reading process interpolates values for the given time (or another state variable) that it is reading, so that continuity in the coupling may be better preserved.

DTK IS SIMULATOR GLUE

The development of this crane simulator is driving the need for innovative HPC software tools like DIVERSE. DTK is the "glue" that ties together all of the programs in the simulator. The DTK server manages the inter-process shared memory, provides an interface for other programs to serial hardware devices via shared memory, and a seamless interface to IP networked shared memory. All programs in the simulator use the DTK C++ client API. The DTK server is central to the system in that it is the only program required to be running at all times for the simulator to be operational. All of the other programs, or modules, can be developed and run independently of most other modules. Modules can be emulated, started, and stopped without corrupting other running modules. When the system is running in a steady state, the primary IPC mechanism used is inter-process shared memory. DTK extends inter-process shared memory to Internet remote shared memory without additional user coding, so that the system may be distributed on many computers when necessary.

The largest difference between this programming paradigm and one using MPI is that the programs distributed in this system typically run asynchronously. The network IPC methods in DTK are usually unreliable (i.e., UDP/IP). However, reliable TCP/IP is used for changing the system configuration, like adding an observer to the system or communicating discrete events, such as turning on a light. The UDP/IP methods are used for transferring most of the network information, such as when a model of a hydraulic cylinder is continuously expanding and contracting due to an operator's input. The changes in the cylinder's length are being fed to the network continuously at a regular rate. If a cylinder length network IP packet is lost, there will not be a need to send again that information because, in most cases, the next cylinder length packet will come before a replacement IP packet can arrive. This method is analogous to the way real systems interact. For example, if you blink your eyes, the light that your eyes did not receive is not sent again, but instead the latest light information is sent to your eyes when you reopen them.

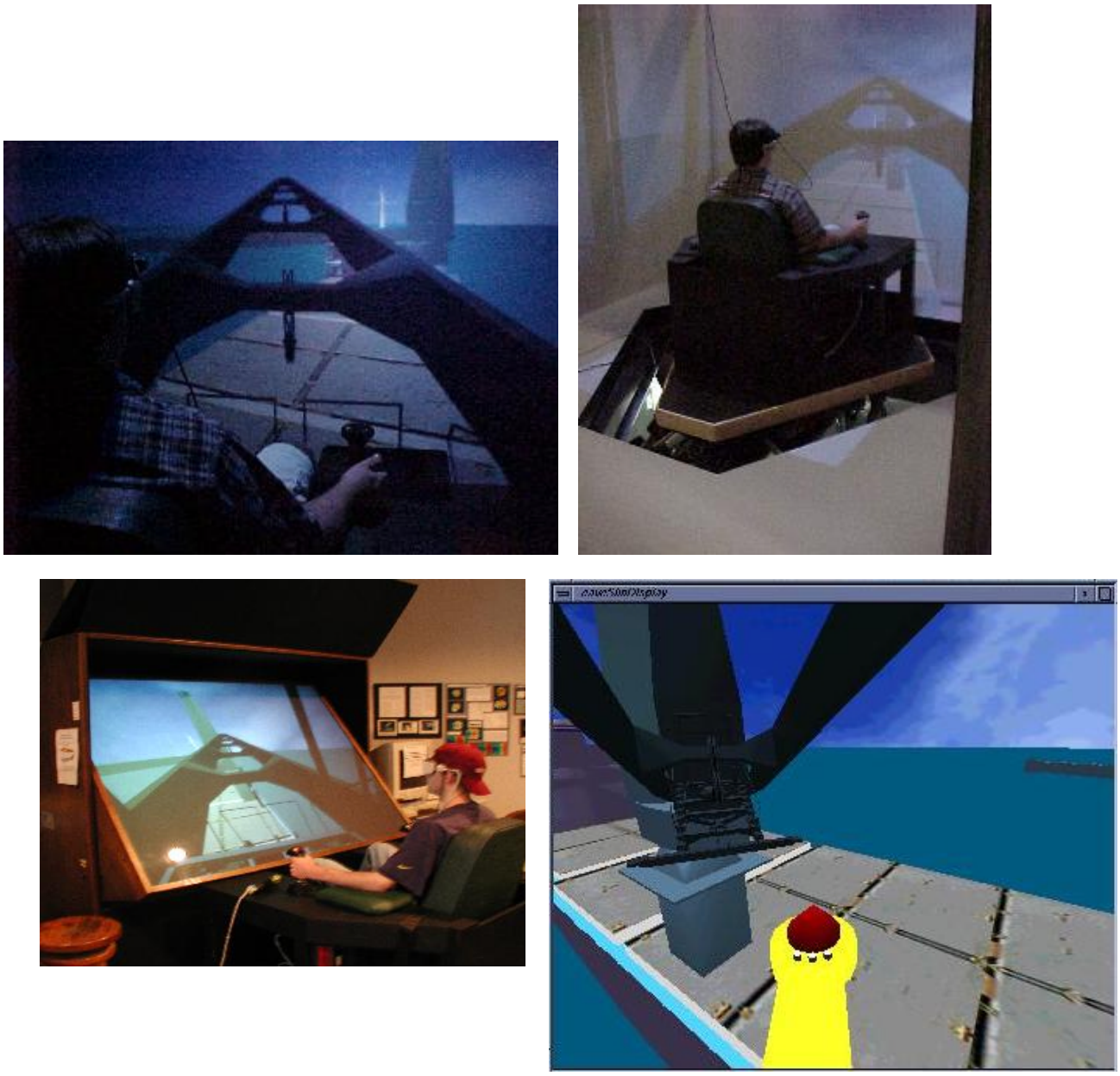


Figure 7, pictures of the ship-based crane simulator

The top two pictures in Figure 7 show the ship-based crane simulator running with a CAVE and motion base. The bottom two photos show the ship-based crane simulator running with an ImmersaDesk and in a window on a desktop.

The simulator programs contain DTK client objects that enable the making of DTK shared memory objects that, in turn, enable communication between the programs in the system. The code in the programs may be written in any combination of languages that may be linked with C++ objects, which includes FORTRAN and C

Most of the programs in the system, after being initialized, run in a cyclic steady state simulation loop. For this case the program's code may be split into two logical functions (subroutines or methods), "initialization" and "cyclic advance." In the case of a C++ implementation a class constructor may be used for initialization. A possible third function can be "closure" for the removal of shared computational resources that may have been allocated when the program runs, so that the program may call this function before exiting. We refer to the code of these functions as "task functions." The writing of the task functions does not require the use of the DTK API. For programs that must run in real-time, the real-time synchronizing code *is not* included in these task functions, but is introduced outside of these task functions, so that the code of these task functions may be used on other platforms, in other applications and testing the code is easier. The task functions provide enough output variables so that any needed changing system state information (for example position, angular acceleration, and time) can be gotten from the output variables. The task functions provide enough input variables so that all changing inputs are passed to the task functions. The code of these task functions may preserve the dynamical state of its task or the dynamical state of its task may be passed to the task functions as a combination of input/output variables. The later case may be more resource (CPU and memory usage) efficient in some cases. When possible the cyclic advance task function should allow the user to specify the time to advance to. Configuration files may be read by the initialization task function.

TESTING MODULES

Simple test programs that use these task functions are an important part of the development of the simulator system. They provide a verification of whether a said module is really a module with a clearly defined interface in addition to independently verifying the model that the module is simulating. When working with a diverse group of developers this is of the utmost importance. Simple test programs just print variables to standard output; their inputs are computed in the simple test program (for example from a sine function with a reasonable scaling period). Simple test programs are written in C or C++, though the modules that are being tested may be written in C, C++, or FORTRAN. Simple test programs are written in one source file that does not include any of the source code of the task functions.

Pseudo code for testing the wombat400 dynamics module is shown in Figure 8. The two task functions being tested are: wombat_init() and wombat_advance(). This particular example passes the wombat state variables in and out and does not keep a copy of the state variables in the code of the task functions. This saves having to copy variables in the task functions, but assumes that the user will not change them between wombat_advance() calls.

When modules can be tested using the form of the above pseudo code, they can then be easily integrated into the simulator system and run with simulation time synchronized with wall clock time (real-time). It is also preferable that the requested simulation time be able to advance with a variable time step, even if the time step is restricted to some small range of values.

```
time  data type   t = 0.00
state data type   x
input data type   i

wombat_init(time, i, x)

while 1 {

    // set up next input value i.
    i = sin(0.07 * t)

    // advance the requested time
    time = time + 0.01

    // This advances the wombat dynamics time and
    // sets input to i.  The state I/O is x.

    wombat_advance(time, i, x)

    print(time, i, x)
}
```

Figure 8, pseudo code for testing the wombat400 dynamics module

LIST OF SIMULATOR SYSTEM MODULES (PROGRAMS)

Most of the simulator system modules are separate programs that are connected to the other modules by DTK remote shared memory. The crane ship VR simulator system modules are:

- The **Simulation Launcher** is a shell script that runs all the needed programs for this simulator on a given hardware platform.
- A **DTK server** provides device I/O and DTK shared memory for interconnecting simulator modules.
- **Render** is a dgiPf program that draws to the visual display.
- **Wave** computes ocean wave height and ship's x and y position relative to the ocean grid.
- **LAMP** ship hull dynamics model, from SAIC that is written in FORTRAN [LAMP].
- **Fun Filter** visual display and motion base relative motion filter.
- **Crane Model** crane dynamics model.

- **Crane Controller** filter operator input to crane model
- **Motion Base Controller** uses the DTK server, a GUI client program to send non-position commands, and the **Fun Filter** to feed position commands to the motion base.
- **Sound** provides simulation-activated sound to the operator.

FUTURE DIRECTIONS

We currently have plans to extend the DIVERSE project to include new modules, extensions to existing modules, and more interfaces to other existing packages.

DTK

We are currently extending DTK to include support for collaborative tools- these are tools that will allow collaborative applications to be more easily written. Extensions will be made to DTK's remote shared memory to support remote shared objects, and remote shared events. Remote shared objects will be used to support such things as model file loading, unloading and manipulation. Remote shared events will be used to facilitate reliable state changes to all collaborative participants.

In support of the ship-crane project, a motion queuing filter will be written. The filter, a special case of navigation, will receive input from various sources and from them modify a scenegraph node and generate commands to a motion platform. The motion commands will provide physical forces to the user to increase the sense of immersion. Washout will automatically be provided to optimize motion queuing flexibility. The graphics display will be modified to accommodate this washout, as well as compensate for the fact that the motion platform might move independently of the display screens.

New hardware devices will be supported, and new hardware emulators will be written:

- We are in the process of integrating a Phantom haptic device into DIVERSE [Phantom]. VRPN is being used to provide the PC/Windows to Unix interface [VRPN].
- We will be writing a driver for a network-enabled PDA (Portable Digital Assistant, a PalmPilot™ or Pocket PC™, for example), which can act as a high-level data-rich interaction device. This makes possible a wide set of complex interactions such as menus, buttons, sliders, drawing canvasses, and textual displays. During a collaborative VE session the PDA can be used to upload and download data such as object properties, remarks from other participants, and system messages. Attaching a position sensor to the PDA creates even more potential interaction techniques, such as a data probe for virtual objects.
- Xwand is a DIVERSE client program under development that emulates a desktop mouse using the CAVE wand device commonly found in immersive environments. Xwand facilitates easy interaction with regular desktop windows, which might contain items such as web browsers, menus, buttons and sliders; these windows can therefore be

incorporated into a virtual environment. Xwand will allow a desktop interface to be transparently ported to an immersive environment.

A single process, interrupt driven, multithreaded version of the DTK server is being written. The interface to this new version of the server will be identical to the current server's interface. The multithreaded DTK server may use less overall CPU time and have less service latency time due to the removal of the select() system call. This gain is mitigated by the overhead of thread management overhead, lower scalability, and increased code complexity. We suspect that the results of performance comparisons will vary depending on the operating systems (OS) used, DTK services loaded, and how the DTK server is used. Overall, a large increase in performance is not expected.

DGIPF

To better support the visualization of 3-D data sets we will be creating interfaces to other visualization packages. We plan to first write interfaces to:

- VTK, or the Visualization ToolKit [VTK]
- OpenGL Volumizer [Volumizer]

To support networked collaboration, we will be writing tools to help support the creation of graphical collaborative systems. A collaborative application will allow multiple users on multiple machines to share data sets and interact with them in the same virtual world. A simple example of a collaborative tool might be a new type of scenegraph node that can be transparently shared amongst users. Another example could an "awareness tool," such as DSOs that displays status information about the shared virtual world and its inhabitants.

We are also currently in the process of writing more navigation and simulation techniques. Navigation techniques will include features such as collision detection. Simulators will be intuitive and anthropomorphic, better simulating a CAVE experience. All of these will take the form of DSOs, so they will be readily available to any application.

We are continually adding to the diversifly program. We will provide a GUI (Graphical User Interface) front-end, so it will be easier for users who aren't familiar with Unix to access immersive environments.

DGL

A new DIVERSE module is being written which will augment the OpenGL interface in the same manner that dgiPf augments the Performer interface. DGL will allow us to port DIVERSE to many new platforms that are not currently supported by Performer.

DGL-OI

OpenGL does not support a scenegraph. To remedy this deficiency a new module called DGL-OI will be written to augment the DGL interface to allow Open Inventor™ scenegraphs to be loaded [Inventor].

SUMMARY

By way of the Navy Crane-ship implementation example, DIVERSE has been shown to be an effective platform for facilitating the implementation of device-independent VE and graphic systems. There are several features of the DIVERSE design that contribute to this effectiveness:

- DIVERSE is highly modular, and you only use the parts you need. This is demonstrated at the highest level by DIVERSE's compartmentalization into a non-graphical component, DTK, and a graphical component, dgiPf. Modularity also facilitates the creation of new components, such as ones that augment alternative graphics APIs.

- Modularity is taken one step further by the use of DSOs, which enable functionality to be transparently inserted into an application at run-time. This feature allows navigation and interaction techniques to be developed independently of the application, which fosters software reuse and inter-application consistency.

- DSOs are also used to specify hardware configurations for graphical display and input devices. Since DSOs are loaded at run-time, an application can run on any supported platform without modification. This flexibility also allows application development and testing to be performed on relatively inexpensive desktop units. DSOs also allow new hardware devices and configurations to be readily integrated into existing systems.

- In order to maximize existing code re-use, and to *keep out of the programmer's way*, DIVERSE augments existing APIs instead of inventing new ones. As much as possible, DIVERSE doesn't impose a particular programming paradigm, allowing developers to determine for themselves the optimal design for their particular needs.

Last, but certainly not least, since DIVERSE is free (open-source), users can share their work with others without burdening them with licensing issues. Users are also free to modify and expand DIVERSE to suit their particular needs, now or in the future [OSBenefits].

We encourage anyone writing software with DIVERSE to contact us if they have any software, be it nifty applications, examples, DSOs or DTK services they wish to contribute to our distribution.

We also encourage all current and potential DIVERSE users to contact us with any questions or comments they might have, or visit our web site at www.diverse.vt.edu. We can be reached by email at diverse@vt.edu.

REFERENCES

[Aviation] Aviation Week & Space Technology, January 17, 1983. Special issue on visual simulation.

[Carter] John B. Carter, Dilip Khandekar, and Linus Kamb, Distributed shared memory: Where we are and where we should be headed, 1995

[CAVE] Cruz-Neira, C., Sandin, D.J., DeFanti, T.A., Kenyon, R., and Hart, J.C. The CAVE, Audio Visual Experience Automatic Virtual Environment.

Communications of the ACM, June 1992. pp. 64-72. The CAVE is available through FakeSpace Systems, at http://www.fakespacesystems.com/products/cave.html

[CAVElib] VRCO's CAVElib software is documented at http://www.vrco.com/products/cavelib_main.html

[DVR] A long list of VR and distributed VR software: http://www.diverse.vt.edu/VRSoftwareList.html

[GPL] The GNU GPL and LGPL licenses are documented at http://www.gnu.org/philosophy/license-list.html

[Hartling] P. Hartling, C. Just, C. Cruz-Neira, Distributed Virtual Reality Using Octopus, IEEE Virtual Reality 2001, Yokohama, Japan, March 13-17, 2001

[HMD] Sutherland, I. A Head-mounted Three Dimensional Display. Proceedings of the Fall Joint Computer Conference, 1968, pp. 757-764.

[InterSense] Information about the InterSense line of trackers can be found at http://www.isense.com/

[Inventor] Open Inventor is documented at http://www.sgi.com/software/inventor/

[IRIX] Silicon Graphics documents its IRIX Operating System at http://www.sgi.com/software/software.html#IRIX

[LAMP] The LAMP software was written by SAIC, at http://www.saic.com/

[Lehner] V. Lehner, and T. DeFanti, Distributed virtual reality: support remote collaboration in vehicle design, IEEE Computer Graphics and Applictaions, 1997, vol 17, no. 2, pp. 13-17.

[Linux] GNU/Linux is available from many sources. A good starting point is http://www.linux.org/

[Moog] Information about the Moog Corporation and its products are at http://www.moog.com/

[MPI] MPI is documented at http://www-unix.mcs.anl.gov/mpi/index.html

[MPIRT] MPI/RT is documented at http://www.mpirt.org/

[OpenGL] OpenGL is supported by the OpenGL Foundation, at http://www.opengl.org/

[OSBenefits] Benefits of Open Source Software, http://www.theaceorb.com/product/benefit.html, The ACE ORB, Commercially Supported by Object Computing, Inc.

[Performer] OpenGL PerformerTM, white paper at http://www.sgi.com/software/performer/presentations/perf_wp_clr.pdf http://www.sgi.com/software/performer/.

[Phantom] The Phantom haptic device is sold by Sensable technologies, at http://www.sensable.com/haptics/haptics.html

[Schacter] Bruce Schacter, and Narendra Ahuja, A History of Visual Flight Simulation Computer Graphics World, May, 1980. pp. 16-31.

[Stevens] W. Richard Stevens, UNIX Network Programming, Volume 2, 2nd Edition, Prentice Hall PTR, 1999. Page 321 (shm/test2.c).

[Volumizer] OpenGL Volumizer, http://www.sgi.com/software/volumizer/

[VRPN] VRPN is available at http://www.cs.unc.edu/Research/vrpn/

[VTK] The Visualization ToolKit (VTK), http://www.kitware.com/