John Kelso
kelso@vt.edu
Department of Computer Science
Virginia Tech
Blacksburg, VA USA

Steven G. Satterfield
steve.satterfield@nist.gov
National Institute of Standards and Technology
Scientific Applications and Visualization Group
Gaithersburg, MD USA

Lance E. Arsenault
lanceman@vt.edu
Department of Computer Science
Virginia Tech

Peter M. Ketchan
peter.ketcham@nist.gov
National Institute of Standards and Technology
Scientific Applications and Visualization Group

Ronald D. Kriz
rkriz@vt.edu
Department of Engineering Science and Mechanics
Virginia Tech

DIVERSE: A Framework for Building Extensible and Reconfigurable Device-Independent Virtual Environments and Distributed Asynchronous Simulations

Abstract

We present DIVERSE, a highly modular collection of complimentary software packages designed to facilitate the creation of device-independent virtual environments and distributed asynchronous simulations. DIVERSE is free/open source software, containing both end-user programs and C++ application programming interfaces (APIs).

DPF is the DIVERSE graphics interface to OpenGL Performer. A program using the DPF API can run without modification on platforms ranging from fully immersive systems such as CAVEs to generic desktop workstations.

The DIVERSE toolkit (DTK) contains all the nongraphical components of DIVERSE, such as networking utilities, hardware device access, and navigational techniques. It introduces a software implementation of networks of replicated noncoherent shared memory. It also introduces a method that seamlessly extends hardware drivers into interprocess and Internet hardware services.

We will describe the design of DIVERSE and present a specific example of how it is being used to aid researchers.

1 Introduction

We introduce DIVERSE (Device-Independent Virtual Environments—Reconfigurable, Scalable, Extensible), a highly modular collection of complimentary software packages, containing both end-user programs and C++ application programming interfaces (APIs), designed to integrate distributed asynchronous simulations with heterogeneous virtual environments (VEs) (Arsenault, Kelso, Kriz, & Das Neves, 2001).

DIVERSE is modular: you only have to use the parts that you need, and you can omit the parts that you don't need. The DIVERSE graphics interface to

Presence, *Vol. 12, No. 1, February 2003,* 19–36

```
#include <dpf.h>
int main() {
   dpf *app = new dpf ;
   app->config();
   app->display()->world()->
      addChild(pfdLoadFile("model.pfb"));
   while(app->state & DTK_ISRUNNING)
      app->frame();
   delete app ;
   pfExit();
   return 0;
}
```

Figure 1. *A complete DPF program.*

Performer (DPF)[1] is a module of DIVERSE that augments OpenGL Performer to facilitate the creation of device-independent graphical applications.

Figure 1 is an example of a complete DPF program that uses the DPF API to load a model file. By loading various dynamic shared objects (DSOs) at runtime, the user of this program can

- run with one or more immersive, or nonimmersive graphical display systems, or desktop simulators of immersive environments, such as a CAVE, RAVE, ImmersaDesk, head-tracked head–mounted display (HMD), desktop or laptop (Fakespace Systems, 2002);
- dynamically load/unload and switch between any number of navigational and interaction techniques;
- use input devices, either real or virtual, local or remote; and
- access the functionality of other APIs such as math packages, sound servers, and simulation tools to animate and enhance the virtual environment.

The code in figure 1 is basically a small Performer program. The configuration code typical to a Performer application has been removed, and seven lines of DPF code, in bold, were added.

The DIVERSE toolkit (DTK) contains all the nongraphical components of DIVERSE; it is not a virtual reality toolkit per se. DTK is a standalone package in its own right and can be used to completely implement distributed asynchronous simulations. DTK contains

- a software implementation of replicated noncoherent shared memory networks using Internet protocol (IP) networking;
- an extensible server that provides service to and from hardware peripheral devices;
- a C++ client API; and
- wall-clock time process synchronization, and other standard system library wrappers that aid in the development of distributed dynamical simulations.

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

1.1 Motivation

DIVERSE was not created as a research project; it grew out of a real-world need to provide our users with a highly reconfigurable device-independent system to support the immersive display of data from asynchronous distributed simulations.

We are fortunate to have access to a wide range of immersive and nonimmersive systems. We desired a software solution that would allow a user to run an application, unmodified, on all of these systems.

We also have a MOOG 2000E Six-DOF motion platform in the floor of our CAVE, and desired a system that would allow us to provide motion cuing by way of a robust collection of cooperating asynchronous networked processes.

We desired a free and open source system that could be readily shared, a system that would encourage the growth of a mutually supportive user community that leaves no one out due to financial constraints.

It was also recognized that high-end systems such as

1. In previous versions, DPF was abbreviated as dgiPf.

the CAVE are not cost effective debugging and development platforms. For reasons of both efficiency and convenience, users seem to prefer to spend most of their time in their offices and labs. Only when they perceive a benefit from running their software in an immersive system will they do so. Furthermore, many members of our VR community are not programmers; they develop content using their tools of choice and hope to use immersive systems to visualize this content without having to write program code. Therefore, the software we develop should

- allow an application to be run, unmodified, on all supported platforms;
- support application-independent interfaces optimized for each platform (for example, a desktop interface can be used on a desktop system, and an immersive interface can be used in an immersive system, without needing to modify the underlying application);
- provide emulators of immersive systems to support development and debugging of immersive systems on nonimmersive systems:
- promote the creation of reusable components to facilitate experimental studies, the evaluation of hardware and software interfaces, and the creation of networked simulations; and
- include tools that allow nonprogrammers to display and interact with their data.

These goals imply that our software should automatically handle all of the details of graphics display and device access. The goals also imply that our software should have the ability to support the separate development of applications and user interfaces. In addition to these goals, we added the following design directives.

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. This also implies that the software should be robust and work at all times.
2. *Stay out of the user's way:* This criteria also means to stay out of the application programmer's way.

   We 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. Because the second directive fosters innovation and flexibility, we tend to favor it over ease of use. To us, "easy to use" also implies "easy to install" and "easy to debug."

These criteria lead to specific design features:

1. *Highly modular design:* A module is a collection of one or more interchangeable components. By using a highly modular design we are able to achieve
   - *Selectivity:* The programmer needs to use only the parts of DIVERSE that are relevant to their application. The end user needs to use only the parts of DIVERSE that are relevant to the application's specific execution environment or desired modes of behavior.
   - *Interchangeability:* Modules and components interact with each other's modules via well-defined interfaces. This allows the abstraction of such things as hardware devices and navigational techniques.
   - *Flexibility:* Modules and modular components make DIVERSE easy to extend and reconfigure. This also applies to applications that use DIVERSE. 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 modules and instances without requiring program modification.
2. *Augment; don't replace:* DIVERSE augments and uses existing software libraries instead of creating

new nonstandard implementations. This allows us to take advantage of the quality software efforts of others and focus on our specific goals. It also often makes it easier to port existing applications into DIVERSE, as they don't have to be completely rewritten. For example, as shown in figure 1, DPF augments Performer.

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, but will require different system-specific configurations to be loaded at runtime.

1.2 Availability

DIVERSE is available for download from the DIVERSE homepage at `http://www.diverse.vt.edu/`. Included with the distribution are the source code, documentation, utility and application programs, as well as dozens of example programs. DIVERSE currently runs on all platforms supported by Performer: SGI IRIX and GNU/Linux systems (Silicon Graphics, Inc., 2002a; Linux Online, 2002). Multiple IRIX binary types are supported (o32, n32, and 64).

DIVERSE is free/open source 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) (Free Software Foundation, 2002).

1.3 Other Related VR Systems

Before deciding to embark on the expense and toil of creating our own VR system, we surveyed available VR systems to see if one would fit our needs. If we had found one, we would have used it. Since DIVERSE was first released, we have continued to survey other VR packages, but as yet none have met all of our criteria, which are

- a free and open source application domain-independent API, with tools for nonprogrammers, to facilitate the creation of immersive and nonimmersive VEs and asynchronous distributed simulations;
- highly flexible, modular, extensible, and reconfigurable, allowing applications to run unmodified on all supported platforms; and
- able to easily use other packages, and be used by other packages.

Many excellent products exist, such as the CAVElib (VRCO, 2002), dVise (PTC Corporation, 2002), WorldToolKit (Sense8, 2002), Vega (MultiGen-Paradigm, 2002), MR Toolkit (White, 2002), AVOCADO (Tramberend, 1999), SVE (Kessler, Bowman, & Hodges, 2000), and RAVEL (Kessler, Hodges, & Ahamad, 1998), but none meet our needs because it is not clear that they are free, either in the sense of cost or of redistribution.

Other packages, such as MAVERIK (Hubbold, Dongbo, & Gibson, 1996), Bamboo (Watsen & Zyda, 1998), AVIARY (Snowdon & West, 1994), and NPSNET (Macedonia et al., 1995) meet our distribution criteria, but we feel they would be too difficult to modify or extend to suit our purposes, or because they address a specific problem domain.

Although VRJuggler (Cruz-Neira, 2002) currently meets our distribution criteria, at the time of our survey in July 1999, it had not yet been released open source. When VRJuggler was OpenSourced in January 2000, we decided to continue DIVERSE development because VRJuggler, with its callback architecture, takes the "`main`" loop away from the user, which we felt was not adequately staying out of the user's way.

An additional shortcoming of all the surveyed packages was that, although they addressed immersive graphical environments, none of them met our needs for distributed asynchronous simulations, and it wasn't clear how readily they could be extended to meet this need.

1.4 DIVERSE Modules

DIVERSE currently consists of two main modules: DPF is a module that implements graphic display output and graphical user interactions in VEs, and DTK

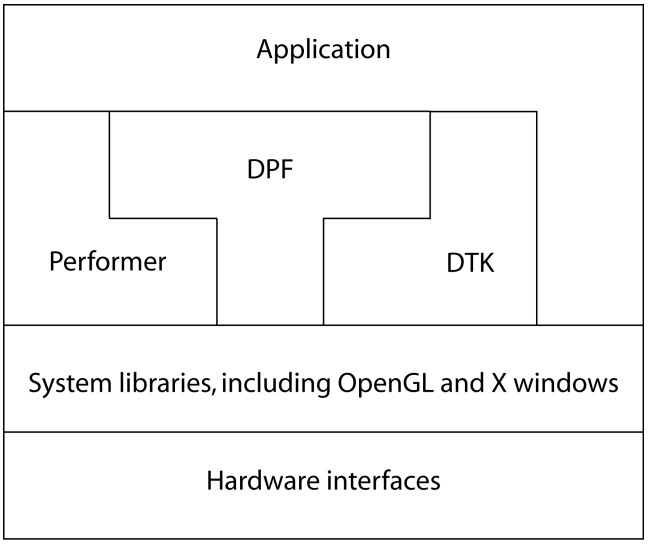

Figure 2. *DPF Library Dependencies.*

implements all nongraphical tasks such as peripheral hardware services, networking, navigational techniques, and nongraphical user interactions. The segregation of DIVERSE into graphical and nongraphical modules allows new graphical APIs to be incorporated into DIVERSE as needed, and simulations that require no graphical output do not need to use any DIVERSE graphics modules.

DPF adds immersive and nonimmersive graphics to simulations by augmenting OpenGL Performer, a high-level scenegraph-based graphics API built using OpenGL (Silicon Graphics, Inc., 2000, 2002c; OpenGL.org, 2002). DPF uses DTK for all of its nongraphical functions.

Figure 2 illustrates how DPF and DTK interface with the application and other software libraries. Notice that the application can directly call DPF, DTK, Performer, and the system libraries—another example of staying out of the user's way.

2 Functionality

The following subsections describe the various functionalities that compose DIVERSE. Some directly provide utility, such as networking services and access to hardware devices, whereas others provide a framework for developing DIVERSE modules and applications by way of C++ classes.

2.1 Remote Shared Memory

DTK remote shared memory can be thought of as a software version of SCRAMNet (Moorthy, 1998) that uses IP networks. The basic idea is that an application reads and writes to a DTK remote shared memory file, and changes to the file are automatically and transparently propagated to other processes connected to the same remote shared memory file. The process is unaware that data it reads might be modified by other processes, nor is it aware that its data modifications might be propagated to other processes.

DTK remote shared memory files map to a hierarchical directory structure. Certain de facto standards are used for some shared memory filenames, such as `head` and `wand`, but users can adopt any naming convention that best suits their needs. For example, suppose a system has two head trackers; they could be named `tracker1/head` and `tracker2/head`. Open source and the modularity of DSOs make these types of changes fairly straightforward.

DTK remote shared memory does not provide a coherent memory state between networked memory images, as is common to most distributed shared memory (DSM) systems (Carter, Khandekar, & Kamb, 1995; Lu, Dwarkadas, Cox, & Zwaenepoel, 1997). Instead, DTK remote shared memory was developed specifically for building distributed interactive dynamical simulations. Distributed VR applications are one example of this type of distributed simulation (Deligiannidis, 2000; Hartling, Just, & Cruz-Neira, 2001). This kind of simulation shares data by running distributed programs asynchronously and is typically concerned more with the speed of data transfer than the coherence. The use of a data interpolation method, like dead reckoning, to maintain apparent data consistency, is preferred to strict distributed data coherence and leads to smoother remote dynamical motions. Shared memory queuing is

used by DTK to provide for data interpolation or extrapolation.

For sharing data that represents dynamics, simulated or real, the main concern is not just how well the networked data is correspondent with the local image data, but also how well the data's continuity is preserved over time. Put simply, getting data quickly and smoothing the data is more important than getting a perfect image of the data.

This type of application lacks the need for distributed program synchronization and data coherence. This approach leads to much higher performance by not blocking processes and reducing messaging overhead, as is required in traditional DSM systems.

2.2 Dynamic Shared Objects

Much of DIVERSE's flexibility comes from the use of DSOs. A DIVERSE DSO is a separately compiled C++ program containing a standard entry point. DSOs can be loaded and unloaded at any time an application is running and their behavior is cumulative; one DSO can modify the behavior of other DSOs, or load and unload other DSOs.

An application can be reconfigured without needing to be recompiled simply by loading different sets of DSOs. This facility 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, just by loading different sets of DSOs.

The very direct approach of DSOs conveys the flexibly of the native API interface to the configuration interface, so anything that can be done when using the API can automatically be done when using the configuration interface. DSO objects can execute any arbitrary code, including using other APIs.

By contrast, a scripted configuration is either a set of static start-up values, such as the CAVElib's `caverc`, or if dynamic, usually requires the programmer to learn yet another language. A scripted configuration language must be integrated into the API itself, making extensibility more difficult. A scripted configuration must necessarily be limited in its scope, as it is impossible to have hooks into all other possible system and API calls that might be used. Any new features that are introduced into a package or application requires that the embedded parser be modified, if the new feature is not already handled by the parser. In practice, this can make the parser a large part of the project itself. In DIVERSE, no additional parser is needed for the configuration of programs; therefore, the complexity of DIVERSE is relatively reduced.

The DIVERSE release provides two embedded configuration parser examples, implemented as DSOs, for those who wish to use such a system. DIVERSE's extensibility easily accommodates this.

2.2.1 The Augment Class. DSOs are typically implemented as objects of DTK's Augment class. The Augment class contains methods to augment DIVERSE by means of inserting functionality at predefined points. An Augment object inherits four virtual methods, any or all of which it may overwrite. These methods are designed to accommodate the structure of the typical display-loop graphics program or of the cyclic dynamical simulation code. In the case of DPF, these four methods are invoked before and after Performer methods, allowing the Augment object to configure and modify (that is, augment) the Performer application program's behavior. The four methods and their relationship to Performer are

- `preConfig`, which is invoked once, just before Performer's `pfConfig` function is called;
- `postConfig`, which is invoked once, just after Performer's `pfConfig` function is called;
- `preFrame`, which is invoked every graphics frame, just before Performer's `pfFrame` function starts to traverse the scenegraph; and
- `postFrame`, which is invoked every frame, just after Performer's `pfFrame` function returns from traversing the scenegraph.

Figure 3 is pseudo-code showing where the methods are invoked within a DPF program. There are no limitations placed on what the code in an Augment object does. The importance of the Augment class cannot be

```
dpf *app = new dpf ;
// all Augment objects' preConfig
// methods invoked here
app->config() ;
// all postConfig methods invoked
while (1) {
   // all preFrame methods invoked
   app->frame() ;
   // all postFrame methods invoked
   // all graphic displays sync'd
}
```

Figure 3. *Placement of Augment class methods.*

overstated: it is a major part of the foundation of DIVERSE's flexibility.

2.2.2 The Manager Class. An object of the Manager class loads and unloads DSO files containing objects based on the Augment class, and invokes the four virtual methods of each loaded Augment object at the appropriate times. A returned status value tells the Manager class object what actions to take with respect to the Augment object's callback methods. The possible actions are

- `continue`, which signifies the method should be reinvoked when appropriate;
- `remove_callback`, which signifies that the method should be removed from the Manager object's list of callbacks;
- `remove_augment`, which signifies that the Augment object containing the method should be unloaded; and
- `error`, which signifies that the method encountered an error condition.

The Manager class is implemented in DTK and is used by DPF as a base class for its Dpf class. The Dpf class has methods that return pointers to objects of other classes, such as Display and other currently loaded Augment objects. A DPF application creates a single instance of the Dpf class, which is an application's entry point into the DPF API.

Some of the Manager class'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 programming errors due to methods not being invoked, or being invoked at the wrong time.

2.2.3 Specifying Which DSOs to Load. The set of DSOs to be loaded during any particular execution of a DPF program can be specified in two ways. The first way is to specify a list of DSO names to be loaded in the Manager object's constructor. If no DSO names are passed to the constructor, then the default DSO, a simple desktop window, will be loaded. The second way is to specify a list of DSOs to be loaded using the `DPF_DSO_FILES` environment variable; its value is a colon-separated list of DSOs to be sequentially loaded, processing from left to right through the list. This variable, if set, overrides the list of DSOs passed to the Manager object's constructor.

DSO callbacks are invoked by the Manager object in the same order that the DSOs were loaded. They are unloaded by the Manager object's destructor in the reverse order. For example, if DSO `bar` requires something created by DSO `foo`, `foo` is loaded before `bar`. At object destructor time, `bar` is unloaded before `foo`. This is done as a safety measure, as to do otherwise might allow `bar` to try to access something in the unloaded `foo`.

Each DSO is loaded by the Manager object using the `dlopen` library call. A pointer to the Manager object is passed as a parameter, allowing DSOs to invoke methods to query and modify the application that loaded it.

2.3 The Server and its Services

DTK provides a general hardware peripheral device service mechanism. This consists of a server program that can load and unload services that are DSOs, and a C++ API that is used to build clients whose primary interface to the served data is through DTK shared memory.

2.3.1 DTK Server. DTK distinguishes between a DTK server and DTK services, which are provided by a DTK server. A DTK service can be loaded or unloaded at any time while a DTK server is running, and multiple instances of the same DTK service can be loaded at the same time.

The DTK server program runs as a interrupt-driven `select` server, either as a daemon or as a regular process as specified via command line options. Multiple DTK servers can run concurrently on a single computer. Command line options can specify any number of DTK services to be loaded at DTK server start-up.

2.3.2 DTK Services. DTK services are DSOs that are loaded by a running DTK server; their value is that they extend the server's functionality at runtime. New hardware services can be created without needing to change, or even restart, the server or the clients that are using the service.

DTK services are based on a pure virtual base class, which defines the standard interface that the DTK server uses. A service implements a constructor and a data server method and specifies a file descriptor to be used by the server. The service's server method is invoked by the DTK server any time data associated with the specified file descriptor are available for reading.

A typical DTK service DSO declares a constructor and destructor that opens and closes one file; this may be an IP socket, serial port, or other device. The data server method reads and parses the file's data and then writes its output to DTK shared memory. In this way, there is no difference between local and networked DTK services; DTK shared memory can be seamlessly networked on the fly.

2.4 Input

DTK implements an extensible set of logical input devices; each device is implemented as a class based on the Input class. Additional device types/classes can be created as desired, allowing virtually any type of input device to be defined. Because the source of the input data is the DTK server, the devices are automatically

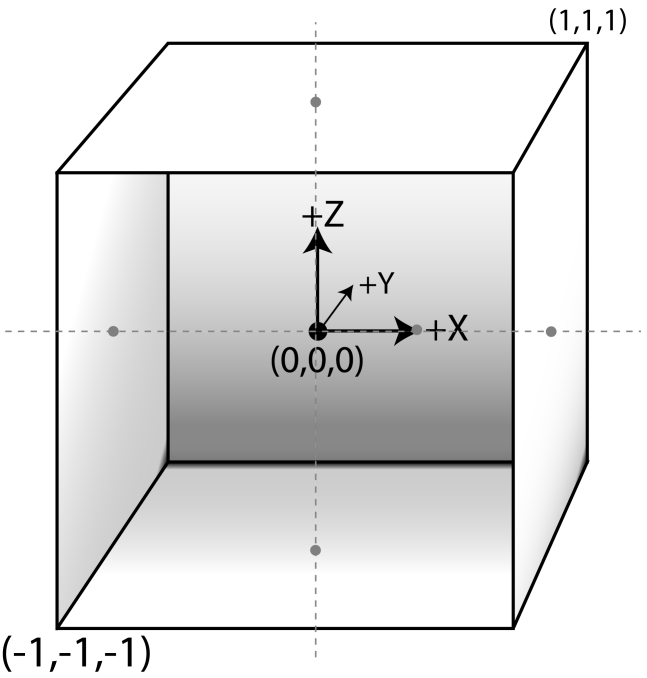

Figure 4. *The DPF coordinate cube.*

and transparently networked. DTK currently implements the following logical devices.

- A `locator` is a device providing three-space positional information, represented as a location and orientation, typically provided by a tracker. Location data are normalized to be consistent with the DPF coordinate cube. (See figure 4 in subsection 2.5.) Orientation data are HPR Euler angles, in degrees.
- A `keyboard` is a device returning a value representing a key on a keyboard.
- A `button` is a device returning an array of bits, each having a value of 1 or 0.
- A `valuator` is a device returning an array of floating-point values. The `locator` class is based on the `valuator` class; its data are normalized to the range [−1,1]. A mouse can be used as a `locator` device, returning normalized values regardless of the size and shape of the window containing it.
- A `selector` is a device returning an array of integer values.

Any device can be either queued or polled, as specified by the application reading the data. A queued device, typically a button or keyboard, queues each state change. A polled device, typically a locator or valuator, merely maintains a state that can be read at any time.

Event records contain a snapshot of the data associated with a set of input devices. Each change to the state of a queued input device queues an event record. In this way, it is possible to get the values of other devices at the time the event occurred.

The source of the input data is specified using DTK services. Similar input devices—be they actual or simulated, local or remote—can be exchanged, restarted, and reconfigured using DTK, simply by moving services in and out of the DTK server, without the application being aware of the modifications. To maintain transparency, each service needs to normalize its data in a manner consistent with the type of logical device it is implementing.

An example: a `Wombat9000` tracker conveys positional data via a serial port. A `Wombat9000` service can be written to communicate to the hardware device using the `Wombat9000`'s communication protocol using a specified serial port. The service is invoked by the server each time data arrive on the serial line. The service interprets the data from the hardware device, normalizes it, and writes it to a DTK shared memory file named `locator`. As long as a program is reading its tracker data from the DTK shared memory file named `locator`, it is unaware that it is using a `Wombat9000`. The `Wombat9000` service may be unloaded, and a new service loaded that also writes to the DTK shared memory file named `locator`. The application is unaware that it is receiving data from a different source.

2.5 Display

DIVERSE implements a Display class that is used to configure and query the characteristics of the graphical displays by use of an underlying graphics API. DTK provides a base Display class with virtual methods that are inherited by DPF, which implements these virtual methods and adds additional methods using Performer. By invoking the methods of DPF's Display class, applications will be created that are device independent.

An application that doesn't require graphical output does not have to use the Display class. Conversely, an application that uses the Display class for a particular API, such as DPF, can have its graphical output suppressed by loading a configuration that defines no graphical displays.

DPF's Display class can create any number of graphical windows, each containing any number of mapped areas, or screens, placed at an arbitrary size, position, and orientation with respect to the virtual world. Each screen contains either a symmetric or asymmetric viewing frustum, displaying a single mono image or a stereo pair with parallax offset. DPF automatically calculates the frusta and parallax values every frame: all the program needs to do is specify the configuration of the graphical screens.

Display spatial configuration is specified in a normalized right-handed coordinate system whose origin is at the center of the virtual world. As with Performer, the $+x$ axis is to the right, the $+y$ axis is straight ahead, and the $+z$ axis goes up. Typically, a coordinate cube two units on a side encompasses the VR system's display area. The actual size of these units will vary from system to system, and can be defined as best suits the system's configuration. Figure 4 illustrates this coordinate system. An application can specify its own world coordinate system using any arbitrary translation, rotation, and uniform scaling, allowing the VE coordinate system to accommodate the coordinate system of the underlying model data.

DPF creates a small Performer scenegraph, illustrated in figure 5, containing three nodes under which the user can create geometry. Each node corresponds to a different type of coordinate system. The root of the scenegraph, called the `scene` node, is used for normalized coordinates; objects placed under this node will be displayed with the same size and orientation regardless of the application's world coordinate settings. Under the scene node is the `ether` node, which is used for objects that will be displayed in world coordinates but which will not be subject to navigational offsets. This node is useful for unreachable objects such as a sky dome, or a

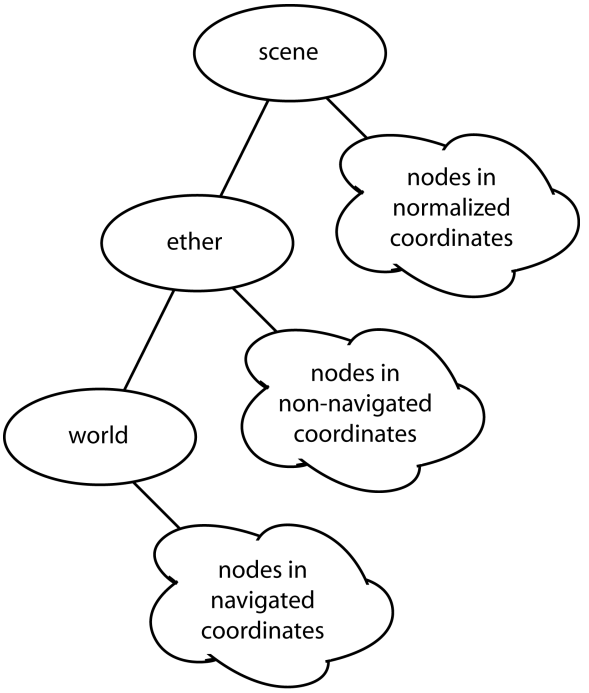

Figure 5. *The DPF scenegraph.*

cylinder containing a texture map of a mountain range representing a horizon. Below the `ether` node is the `world` node. Objects placed under this node are displayed in navigated world coordinates; applications typically place most of their geometry under this node.

DPF's Display object is a factory at the root of a tree-structured hierarchy of DPF class objects. Each object creates and manages the objects belonging to the class directly below it in the tree. The number and attributes of the objects in each class determine the configuration of a system's graphical displays. This tree structure minimizes errors that are caused by misconfiguring Performer objects. Figure 6 illustrates the configuration of these objects in Virginia Tech's CAVE. The classes are arranged as follows.

- The Display object creates one or more Pipe objects, each of which configures and controls a Performer `pfPipe` object. (A *pipe* refers to a graphics pipeline; there is one pipe for each hardware graphics controller in use.)
- A Pipe object creates one or more Window objects, each of which creates and controls a Performer `pfPipeWindow` object. A window is mapped to a rectangular portion of the display area managed by the pipe.
- A Window object creates and controls one or more Screen objects. A screen is a rectangular portion of a window containing a view frustum. In immersive systems with fixed display surfaces, such as a CAVE, a screen usually corresponds to a single display surface and contains an asymmetric frustum. In desktop systems, a screen is usually used as a viewport and contains a symmetric frustum. Head-tracked HMDs usually use a screen with a symmetric frustum that fills the entire viewable area.
- A Screen object creates and controls Performer `pfChannel` objects. If the Window object that created the Screen object is in mono, a single `pfChannel` is created. If the Window is in stereo, two `pfChannel` objects are created, one for each binocular eye.

An application will typically invoke methods of only the Display class; in this way it is independent of the graphic display configuration. A DPF Display DSO file will use the methods of the Display class and all of the classes below it in the tree to configure the graphical displays at run time.

2.6 Navigation

The Navigation class, based on the Augment class, is part of DTK as it is not directly related to graphical output. An object of the Navigation class modifies parameters such as position, orientation, pivot point, scale, and response, which are then converted to a transformation matrix by the Navigation class's `postFrame` method. A Navigation object can also directly specify its own matrix if desired. A Navigation object can set navigational parameters based on anything from device input, such as a desktop mouse or a CAVE's wand and joystick, to the output of a simulation process. A DTK time class allows for the easy implementation of rate-based, rather than frame-based, navigations.

Navigation objects are usually written as DSOs. Multiple Navigation objects can be loaded and are accessed

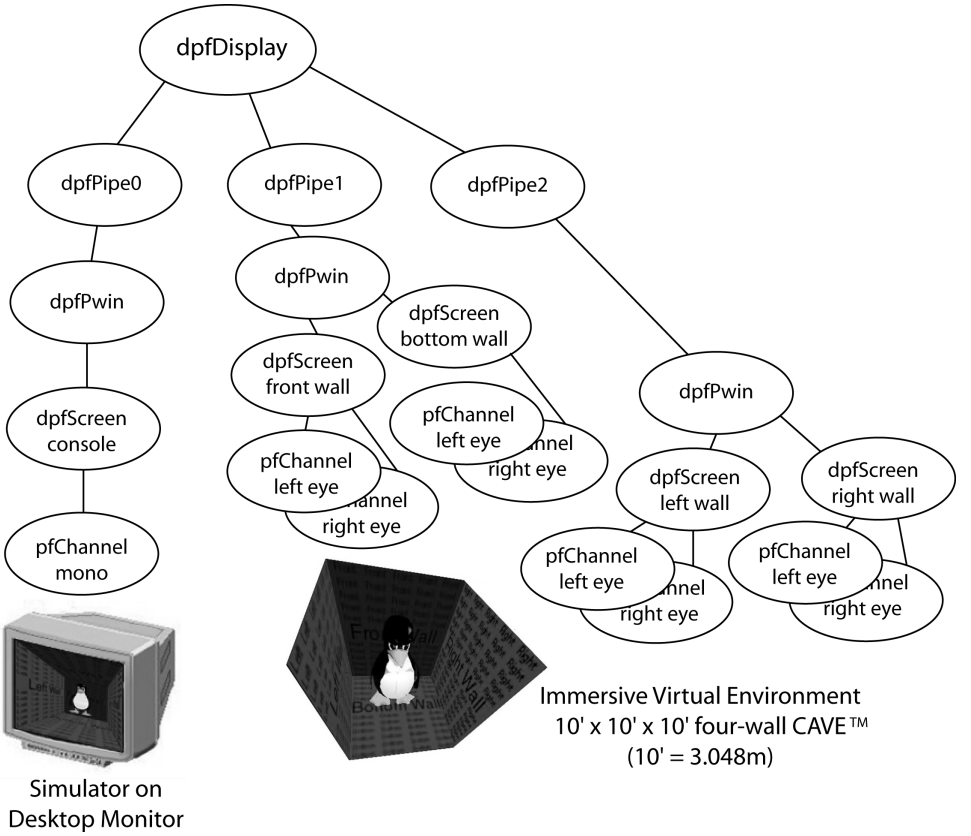

Figure 6. *Example Display class hierarchy.*

by methods of the NavList class. Any single Navigation object (or no Navigation object (the NULL Navigation)) can be chosen as the current navigation. DSOs can load and switch between Navigation objects, and modify the current navigation's parameters, based on user input or state data.

In most cases, classes based on the Navigation class usually implement just a `postFrame` method, which converts input data into the current navigation's transformation matrix. The Navigation class's virtual `preFrame` method copies this matrix into the `world` scenegraph node's matrix.

3 Writing a DIVERSE Application

The first decision that should be made when designing a DIVERSE application is where should the functionality go. By this we mean that the designer needs to determine what pieces of the application can be written into DSOs and what pieces need to be part of the application itself. The designer also needs to determine how many processes should comprise the application, and what data need to be shared between the various processes.

We have determined over time that, in general, as much code as possible should be written as DSOs. The pieces that should be written as DSOs are those that can be reused—either by other applications, such as a navigational technique or data filter, or as an interchangeable component of the application. In this way, DIVERSE promotes the creation of small and more easily maintainable application programs.

DIVERSE comes with dozens of small example programs; each one is designed to demonstrate a specific feature, and each can be easily moved, modified, and

recompiled. Every example is based on the same small "hello world" program, similar to the code in figure 1; in this way, the added functionality is readily apparent. Source code of examples, standalone programs, and loadable DSOs are included with the distribution.

3.1 Command Line Tools

DIVERSE comes with a number of programs to facilitate the creation and management of applications, both for the programmer and nonprogrammer.

`dtk-config` and `dpf-config` are small applications that display installation-specific text strings. They can return the current version, flags needed for compiling and linking an application, and compiled-in values, such as the default DSO directory. Their use in shell scripts and `Makefiles` will increase application portability and compatibility.

DTK comes with a number of application tools to access and manipulate remote shared memory and the server. Some are text based, such as

- `dtk-readFloats`, which repeatedly reads a remote shared memory file, interprets the data as an array of floating-point values, and displays the values;
- `dtk-pingServer`, which tests if the DTK server is running; and
- `dtk-loadService`, which loads a service into the server.

Others have a GUI interface, including

- `dtk-gnomonDisplay`, which graphically displays the data from DTK shared memory containing locator data;
- `dtk-floatSliders` and `dtk-doubleSliders`, both of which write floating-point data into DTK shared memory using slider widgets; and
- `dtk-launcher` and `dtk-demoRunner`, which are useful for starting and stopping executable programs, including the DPF example programs.

3.2 Diversifly

`Diversifly` is a DPF program that allows nonprogrammers to load and navigate through model files and apply global transformations and lighting effects; it is the DPF analog of Performer's `perfly`. In form, `diversifly` is similar to the program in figure 1; the main difference is that `diversifly` contains command line argument parsing and additional error checking. `Diversifly` also invokes methods of the Display class to configure the world coordinate system, and contains some Performer code to place light nodes into the scenegraph. Like any other DIVERSE program, the display format, input devices, and navigational and interaction techniques are loaded as DSOs. `Diversifly` has become popular as an easy way to visualize a model in various immersive and nonimmersive settings. It has also proven useful as a general-purpose program that can be used with functionality that is encapsulated as DSOs.

For example, we recently wrote a multiuser collaborative system for battlespace simulation. It was written as a collection of DSOs using `diversifly`, and so we were easily able to incorporate its collaborative functionality into another existing application that displays time-series molecular data—itself a collection of DSOs that use `diversifly`. See subsection 3.3 for another example of this approach.

3.3 A Concrete Example

The purpose of the Scientific Applications and Visualization Group (SAVG) at the National Institute of Standards and Technology (NIST) is to accelerate scientific discovery through computation and visualization (Griffin, 2002a; Sims et al., 2000; Sims et al., 2002). SAVG provides a framework of hardware, software, and complementary expertise that application scientists can utilize to facilitate meaningful discoveries.

Large and complex data sets are becoming more commonplace at NIST, as high-performance parallel computing is used to develop high-fidelity simulations, and combinatorial experimental techniques are used in the laboratory. Immersive virtual reality (IVR) is an emerging technique with the potential to handle this growing data. IVR is significantly different from traditional desktop visualization and significantly more effective at illuminating such data sets (van Dam, Forsberg,

Laidlaw, LaViola, & Simpson, 2000). However, the benefits of IVR can only be gained when scientists use it, and the key ingredient to making IVR accessible to scientists is to provide the ability to simply and quickly move their data into the immersive environment.

DIVERSE is among the IVR tools used by SAVG to advance the scientific research of its collaborators. NIST (including SAVG) is physically spread across two campuses in Gaithersburg, Maryland, and Boulder, Colorado. The device independence of DIVERSE allows the same applications to be run on nonstereo and stereo-enabled workstations across both campuses, as well as the RAVE located at Gaithersburg. This ability to run on a range of input and output devices is critical for SAVG collaborators. The two-wall RAVE is configured as an immersive corner with two 8 × 8 ft. (2.44 × 2.44 m) screens flush to the floor, oriented 90 deg. to form a corner. DIVERSE handles the details of stereoscopic projection and the I/O for head and wand tracking.

Researchers in the Building and Fire Research Laboratory (BFRL) at NIST are studying high-performance concrete (Building and Fire Research Laboratory, 2002). BFRL is leading the Virtual Cement and Concrete Testing Laboratory (VCCTL) consortium consisting of the major cement producers (Virtual Cement and Concrete Testing Laboratory, 2002). "Concrete is the most widely used man-made product in the world, and is second only to water as the world's most utilized substance" (Vision2030, 2000 p. 1). Any improvements in concrete such as cost, durability, or strength will have a significant effect on the economy. SAVG is a member of the VCCTL consortium and collaborates with its members on their visualization and parallel computing needs.

Figure 7 is a single image from an interactive visualization of flowing concrete implemented with DIVERSE. More specifically, it was implemented as a set of DSOs invoked by `diversifly`. Ellipsoids represent concrete particle motion. Lines represent their full path over the simulation time period. The numerical algorithm (Sims et al., 2000, section 5) simulates the flow of ellipsoidal objects (concrete particles) in suspension. The visualization plays an important role in the validation of the algorithms and the correctness of complex systems like this flow of fluid concrete. A digital animation of this visualization is available at `http://math.nist.gov/mcsd/savg/vis/concrete/` (Griffin, 2002b).

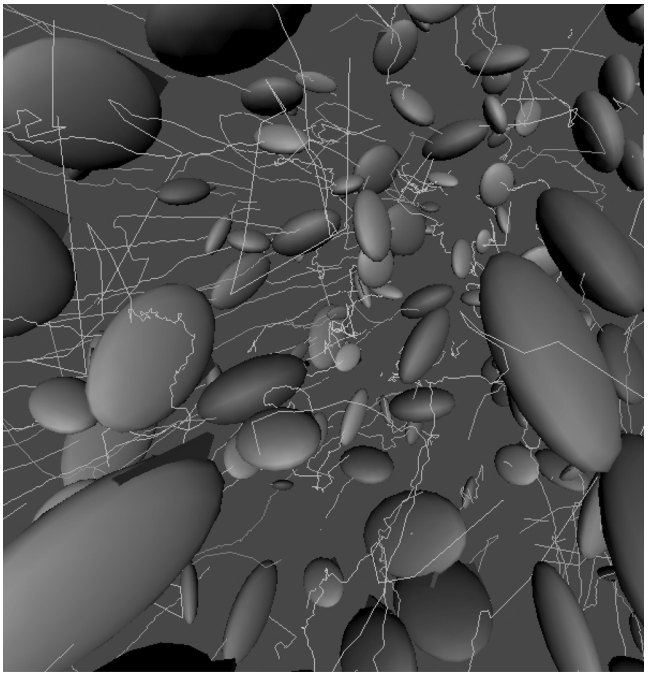

Figure 7. *Visualization of flowing concrete.*

On top of this DIVERSE/`diversifly` infrastructure, SAVG has developed additional tools and techniques for quickly moving research data into the immersive environment, often with little or no special-purpose graphics programming. The approach used is to apply the classic UNIX philosophy (Kerighan & Pike, 1984) of creating small and reusable tools that fit into the immersive environment. These tools lend themselves to be combined in a variety of ways to perform useful tasks such as moving scientific data into the immersive environment and creating IVR applications.

Using this philosophy, SAVG is providing the key ingredient to making IVR accessible to NIST scientists by

- developing simple and reusable graphics file formats implemented as `diversifly` file loaders;
- developing application-specific files that are easily transformed into existing or newly created graphics file formats; and

- developing filters (scripts, small programs, and so forth) to connect into data transformation pipelines.

For the visualization of ellipsoidal objects (such as concrete particles) in suspension, a file loader DSO was created. This DSO implemented the animation as a time sequence of 3D data. For this new `seq` file type, each time step of the sequence was made general to allow all other possible graphics file types to be used. Thus, not only were the requirements for this application met, but a wide variety of other applications involving animation as a time series of 3D data can be quickly implemented by reusing this DSO.

The development of an application-specific file format doesn't always require a DSO extension to `diversifly`. For example, an application that required the display of colored spheres at random locations was satisfied by defining an ASCII file to simply list the *x, y, z* sphere coordinates, the sphere diameters, and the corresponding sphere colors. Using a simple scripting language, a filter named `sphere2iv` was created to convert the sphere list format to an Open Inventor (Silicon Graphics, Inc., 2002b) file (`iv`), which is one of the many file formats compatible with DIVERSE. Although it is possible for a user to directly write the `iv` file, the approach followed here allows a user to concentrate on their application by simplifying the graphics data format to essentially what the user is already doing. Separating the larger task into smaller pieces that fit together is the essence of the classic UNIX philosophy of software tools. Again, the simplicity of this approach provides quick access to the IVR environment for a range of applications requiring the display of colored spheres. Now that this tool has been added to the toolbox, it can be combined with the `seq` file format to produce other types of time-series animations.

In addition to using DSOs for adding new graphics file loaders, a DSO can also be used to extend the functionality of `diversifly`. Using this capability, interactive features can be added. For example, figure 8 shows a visualization of a cloud water data set[2] derived from a sample data set and program provided by the Open Visualization Data Explorer (OpenDX) software package (OpenDX, 2002). With the addition of a file switcher DSO, an interactive user click can be used to turn on (or off) streamlines as illustrated in figure 9. In this case, the feature is used as a simple on/off switch. However, the file switcher DSO is actually programmed to cycle through each of a series of files with each click. The list of switched files is separate from the normal `diversifly` input list of files to display. All of the available graphics file types can be specified in either the

Figure 8. *Cloud example.*

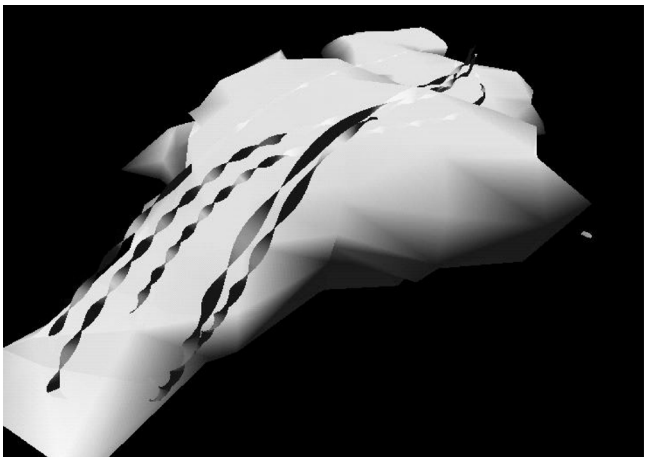

Figure 9. *Click to turn on streamlines.*

2. From Open Visualization Data Explorer (OpenDX) data set `cloudwater.dx` (OpenDX, 2002).

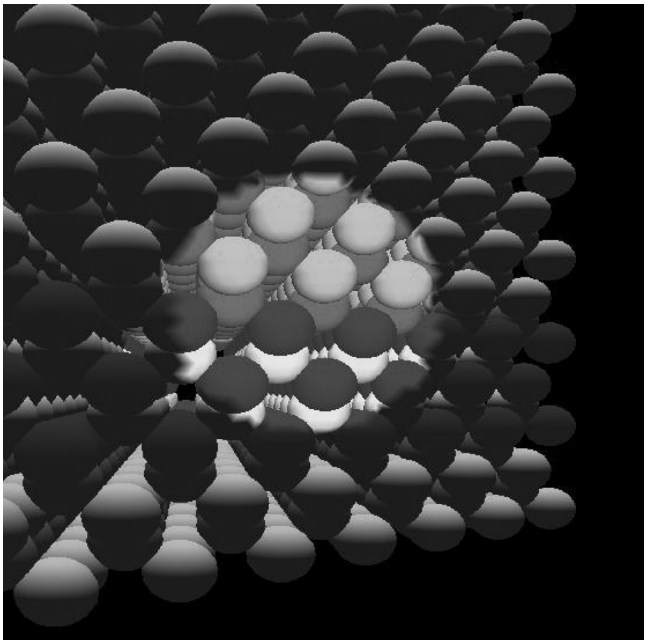

Figure 10. *Flashlight DSO enables the user to shine a* virtual flashlight *on a spot* within *the immersive visualization by merely pointing to it.*

switched or nonswitched list. The most appropriate graphics file type can be used for each view of the visualization. Thus, very complex immersive visualizations can be created at runtime with the addition of this very simple tool.

Another DSO that extends the user interface is a *virtual flashlight*. Figure 10 shows the user view of the virtual flashlight, which is attached to the wand. It operates like a handheld flashlight to explore and highlight the virtual world. This capability is extremely useful both for a single user to focus attention on specific areas of the data and for multiple users in a collaborative session. Just like the real world, a flashlight belongs in everyone's tool box.

A very straightforward definition for "user friendly" is "what the user already knows." The advantage of the software tools approach is that combinations of existing tools with occasional small additions can frequently match the user's natural data to the visualization data requirements.

The success of the cement flow visualization and other applications have shown DIVERSE to be a valuable tool in support of the SAVG collaborators. Using DIVERSE as a foundation, SAVG has built an infrastructure of reusable utilities, data filters, generalized application protocols, and data formats that feed into a single, well-understood viewing environment. This infrastructure provides a quick and almost effortless (no application-specific virtual reality programming) method for a wide range of scientists to get numerical simulations or other research into the immersive virtual reality environment.

4 Recent Additions and Future Directions

We are continually extending the DIVERSE project to include new modules, extensions to existing modules, and more interfaces to existing packages.

With the release of Performer 3.0, a Microsoft Windows version of DPF is possible. (A Windows release should be available in early 2003.)

A DIVERSE module has been released that augments the OpenGL interface in the same manner that DPF augments the Performer interface. This module, the DIVERSE interface to OpenGL (DGL), allows us to port DIVERSE to platforms that are not supported by Performer. DGL is scenegraph agnostic; scenegraph libraries that generate OpenGL can be used with DGL to create VE content. So far, OpenInventor (Silicon Graphics, Inc., 2002b) has successfully been used with DGL to display a scenegraph in a CAVE.

To better support the visualization of 3D data sets, we are creating interfaces to other visualization packages. We have implemented interfaces to SGI's OpenGL Volumizer (Silicon Graphics, Inc., 2002d) and are currently implementing interfaces to Kitware's VTK (Kitware, Inc., 2002).

To support networked collaboration, we have written tools to support the creation of collaborative VEs. A collaborative application allows multiple users on multiple systems 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 among users. Another example is awareness tools, such as DSOs that display status information about the shared virtual world and its inhabitants.

We are investigating techniques to support GNU/Linux-based immersive clusters, building on existing DTK shared memory and collaboration tools. We are also currently in the process of writing more navigational and simulation techniques. Navigational techniques will include features such as physics-based dynamics and intelligent agents. Simulators will strive to be more intuitive and anthropomorphic. All of these will be written as DSOs, so they will be readily available to any application.

Further, we are implementing a system to map from an application's name for a shared memory file to the actual name of the shared memory file, so the application (or DSO) won't have to be modified to use different shared memory files.

We are continually adding functionality to the `diversifly` program. We are creating more command line and GUI tools, so it will be easier for users who aren't familiar with Unix to create and access immersive environments.

5 Conclusions

This paper has described DIVERSE, including its motivation, architecture, functionality, and how it is used to develop applications. Examples were presented that demonstrate how DIVERSE can provide an infrastructure for applying immersive visualization to scientific research. Through the use of well-defined interfaces and an extensible display/explore application, DIVERSE fits naturally into a software tools approach. The ease of use, portability, and extensibility of DIVERSE provides the key ingredients to making immersive visualization accessible to a wide range of applications.

We welcome suggestions and comments from users as to how best to improve DIVERSE.

Acknowledgments

This research was supported in part by the Office of Naval Research (grant N00014-00-1-0549), TASC (grant 2002-042), and by Lockheed Martin Astronautics (grant CK 124405). The authors wish to thank Edith Satterfield for her invaluable support and assistance.

Disclaimer

Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

Arsenault, L., Kelso, J., Kriz, R., & Das Neves, F. (2001). DIVERSE: A software toolkit to integrate distributed simulations with heterogeneous virtual environments. Tech. Rep. 01-10, Department of Computer Science, Virginia Tech. Retrieved May 10, 2002, from http://www.cs.vt.edu/TR/DIVERSE.pdf.

Building and Fire Research Laboratory. (2002). BFRL goal: High performance building materials. Retrieved May 10, 2002, from http://www.bfrl.nist.gov/goals_programs/HPBM_goal.htm.

Carter, J. B., Khandekar, D., & Kamb, L. (1995). Distributed shared memory: Where we are and where we should be headed. *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V),* 119–122. Retrieved May 10, 2002, from http://citeseer.nj.nec.com/carter95distributed.html.

Cruz-Neira, C. (2002). VR Juggler—The virtual reality platform. Retrieved May 10, 2002, from http://www.vrjuggler.org/.

Deligiannidis, L. (2000). DLoVe—A specification paradigm for designing distributed VR applications for single or multiple users. Unpublished doctoral dissertation, Tufts University, Department of Computer Science. Retrieved May 10, 2002, from http://citeseer.nj.nec.com/deligiannidis00dlove.html.

Fakespace Systems. (2002). Fakespace Systems Inc.—Better ways to create & communicate. Retrieved May 10, 2002, from http://www.fakespacesystems.com/.

Free Software Foundation. (2002). Various licenses and comments about them - GNU project - Free Software Foundation (FSF). Retrieved May 10, 2002, from http://www.gnu.org/philosophy/license-list.html.

Griffin, T. J. (2002a). Scientific Applications & Visualization Group. Retrieved May 10, 2002, from http://math.nist.gov/mcsd/savg/.

———. (2002b). Visualization of high performance concrete. Retrieved May 10, 2002, from http://math.nist.gov/mcsd/savg/vis/concrete/.

Hartling, P., Just, C., & Cruz-Neira, C. (2001). Distributed virtual reality using octopus. *IEEE Virtual Reality 2001,* 53–62.

Hubbold, R. J., Dongbo, X., & Gibson, S. (1996). MAVERIK—The manchester virtual environment interface kernal. In M. Göbel, J. David, P. Slavik, & J. J., van Wijk. (Eds.), *Virtual Environments and Scientific Visualization '96* (pp. 11–20), Wien: Springer-Verlag.

Kerighan, B. W., & Pike, R. (1984). *The UNIX programming environment.* Englewood Cliffs, NJ: Prentice Hall.

Kessler, G. D., Bowman, D. A., & Hodges, L. F. (2000). The simple virtual environment library, an extensible framework for building VE applications. *Presence: Teleoperators and Virtual Environments, 9*(2), 187–208.

Kessler, G. D., Hodges, L. F., & Ahamad, M. (1998). Ravel: A support system for the development of distributed multiuser VE applications. *IEEE VRAIS '98,* 260–267.

Kitware, Inc. (2002). Kitware Inc.—Leaders in visualization technology. Retrieved May 10, 2002, from http://www.kitware.com/.

Linux Online. (2002). Retrieved May 10, 2002, from http://www.linux.org/.

Lu, H., Dwarkadas, S., Cox, A. L., & Zwaenepoel, W. (1997). Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing, 43*(2), 65–78. Retrieved May 10, 2002, from http://citeseer.nj.nec.com/lu97quantifying.html.

Macedonia, M. R., Brutzman, D. P., Zyda, M., Pratt, D. R., Barham, P. T., Falby, J., & Locke, J. (1995). NPSNET: A multi-player 3d virtual environment over the Internet. *Symposium on Interactive 3D Graphics, 210,* 93–94.

Moorthy, V. (1998). Low latency message passing on workstation clusters using SCRAMNet. Tech. Rep. 10/98-TR42, The Ohio State University, Department of Computer and Information Science, Columbus, Ohio 43210, USA. Retrieved May 10, 2002, from http://citeseer.nj.nec.com/222981.html.

MultiGen-Paradigm (2002). MultiGen-Paradigm—Vega. Retrieved May 10, 2002, from http://www.paradigmsim.com/products/runtime/vega/index.shtml.

OpenDX (2002). Open visualization data explorer. Retrieved May 10, 2002, from http://www.opendx.org/.

OpenGL.org (2002). OpenGL—High performance 2d/3d graphics. Retrieved May 10, 2002, from http://www.opengl.org/.

PTC Corporation (2002). PTC—What matters most? Product first. Retrieved May 10, 2002, from http://www.division.com/.

Sense8 (2002). Sense8 virtual reality 3D software. Retrieved May 10, 2002, from http://www.sense8.com/.

Silicon Graphics, Inc. (2000). *OpenGL Performer programmer's guide.* Silicon Graphics, Inc. Also available online at http://techpubs.sgi.com/.

———. (2002a). SGI—Developer central operating systems. Retrieved May 10, 2002, from http://www.sgi.com/developers/technology/irix/.

———. (2002b). SGI—Open Inventor. Retrieved May 10, 2002, from http://www.sgi.com/software/inventor/.

———. (2002c). SGI—OpenGL Performer. Retrieved May 10, 2002, from http://www.sgi.com/software/performer/.

———. (2002d). SGI—OpenGL Volumizer. Retrieved May 10, 2002, from http://www.sgi.com/software/volumizer/.

Sims, J. S., George, W. L., Satterfield, S. G., Hung, H. K., Hagedorn, J. G., Ketcham, P. M., Griffin, T. J., Hagstrom, S. A., Franiatte, J. C., Bryant, G. W., Jaskólski, W., Martys, N. S., Bouldin, C. E., Simmons, V., Nicolas, O. P., Warren, J. A., am Ende, B. A., Koontz, J. E., Filla, B. J., Pourprix, V. G., Copley, S. R., Bohn, R. B., Peskin, A. P., Parker, Y. M., & Devaney, J. E. (2002). Accelerating scientific discovery through computation and visualization II. *Journal of Research of the National Institute of Standards and Technology, 107*(3), 223–245.

Sims, J. S., Hagedorn, J. G., Ketcham, P. M., Satterfield, S. G., Griffin, T. J., George, W. L., Fowler, H. A., am Ende, B. A., Hung, H. K., Bohn, R. B., Koontz, J. E., Martys, N. S., Bouldin, C. E., Warren, J. A., Feder, D. L., Clark, C. W., Filla, B. J., & Devaney, J. E. (2000). Accelerating scientific discovery through computation and visualization. *Journal of Research of the National Institute of Standards and Technology, 105*(6), 875–894.

Snowdon, D. N., & West, A. J. (1994). Aviary: Design issues

for future large-scale virtual environments. *Presence: Teleoperators and Virtual Environments, 3*(4), 288–308.

Tramberend, H. (1999). Avocado: A distributed virtual reality framework. *IEEE Virtual Reality 1999,* 53–62.

van Dam, A., Forsberg, A., Laidlaw, D., LaViola, J., & Simpson, R. (2000). Immersive VR for scientific visualization: A progress report. *IEEE Computer Graphics and Applications,* 26–52.

Virtual Cement and Concrete Testing Laboratory (2002). Concrete publications online. Retrieved May 10, 2002, from http://www.bfrl.nist.gov/862/vcctl/.

Vision2030 (2000). *Vision 2030: A vision for the US Concrete Industry.* Concrete Vision Workshop, Strategic Development Council. Retrieved May 10, 2002, from http://www.ees.lanl.gov/pdfs/ConcreteVision2030.pdf.

VRCO (2002). VRCO. The producers of the CAVE libraries, or CAVElib. Retrieved May 10, 2002, from http://www.vrco.com/.

Watsen, K., & Zyda, M. (1998). Bamboo—A portable system for dynamically extensible, real-time, networked, virtual environments. *IEEE VRAIS '98,* 252–259.

White, L. (2002). MR Toolkit. Retrieved May 10, 2002, from http://www.cs.ualberta.ca/graphics/MRToolkit/.