DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments

John Kelso, Lance E. Arsenault
Virginia Tech
Department of Computer Science
kelso@vt.edu, lanceman@vt.edu

Steven G. Satterfield
National Institute of Standards and Technology
Scientific Applications and Visualization Group
steve@nist.gov

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

Abstract

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

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

We will describe DgiPf's design 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++ APIs (Application Programming Interfaces), designed to integrate distributed simulations with heterogeneous virtual environments (VEs) [1].

DgiPf (DIVERSE graphics interface to Performer) 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 DgiPf program that loads a model file. The user of this program can:

- run with one or more immersive, or non-immersive graphical display systems, or desktop simulators of immersive environments,
- select from navigation and other interaction techniques,
- configure input devices, either real or virtual, local or remote.

The code in Figure 1 is basically a Performer program. All graphical configuration code typical to a Performer application has been removed, and four lines of DgiPf code, in **bold**, were added. This program can run on a CAVE™, RAVE™, ImmersaDesk™, head-tracked HMD (head mounted display), desktop or laptop, using any available navigation or interaction technique that the user chooses [2].

```
#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 1. A complete DgiPf program

1.1. Motivation

DIVERSE grew out of a need to provide a highly reconfigurable device independent system to support the display of data from asynchronous distributed simulations.

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

We also 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 restraints

It was also recognized that high-end systems such as the CAVE are not cost-effective debugging and development platforms. In addition, 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. 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 non-immersive systems, and,
- include tools that allow non-programmers 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. From these goals came our list of design criteria.

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. But, since the second directive fosters innovation and flexibility we tend to favor it over ease of use.

These criteria lead to specific design features:

1. **Highly 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 DIVERSE 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 feature allows the abstraction of such things as hardware devices and navigation techniques.

     For example: 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 between 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.

     Another example: a navigation module allows locomotion through the virtual world. Encapsulated navigation modules allow them to be developed and used independently of the application program. This also allows on-the-fly switching between navigation techniques.
   - Flexibility– Modules 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 module types and instances without requiring program modification.

2. **Augment, don't replace**

   DIVERSE augments and uses existing software libraries instead of creating new non-standard 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, DgiPf 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 run-time.

1.2. Availability

DIVERSE is available for download from the DIVERSE home page: http://www.diverse.vt.edu/.

Included with the distribution are utility and application programs, as well as dozens of example programs. DgiPf currently runs on all platforms supported by Performer: SGI IRIX™ and GNU/Linux systems [3][4]. 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) [5].

1.3. Other related VR systems

Before deciding to embark on the expense and toil of creating our own system we surveyed the available VR systems to see if one would fit our needs. Many products, such as the CAVElib™[6], dVise™[7], WorldToolKit™[8], Vega™[9] and MR Toolkit[10] were rejected because they were not free, both in the sense of cost and of redistribution. Although VR Juggler[11] currently meets this criteria, at the time of our survey in July 1999, it had not yet been released open source. When VR Juggler was OpenSourced in January 2000, we decided to continue DIVERSE development because VR Juggler, 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".

1.4. DIVERSE Modules

DIVERSE currently consists of two modules; **DgiPf** is a module that implements graphic display output and user interactions in VEs. **DTK,** the DIVERSE ToolKit, implements non-graphical tasks such as peripheral hardware services and networking. The segregation of DIVERSE into graphical and non-graphical modules allows new graphical modules to be incorporated into DIVERSE as needed, and simulations that require no graphical output do not need to link to any particular graphics packages.

DgiPf adds immersive and non-immersive graphics to simulations by augmenting OpenGL Performer™, a high-level scenegraph-based graphics API built using OpenGL [12][13]. DgiPf uses DTK for utility functions, non-graphical hardware access and I/O facilities.

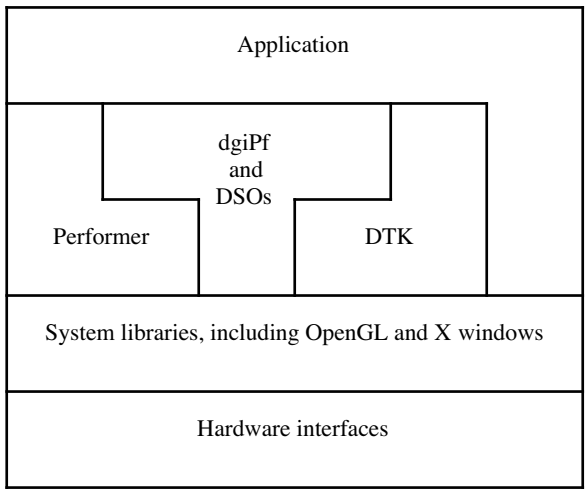

Figure 2. Library dependencies

Figure 2 illustrates how DgiPf interfaces with DTK, the application, and other software libraries. Notice that the application can directly call DgiPf, DTK, Performer and the system libraries.

2. DgiPf Design

DgiPf, being a C++ API, is built using classes containing methods and data. There are four classes that will be used in most DgiPf applications, as described below.

2.1. The Display Class

The Display class is used to configure and query the characteristics of the graphical displays. By querying the methods of the Display class, applications will be created that are device independent.

One or more graphical displays can be specified, and each one can have an arbitrary size, position and orientation with respect to the virtual world. Each display presents a single mono image, or a stereo pair with parallax offset. Each graphical display contains one or more mapped areas, or viewports. Each viewport contains either a symmetric or asymmetric viewing frustum.

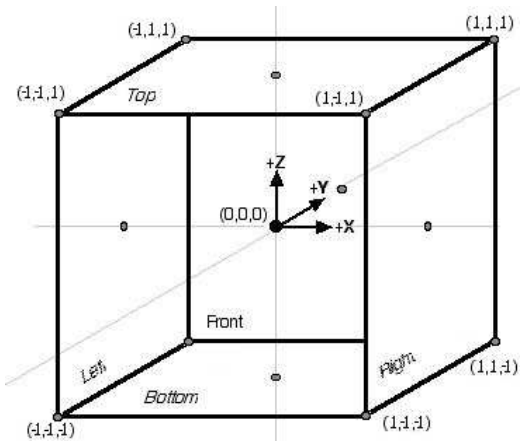

Figure 3. The DgiPf coordinate cube

Symmetric frusta, typically used in non-immersive configurations such as desktops, have a fixed shape, and a base that moves to accommodate changes in the viewing position and orientation. Asymmetric frusta, typically used in immersive configurations, have a base at a fixed position, and a shape that changes with the viewing position, but not the viewing orientation.

Display 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. A coordinate cube two units on a side encompasses the VR system's display area. Figure 3 illustrates this coordinate system. The application can specify its own world coordinate system using any arbitrary translation, rotation and uniform scaling.

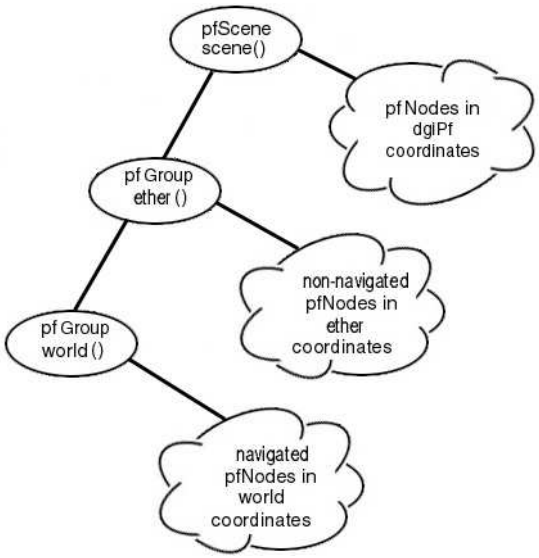

Figure 4. The DgiPf scenegraph

The Display class creates a small Performer scenegraph, illustrated in figure 4, 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, 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 cylinder containing a texture map of a mountain ridge 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.

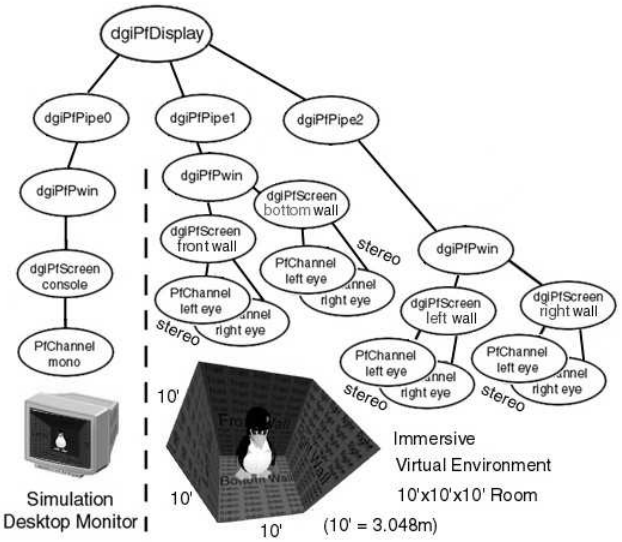

Figure 5. Example Display class hierarchy

The Display object is the root of a tree-structured hierarchy of DgiPf class objects. Each object creates and manages 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 5 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 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. In immersive systems a Screen usually corresponds to a physical display surface, such as a wall of a CAVE. In non-immersive systems a Screen is usually used as a viewport.
- 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 eye.

An application will typically only invoke Display class methods; in this way it is independent of the graphic display configuration. An Augmentation object (see section 2.3) that configures a set of graphical displays will use the methods of the Display class and all of the classes below it in the tree.

2.2. The Input Class

The Input class is used to read data from an extensible set of generic 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. DgiPf currently implements the following device types:

- A *locator* is a device providing 3-space positional information, represented as a location and orientation, typically provided by a tracker.
- A *keyboard* is a device returning an X-windows KeyCode, representing a key on the keyboard.
- A *button* is a device returning an array of bits, each having a value of one or zero.
- A *valuator* is a device returning an array of floating point values. A mouse is an example of this device; its position is represented by two floating-point values.
- A *selector* is a device returning an array of integer values.

Any device can be either queued or polled, as specified by the application. A queued device, typically a button or keyboard, queues each state change. A polled device, typically a locator or valuator, 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 queued input device queues an event record. In this way it is possible to get the values of all devices at the time the event occurred.

The source of the input data is typically specified externally to DgiPf using DTK. Similar input devices, be they actual or simulated, local or remote, can be exchanged, restarted and reconfigured using DTK, without the DgiPf application being aware of the modifications.

2.3. The Augment Class

The Augment class contains methods to augment DgiPf functionality by means of pre-defined callbacks. An Augment object inherits four virtual methods, any or all of which it may overwrite:

- prePfConfig, which is called once, just before Performer's pfConfig function,
- postPfConfig, which is called once, just after Performer's pfConfig function,
- prePfApp, which is called every frame, just before Performer's pfApp function starts to traverse the scenegraph, and
- postPfApp, which is called every frame, just after Performer's pfApp function returns from traversing the scenegraph.

There are no limitations placed on what the code in an Augment object does. The importance of the Augment class cannot be overstated. Augment objects, and classes derived from the Augment class, are used to:

- implement the Input class,
- configure the Display and Input classes, either directly in the Augment object itself, or indirectly by having the Augment object read interpreted configuration files,
- implement navigation and interaction techniques,
- create desktop simulators of immersive systems, and,
- implement new Performer node types, facilitating a data-driven programming paradigm. For example, an object is automatically positioned in the virtual world based on the output of a networked simulation process.

2.4. The DgiPf Class

A DgiPf application creates a single instance of the DgiPf class, which is the application's entry-point into the DgiPf API.

The DgiPf object manages other C++ objects. It loads and unloads pre-compiled DSO (Dynamically Shared Object) 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 DgiPf object what actions to take with respect to the Augment object.

Since DSOs are loaded at run-time, and their behavior is cumulative, the application itself can be reconfigured without needing to be recompiled 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.

The DgiPf class also creates a single Display object, optionally configured by the loaded Augment objects, which generates graphical output by invoking Performer methods. In order to queue event data from input devices the DgiPf class also creates a single event record object. The DgiPf class unloads and deletes all objects that it creates, when 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.

3. Writing a DgiPf Application

The first decision that should be made when designing a DgiPf application is: "*Where does the functionality go?*" By this we mean that the designer needs to determine what pieces of the application can be written into Augment DSOs, and what pieces need to be part of the application itself. 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 navigation technique, or as an interchangeable component of the application.

DgiPf comes with dozens of example programs; each one is designed to demonstrate a specific feature. Source code of examples, standalone programs and loadable DSOs are included with the distribution.

3.1. Example: Diversifly

Diversifly is a small program that allows non-programmers to load and navigate through model files and apply global transformations and lighting effects; it is the DIVERSE analogue 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 Display class methods 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, navigation and interaction techniques are loaded as DSOs. Diversifly has become popular as an easy way to see how a model will look in various display settings. It has also proven useful as a generic program which can be used to test new interfaces that are encapsulated as DSOs.

3.2. 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 [14][15]. SAVG provides a framework of hardware, software and complementary expertise that application scientists can utilize to facilitate meaningful discoveries.

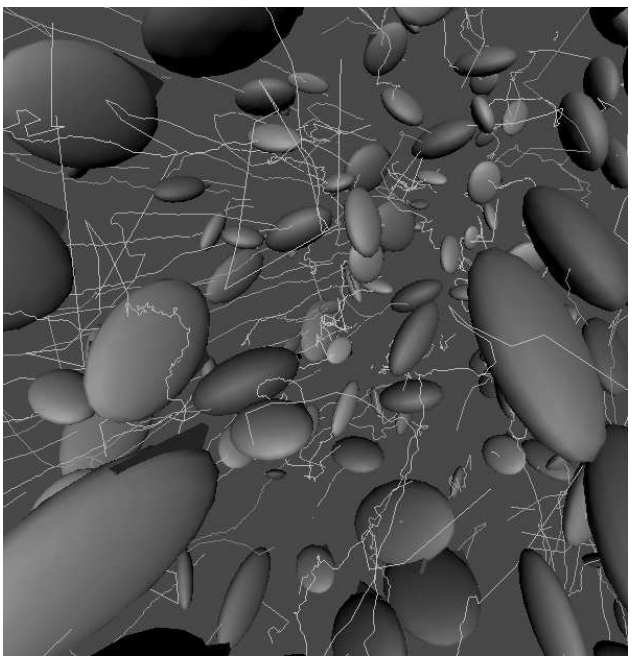

Figure 6. Visualization of flowing concrete

Immersive Virtual Reality (IVR) [16] is an emerging technique with the potential to handle the growing amount of data from large parallel computations or advanced data acquisitions. DIVERSE is among the IVR tools used by SAVG to advance the scientific research of its collaborators. NIST (including SAVG) is physically located across two campuses in Gaithersburg, MD and Boulder, CO. The device independence of DIVERSE allows the same applications to be run on non-stereo 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' x 8' (2.44m x 2.44m) screens flush to the floor oriented 90 degrees 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 [17]. BFRL is leading the Virtual Cement and Concrete Laboratory (VCCTL) consortium consisting of the major cement producers [18]. "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 [19]." Any improvements in concrete such as cost, durability, strength will have a significant impact on the economy. SAVG is a member of the VCCTL consortium and collaborates with the visualization and parallel computing needs.

Figure 6 is a single image from a diversifly based interactive visualization of flowing concrete. Ellipsoids represent concrete particle motion. Lines represent their full path over the simulation time period. The numerical algorithm [20] 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 movie of this visualization is available at: http://math.nist.gov/mcsd/savg/vis/concrete/ [21].

The virtual reality simulation of concrete flow was implemented with DIVERSE. More specifically, it was implemented using diversifly, so no application specific DIVERSE or Performer programming was required. Two general purpose and very simple ASCII file formats were defined. Performer file loaders were implemented as DSOs to provide an interface between the numerical simulation and the immersive environment. The file formats are suitable to a wide range of application areas.

The numerical simulation algorithm was implemented to execute in parallel on both shared memory systems and distributed memory clusters. The output of the simulation is the position, orientation and size of the particles for a series of time steps. Utilizing shell scripts and common filters/tools, the simulation data is transformed into the suitable formats to be loaded, viewed and navigated with diversifly.

The success of this application has shown DIVERSE to be a valuable tool in support of the SAVG collaborators. With this infrastructure in place, SAVG has found a quick and almost effortless (no application specific VR programming) method for a wide range of scientists to get numerical simulations or other research into the immersive virtual reality environment.

4. Future Directions

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

A new DIVERSE module is being completed which augments the OpenGL interface in the same manner that DgiPf augments the Performer interface. This new module, named DGL, allows us to port DIVERSE to platforms that are not supported by Performer. DGL will be scenegraph agnostic; scenegraph libraries that generate OpenGL can be used with DGL to create VE content.

To better support the visualization of 3-D data sets we are creating interfaces to other visualization packages. We plan to first write interfaces to VTK [22], and SGI's OpenGL Volumizer™ [23].

To support networked collaboration we are writing tools to support the creation of collaborative VEs. A collaborative application will allow 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 amongst users. Another example could an "awareness tool," such as a DSO 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 and dynamics. Simulators will be more intuitive and anthropomorphic. All of these will be written as DSOs, so they will be readily available to any application.

We are continually adding to the diversifly program. We are creating a GUI (Graphical User Interface) front-end written in FLTK [24], so it will be easier for users who aren't familiar with Unix to access immersive environments.

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

5. Acknowledgements

This research was supported in part by the Office of Naval Research, grant N00014-00-1-0549, and by Lockheed Martin Astronautics, grant CK 124405.

6. 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.

7. References

All references taking the form of URL web pages were verified on December 4, 2001.

[1] http://www.cs.vt.edu/TR/TR1999.html
Technical Report TR-01-10, Department of Computer Science, Virginia Tech, more completely documents DIVERSE than is possible in this paper.

[2] http://www.fakespacesystems.com/
the home page for FakeSpace Systems, manufacturer of immersive VR systems including the CAVE, RAVE, and ImmersaDesk.

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

[4] http://www.linux.org/
a good starting point for GNU/Linux information.

[5] http://www.gnu.org/philosophy/license-list.html
documents the GNU GPL and LGPL licenses.

[6] http://www.vrco.com/
is the home page of VRCO, the producers of the CAVE libraries, or CAVELib

[7] http://www.division.com/
is the home page of Division, the makers of dVise

[8] http://www.sense8.com/
is the home page of Sense8, the makers of WorldToolKit

[9] http://www.paradigmsim.com/products/runtime/vega/index.shtml
is the home page of MultiGen-Paradigm's Vega product

[10] http://www.cs.ualberta.ca/~graphics/MRToolkit.html
is the home page for the University of Alberta's MR Toolkit

[11] http://www.vrjuggler.org/
is the home page of Iowa State's VR Juggler package

[12] http://www.sgi.com/software/performer/
documents the OpenGL Performer™ API.

[13] http://www.opengl.org/
the home page for the OpenGL Foundation.

[14] http://math.nist.gov/mcsd/savg/
T. J. Griffin, NIST Scientific Applications and Visualization Group.

[15] James S. Sims, John G. Hagedorn, Peter M. Ketcham, Steven G. Satterfield, Terence J. Griffin, William L. George, Howland A. Fowler, Barbara A. am Ende, Howard K. Hung, Robert B. Bohn, John E. Koontz, Nicos S. Martys, Charles E. Bouldin, James A. Warren, David L. Feder, Charles W. Clark, B. James Filla and Judith E. Devaney, Accelerating Scientific Discovery Through Computation and Visualization, Journal of Research of the National Institute of Standards and Technology, 105 (6), 875-894, 2000.

[16] Andries van Dam, Andrew Forsberg, David Laidlaw, Joseph LaViola, and Rosemary Simpson, Immersive VR for Scientific Visualization: A progress report, IEEE Comput. Graph. Appl., Nov./Dec. 2000, pp. 26-52.

[17] http://www.bfrl.nist.gov/goals_programs/HPBM_goal.htm
BFRL Goal: High Performance Building Materials.

[18] http://www.bfrl.nist.gov/862/vcctl/
the Virtual Cement and Concrete Testing Laboratory.

[19] Vision 2030: A vision for the US Concrete Industry, Concrete Vision Workshop, concrete industry's Strategic Development Council, September 27, 2000.

20 James S. Sims, John G. Hagedorn, Peter M. Ketcham, Steven G. Satterfield, Terence J. Griffin, William L. George, Howland A. Fowler, Barbara A. am Ende, Howard K. Hung, Robert B. Bohn, John E. Koontz, Nicos S. Martys, Charles E. Bouldin, James A. Warren, David L. Feder, Charles W. Clark, B. James Filla and Judith E. Devaney, Accelerating Scientific Discovery Through Computation and Visualization, Journal of Research of the National Institute of Standards and Technology, 105 (6), 875-894, 2000, section 5.

[21] http://math.nist.gov/mcsd/savg/vis/concrete/
T. J. Griffin, Visualization of High Performance Concrete.

[22] http://www.kitware.com/
the home page for Kitware's Visualization ToolKit (VTK).

[23] http://www.sgi.com/software/volumizer/
documents SGI's OpenGL Volumizer™.

[24] http://www.fltk.org/
documents FLTK, the Fast Light ToolKit.