A Software System for Inexpensive VR via Graphics Clusters

Benjamin Schaeffer

Integrated Systems Lab
Beckman Institute
University of Illinois
Urbana, IL 61801

Abstract

A scene graph API designed for tightly-coupled distributed 3D graphics is presented. When augmented with other software, like that providing network synchronization functions, this code forms the basis of the Distributed Graphics Database (DGD) system. Experience shows this system is suitable for driving VR displays, providing an inexpensive alternative to SGI computers. We explore approaches to networked graphics and motivate the choice of a scene graph API.

We explore applications of this system using commodity PCs over 10Mbps LANs. In particular, we describe experiences driving a head-mounted display with a 3 PC cluster, the system in question being used to provide an interactive virtual environment for human factors research. We also examine the potential for such inexpensive VR systems to do scientific visualization. Reasonably demanding examples are presented. Additionally, we give benchmarks comparing the system's performance on these examples to that of GLX, including information about data throughput.

CR Categories and Subject Descriptors: I.3.m [Computer Graphics]: Miscellaneous; **Additional Key Words:** Distributed

1 Motivation

This paper focuses on applications of multiple display systems to virtual reality, especially the scenario where two mono video streams need to be generated for a head-mounted display. In many ways, this is the simplest case, requiring the least complicated hardware solution. In contrast, displaying active stereo using images from multiple PCs requires each PC to have a video card capable of receiving an external genlock signal.

Still, the architecture presented can be generalized to such VR systems. For instance, in the networked PC case, the graphics clients on each rendering computer could be easily modified to display images using active stereo, external genlock signals would synchronize the vertical refreshes so that an intelligible active stereo image would show across all the displays, and these displays would have consistency maintained via networked synchronization functions.

A compelling reason to examine different architectures for multiple display systemsis the broad cost range they span. Such systems driven by PC clusters can be between one and two orders of magnitude less expensive than systems based on a graphics supercomputer. As such, cluster-based solutions have the potential to put high-performance VR within the reach of more researchers. Going even further in this direction, with the right software, low-end PC clusters, using slow networks, can give surprisingly good performance. Instead of a dedicated VR machine, it is possible to take random computers from around the lab, already connected by the lab's LAN, string some additional video cabling and get a new display system.

Since software is the primary obstacle to constructing such systems, this paper presents a software project, the Distributed Graphics Database (DGD) system, whose aim is to produce reasonable networked graphics performance on low-end PC systems.

Before discussing the software architecture and implementation, we discuss the uses of multiple display systems and their hardware requirements, making an argument for the use of cluster-based solutions. We then conduct an analysis of existing methods for doing computer graphics over the network. This motivates the design of the DGD system, notably its use of a scene graph API. AFter describing the DGD system, we close with some benchmarks comparing it to GLX for two nontrivial scientific visualizations.

The genesis of this research was a desire to build an efficient, low-cost system for providing left-eye/ right-eye views for an HMD. We needed to provide a virtual environment for a psychology experiment studying the way subjects navigate around complex environments. No SGI computers were available to be dedicated to this purpose and funds were limited. In the end, a workable system was built for about $3000.

2 Overview

Many types of graphics displays involve more than one image. Head-mounted displays need both a right-eye and a left-eye image. Video walls can have anywhere from 4 to 100 screens. CAVE-like immersive environments involve 4 to 6 projectors [4]. Since there is likely no limit to the hunger for more immersive displays with higher and higher resolutions, multiple image displays will become more and more common.

There are three basic system architectures for a multiple display system. The simplest possibility is to split the video signal and send the various pieces to an array of monitors. This approach is used for most public video walls. Unfortunately, the intrinsic resolution of the image remains the same, making this method useful when the only desired outcome is a gigantic video image, but not useful when higher resolution is desired or the display, like a CAVE, is nonplanar.

Another possibility is to have a single powerful computer with multiple graphics cards. This approach places heavy stress on the computer. At the high end, current CAVE systems or high resolution video walls are often driven by SGI Onyx2 computers with 4, 6, or even more InfiniteReality2 boards. On the low end, graphics design professionals commonly work with multiheaded PCs.

Unfortunately, complicated 3D scenes need significant CPU time and bus bandwidth to render with acceptable performance. A good rule of thumb is to have one more CPU than the number of graphics cards. This makes suitable computers, even PCs, very expensive. Single processor motherboards are cheap. Dual processor ones are not, and quad processor boards are even worse. Also, only one AGP slot exists in current PC motherboards. Since AGP overcomes per-

formance limitations associated with the PCI bus, this casts doubt on the ability of commodity PC motherboards to deliver peak performance across multiple graphics cards.

The final basic architecture is to use networked computers, each driving a subset of the available screens. Initial CAVEs, for instance, used this model as early Onyx computers could only drive two graphics boards. This methodology is very cost-effective when using networked commodity PCs, each holding a single graphics card. A notable project taking this perspective is the Princeton Display Wall [5]. There, researchers are investigating rendering images across 8 Windows NT PCs connected by Myrinet. Other projects, such as the Stanford Interactive Mural [6], are using Linux PCs in similar ways.

Conceptually speaking, each display PC becomes a network appliance provding graphics services. The obstacles now become ones of software. Applications need to be designed so that network bandwidth can be minimized. While gigabit networking technologies are commonly available, these are still costly compared to 100Mbps ethernet. A reasonable goal is to see how far one can go with commonly available hardware. Ideally, a hobbyist should be able to take several standard boxes sitting around the house, connect them with ethernet, and have a small-scale graphics supercomputer.

In addition to bandwidth issues, another motivating factor for chosing networking technologies like Myrinet is their low latency. Multiple screen displays, especially those constructed from heterogeneous components, need to be coordinated. For instance, image updates need to be synchronized. It turns out that TCP/IP over ethernet can easily support 50 synchronizations per second using the network synchronization software developed for the DGD system, which is sufficient for most purposes. This is fortunate, since it means that a quality visual experience can be produced with common hardware.

Other software issues include portability, flexibility, and modularity. Keeping overall costs in mind, systems should be deployable using whatever components exist in the lab, be they Linux, SGI, or Windows. Furthermore, heterogeneous display systems should be no more troublesome than homogenous ones. The system needs to be dynamically reconfigurable, so that more components can be added on the fly, and robust, so that failure of a display component will not effect the operation of the whole. As networked components are developed, servers and clients need to be startable in any order and immune to crashes of remote systems.

Given our goal of constructing a flexible, low-cost system, a cluster of commodity PCs connected by ethernet is the right platform to pursue. The hardware components are ubiquitous, and the ability to retask individual PCs for other uses adds to the value of the system. Indeed, the infrastructure might already exist in a given locale, with only software and display devices needing deployment before a visualization center can be created. We focus on this hardware architecture for the rest of the paper.

3 Closely Related Research

There are two areas of active research that touch on the topic of this paper. First, research in distributed virtual environments has received much interest recently. Two systems deserve special mention: Avango [12] and Repo-3D [7]. Both provide a software architecture for sharing scene graph information between nodes in a networked virtual environment. In that, they are similar to the DGD system described in this paper, which also provides a transparent mechanism for sharing a scene graph between multiple processes over a network. The main difference between the research in distributed VEs and the research in cluster-based VR presented in this paper is the degree of coupling between the computers participating in the simulation. The cluster-based VR solution requires a tightly coupled system that can effectively behave as one box.

The second related area of active research is in distributed rendering. Here an especially interesting project is WireGL [3], as pursued by the Stanford Interactive Mural group. In this case, an OpenGL wrapper library is created that distributes OpenGL commands over a network to rendering clients. WireGL also does caching to reduce network traffic, a critical optimization as the true bottleneck to distributed graphics is the network bandwidth. While the cluster-based VR research described in this paper also describes a tightly-coupled distributed graphics system, this research differs in that it is directed towards a VR system and also in that it investigates a scene graph API. We hope to demonstrate that a scene graph API is suitable for highly animated networked displays.

4 Various Existing Software Architectures

The DGD system chooses a scene graph API over an OpenGL-style API. It also broadcasts graphics from a central server to dumb display clients. In this section, we attempt to demonstrate how bandwidth considerations and software engineering issues combine to motivate these choices.

There are two basic approaches to running an graphics application across multiple computers. The first is to modify the application so that it runs on each display computer. The second is to modify its graphics routines so that these broadcast graphics information to the display computers. Each approach has its advantages and disadvantages.

Running multiple copies of the application will potentially yield better performance, depending on the usage patterns of the application. For instance, a code that animates a large polygonal mesh, with every vertex changing position each frame, will likely run better as ”multiple copies” than by broadcasting graphics information. After all, the limiting factor is network bandwidth. Graphics bandwidth over the AGP bus, in rendering lit polygons with a consumer graphics card (GeForce) can easily hit 200 Mbps. A simple architecture where a server redundantly sends information to multiple clients would thus max out at 3-5 clients, even given 1 Gbps networking technology.

Various tricks can be played to wring more out of the interconnect, like having a tree of rebroadcasting nodes or switching topologies of varying degrees of sophistication, but bandwidth is always going to be a problem. This is especially significant in light of the fact that we want to be able to deploy high performance solutions over consumer technologies, and moveover want to be able to flexibly deploy in a variety of situations. By running ”multiple copies”, we sidestep the bandwidth issue associated with broadcasting display information.

The ”multiple copies” approach does, however, have disadvantages. For instance, there is the problem of portability. Under the broadcasting model, the display machines merely need to run clients that understand the graphics protocol spoken by the controlling application. Under the ”multiple copies” model, the application itself needs to be able to run on the display computers. Since the majority of boxes in the system are likely to be display boxes, it is advantageous to impose as few constraints as possible on these. For instance, we might be using PCs and midway through the project decide that Windows NT offers a better driver/ card availability situation than Linux. Under the broadcast model, it will be much easier to make such a transition than under the ”multiple copies” model. Similarly, the broadcast model allows us to use a heterogeneous system, that is relatively loosely coupled.

Other disadvantages of the ”multiple copies” approach are I/O and synchronization. Any display technology spanning multiple boxes requires display synchronization. However, to maintain consistency of the world, the ”multiple copies” model requires that we add compute synchronization to our application. Also, device drivers need to be written to broadcast user input to the various application instances. Finally, any issues with retrieving application-specific data like configuration files, data files, or distributing an incoming network data stream need to be addressed. In general, the problem with the ”multiple copies” approach is in maintaining a consistent state across the multiple application instances.

Note that the ”multiple copies” model is already in heavy use in the commercial world for graphics-intensive internet gaming like Quake Arena. While one computer in the game will act as a server, it coordinates relatively low-bandwidth information like character location. Each copy runs the world from internal data structures. The analogy is imperfect since, for instance, the various computers do not collaborate to produce a single coherent view of the world nor do they attempt to deal with the issues of I/O broadcasting and synchronization. Still, the architecture is similar, and, since the main attraction of running multiple copies of the application is in reducing communications bandwidth, it is not surprising that multiplayer internet games are structured according to this paradigm.

The main advantage of the broadcast model is the way it seperates the controlling application from the display cluster. This gives us more flexibility in the design of our display system. It also potentially impacts the portability of an application to the new display device. Under this model, one confines rewrites of the application to the graphics subsystem, instead of the whole application, as is basically inevitable in the ”multiple copies” model. Furthermore, this allows for dynamic reconfiguration of the display device, without disturbing the running application. Such might be convenient if one wanted to add a head-mounted display for seperate viewing of a world already displayed on a video wall. Similarly, this would be effective for multi-participant shared virtual worlds.

The main disadvantage of the broadcast model is its potentially higher bandwidth requirements over the ”multiple copies” model. Since we are interested in solutions that will provide high performance even over common consumer technologies, this is problematic. Network bandwidth and PCI bus bandwidth are clearly the limiting factors in distributed graphics. Still, via clever design of a broadcast system, one can take steps to minimize the network traffic for a broad class of applications. While moving every vertex every frame will never be suited to a broadcast model, this is, in fact, not the common case.

The additional flexibility mentioned above was an important factor in deciding to base the DGD system on a broadcast model. While there is some concern about bandwidth utilization, many VR applications change relatively little scene information per frame, possibly even just the viewing matrix or, in the case where objects in the scene are being manipulated, the transformation matrix of a single object. Indeed, an important class of scientific visualizations changes only small percentages of the on-screen vertices per frame. The DGD system is designed to take advantage of this fact to minimize the amount of data sent over the network. Later in this paper, we present two pertinent examples, namely a time tunnel visualization of a parallel computer and a simulated visualization of network traffic averages over a thousand nodes.

Approaches that follow the broadcast model can be differentited by the kind of information they broadcast, be it pixel or graphics primitives, and the degree to which screen state information is retained on the display device. These categories determine the bandwidth needs of the applications, and, since communications bandwidth is almost certainly a bottleneck on the high performance end, they determine the overall efficiency of a given application.

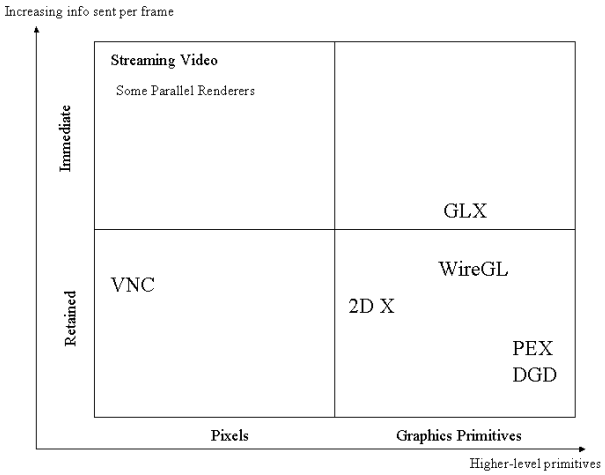

Figure 1 Architecture Classification

Consider the type of data sent to draw each frame. There is both an immediate mode approach where complete information to draw each frame is sent from a server (or client if one prefers X terminology), and a retained approach, where the rendering computers hold most information necessary to draw the frame and receive only small amounts of coordination from a central server or each other. Both modes modes have variations based on the type of data sent, be it pixels or graphics primitives.

There are two good examples of immediate mode pixel broadcast. The first is a ubiquitous one, streaming video. The second is more exotic. Samanta et al. describe a parallel rendering system where a cluster of computers each cooperate in rendering part of a geometry-intensive scene, sending blocks of pixels to a single computer for display [10]. Note that neither example attempts to fill multiple high resolution screens, for fairly obvious reasons. Ignoring compression, sending 20 frames per second of 24-bit color 1024x768 images takes nearly 50 MB/second of bandwidth, putting us into the realm of gigabit networking technologies. Consequently, it seems unlikely that current technology will be useful for broadcasting multiple screens worth of information at frame rates of interest. This consideration eliminated pixel broadcast from consideration during the design phase of the DGD system.

A good example of retained mode pixel broadcast is VNC [9]. A large VNC virtual desktop can be broadcast to VNC clients on display machines to produce a unified high resolution display. A disadvantage to this method is the relatively large bandwidth required, though it has the advantage of being portable over a wide variety of display devices and compatible with existing software. VNC operates in retained mode because it does not rebroadcast every pixel every frame, just those that corresponding to changing areas of the desktop.

X windows provides an example of a system that sends graphics primitives over the network and works largely in retained mode. An X server can support multiple attached displays, so it is possible to display a very large desktop on a single computer. This would, for example, be the case with Onyx2-powered video walls. Since clients send display information to a single X server and the X server cannot be naturally run across multiple machines, it is not obvious how to get a large X display running across multiple networked displays. One approach would be to follow work by the Princeton Display Wall group in producing a large Windows desktop [5]. They create a device driver which distributes graphics primitives across the display machines in the cluster.

GLX gives an example of a largely immediate mode system that broadcasts graphics primitives over the network. WireGL adds more retained mode features to GLX and creates the possibility of having multiple rendering devices connected to a single application. Still, the ability of WireGL to operate in retained mode is limited by its OpenGL roots, raising potential issues of network usage.

In contrast, PHIGS/ PEX is a system that displays 3D images across the network using retained mode. Unfortuantely, like GLX, it is adapted to display on a single target machine. PHIGS provides database functionality for the 3D scene so that the scene can be altered by altering portions of the databse. PEX, the PHIGS X extensions, provide the ability to transmit PHIGS commands over the network, allowing a database to be built on the display side of the network connection.

Since concerns about flexibility constrain us to use the broadcast model for the DGD graphics system, we need to be very careful about use of the available network resources. Retained mode graphics systems, where only the scene changes are sent over the network, offer the best network characteristics. Furthermore, we can further reduce network usage by chosing high level primitives. This is the approach taken by the DGD system. In many ways, our system is conceptually closest to PEX, adding the capability for rendering by multiple boxes.

5 Software Architecture Used in this Project

The most important goals of this project were to produce a readily portable display system, that could run on heterogeneous hardware, that could be flexibly reconfigured, and that would yield good performance on common consumer hardware. Consequently, a broadcast model using a retained scene database on the display side seemed most appropriate. The scene database resides on both the geometry server and the display client. This database is altered by a stream of data packets, corresponding to a simple graphics language. When a display client connects to the geometry server, the server sends the current state across the wire to the new view.

As alterations to the scene are made, these are relayed to the connected clients. Synchronization of screen refreshes is handled by a seperate barrier server/ barrier client pair, communicating using the same infrastructure as in the graphics client/ server pair. In each case, clients and servers can be started in any order and new clients can join display groupings and leave display groupings dynamically without disrupting the visualization. The display client is designed to run permanently. Conceptually, it turns the display computer into a network appliance for displaying 3D graphics. After the geometry server quits, it reverts to a waiting mode until a new geometry server connects.

Part of the challenge of building this system is designing the wire protocol that supports the Distributed Graphics Database. Since, as a rule, the transmitted records are complex, representing a variety of high level graphics concepts, and must be transparently transmitted between a wide variety of machines, with consequent attention paid to data format conversion, there needs to be a semantic abstraction layer between the data and the programmer. This facilitates rapid prototyping of the communication records and keeps the programmer, once the data communications protocol has been written, from needing to consider the details of how complex data is transmitted. Instead of the byte stream given by sockets, we have a stream of structured data records of various types. The component fulfilling this function in the DGD system is the Complex Data Communication Language, or CDCL.

While communications libraries like MPI have means of transmitting structured data, the CDCL library has methods for manipulating semantics in a high level fashion, thus making programs more intelligible and maintainable. The semantics of a particular data type is stored in a data template object. Dictionaries of data templates, as would occur in a complete protocol, can be defined and manipulated, and individual record fields can be accessed via semantic designations. This is similar, for instance, to the data abstractions provided by CORBA or any number of other projects, like in Motif widget definitions.

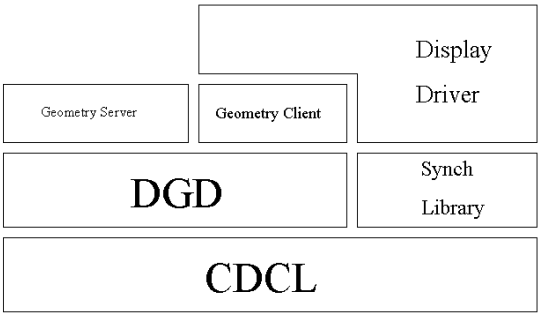

Figure 2 DGD Software Architecture

However, here the CDCL library provides only the minimum functionality necessary to transfer a stream of structured data, with the added ability of the user to perform semantic manipulations. In spirit, it is very similar to the SDDF data format developed by the Pablo group [1], except that it is a very small library and provides network communication functionality that the Pablo system provides only via an additional library, Autopilot, that runs on top of Globus [8]. The CDCL library has the advantage of being independent of any other toolkit. This plus its small size makes it easy to port, a very important consideration.

A language for manipulating the Distributed Graphics Database is built on top of the CDCL protocol. The DGD language is a collection of data records that specify changes to a database, either creating a new entity or modifying an existing entity. The records in the database are arranged in a tree and are associated with both a name and an ID. Not every record is drawable. This allows easy expandability of the graphics primitives without changing the records of the language. For instance, instead of having a triangles record, containing information about vertices, normals, and colors, the DGD language has vertex records, normal records, and color records. The color record is drawable and refers to its ancestor nodes in the database tree for normals and vertices.

In the example of Figure 3, a mesh of colored triangles is attached to the ”world” node in a DGD database. each record is created by a call to a utility function, like makePointsData (...), and is sent to the database via the alter (...) method. In each utility function call, the second field refers to the name of the new node to be created and the third field refers to the parent name. In this case, a branch of four nodes hangs from the ”world” node, terminating in the node ”mesh colors”. The ”mesh points” node stores a collection of point positions keyed by ID. Similarly, the ”mesh triangles” node stores a collection of point ID triples, keyed by their own ID and refering to point data stored in its nearest ancestor point node. The ”mesh normals” and ”mesh colors” nodes store normal and color information respectively keyed by triangle IDs, where these IDs refer to triangles stored in the nearest ancestor triangle node.

```
theData = theDatabase->makePointsData
  (NULL,"mesh points",0,"world",
   numberPoints,pointIDs,pointPositions);
theDatabase->alter(theData);

theData = theDatabase->makeTrianglesData
  (NULL,"mesh triangles",0,"mesh points",
   numberTriangles,triangleIDs,
   vertices);
theDatabase->alter(theData);

theData = theDatabase->makeNormalsData
   (NULL,"mesh normals",0,"mesh triangles",
    numberTriangles,triangleIDs,normals);
theDatabase->alter(theData);

theData = theDatabase->makeColTrianglesData
   (NULL,"mesh colors",0,"mesh normals",
    numberTriangles,triangleIDs,colors);
theDatabase->alter(theData);
```

Figure 3 Code to Create Triangle Set

```
theData = theDatabase->makePointsData
  (NULL,"",pointNodeID,"",
   numberPoints,pointIDs,pointPositions);
theDatabase->alter(theData);
```

Figure 4 Code to Alter a Point Set

Nodes in the DGD database can be editted or deleted after creation. Suppose we want to animate a triangle mesh by moving the vertices. This is accomplished by sending a DGD record to the points node in question. We can query the database to get the node's ID, "pointNodeID", and then, with the IDs of the points to be modified in the array "pointIDs" and the corresponding new positions in the array "pointPositions", we issue the command in Figure 4 to perform the modification.

By making each record of the DGD language a self-contained database modification, we achieve three useful properties. First, the DGD language is inherently a streaming language. Streaming languages are important for sending complex 3D animations over low bandwidth connections like the internet. Secondly, two instances of a database can be kept synchronized by ensuring that they receive the same stream of DGD language records. This is critical for large scale displays powered by multiple rendering CPUs. Each CPU needs to render a database synchronized with the master database residing on the geometry server. Third, the graphics system is inherently designed so that only scene changes need to be sent between frames. Since network bandwidth is the limiting factor in interactive distributed graphics, this feature is critical for the success of any distributed graphics system.

Geometry server and geometry client objects are built on top of the DGD language and the CDCL tools. The geometry server accepts a DGD language input stream, thus altering the state of its internal database, and accepts connections from geometry clients. When a client connects, the server transmits the current state of the database to that client (as a DGD language stream) and thereafter relays its DGD language input stream to the client. The client handles the chores of maintaining its local database. Server and client can be started in any order, new clients can leave or join the server's group at any time, and the software is designed so that any component of the client/ server group can crash at any time without effecting the health of the rest of the system.

Under this architecture, an application wishing to use the distributed display system includes a geometry server object. The applications

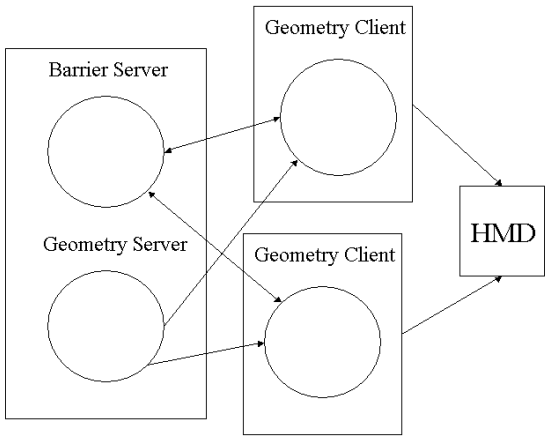

Figure 5 HMD System Diagram

graphics are adapted to fit the DGD language, and DGD records are sent to the geometry server, both to initialize the environment and to perform animations. The application then does not need to be concerned with the nature of the clients connecting to it, even if this is a dynamically changing factor. In fact, all network information is abstracted away from the application, which just sees a scene graph style API.

The next important component is a network-based synchronization barrier, needed to synchronize the various displays. This is important if the display CPUs or graphics cards are imbalanced, for instance if a particluar display machine has better graphics performance than the others in the cluster. Without a synchronization barrier, the user will see the display attached to the high performance machine update before the displays attached to the other machines, causing tearing in the overall display. This is even important with perfectly balanced display machines in a display wall context, but where the different machines are displaying scene fragments of greatly different complexities. For instance, if a scene consists of a room with textured walls, a few geometrically simple objects, and a single object of great complexity, the screen holding the complex object can update with a noticeable lag relative to the other screens.

See Figure 5 for a diagram of how these elements fit together to ower a head-mounted display.

An important characteristic of the design of this system, which it shares with the Stanford WireGL project [6], is that the display computers simply run a display client for the graphics protocol. In other sytems, especially like those for distributed virtual environments, each node in the distributed system will run the application in question. Repo-3D and Avango are good examples of such systems. In these, the library provides distributed shared memory services for the applications. In the case of Avango, a shared distributed scene graph is provided, much like that provided by the DGD [12]. In contrast, here we try to restrict the function of the display computers as much as possible to providing display capabilities only. The display client can be compared to a low-level device driver. It does not know anything about the application connecting to it or what geometry that application will send. It simply receives information according to a well-defined protocol and translates that information into visuals.

This highlights the difference between clustered VR and distributed virtual environments. Namely, the different displays in clustered VR are tightly coupled, while the different displays in distributed virtual environments are loosely coupled. Furthermore, oftentimes the users at various displays can independently modify the virtual

environment, without modifying the world seen by others. A good example of this would be dragging an object in the environment. Generally speaking, the object's position will track the dragging motion on local displays but will only change position on remote displays once the drag completes. This loose coupling reflects the design target for distributed virtual environments, a group of collaborators, each at a seperate display computer with its own I/O devices for interacting with the VE and a degree of autonomy from other participants in the shared environment. In contrast, clustered VR systems need to provide, as much as possible, the illusion of a single system. Displays and I/O need to be tightly integrated across component PCs. This means that individual components can be fairly dumb, only resposible for handling a piece of the VE.

In terms of the systems architecture, the display client handles the geometry client object, the piece responsible for pulling geometry from a geometry server object, and the barrier client object, the piece responsible for keeping the component displays synchronized. It also handles more mundane tasks like display resolution and viewport. Since the display client has exclusive control of a single video display, it acts much like a video driver, with the added configuration functionality needed to let it become a coordinated piece of a multi-screen system. Furthermore, the goal for the display client is for it to be always on, with applications connecting and disconnecting with different content many times.

Portability is a key design consideration. By keeping the software framework simple and the library dependencies small and limited to widely implemented OS services, like sockets, threads, and OpenGL, porting the software is quite easy. The various components of the DGD system work on SGI Irix, Linux, and Win32. Portability is critical in the heterogeneous, cash-strapped, networked environment targeted by this research. One wants to be able to repurpose already existing equipment. One wants to have the flexibility to run any piece of software on any piece of hardware around the lab. Expensive SGI hardware should not go to waste, and choice of a variety of PC operating systems ensures that one will be able to track PC graphics performance and graphics drivers no matter which OS forges ahead.

Clearly some degree of robustness is necessary for components designed to work together over the network, even in a LAN environment. Furthermore, due to the large number of components that must work together, stability becomes quite important. For instance, a typical application might include a controlling application program, display clients running on each display device, a barrier server program, and a device server of some sort, possibly for joystick data or 6DOF tracker data. Care must be taken to simplify the process of starting, connecting, and restarting these components. Otherwise the system becomes too difficult to set-up and use. All components are created so that client/ server pairs can be started in any order, clients can be dynamically added or subtracted from server broadcast groups, and any component can crash or be killed without taking down the rest of the system.

By making sure all client/ server pairs can be started in any order, the application start process becomes much more flexible, ensuring that no rigid start order for the various components. Furthermore, requiring that all servers be able to dynamically add or subtract clients from their broadcast group eliminates a level of configuration. Neither barrier servers or graphics servers need to be told how many clients will connect.

A concrete example pertaining to the DGD system is that graphics displays can be augmented on the fly. One can decide while the application is running to add an HMD display to the video wall. Finally, by ensuring that the whole system can recover from component failure, debugging is simplified and great flexibility in repurposing running components is gained. For instance, with the DGD system one can kill the program providing the graphics data, start up a different graphics server program, and continue as before, without restarting the other system components.

Modularity is also important, both from the point of view of component reuse and from the point of view of system stability. Since components are forced to communicate over the network due to the distributed nature of the system, modularily, via a network communications interface and a wire protocol, is already built into the system. Note that this differs from component frameworks like Bamboo [13] or VR juggler [2] which are based on DSOs and runtime linkers. In these cases, modularily comes from implementing an interface for the manager program that loads and unloads DSOs and facilitates callbacks between them.

Once modular components are created, they are relatively easy to reuse. For instance, part of the system software used at the Integrated Systems Lab is a program for routing 6DOF magnetic tracking data over the network. Since this data transfer is based on a well-defined protocol built using CDCL tools, it is simple to embed the client object in other programs or even to write a completely new client, without changing other infrastructure. This example shows an advantage of the protocol-based approach to modularity over the runtime linker approach. Specifically, the programmer has more freedom to manipulate the system and is not forced to fit his code into a specific framework. Of course, the runtime linker approach has the advantage of speed when components are running on the same box, but this advantage is negligible in a highly distributed environment.

Another important effect of modularity is system stability. In the case of the DGD system, the display client programs are effectively isolated from all other components. If those components fail, for instance the application component crashes, the display clients can remain up and ready to receive a new protocol stream from a restarted application component. In any clustered VR system, decreased reliability of the total system needs addressing. As the number of display devices rises, so to does the possibility that one of them will fail during any given time period. Hardware failure of an individual node should not take down the whole visualization. Modularity allows us to accomplish this goal in the DGD system.

6 Classification of Graphics Applications

Graphics applications can fruitfully be classified based on the amount of data necessary to build a scene and further on the amount of data necessary to build a given frame assuming that the data needed to build the previous frame is already on hand. Note that a large subset of graphics applications need very little new information to draw a new frame. Architectural walkthroughs are good examples of this phenomenon, where the change is in the viewing transformation.

Another class of applications both requires large amounts of information to draw a single frame and relatively little information to draw a new frame. The time tunnel visualization and the network traffic visualization examples of the next section are instances of this class. Such visualizations are especially suited to high performance networked graphics display, provided that the underlying API has the ability to easily cache data on the display side. This is what scene graph APIs are designed to do, and WireGL demonstrates that a caching scheme can also be built into the OpenGL API. Further research would be required to determine whether this can be as general as with a scene graph API.

The opposite end of the application spectrum would be a large precomputed mesh animation, successive frames of which the program attempts to pull out of main memory and display as quickly

Figure 6 Human-Factors Environment

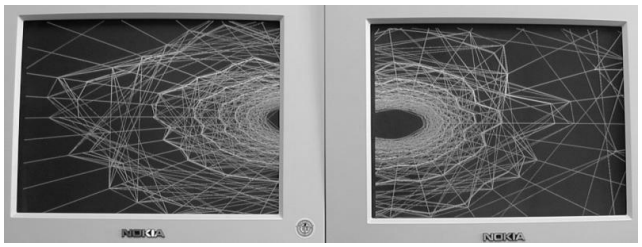

Figure 7 Time Tunnel Scientific Visualization

as possible. Here we have the convergence of a large amount of data per frame and a large amount of data changing per frame. It is unlikely that a simple broadcast model for graphics information will ever provide satisfactory performance for such applications. Instead, methods for retreiving data from local memory or disk would need to be added to the API. Easily integrating this into existing APIs is a topic for further investigation.

7 Practical Uses of the DGD System

We test the system with three distinct types of visualizations. The first is the program that motivated the DGD system, namely a psychology application whose purpose is to study how subjects build a mental representation of a complex virtual environment. The environments are single rooms, large enough so that 4 to 6 are required to perceive the whole horizontal extent and 3 viewport heights are required to perceive the vertical extent. Pictures adorn the walls and various 3D objects are scattered around the floor. The application is split into two independent pieces, an editor program that gives the users a means to create the environments that has a shorter learning curve than a full-featured 3D modeling program and a display program which loads a sequence of environments and, while an environment is loaded, changes the view based on user actions. Both programs run across the DGD system, with the editor program showing that the DGD interface is sophisticated enough to allow a high degree of interaction with the virtual world.

Proper use of display lists could yield good performance for this application under GLX with only one display. There are fewer than 20 objects in the environment, so assuming each is encapsulated in a display list, the bulk of the data sent over the network will be 20 4x4 matrices. While we cannot improve this for the editor program, as it must be able to manipulate every object, we can improve this for the display program. In this instance, only the global transformation matrix needs to be changed. Everything else can be placed in a display list.

Still, the DGD system offers distinct advantages over GLX. First, to view the environment over an HMD, we need to be able to connect the application to two display programs, something GLX cannot do as it is designed to send one stream to one display device. Second, we need to be able to synchronize the two displays, another capability missing from the GLX framework.

The computer system used to run this human factors experiment consists of 3 600Mhz PIII PCs with Voodoo3 graphics cards and 128MB of RAM. The PCs all use the Linux OS. One PC produces the left-eye image, another PC produces the right-eye image, and the third PC runs the experiment, both by sending graphics information and coordinating the framebuffer swaps of the display computers. A Virtual Research head-mounted display is connected to the two display PCs. By using this set-up, we save substantial money over the alternative of using SGI Onyx-class machines for our VR application. Figure 6 shows left half/ right half images instead of the left-eye/ right-eye images that would be sent to the HMD.

The second type of visualization stresses the system much more than the first. Whereas the psychology experiment programs send at most one hundred matrix updates a second, this visualization tends to use the totality of the data processing capabilities available. The Pablo group at the UIUC has developed a metaphor for visualizing the operation of MPI codes on supercomputers called a time tunnel [11]. In this metaphor, the individual processors have their activity logged along lines stretching along the long axis of a cylinder. MPI message activity is visualized as chords stretching between the individual processor lines and through the interior of the tunnel. As new events occur, they are added to the front of the time tunnel, while already added events scroll along, eventually disappearing off the back edge of the time tunnel. The overall effect is of a snapshot of the operation of a supercomputer over a fixed time interval, with that interval travelling through time. Overall, this visualization is highly animated with large amounts of data on-screen at any one time. Here there is a definitive advantage over the GLX-based approach. Due to the dynamic nature of the scene, display lists cannot be used. Consequently, rendering is much faster for the DGD-based approach than the GLX-based approach. See the benchmarks below.

We tested this visualization with data gathered from a solid rocket simulation code produced by the Center for the Simulation of Advanced Rockets at the University of Illinois The particular code is rocflow, which simulates the fluid dynamics of the gases produced by the fuel ignition.

The third type of visualization also tries to stress the system, and is furthermore an example of something that would be difficult for an OpenGL-based API like WireGL to handle over a slow network connection. Specifically, consider a large graph with a thousand cube-shaped vertices, representing a computer network. The cube faces can display various glyphs representing the state of each node, and rotation of each cube can also be used for data display. This fine-grained control makes putting the whole scene in a display list problematic. Hence, an OpenGL-based API needs to broadcast a large amount of matrix data per frame to draw the scene. A scene graph API, on the other hand, only needs to broadcast matrices that are currently being manipulated. Since that is potentially far fewer than the total, this is a large bandwidth savings, which would be especially important for operating over low bandwidth connections such as the internet or 10Mbps LANs. To investigate this situation, we have a program that randomly places 1000 cubes in space and then randomly rotates various cubes and also randomly changes face textures. See benchmarks below for performance versus GLX on this simulated network visualization test.

8 Benchmarks of the DGD System versus GLX

The test machines are jack, steel, and isl25. Jack is a 4 processor Onyx2, isl25 is a dual processor Octane, and steel is a 600Mhz PIII PC with a GeForce graphics card running Linux. These machines are connected by a 10Mbps building network. While SGI is not the main target for this software, it does allow easier comparisons

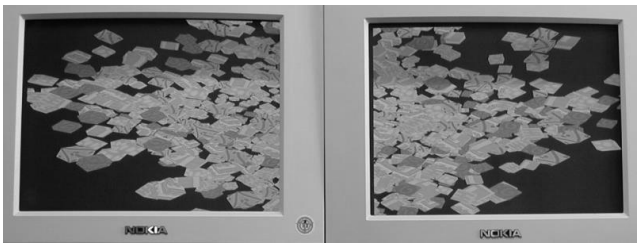

Figure 8 Simulated Network Visualization

between the DGD protocol and GLX since Irix has good support for remote rendering over GLX. We stick to a 10Mbps network to demonstrate the DGD software's ability to operate in a low-end network environment.

Our data first lists the machine pair executing the distributed graphics, with the machine producing the data listed first, followed by the machine consuming the data. The protocol is either DGD or GLX. To eliminate testing irregularities from different code bases, the only factor varied between protocol tests is how the graphics data is rendered. In each case, the visualization code sends data to a local DGD database. If the DGD protocol is used for remote rendering, the protocol relays these modifcations to the connected remote DGD database, which is rendered by the display client program. If GLX is used for rendering, the visualization code draws its local DGD database.

We give estimates for the remote rendering bandwidth used by the GLX and DGD protocols, as appropriate to the test. Finally, we report the framerates achieved via these various rendering methods.

We first test a time tunnel visualization of a solid rocket simulation code, using software and data from [11]. The only visual elements are lines, representing the various events occuring with the supercomputer as the code executed. Approximately 27,000 lines make up the tunnel during the steady state of the visualization.

machine pair	Protocol	Protocol Bandwidth	Framerate
jack → isl25	GLX	700 KB/s	0.8 fps
jack → isl25	DGD	900 KB/s	14 fps
jack → steel	DGD	1000 KB/s	18 fps

Now we test the simulated network visualization, as described in the previous section.

machine pair	Protocol	Protocol Bandwidth	Framerate
jack → isl25	GLX	800 KB/s	1 fps
jack → isl25	DGD	1000 KB/s	8 fps
jack → steel	DGD	1000 KB/s	8 fps
steel → isl25	DGD	500 KB/s	8 fps

In conclusion, using the DGD protocol can lead to significant framerate improvements over using GLX. This is largely due to the fact that GLX needs to transfer the complete scene over the network at each frame, while DGD only needs to transfer the scene's changes. Of interest in these tests is the graphics performance of the Linux box relative to the SGI Octane. Steel is a sub-$1000 PC outfitted with a high-end consumer graphics card, and it gives comparable graphics performance to the high-end SGI box. Also interesting is the Linux box's ability to serve data when compared to a fairly powerful Onyx2, jack. While jack wins hands down, the price/ performance ratio is quite unfavorable to the SGI. It seems possible that clustered PC solutions could provide comparable VR performance to SGIs at a much lower cost.

9 How to Get the Software

The Integrated Systems Laboratory website, www.isl.uiuc.edu, has source code for the core software, plus selected demos. Documentation for the classes and various software components is also available.

References

[1] R. Aydt. *The Pablo Self-Defining Data Format*. www.pablo-cs.uiuc.edu, 1992.

[2] A. Bierbaum, C. Just, P. Hartling, and C. Cruz-Neira. Flexible application design using vr juggler. In *Conference Abstracts and Applications*. SIGGRAPH, 2000.

[3] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[4] C. Cruz-Neira, D. Sandin, and T. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In *Computer Graphics*. SIGGRAPH, 1993.

[5] K. Li et al. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, 20(4):29–37, Jul/Aug 1999.

[6] G. Humphreys and P. Hanrahan. A distributed graphics system for large tiled displays. In *IEEE Visualization*, 1999.

[7] B. MacIntyre and S. Feiner. A distributed 3d graphics library. In *Computer Graphics*, pages 361–370. SIGGRAPH, 1998.

[8] R. Ribler and R. Aydt. *Autopilot User's Manual*. www.pablo-cs.uiuc.edu, 1998.

[9] T. Richardson, Q. Staffor-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, Jan/Feb 1998.

[10] R. Samanta, T. Funkhouser, K. Li, and J. Singh. Sort-first parallel rendering with a cluster of pcs. In *Conference Abstracts and Applications*. SIGGRAPH, 2000.

[11] E. Shaffer, D. Reed, S. Whitmor, and B. Schaeffer. Virtue: Performance visualization of parallel and distributed applications. *IEEE Computer*, 32(12):44–51, Dec 1999.

[12] H. Tramberend. Avocado: A distributed virtual reality framework. In *Virtual Reality*, pages 14–21. IEEE, 1999.

[13] K. Watsen and M. Zyda. Bamboo- a portable system for dynamically extensible, real-time, networked, virtual environments. In *Virtual Reality*. IEEE, 1998.