Networking and Management Frameworks for Cluster-Based Graphics

Benjamin Schaeffer

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

Abstract

Much interesting work has been done in recent years in using PC clusters to power multipipe graphics displays. Tiled display walls and CAVE virtual environments are a natural fit with this research. Examining the many successful system points to problems each needs to surmount to achieve its objectives. Accordingly, this paper attempts to explore the underlying infrastructure, both in software component management and in networking software, needed to make cluster-based graphics systems easy to create, extend, and use.

These ideas have been implemented in Syzygy, a toolkit for distributed graphics used in the Integrated Systems Lab's 6-sided CAVE. Two very different frameworks for doing distributed graphics applications have been built using this toolkit, one a distributed scene graph and the other a method for writing synchronized applications, are described here. The reuse of basic software components improves the maintainability and manageability of the resulting software. In addition, new styles of distributed graphics programming can be implemented on top of the existing infrastructure as they become needed.

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

1 Introduction

In recent years, commodity-based systems have become an attractive platform for visualization. The popularity of 3D video games has created inexpensive graphics hardware that is surprisingly versatile. While there is still a place for the graphics supercomputers, especially in codes requiring large amounts of communications bandwidth, clusters of commodity PC's are adequate for many high-end tasks in scientific visualization and virtual reality. Since economies of scale give commodity systems a better price/performance ratio than the traditional supercomputers, meeting project requirements using PC's leads to cost savings.

Consequently, there has been intense interest in learning how to exploit PC clusters for interactive graphics. A few important types of displays need multiple screens, each driven at high performance. Various flavors of virtual reality, from CAVE immersive displays [6] to simulators to head-mounted displays, have this requirement. Very high resolution screens can be created via a tiled display wall, like the PowerWall [21] and more recent cluster-driven versions that have been produced at Princeton [7], Stanford [12], UIUC [14], among other locations.

Software design is an important challenge in using PC clusters for interactive graphics and visualization. A number of excellent systems have been produced. WireGL, and its successor Chromium, replace the OpenGL shared library with a stub that sends the draw commands to connected render nodes [11]. Scalability is achieved by limiting network traffic, for instance by only sending primitives

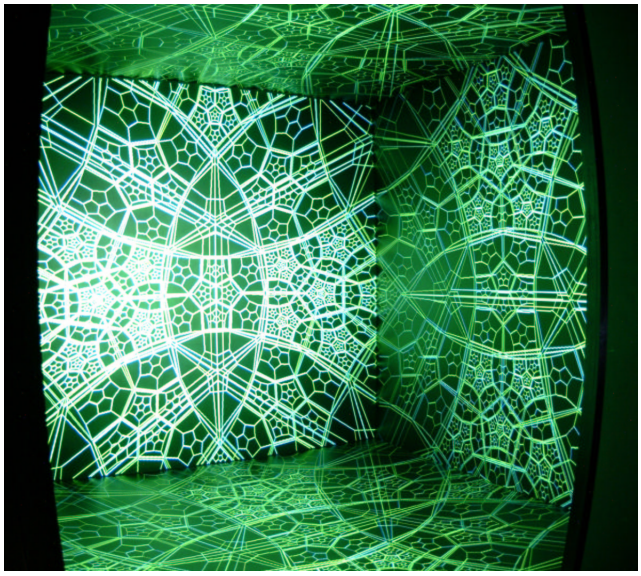

Figure 1 Hyperbolic Geometry in the ISL Cube

to the nodes on which they are displayed [4]. The Princeton Omnimedia group also has a shared library, DGL, that replaces the OpenGL shared library and distributes OpenGL commands over the network [7]. They have also done interesting work in load balancing for cluster rendering [17], and they have investigated software methodologies for writing graphics applications that can have multiple instances running synchronized, one on each render node [5]. Finally, the versatile VR library, VR Juggler, as described in [2] and [3], has an extension, Net Juggler [15] that allows many VR Juggler applications to run unaltered on a cluster.

The Integrated Systems Lab at the University of Illinois at Urbana-Champaign has a six-walled CAVE driven by a PC cluster. The purpose of this structure is to provide an environment both for scientific visualization and for experiments in human perception. As such, we need to be able to support a diversity of graphics application types on our cluster. Furthermore, the cluster-based VR software we produce must be manageable by researchers in psychology with a minimal level of technical expertise. The resultant requirements of high performance, flexibility, and ease of use led to the design of the Syzygy software used at the ISL to power its six-walled CAVE.

2 The Problem

These successful examples of software design for PC cluster graphics, when taken as a whole, teach two important lessons. First of all, there are several valid approaches, each best suited to a particular problem domain. The limited network bandwidth between nodes, relative to the internal memory bandwidth in a shared mem-

ory SMP computer, means that some applications can best achieve high cluster performance with custom communications protocols. Second, there is a nontrivial software infrastructure common to all. Application software components deployed in the cluster need to be managed, possibly across multiple OS platforms, in a simple way. Finally, many of the networking infrastructure needs are common across various software designs for distributed graphics.

Designs that replace the OpenGL shared library have the advantage of excellent software compatibility but can require large amounts of network bandwidth to function well. On the other hand, approaches that focus on running synchronized copies of an application, one instance on each render node, have minimal bandwidth requirements but suffer from reduced software compatibility. The programmer may need to make modifications to the application's source code. For instance, the synchronization may place restrictions on the resources available to the program, like prohibiting threading, or demanding that all state change occur in a defined data path.

In this paper, we look at two basic types of distributed graphics applications. The first is similar to in architecture to WireGL. Here, a generic rendering program runs on each graphics node, turning a received data stream into graphics. An application merely needs to speak to the rendering programs using the appropriate protocol in order to produce graphics on the cluster. This paper labels this architecture as client/server. We deviate from X windows terminology and label the component that produces the data as the server and the components that consume the data as the clients. This is intuitive for cluster-based graphics since there is a single producer and many consumers. Others have classified this architecture as data distribution [5].

The second architecture for distributed graphics applications is to have multiple copies of an application running synchronized, one of each of the rendering nodes. In this case, one copy will control the application, distributing input or other control information to the other application instances. This paper refers to this architecture as master/slave, with the master node passing the information to the slave nodes. Others have classified this architecture as control distribution [5]. Good examples of this architecture can be found in Net Juggler or some of the products of the Princeton group.

The first important common infrastructure is networking code. Messages need to be sent and buffered. Operations on multiple computers need to be synchronized. Connections between components must be managed, preferably transparently to the programmer. Heterogeneous systems built from components with different architectures should be supported, specifically by providing a mechanism to translate message contents between different binary formats. Finally, for performance reasons, it is desirable to be able to overlap network communications threads with computation threads. The underlying framework should support this automatically.

The second important part of a common infrastructure is software component management. The various components comprising the cluster application need to be launched and killed when the application is finished. Components need to be configured in a flexible way when launched and must be able to form connections in a robust manner. If components need to be started in a fixed order, depend on a multitude of configuration files scattered across the cluster nodes, or cannot tolerate another component crashing, the resulting applications built from those components will be fragile and difficult to administer.

In this paper, we describe the Syzygy architecture, starting with its basic communications infrastructure. We show how this basic infrastructure is used to build two types of distributed graphics applications, both a distributed scene graph and a reusable framework

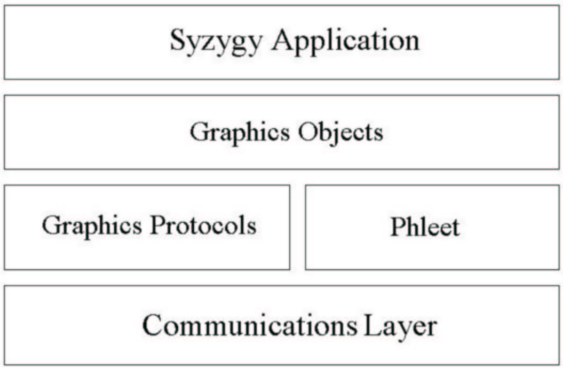

Figure 2 Syzygy Architecture

for building master/slave applications. Next, we show how a basic distributed operating system can ease the management challenges of using heterogeneous graphics software on a cluster. Note that all the software described runs on Win32, Linux, and SGI Irix. This cross-platform compatibility is achieved by using a modular software architecture. The Syzygy architecture is pictured in Figure 2.

3 Communications Infrastructure: Messaging

Several requirements inform the design of Syzygy's communications infrastructure. Distributed graphics programs must be able to be built from components spanning different machine architectures. Furthermore, no single protocol will cover all distributed graphics needs efficiently. The programmer needs to be able to define new communications protocols easily and in a way that is robust with respect to minor changes.

Also, the programmer should not have to worry about the connection management tasks that are inevitable in any distributed system. A common abstraction in distributed graphics involves having a central server feeding updates to clients on the render nodes. A large part of the server's job is managing the resulting connections. Hence, it is especially helpful to have server objects that can deal with multiple connections, each opening and closing at unpredictable times. Finally, synchronization is critically important to distributed graphics. Reality presents a consistent view and the cluster-based display needs to do so as well.

These requirements are not difficult to meet. A framework for defining languages can aid in protocol design while, at the same time, giving the semantic information necessary to do binary format translation from, for instance, big-endian to little-endian machines. Objects with connection management functionality can be designed, as can objects to robustly manage synchronization.

All of the Syzygy components are built on a common communications infrastructure. This infrastructure is message-based and includes objects to handle message semantics. Many similar schemes have been proposed in recent years, including SDDF [1], CORBA's IDL, or even the XML schema underlying SOAP. At a fundamental level, the arDataTemplate object defines the semantics of a particular type of message, be it an array of vertices being sent to a scene graph database or a navigation matrix shared across a synchronized graphics application. Each message type has a name, a numerical ID, and a collection of fields, each with its own name and ID. The fields each hold an array of integers, floats, bytes, doubles, or long

integers.

The arTemplateDictionary object organizes and manages the arDataTemplates into a coherent language for use in a communications protocol. For instance, the Syzygy distributed scene graph API is built on top of a language managed by an arTempateDictionary and comprised of messages whose semantic content is defined by arDataTemplate objects. Raw data storage, including moving to and from the byte stream representations useful for network transfer, is provided by arStructuredData objects. These are initialized by arDataTemplate objects to hold a particular type of message data. The fundamental message construct exposed to the user is an arStructuredData object, which is used to pass data into many Syzygy object methods.

Messaging between components of the distributed system is handled by arDataServer and arDataClient objects. These objects handle three basic functions: connection management, data transfer, and data translation. An arDataServer object is designed to handle many simultaneous connections with arDataClient objects, each connection involving full-duplex communication. A connection thread runs automatically in the background, invisible to the applications programmer, and, as arDataClient objects request connections with the arDataServer, it adds those connections to a database, tagging them with both an ID and a program-supplied string. When an error occurs in a send or receive on one of these connections, the arDataServer automatically terminates that connection and removes it from the database, invoking a user-defined callback on disconnect, if such exists. The szgServer, central controller for the Phleet distributed operating system described later in this paper, is just a thin wrapper around an arDataServer object.

These objects can transfer either single arStructuredData objects or buffers of arStructuredData objects between themselves. Single arStructuredData records are better suited to simple messaging while buffer transfers make more sense for high performance transfers of graphics information, like, for instance, the updates to a scene graph database. An arDataServer can send to a particular connected client, to a list of connected clients, or to all connected clients. The last mode is very useful since some protocols are naturally coded by having a controlling object sending the same data to each connected client. For instance, the Syzygy distributed scene graph works by collecting scene graph modifications during each frame, sending them to all connected clients, each of which then uses that information to draw its next frame. The arDataServer can also receive arStructuredData messages from the connected clients, passing them to user-defined callbacks for processing.

Finally, data translation between different machine architectures is supported at the arDataServer/arDataClient level. For performance reasons, arStructuredData objects are transferred in binary format, which obviously presents problems in interoperability. Each end of a connection between an arDataServer and an arDataClient knows the native binary format of the other. Consequently, given the semantic information contained in an arDataTemplate, the receiving object can translate the incoming byte stream representation of an arStructuredData message into native format. Note that this translation only occurs if necessary, so that all communications between machines having the same architecture approximate the efficiency of a pure memory transfer. By adding this translation functionality, distributed graphics systems can be built from machines with different architectures, for instance MIPS-based and x86-based computers. While communications between the two will never be maximally efficient, certain styles of distributed graphics, like master/slave, require very little bandwidth.

4 Commnications Infrastructure: Synchronization

In addition to data manipulation and transmission, distributed graphics programs require synchronization. Even if the render nodes are evenly matched in power, different views of a scene will render at different speeds, destroying the illusion of a unified display. Consequently, a synchronization call needs to occur immediately before the graphics buffer swap. Experience shows that this synchronization can occur over an ethernet network with latencies of about 1 ms, clearly sufficient for interactive graphics. Of course, software synchronization is only one piece of the puzzle. The buffer swap will complete on each render node at the video card's vertical retrace. If these signals are not synchronized, jitter will occur. The solution is to either use video cards that accept an external genlock signal, like the Wildcat 4210 boards from 3Dlabs, or to use a custom genlock solution like SoftGenLock [16]. With genlock and the Syzygy software, the graphics boards have synchronized vertical retraces and the resulting image is flawless across the cluster display, even under highly animated conditions. Without genlock, jitter is apparent but acceptable, once again, even with highly animated scenes.

Furthermore, in addition to synchronized buffer swaps, the distributed graphics application depends on synchronized message consumption. The basic model for distributed graphics is of a server sending a sequence of messages to multiple rendering objects. Each message in the sequence corresponds to the information needed to draw the next frame. One problem with using TCP networking as a communications fabric is the implicit buffering it does. The completion of a send call on a TCP socket only means that the data has been successfully buffered. Consequently, if the software does not guarantee that rendering objects consume data in synchronized fashion with its production at the server, views on render nodes can lose coherence due to different buffering behavior.

Robust synchronization primitives, arBarrierServer and arBarrierClient, can be built on top of arDataServer and arDataClient. One important characteristic for distributed software is fault-tolerance. Ideally, a synchronization group should be dynamic, with the barrier continuing to function as members enter and leave the group. In combination with the connection management functionality of an additional arDataServer object, say an object that serves graphics information, this feature allows us to build a server that can push synchronized updates to a dynamically changing group of rendering clients. This lets the user, for instance, change displays while an application is running. A program based on the Syzygy distributed scene graph can start displaying in a CAVE, then move to a display wall, and end up on a desktop.

The arBarrierServer object operates largely independently of user intervention. It uses TCP sockets for connection management and exchanging control information with clients, but it uses UDP sockets for receiving notification that a client is at the barrier and for broadcasting the barrier release. The programmer simply tells the arBarrierServer the interface to which it should bind and specifies a block of ports for the various sockets. After this, calling the object's start() method spawns threads that operate the service.

The arBarrierClient object also has relatively few methods. The object connects to an arBarrierServer at a specified IP/port address. It can subsequently operate in one of two ways: either it will automatically join the synchronization group or it will wait until explicitly told to do so. Once the arBarrierClient has joined the synchronization group, it can invoke a barrier by calling its synch() method. This call first sends a UDP packet to the arBarrierServer and then blocks while waiting for a broadcast release packet from the arBarrierServer. The arBarrierServer broadcasts this packet when all clients currently active in its synchronization group have notified it

that they are waiting at the barrier.

We focus on the case where the arBarrierClient takes explicit action to join the synchronization group. When the arBarrierClient wishes to join, it invokes its requestActivation() method, which blocks until a three-way handshake with the arBarrierServer can be completed. This call first notifies the arBarrierServer that it wants to become part of the group. On its side, the arBarrierServer also must explicitly accept the pending additions to the synchronization group using its activatePassiveSockets() method. This method sends the second round of the handshake to the connected arBarrierClient object and blocks until it receives the third round. Once the arBarrierClient receives the second handshake round, still in the requestActivation() method, it sends the third round to the arBarrierServer and returns. When the arBarrierServer gets the third round it returns as well. The arBarrierClient has now joined the synchronization group. The importance of the three-way handshake is that, after completion, each connected object now knows where the other is in its code, and can predict what the effects of a client's synch() call will be.

The case where the arBarrierClient explicitly joins the synchronization group is important. It is useful when we want to synchronize a process that can have components entering and leaving dynamically, as in, for instance, the distributed scene graph's support for rendering clients connecting and disconnecting at will while the application is running. The explicit join also allows us to guarantee that the scene update messages are synchronized in their distribution to the clients and in their consumption on the clients. To accomplish this, code on the client needs to know what code is currently executing on the server and vice versa. The 3-way handshake described above accomplishes this.

Finally, we describe the arSyncDataServer and arSyncDataClient objects. The arSyncDataServer combines an arDataServer with an arBarrierServer to create an object that can perform synchronized message buffer transfers with connected arSyncDataClient objects, which are a similar combination of arBarrierClient and arDataClient. Synchronized message buffer transfers on the server-side, when matched with synchronized message buffer consumption on the client-side, forms the abstract basis for many distributed graphics programs. Consequently, this code can be heavily reused.

On the arSyncDataServer side, the object uses double-buffering, a queuing buffer and a send buffer, to overlap data processing and network traffic. An application continually sends messages to an arSyncDataServer embedded within it using the receiveMessage(arStructuredData*) method. These messages are automatically queued into a send buffer in one thread, while another thread sends the previously queued send buffer to connected clients. When the connected arSyncDataClients have finished consuming a buffer, they call their embedded arBarrierClient's synch() method. With graphics applications, this occurs at the buffer swap. Once the arBarrierServer object embedded in the arSyncDataServer releases the barrier, arSyncDataServer swaps buffers and starts sending the data that had previously accumulated in the old queuing buffer, now the send buffer, and begins queuing data in the new queue buffer.

The arSyncDataServer can handle stateful protocols. In this case, a newly connected client needs to have its state synchronized with that of the server. This is true, for instance, with the Syzygy distributed scene graph. Upon connection, the current scene graph state needs to be transferred to the client. This is accomplished by a program-registered connection callback.

The arSyncDataClient side also uses double-buffering to overlap processing and network traffic. A receive thread continually fills the back data buffer with transmitted information. A second consumption thread is continually spinning, first consuming the front data buffer, using a program-defined callback, and then synchronizing using an arBarrierClient object. Once the front data buffer is consumed and the back data buffer is ready, the buffers swap automatically, and the consumption loop continues. In the case of a graphics application, the consumption callback uses the message buffer to alter internal data structures and then draw the frame using the new information. See Figure 3 for timing details of these operations.

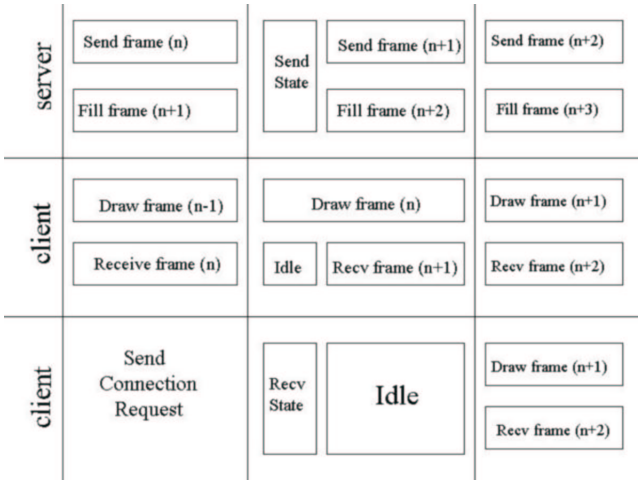

Figure 3 Timing Diagram for Buffer Transfers

5 Client/Server Application Architecture

The above infrastructure is used to simplify experimenting with a distributed scene graph API for Syzygy. Several distributed scene graphs have appeared over the years, mainly in the context of shared virtual environments. Two notable efforts in this direction are Avango [20] and Repo-3D [13]. Of course, here our focus is different, namely cluster-based graphics. The distributed scene graph API fits the client/server architecture. A copy of the scene graph database is manipulated by the application, and generic rendering clients connect to it, thus maintaining synchronized local copies for graphics production.

Using the communications toolkit described in the previous section, one can easily define a protocol for transmitting scene graph information from a server database to multiple client databases. Indeed, connection management, machine architecture independence, synchronization, and message formatting all come for free. Thus, the developer can focus on implementing functionality. An earlier version of the Syzygy scene graph API was detailed in [18]. Pieces of the current version are described below. It is still rough but is sufficient for implementing human perception experiments in the Integrated Systems Lab's facilities.

Using a distributed scene graph is an excellent way to cut down the network traffic between the server and the connected rendering clients. Upon connection, the server can dump its database state and send it to the newly connected client. Subsequently, keeping the databases synchronized merely requires sending changes made to the server's copy, and these changes naturally map to scene graph API calls.

The object arGraphicsServer, a subclass of arSyncDataServer, holds the application's copy of the scene graph database. At the most basic level, the application sends the arGraphicsServer a stream of messages using arGraphicsServer's receiveMessage(arStructuredData*) method. These messages correspond to database modifications and are buffered and sent to connected ar-

GraphicsClient objects, a subclass of arSyncDataClient, after being used to modify the local database. The received message buffer is then used to alter the arGraphicsClient's scene graph database, keeping it synchronized with the application's copy. After this synchronization, the arGraphicsClient draws the database, thus completing the arSyncDataClient's consumption loop described in the previous section.

The scene graph database is a tree, with a special root node. Each node has a type, like texture node, a name, an ID, and a parent. Conceptually, the programmer does two things to the database: creates nodes and sends them messages based on their ID. Texture nodes, for instance, can receive messages telling them to load a new texture. Visibility nodes can receive messages telling them to toggle the drawing of the sub-tree below them. The scene graph is drawn via depth-first traversal.

We now enumerate the node types in this simple scene graph API. Some of these result in drawing, some do not. Drawable nodes include a colored lines node, a textured triangles node, a colored triangles node, and a text billboard node. Some nodes do not directly cause drawing but instead provide information used by the drawable nodes. These nodes include a texture node, a transform node, a points node, a bounding sphere node, a visibility node, and a triangles node. Transform nodes put a matrix on the stack, modifying the rendering of the subtree below it. Points, triangles, and normals node provide geometry for drawable nodes occurring in the subtree below them.

For instance, a colored triangles node holds a sequence of ID's of triangles to be drawn and a sequence of colors. The ID's refer to the nearest ancestor triangles node, which contains an array of point ID triples, ordered by triangle ID. This, in turn, refers to the nearest ancestor points node, which contains an array of coordinates ordered by point ID. Along with the normals contained in a normals node, indexed by triangle ID, this suffices to draw the lit triangles. Similarly, textured triangles nodes contain a sequence of ID's of triangles to be drawn along with a packed sequence of texture coordinates. The geometric information is collected as in the colored triangles node, with the addition that the nearest ancestor texture node is used to provide the texture.

A graphics API wraps the process of sending raw messages to the database, in order to make life easier for the programmer. These commands generate the underlying messages and send them to the arGraphicsServer. An example of how to attach a collection of 200 textured triangles to a transform matrix whihc is in turn attached to the root node follows. In each call, the first parameter refers to the name of the new node and the second parameter refers to the name of its parent. In the case of the dgPoints, dgTriangles, and dgTexTriangles commands, the third parameter refers to the number of elements.

```
dgTransform("world", "root", matrix);
dgTexture("tex", "world",
          "texturefile.ppm");
dgPoints("points", "tex", 600, pointsIDArray,
         coordArray);
dgTriangles("tri", "points", 200,
            trianglesIDArray, pointsIDArray);
dgTexTriangles("textri", "tri", 200,
               trianglesIDArray,textureCoordArray);
```

When a new node is created, the call returns a node ID. This ID can subsequently be used to alter the contents of the node. For instance, suppose a node with ID transformID contains a transformation matrix. This matrix can be changed by a call to:

```
dgTransform(transformID, matrix);
```

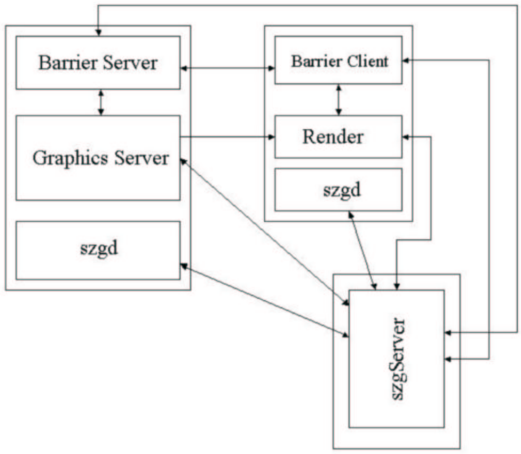

Figure 4 Distributed Scene Graph Software Components

Furthermore, one can conserve bandwidth on mesh changes by only sending the vertices that change instead of the whole mesh. This can be important for highly animated meshes, like the time tunnel application mentioned in the final section of this paper. To simply change some of the points in a given mesh, as stored in the node with ID pointsID, pack the ID's of the nodes to be changed in an array pointIDArray and the coordinates into an array pointsCoordArray and issue the following call:

```
dgPoints(pointsID,numberPoints, pointsIDArray,
         pointsCoordArray);
```

SZGRender is the generic drawing program for this protocol. It is really just a thin wrapper around an arGraphicsClient object. It provides configuration services to the arGraphicsClient, using the Phleet infrastructure described later in this paper. The same Phleet capabilities are used to make the SZGRender program a manageable component of the distributed system, for instance letting it shutdown or reload its parameters in response to a remote signal. See Figure 4 for a diagram of the software components active in this system and the connections between them. This diagram includes the related distributed operating system components. See the section on Phleet for an explanation of their function.

6 Master/Slave Application Architecture

The communications infrastructure can also simplify creating an API for coding master/slave graphics applications. The basic requirement of transferring messages between a server and several rendering clients is the same as in the distributed scene graph, as is the need for this operation to be synchronized with the drawing on the render nodes and for that drawing to occur in lockstep. Furthermore, by reusing the infrastructure, we gain certain benefits for free, namely the ability to add and remove slaves dynamically. This lets us add or even change displays while the application is running.

The master/slave application architecture is realized by an arServerFramework object, which, in turn, is built directly on top of arSyncDataServer and arSyncDataClient objects. An application registers various callbacks with the arServerFramework, sets a few flags, like whether this particular instance will act as a master or a slave, then calls the object's start() method, at which point control passes permanently into the arServerFramework object's hands. If the object acts as a master, it activates an arSyncDataServer, but if the object acts as a slave, it uses an arSyncDataClient. The various callbacks that can be registered are listed below.

```
void sendMessage(arStructuredData*);
void receiveMessage(arStructuredData*);
void updateState();
void draw();
```

The arServerFramework event loop is different depending upon whether or not the object has been started as a master or a slave. If it has been started as a slave, the event loop is recieveMessage, followed by updateState, then draw. If it has been started as a master, the event loop is sendMessage, the updateState, and finally draw. Note that all synchronization considerations now occur transparently to the programmer, and in basically the same manner as in the raw arSyncDataServer and arSyncDataClient pair described in the communications layer section.

To use the arServerFramework object, the programmer defines a simple language, consisting of one type of data record. The arServerFramework object passes a pointer to an empty arStructuredData suitable for holding such data into the sendMessage callback. The application then has the responsibility to fill it with suitable data, possibly depending on the state of I/O devices, the current timestamp, or other considerations, before returning. The arServerFramework object then sends this to all connected clients. On each slave instance of the application, the local arServerFramework object receives the message and passes it into the registered receiveMessage callback. Here, the application can decode the received message and transfer its contents into local storage.

Next, in both modes of operation of the arServerFramework object, an updateState callback occurs, allowing the program to perform an action on the transferred information, if necessary. Once the information is so processed, the next frame can be safely drawn. The view, for instance which CAVE wall, is configured via Phleet, as outlined in the next section, and routines for managing stereo and calculating view frusta are transparent to the programmer.

7 Phleet, A Distributed Operating System

This distributed operating system work has similarities with the Globus [8] and Legion [10] projects. However, these systems attempt to provide infrastructure for supercomputing grids instead of PC clusters.

Two programs form the backbone of the Phleet system, szgServer and szgd. The szgServer controls the distributed operating system. It is built directly on top of an arDataServer object, thus leveraging the communications infrastructure code base. Every software component in the system contains an arSZGClient object that connects to the szgServer. As for szgd, this program is a remote execution daemon with uniform operation across Unix and Win32. Typical operation of a Syzygy system has a copy of szgd running on each computer in the cluster and a single controlling copy of szgServer.

The szgServer performs three functions. First, it holds a database of configuration parameters. Second, it facilitates message transmission between separate components in the distributed system. Third, it keeps a registry of all running Syzygy software components. This simple functionality suffices to mimic many of an operating system's standard functions.

A configuration parameter is a 4-tuple of strings, the first a computer name, the second a parameter group, the third a parameter name, and the fourth a parameter value. For instance, some parameters for a VR rendering program are displayed in Figure 5.

In this example, we show partial configuration information for two render nodes. The computer rendernode1 is displaying the CAVE's front wall, is running in stereo, is showing a window with resolution 640 by 480, and is connecting to IP address 192.168.0.1 at port 10000 to receive geometry information for display. The computer rendernode2 is configured similarly except that it displays the CAVE's right wall.

```
(rendernode1,SZG_RENDER,wall,front)
(rendernode1,SZG_RENDER,stereo,true)
(rendernode1,SZG_RENDER,size,640/480)
(rendernode1,SZG_RENDER,geometry_IP,192.168.0.1)
(rendernode1,SZG_RENDER,geometry_port,10000)
(rendernode2,SZG_RENDER,wall,right)
(rendernode2,SZG_RENDER,stereo,true)
(rendernode2,SZG_RENDER,size,640/480)
(rendernode2,SZG_RENDER,geometry_IP,192.168.0.1)
(rendernode2,SZG_RENDER,geometry_port,10000)
```

Figure 5 Sample Phleet Database Parameters

```
void arSZGClient::setParameter(string computer,
                               string group,
                               string parameter,
                               string value)
string arSZGClient::getParameter(string computer,
                                 string group,
                                 string parameter)
string arSZGClient::getProcessList()
void arSZGClient::sendMessage(string messageType,
                              string messageBody,
                              int destinationID)
```

Figure 6 arSZGClient Methods

The szgServer can also transfer simple messages between Syzygy programs. These messages are a 2-tuple of strings, the first being the message type and the second being the message body. Every Syzygy software component understands two messages, (*reload*, *NULL*) and (*quit*, *NULL*), where the first causes the component to reload its parameters from the database and the second causes the component to stop operation. The szgd program also understands an (*exec*, $<$ *program* $>$) message that causes it to attempt to start a new process with the named executable.

Finally, the szgServer keeps a registry of connected Syzygy software components. Every Syzygy component has an arSZGClient object. On start-up, the arSZGClient opens a network connect to the szgServer and registers itself with a descriptive label. The szgServer assigns the connected client a numerical ID. This ID is the basis of routing messages from one Syzygy software component to another. Furthermore, the software component registry includes information about the computers on which the computers run.

What follows is a sample dump of a component registry, each line starting with computer name, followed by program name, followed by program ID. This particular listing shows an application program running on computer controlnode which is displaying its graphical output via two instances of the rendering program SZGRender, one running on rendernode1 and the other running on rendernode2.

```
rendernode1/szgd/1
rendernode2/szgd/2
controlnode/szgd/3
rendernode1/SZGRender/5
rendernode2/SZGRender/6
controlnode/application/23
```

The arSZGClient object has a simple API that enables it to control the system. This is displayed in Figure 6.

The setParameter method allows the arSZGClient object to set a parameter tuple in the database residing in the szgServer. The getPa-

rameter method allows the object to query that database. By passing the string ”NULL” for the first parameter, the object will return a parameter value corresponding to the computer on which the method executed. This is the most common call type and is used by Syzygy components in their configuration phase. Many parameter values depend on the host computer. For instance, the tile in a cluster-driven tiled display or the wall in a cluster-driven CAVE. Referring to a particular computer in the getParameter method is usually reserved for Syzygy systems programming.

The getProcessList call enables the arSZGClient object to access the registry of the running Syzygy components and, from this information, allows it to determine the ID of a particular software component, as identified by component name and host computer. Finally, the sendMessage method allows the arSZGClient object to use the szgServer's simple messaging API.

The user needs to be able to manage the system from the command line. Very simple programs that wrap the various methods of arSZGClient can provide that functionality.

```
dls
dget <computer> <group> <parameter>
dset <computer> <group> <parameter> <value>
dbatch <file>
dex <computer> <executable>
dkill <computer> <name>
```

The command dls displays the currently running Syzygy software components. The commands dget and dset are wrappers around the getParameter and setParameter methods of arSZGClient. The user can set a large number of parameters at once by sending an appropriately formatted text file through dbatch. Also, an executable can be remotely launched via the dex command and killed via the dkill command. Together, these functions provide the functionality required to control the distributed system. Importantly, they can be used as components of scripts, allowing complex control of the distributed system's behavior.

To speed deployment, the arSZGClient objects automatically discover and connect to the szgServer. When an arSZGClient object launches, it first checks a cached configuration file to find the IP address and port number where the szgServer listens for connections. If this file does not exist, it sends a broadcast packet to a well-known port. The szgServer continually listens on this port and responds with the IP/port pair the arSZGClient needs to make a connection.

Finally, we conclude the section by showing how this system can be used to manage cluster-based graphics programs. To begin, we have szgd daemons running on the various nodes of our system, in addition to a single szgServer. The first example application uses the client/server model. The geometry server object embedded in the application controls a scene graph database that is synchronized with copies running in generic rendering clients, in this case the SZGRender program, that handle the display.

A simple script launches the components on each node in the system. For simplicity we assume 2 render nodes.

```
dex controlnode application
dex rendernode1 SZGRender
dex rendernode2 SZGRender
```

When the various components launch, they connect to the szgServer via arSZGClient objects, download parameters from the parameter database, and use these parameters to configure themselves according to the node on which they reside. Scene graph updates transfer

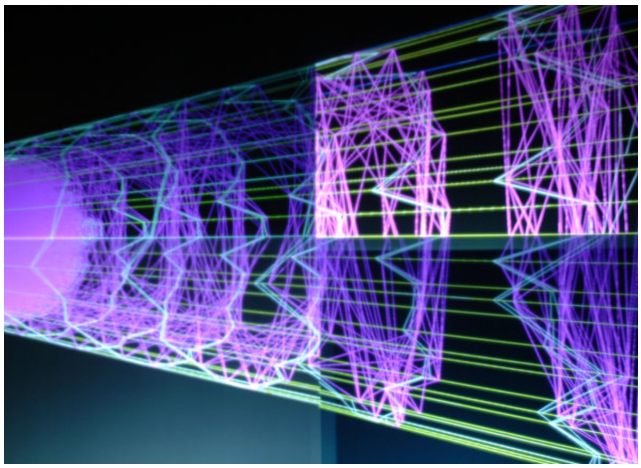

Figure 7 Visualization of MPI Code

from the application to the SZGRender programs using direct network connections, as set up using parameters from the database. Unlike a CORBA architecture, where all messages in the system pass through an ORB, only operating system level messages pass through the szgServer. Once the components have been configured, they communicate with each other directly.

It is possible to reconfigure the system while it is running. As a simple example, suppose the user wants to change the view displayed by a render node in a CAVE-like environment. The user then alters the appropriate parameter in the database and sends the render program a reload message. In response, the render program reloads its parameters from the database and begins displaying the changed view.

To stop the application, we execute the following script, which sends kill messages to each of the components.

```
dkill controlnode application
dkill rendernode1 SZGRender
dkill rendernode2 SZGRender
```

A similar process can be used to start a master/slave application. In this case, we execute an application copy on each render node. A database parameter, SZG_RENDER/master, is used to indicate the node that will control the synchronized run. Different views for the different displays are configured in the same way as the previous application. In this way, the uniform model of software component management aids in moving between applications done in different programming styles and simplifies the overall administration problem.

8 Practical Experiences

We conclude by briefly describing some experiences using this framework to implement applications. First, Paul Rajlich has modified Steve Taylor's open source Quake 3 level viewer to make it suitable for displaying on virtual reality hardware. The resulting software, CAVE Quake 3, was easy to port to Syzygy and hence into the ISL's 6-walled CAVE. Since the program's draw state only depends on a navigation matrix and a timestamp, the master/slave architecture was chosen to do the port.

We also took a scientfic visualization designed for an SGI-driven CAVE and ported it to Syzygy. The ”time tunnel” pictured in Figure 7 is a metaphor designed by the Pablo group at UIUC to visualize the behavior of MPI numerical analysis codes [19]. Events are

pictured as lines, with the long axis of the cylinder representing time. Messages passed between processors are visible as chords cutting through the cylinder's interior. As new events occur, they force old events to scroll away, eventually disappearing. While the line set is thus highly animated, only a small percentage of lines change per frame. Consequently, this is efficiently implemented as a Syzygy distributed scene graph, using the partial mesh update messages mentioned previously in this paper. The data pictured here is courtesy of CSAR, the Center for Study of Advanced Rockets at UIUC.

Another scientific visualization is that of hyperbolic geometry [9], pictured in Figure 1. This visualization is the work of George Francis, using data provided by the Geometry Center at the University of Minnesota. The view only depends on a navigation matrix so this application is efficiently implemented in the master/slave framework.

Finally, we have implemented a psychology experiment that studies how subjects navigate a complicated scene, in this case a room with numerous pictures on the walls, displayed to them in a head-mounted display. This environment was implemented using the Syzygy scene graph API.

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 about the various software components, including how to get a system up and running quickly, 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] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. Vr juggler: A virtual platform for virtual reality application development. In *IEEE VR*, 2001.

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

[5] Y. Chen, H. Chen, D. Clark, Z. Liu, G. Wallace, and K. Li. *Software Environments for Cluster-based Display Systems*. http://www.cs.princeton.edu/omnimedia, 2001.

[6] 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.

[7] K. Li et al. Early experiences and challenges in building and using a scalable display wall system. *IEEE Computer Graphics and Applications*, 20(4):671–680, Jul/Aug 2000.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.

[9] G. Francis, C. Hartman, J. Mason, U. Axen, and P. McCreary. *Post-Euclidean Walkabout*. SIGGRAPH VROOM - the Virtual Reality Room, 1994.

[10] A. Grimshaw and W. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[11] G. Humphreys, M. Eldridge, I. Buck, G. Stoll amd M. Everett, and P. Hanrahan. Wiregl: A scalable graphics system for clusters. In *SIGGRAPH*, 2001.

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

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

[14] NCSA. *Display Wall in a Box*. http://www.ncsa.uiuc.edu/TechFocus/Deployment/DBox.

[15] B. Raffin. *Net Juggler*. http://sourceforge.net/projects/netjuggler.

[16] B. Raffin. *SoftGenLock*. http://sourceforge.net/projects/softgenlock, 2001.

[17] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. Singh. Load balancing for multi-projector systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1999.

[18] B. Schaeffer. *A Software System for Inexpensive VR via Graphics Clusters*. www.isl.uiuc.edu/ClusteredVR/ClusteredVR.htm, 2000.

[19] 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.

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

[21] P. Woodward, T. Ruwart, D. Porter, K. Edgar, S. Anderson, M. Palmer, R. Cattelan, T. Jacobson, J. Stromberg, and T. Varghese. *PowerWall*. http://www.lcse.umn.edu/research/powerwall/powerwall.html.