# Syzygy: Native PC Cluster VR

Benjamin Schaeffer, Camille Goudeseune
*Integrated Systems Laboratory, Beckman Institute, Univ. of Illinois at Urbana-Champaign*
*{schaeffr,camilleg}@isl.uiuc.edu*

## Abstract

*The Syzygy software library consists of tools for programming VR applications on PC clusters. Since the PC cluster environment presents application development constraints, it is impossible to simultaneously optimize for efficiency, flexibility, and portability between the single-computer and cluster cases. Consequently Syzygy includes two application frameworks: a distributed scene graph framework for rendering a single application's graphics database on multiple rendering clients, and a master/slave framework for applications with multiple synchronized instances. Syzygy includes a simple distributed OS and supports networked input devices, sound renderers, and graphics renderers, all built on a robust networking layer.*

## 1. Introduction

In recent years, commodity PC platforms have become useful tools for scientific visualization and virtual reality. Unfortunately the hardware of a single PC is too limited to support high-end applications. Multi-projector immersive environments need multiple high-performance graphics cards to realize their full potential, an impossibility for a PC with only one AGP bus. This limitation can be overcome by clustering multiple commodity computers together. PC clusters are a viable alternative for high-end VR applications (figure 1).

The PC cluster hardware platform is very different from shared-memory SMP systems that have commonly been used for VR. As is common in engineering, this poses a trade-off: PC clusters have a favorable price/performance ratio at the cost of increased software complexity. For instance, application management grows more complicated since built-in OS tools no longer suffice, especially for heterogeneous clusters. Also, communications bandwidth between components of the application can now be a bottleneck and synchronizing render nodes is more difficult, especially for highly animated applications using real-time data.

Syzygy addresses the issues that make writing and running VR applications on a cluster more difficult than on a single computer, since it is designed from the ground up to support PC clusters. Instead of adding cluster support to a library primarily designed for single-computer operation, it focuses on providing tools that directly address the challenges of cluster VR software. This design poses another engineering tradeoff: although Syzygy runs on a single computer, other VR libraries may be better suited to such a configuration.

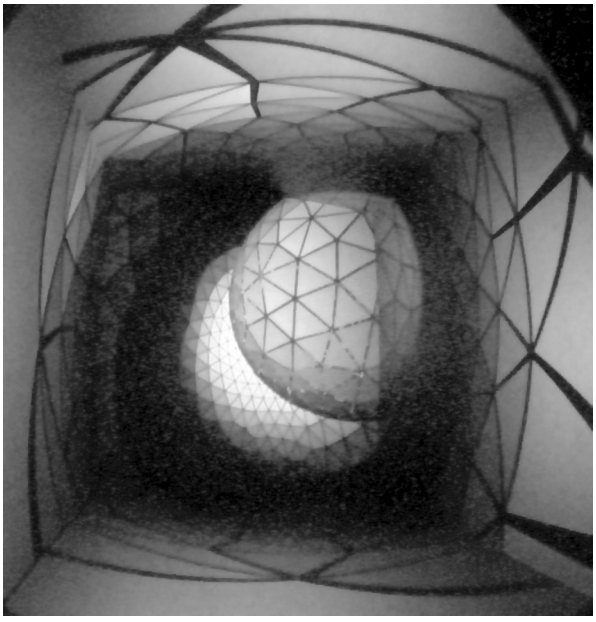

Figure 1. The ISL Cube running a Syzygy port of the "Optiverse" geometrical visualization [7].

Syzygy presents multiple VR application frameworks. Different applications may need different structures to achieve high performance despite the cluster's limited communications bandwidth. Another concern is keeping

render nodes synchronized under the stress of highly animated applications. Syzygy's different application frameworks address these problems by optimizing network usage patterns and synchronization methods for particular classes of applications.

For applications that are highly animated or depend on external data sources, Syzygy provides a distributed scene graph framework. On the other hand, for applications needing small amounts of shared data per frame, Syzygy's master/slave application framework is ideal. With this framework the programmer can define a custom data sharing protocol. In fact this framework can even be useful for applications that require more data sharing, for instance if the shared data can be compressed. Both of these frameworks support networked input devices and sound rendering.

In keeping with its focus on clusters, Syzygy departs from the shared memory model used by most VR libraries. Instead it uses message passing, a better fit to the cluster environment. This message-passing architecture also eliminates common shared-memory programming errors and lets the underlying application framework more tightly control how data is shared and synchronized between application components.

The cluster setting's unique application management problems are handled by a simple distributed operating system, Phleet. This system handles process and configuration management. Phleet particularly simplifies configuration of the distributed system by storing the entire cluster's configuration information in a network-accessible database. This database can be altered from the command line anywhere on the network, and avoids scattering configuration files across all the cluster PC's.

C++ details are found in the Syzygy references, tutorials and source code [16].

## 2. Previous work

The past decade has seen much effort expended on VR library development, both commercial and open source. Of particular note is the CAVELib, developed in the early 1990's at the Electronic Visualization Lab [5]. Initial versions of its immersive projection environment ran on a pair of networked SGI computers. Consequently, the CAVELib includes support for VR clustering. The library shares some fixed data through the cluster, such as a navigational matrix and input device values. Further data sharing can be done through a means of transferring blocks of memory between nodes. In contrast, Syzygy provides more elaborate mechanisms for data sharing and makes these mechanisms central to its API.

Another interesting software effort in this area is VR Juggler [3], a VR library similar in function to the CAVELib. Two extensions provide clustering support, Net Juggler [14] and Cluster Juggler [13]. In each case, a full copy of the VR Juggler application runs on each render node of the cluster. Applications share input events and are synchronized at the end of each drawn frame. Thus, application state remains consistent only if it solely depends on VR Juggler input events.

While this approach has many advantages, notably code reuse, it prevents the developer from exploiting the cluster's full power with highly optimized, application-specific protocols. Also, all nodes of the cluster must start at the same time, with new nodes unable to join later. In contrast, nodes of a Syzygy application can join or leave the cluster during application operation. The application's display can even migrate across VR devices while it is running.

Still another VR library is DIVERSE [10]. While this library does not explicitly support clustering, it does concern itself with networking infrastructure. DIVERSE provides a framework for networked shared memory built around a push model. This networked shared memory can be used to connect input devices to a VR application, or to construct shared virtual environments.

Some projects in distributed graphics are also intimately related to this work. For instance, the Avango project has investigated using a distributed database to implement shared virtual worlds [19]. Recently, they have adapted their research to clustered-based graphics as well. Other projects like Repo 3D are also based on databases or scene graphs [12]. In addition, graphics projects like WireGL [9] and DGL [11] implement software infrastructure which drives a graphics cluster by distributing OpenGL primitives.

Finally, Syzygy's support for networked input devices is similar to the VRPN library [18]. Syzygy's networked sound rendering has a precedent in the client/server model of the VSS sound server [2].

## 3. Infrastructure

Syzygy's layered architecture facilitates reuse of software components. At the foundation is a communications layer. This contains a wrapper library that hides the differences between Windows, Irix, and Linux sockets, threads, serial I/O, and other OS features, increasing application portability. It also contains the Syzygy messaging infrastructure. Syzygy components communicate through efficient binary-encoded messages, and this communications layer contains functions for defining message types, collections of message types (languages), translation between different machine-dependent formats (*e.g.*, endianness), byte-stream parsers, and so on. The messaging infrastructure owes much to systems like CORBA, RPC, and lesser known packages like SDDF [1].

The communications layer also has networking components that send and receive data via Syzygy messages. Server components automatically manage connections and communications with multiple clients; client components connect to just a single server. Robustness is built into this fundamental level by ensuring that components can connect in any order and can survive their connections disappearing at arbitrary times. If a server disappears, its formerly connected clients will keep running, searching for a new server to which they can connect and resume operation. This low-level functionality is reflected, for instance, in the ability of Syzygy applications to add or remove display computers at run time.

Specific functional units are built on top of the communications layer (figure 2). First there are media protocols. There are client/server pairs for managing networked synchronization primitives. There are also units for transmitting and displaying graphics information and sound. Next, Syzygy's distributed operating system, Phleet, uses the same client/server object pairs. Finally, on top of the communications layer Syzygy builds a networked device I/O layer for moving data from input devices like trackers to the virtual environment.

By themselves these client/server pairs for the media protocols or the I/O framework are not very useful in a distributed setting. However, when management and configuration functionality is added to them via Phleet, they become substantially more useful. These network media objects can be semi-autonomous. A sound renderer can be managed remotely and receive sound play requests from anywhere on the network. Similarly, a graphics renderer can have its viewport reconfigured on the fly, while it receives and displays graphics data from first one source and then another. These renderers can be killed and relaunched remotely using Phleet, as well as moved around from node to node in the cluster. Servers for streams of input device events can also be managed from anywhere in the cluster while the application runs.

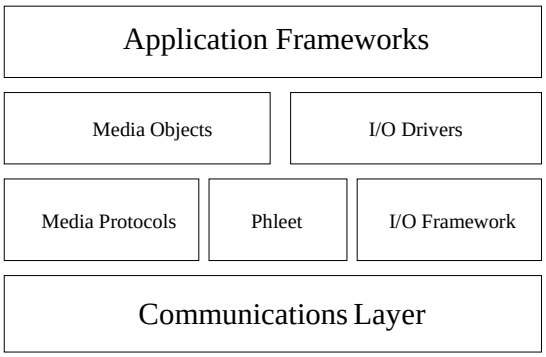


Figure 2. Software layers in Syzygy.

These basic building blocks for graphics, sound, and input devices in the cluster can be manipulated remotely through Phleet, either from the shell prompt or from C++ code. This allows the creation of sophisticated network-aware application frameworks that coherently combine all the components for the programmer. This can be done in different ways, given different application requirements; Syzygy currently implements two different application frameworks.

## 4. Phleet

Phleet is Syzygy's distributed operating system. In concept, it is influenced by Grid operating systems Globus [6] and Legion [8]. Most importantly, it provides facilities for stopping, starting, and inspecting processes on various nodes in the cluster. It has a built-in mechanism for passing messages to running processes. Phleet also provides a network-accessible parameter database that replaces the configuration files that might otherwise be scattered across all the machines in the cluster. This greatly simplifies administration. Finally, Phleet's global locks let complicated manipulations of cluster state be done atomically. These are especially important for application launching. Syzygy's application frameworks examine cluster state on launch, automatically executing needed helper programs and terminating incompatible ones. Cluster-level locking prevents other applications from interfering with this process and leaving the cluster in an inconsistent state.

Because it is built on Syzygy's core networking layer, Phleet can forge a heterogeneous system into a single cluster. It also has a flexible interface: all of its functionality can be accessed from either the command line or from C++.

### 4.1. Phleet Infrastructure Programs

From the shell prompt, Phleet looks like a collection of executables. At its core is the szgserver program that manages the Syzygy cluster. This program stores an online database of attributes of each host in the network. Different attributes are assigned to different Phleet users, so several developers can simultaneously "log in" to the cluster and then independently query and modify the database. Thus, each developer can run the same program on the same cluster with their own configuration.

The szgserver program also maintains a process table of which programs are running on which hosts. Each entry contains the host name, the application name, and a numeric ID unique across the cluster (much like a Unix process ID). Entries are automatically added to and removed from the table as programs start up and terminate. The szgserver can also transfer messages between connected processes. However, for efficiency most Syzygy

components only use the szgserver to configure direct connections amongst themselves.

Several features of the szgserver program ease the creating and maintaining of clusters. Multiple szgservers can run simultaneously on a network, each defining a cluster. Cluster membership is dynamic: a node can switch clusters at any time. In addition, an automatic discovery mechanism lets nodes spontaneously form clusters, without specific configuration. Finally, the szgserver can provide DNS functionality, which is convenient if a server is not available or if noncanonical host names are desired.

Phleet includes many other programs. A critical one is szgd, a remote execution daemon with uniform operation across Syzygy's supported host operating systems. Each node in a Syzygy cluster runs an instance of szgd, letting distributed applications be managed from anywhere on the network. Phleet also includes a host of command line programs for managing the distributed operating system: querying and altering the configuration database, launching or killing processes on a cluster node, listing the running processes on the cluster, and so on.

### 4.2. Phleet from C++

From the programmer's point of view, Phleet's interface is a single C++ class. Constructing an object of this class implicitly connects the program to the cluster. This object's methods can then be called to do anything in Phleet that is possible from a shell prompt, though with increased flexibility. In fact most of Phleet's command-line programs simply wrap these methods.

The C++ interface can do things impossible from the command line. While the command line programs can be scripted to perform simple tasks, their utility is limited. For instance, such a shell script could launch a program on every node of the cluster to start a distributed application. Similarly, a script can terminate a distributed application by sending halt commands to each of the cluster's nodes. But more sophisticated manipulations of the cluster are often desired. For instance, a launching application may scan the cluster, determine if any incompatible programs are running, kill only those, and automatically start any services it needs. This behavior is most easily coded using Phleet's C++ interface.

## 5. Input Devices

Syzygy's I/O framework for sharing input devices over a network uses its core communications layer. Several other software packages have facilities for sending input device data over a network. VRPN provides this functionality and supports a large array of input devices [18]. Cluster Juggler adds cluster support to VR Juggler by mirroring input device data over a network from a device server to the render applications [13]. CAVELib has also long had support for a tracker server that distributes tracker data over a network.

Syzygy's input device framework has several important features: integrated administration, the ability to build virtual devices, robustness, and integrated data filtering. Its input device servers are managed and configured through Phleet, in the same way as all other components of the distributed system. This simplifies cluster administration and tightly couples input device managers with application frameworks, simplifying use of the library. Also, in Syzygy, several input devices on different cluster nodes can combine their outputs to form a virtual device, which appears to the application as a single device. Furthermore, because the input servers are built on the same networking technology as the rest of Syzygy, they can stop, disappear, and restart while the application is running, increasing the application's robustness. Finally, the input device framework provides data filtering. This can be used either for device calibration or for changing input device events from one type into another, letting different input devices drive the same programmatic interface.

### 5.1. Input Device Driver Components

The three most important software components of the I/O framework are input sources, input sinks, and input nodes. An input source produces a stream of input events, formatted as Syzygy messages. Data-producing devices such as joysticks, motion trackers, or RS232 devices interface with the rest of Syzygy via an input source. Special input sources pull in data streams from servers on the network. Input sinks, on the other hand, absorb data. A network input sink streams its received data onto the net. Other types of input sinks provide interfaces to libraries besides Syzygy. For instance, constructing an input sink that posts properly formatted data into a shared memory segment can support CAVELib applications. Finally, input nodes glue together a collection of input sources and input sinks. An input node contains a set of buttons, axes, and matrices whose values are changed by data received from its connected input sources. An input node also has management functions to start, stop, and reset the flow of data through it by manipulating its connected input sources and sinks.

The I/O filters are also important objects. They provide hooks for modifying a data stream, for instance sending raw motion tracking data through a calibration routine, smoothing joystick values with a moving-average filter, or removing extraneous button presses. Chains of these filters are registered with an input node, and they are applied, one by one, to the streams of data

coming from the input sources before those streams are passed on the input sinks.

### *5.2.* Virtual devices

Using these sources and sinks, a virtual input device can be built out of individual physical devices. A familiar example is a VR wand consisting of a motion-tracking sensor and a collection of buttons and axes. The buttons and axes might be derived from a standalone joystick whose driver runs on one cluster node, while the motion tracker's transformation matrix is posted to the network from a different cluster node. Each device sends its stream of input events to the network via network input sinks; the streams are combined at an input node connected to the application, presenting the illusion of a single input device (figure 3).

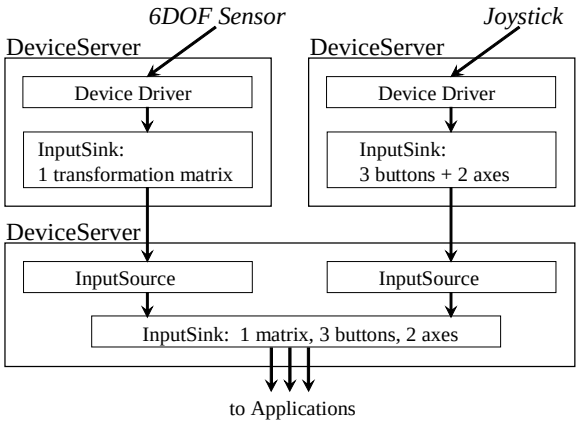


Figure 3. Tree of input devices for a VR wand.

### 5.3. Filters

The Syzygy input device framework includes support for input event filtering. Each filter operates on a stream of input events, possibly maintaining internal state while it operates, and outputs a stream of input events. Multiple filters can be chained together to produce a variety of effects. Calibration of input devices can be represented as a filter on a data stream. For instance, a filter can encapsulate rotating a 6DOF device's sensor values before they are reported to an application. Filters can also work around OS idiosyncrasies. For instance, joystick values are reported using different ranges and offsets on Windows and Linux. Separate filters for each OS easily convert these values to a standard range.

Filters separate input device drivers from the interface requirements for a given class of applications. It is inconvenient to write a separate application interface for each input device. However, due to the expense of 6DOF input devices, VR applications need multimodal input support, as is found in the CAVELib simulator. However, in this case application functionality is changed as well as input modality [5]. DIVERSE's 6DOF simulator connects to a VR application, using a 2-D GUI to manipulate the position and orientation of sensors [10]. Syzygy has a similar 6DOF input device simulator, although here a 3-D representation is directly manipulated. The Syzygy simulator can be conceptualized as a filter that converts mouse/keyboard events into 6DOF tracker events. Similarly, joystick support can be added to a VR application by using a filter to translate joystick events into tracker events.

## 6. Sound

Sound in Syzygy uses the same distributed scene graph infrastructure that is used in the distributed scene graph application framework. Of course, here the scene graph is concerned with sound data instead of graphics. One node in the cluster is a server, generating a scene graph; this is propagated to other nodes running a SoundRender program that turns the scene graph into actual sounds. The sound scene graph includes a stack of OpenGL-style transformation matrices, so sounds can be placed relative to any frame of reference used by the visual scene. A sound's position can be fixed with respect to the world; with respect to a navigation matrix if the user flies through the world; or with respect to some independently moving (graphical) object. If the position and orientation of the user's head is tracked, all playing sounds are properly rendered from that point of view, updated at the same rate as the tracking data.

Nonspatial attributes of a sound are conventional: name of sound file, amplitude scalar, flags indicating triggered (one-shot) or looping behavior. Global attributes of the sound library are also conventional: which sound card to use, sampling rate, overall amplitude, and amplitude roll-off as a function of distance. This simple design is consistent with the "commodity" philosophy of Syzygy: it is no more tied to a single sound card or a sound library than it is to a single graphics card or operating system.

Of course to such sound libraries Syzygy brings its usual features: dynamically restartable renderers, synchronized rendering from multiple viewpoints, and so on. Restartable renderers accelerate development: without affecting the VR application one can modify sound rendering aspects such as sampling rate, CPU load, sound card, or, for that matter, swap out the entire PC. Going a step farther, several rendering PC's can even be driven simultaneously from the host application for rigorous performance comparison from the same live data stream. A practical application of multiple-viewpoint playback is providing a multi-headphone sound display, with the

same sensory data played on each set of headphones. This can provide a high-quality, shared sonic experience in an echo-intensive environment like a rigid-walled, six-sided CAVE.

Network-based sound tools for VR such as VSS [2] can also have multiple renderers for a single VR scene. If synchronization is tight enough, parallel rendering allows individual components of a complex soundscape to be rendered on several low-cost computers, combining their stereo outputs through a small mixer (the inexpensive audio equivalent of a video compositor). However, Syzygy is unique in supporting multiple listening viewpoints and allowing sound renderers to restart without affecting the host application.

## 7. Application frameworks

Syzygy currently supports two application frameworks built using its lower-level, reusable components. These frameworks do the traditional work of a VR library, managing input devices, sound sources, and configuring displays. One framework is a distributed scene graph. The other is a means for constructing applications which can have multiple instances synchronized across the cluster nodes, called the master/slave framework. The Princeton Omnimedia group [4] and others have experimented with cluster graphics using synchronized application instances.

The graphics portions of the Syzygy VR frameworks are described in [17]; an early description of the distributed scene graph framework appears in [15]. We now discuss the trade-offs for using each. No single approach for writing a programming framework can simultaneously optimize flexibility, portability between the cluster and a single computer, and efficiency.

For instance, emphasizing portability between the cluster and traditional single-box VR compromises flexibility. If the same single-box application runs unaltered on each node, it is difficult to access a real-time data stream from a single external network source. Writing output to disk uniquely across the cluster is also difficult. Furthermore, data sharing is restricted to paths implicit in the underlying library, for example through VR Juggler input events. Finally, all nodes may have to begin operation at the same time: this reduces fault tolerance, an important characteristic for cluster-based software. (After all, *n* nodes exhibit defects at least *n* times as often as one node.)

Reasonably general methods of running synchronized copies of an application across cluster nodes cannot guarantee consistent graphics state. However, such methods let the programmer create custom protocols for data sharing between application components. The Syzygy master/slave framework follows this pattern, emphasizing efficiency at the expense of flexibility. In contrast, some application structures guarantee consistency across a cluster's multiple displays by forcing the programmer to funnel all display information through a particular protocol. Syzygy's distributed scene graph framework follows this approach, emphasizing flexibility at the expense of efficiency.

Each application framework contains code to simplify launching distributed applications. When an application starts up, it probes the cluster to determine if other running applications are holding resources it needs, such as graphics screens or input devices. If such are running, it gracefully terminates them.

Next, it launches the services it needs. In the distributed scene graph framework case, szgrender is launched on every render node, DeviceServer is launched on nodes connected to input devices, and SoundRender is launched on nodes that will produce sound. In the master/slave framework case, slave instances of the application are launched on each render node, while input devices and sound are handed as before. During this launch process a lock inside Phleet prevents other applications from launching, lest the distributed system get into an inconsistent state.

In this way, launching an application on the cluster looks to the user exactly like launching one on a single computer, and the application itself contains code to configure the network. This is in contrast to the scripts often used to launch cluster applications. Furthermore, every component of the distributed application can be reliably halted by sending a single kill message to the master application instance. This greatly simplifies application management over, for instance, explicit kill scripts. In demonstrations of a 6-walled immersive projection environment (figure 1) this has been robust enough to cycle through a dozen varied applications, one every minute or two, for ten hours straight.

### 7.1. Distributed Scene Graph Framework

Syzygy's distributed scene graph application framework guarantees visual consistency across the cluster's render nodes, regardless of the application using it. This comes at the cost of visual output being restricted to the scene graph API. Still, this is a reasonable trade-off since the scene graph can be accessed in a multi-threaded fashion, indeed arbitrarily manipulated. In contrast, most other styles of cluster VR impose programming constraints like prohibiting threads, mandating application state modification only through fixed methods, or making rules for using shared memory to maintain visual consistency across render nodes.

The distributed scene graph is built on top of a general distributed database framework, which is shared with the Syzygy sound renderer. Specific database implementations use different languages of messages and different

node types, but they share a common structure. The database essentially is a tree of named nodes. Database nodes are distributed across the cluster by being packed into Syzygy messages, transmitted, and getting unpacked by the recipient. The database is altered by creating new nodes, deleting existing nodes, or sending messages to alter existing nodes.

The database framework also includes functionality for database replication. A sequence of messages to recreate the current database state can be produced at any time. When sent across the network and received by an empty database, this message stream creates a synchronized database copy. Locking prevents database state being altered during replication.

The distributed scene graph includes a variety of node types specific to graphics: transforms, points, triangle sets, line sets, texture coordinates, materials, texture bitmaps, and text billboards, among others. These elements are manipulated via a special graphics language built from Syzygy message types. Instead of forcing the programmer to manually create these messages and send them to the appropriate nodes in the database, Syzygy includes a wrapper graphics API that hides these details of message generation and routing.

Here are three examples of how the graphics database works. First, suppose a texture node has been created, holding a particular bitmap. This bitmap can be changed by sending the node a message asking it to load a different texture file from disk. Second, a "visibility" node's state toggles whether the subtree of nodes below it is rendered. This state can be changed by sending a message to the node. Third, a geometry node, such as a collection of point coordinates, can be altered only partially, thus saving communications bandwidth. For instance, to change the coordinates of only a few points in a points node, it can be sent a message containing just the coordinates that need changing instead of coordinates for every point in the node.

The distributed scene graph application framework goes beyond just the distributed scene graph itself, which is really just an example of a media protocol. First of all, there exists a standalone render client, szgrender, an example of a media object, that runs on each rendering computer in the cluster and accepts a stream of graphics information from whatever distributed scene graph application is currently running. Render clients can be stopped, started, added, or removed at any time while the application is running; active displays remain perfectly synchronized. The application can even be stopped, recompiled, and restarted while the render clients run.

The application framework also manages sound and input devices. The sound media object, SoundRender, and the I/O object, DeviceServer, have all of szgrender's beneficial network characteristics since they use the same underlying communications layer.

The application framework has an API for access to connected input devices and for creating spatialized sounds through the Syzygy sound framework. From the programmer's point of view, the application framework conveniently has no event loop. The framework simply manages configuration of the distributed system. From that point on, the user manipulates the virtual world directly and in an arbitrary manner, with guaranteed consistency in the display. All functionality occurs implicitly: code need not be separated into callbacks for graphics rendering, input events, and so on. The scene graph can be updated in any manner, even from multiple threads; user input can be obtained in any manner. This contrasts with most other methods of constructing VR applications.

### 7.2. Master/Slave Framework

Syzygy provides an alternative application framework for cases where the distributed scene graph application framework described above is inappropriate. This may be because the application's graphics cannot be practically expressed through the scene graph API. The application may also need a more efficient data-sharing protocol, customized to take advantage of its domain knowledge. For instance, a volume visualization application might have a unique way of compressing voxel data.

The master/slave application framework gives the programmer a reasonably general way to construct applications that can have multiple copies running synchronized across cluster nodes. One copy of the application is the master, collecting user input, performing computations on it, and distributing the results to the other nodes, called slaves. As with the distributed scene graph framework, copies of the application can be stopped and restarted at any time, automatically finding each other again and resynchronizing. This again lets the display of a running application move across devices.

The master/slave application framework uses an event loop with callbacks. During initialization, blocks of memory are registered with the framework. The framework will transfer memory from the master to the slaves at a well-defined point in the event loop, translating binary formats as necessary between machines.

We now describe the event loop in some detail. First the master polls the input device, recording the values of its various buttons, axes, and 6DOF sensors. Next, all application instances execute a pre-exchange callback, possibly differently for the master as compared to the slaves. On the master, this callback fills the registered memory blocks with information to be transferred, possibly as computed from input device files, network data, file I/O, or other things. After this, the master transfers memory to the slaves. Then all application instances, master and slaves, execute a post-exchange callback. At

this point, the registered memory is in a consistent state as is every copy's local view of the state of the input device, timestamp, and so on. After this the master executes a sound-rendering callback, and then all application instances then execute a draw callback, synchronize over the network, and finally swap graphics buffers.

Some data sharing is automatic between master and slave application instances: input device information, a timestamp, random number seeds, and a navigation matrix. Consequently, viewer-type applications, especially architectural walkthroughs, can be quickly built using the master/slave framework with no explicit code for data sharing. More complicated data sharing can always be added explicitly.

Data consistency is not guaranteed, because of the added flexibility given by the master/slave framework. We assume that all information necessary to construct the graphics state is shared. We also assume that the master fills the registered memory blocks between the start of the event loop and the start of the memory transfer.

## 8. Conclusion

Syzygy is a VR library that focuses on cluster computing. Instead of trying to present the developer or user with the same picture whether the application is running on a cluster or a single computer, it provides an API tuned to the cluster setting. As such, it makes it easier to write high performance VR software for this challenging hardware platform. Furthermore, by exposing some cluster complexities it can present a unified and simplified management picture. This directly addresses one of the problems of cluster-based VR: how to manage the resulting complex system.

Syzygy is free software licensed under the GNU LGPL, downloadable from the web [16].

## 9. References


[1] R. Aydt. *The Pablo Self-Defining Data Format*. ftp://vibes.cs.uiuc.edu/pub/Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz, 2000.

[2] R. Bargar, I. Choi, S. Das, and C. Goudeseune. "Model-Based Interactive Sound for an Immersive Virtual Environment." *Proc. Intl. Computer Music Conf.* 1994:471-474.

[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." *IEEE Virtual Reality* 2001:89-96.

[4] 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/papers/ccgrid.pdf, 2001.

[5] C. Cruz-Neira, D. Sandin, and T. DeFanti. "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE." *Computer Graphics*, ACM SIGGRAPH 1993:135-142.

[6] I. Foster and C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit." *Intl. J. Supercomputer Applications* 11(2):115-128, 1997.

[7] G. Francis, J.M. Sullivan, and C. Hartman. "Computing Sphere Eversions." In *Mathematical Visualization*, (H.-C. Hege, K. Polthier, eds.), Springer, Berlin, 1998, pp. 237-255.

[8] A. Grimshaw and W. Wulf. "The Legion Vision of a Worldwide Virtual Computer." *Comm. ACM* 40(1):39-45, 1997.

[9] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. "WireGL: A Scalable Graphics System for Clusters." *Computer Graphics*, ACM SIGGRAPH 2001:129-140.

[10] J. Kelso, S. Satterfield, L. Arsenault, and R. Kriz. "DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments." *IEEE Virtual Reality* 2002:183-190.

[11] K. Li, H. Chen, Y. Chen, D.W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J.P. Singh, G. Tzanetakis, and J. Zheng. "Early Experiences and Challenges in Building and Using a Scalable Display Wall System." *IEEE Computer Graphics and Applications* 20(4):671-680. 2000.

[12] B. MacIntyre and S. Feiner. "A Distributed 3D Graphics Library." *Computer Graphics*, ACM SIGGRAPH 1998:361-370.

[13] E. Olson. *Cluster Juggler – PC Cluster Virtual Reality*. M.Sc. thesis, Iowa State University, 2002.

[14] B. Raffin and J. Allard. *Net Juggler*. http://sourceforge.net/projects/netjuggler, 2002.

[15] B. Schaeffer. *A Software System for Inexpensive VR via Graphics Clusters*. http://elim.isl.uiuc.edu/ClusteredVR/paper/dgdpaper.pdf, 2000.

[16] B. Schaeffer. *Syzygy: A Toolkit for Virtual Reality on PC Clusters*. http://syzygy.isl.uiuc.edu/, 2002.

[17] B. Schaeffer. "Networking and Management Frameworks for Cluster-Based Graphics." *Virtual Environment on a PC Cluster Workshop*, Protvino, Russia, 2002.

[18] R. Taylor, T. Hudson, A. Seeger, H. Weber, J. Juliano, A. Helser. "VRPN: a device-independent, network-transparent VR peripheral system." *Proc. ACM Symposium on Virtual Reality Software and Technology* 2001:55-61.

[19] H. Tramberend. "Avocado: A Distributed Virtual Reality Framework." *IEEE Virtual Reality* 1999:14-21.