Adapting Event-Based Applications for Synchronization in VR Clusters

Dmitri K. Lemmerman Andrew S. Forsberg

Brown University Department of Computer Science
PO Box 1910, Providence, RI, 02912, USA
{dlemmerm,asf}@cs.brown.edu

Abstract

We present a method for adapting an event-based virtual reality application originally implemented for shared-memory-based rendering systems to a cluster rendering system. We use a master/slave model: the master node provides synchronized event streams for consumption in duplicated application instances on each slave node. Our main contribution is a simple, MPI-based approach to achieve robust synchronization of an event-based VR application in a cluster. Though polling-based synchronization methods exist, they cannot readily support an event-based paradigm. Our method helps resolve this dilemma.

1 Introduction

Event-based models for developing interactive applications are well known in computer graphics [4]. In an event-based application, user input generates events, which are records of state changes for a given user-controlled device. Events can also be generated by system resources such as timers. Events are stored in a queue and dispatched to event listeners, objects that have registered with the event dispatcher to be notified when a particular class of event is dispatched. The event listeners in turn update the application state based on each event notification. Event-based applications contrast with polling-based models where the application queries a device's state periodically to determine whether its state has changed since it was last queried. VR researchers at Brown have found the event-based model compelling for both conceptualizing and developing interactive applications, and many of the applications driving our 4-walled Cave [3] utilize such a model. These applications were originally developed for a multi-pipe, shared-memory graphics system.

Over the past several years, much work has been devoted to driving multi-display VR setups with distributed, commodity graphics nodes rather than a shared-memory system. The reasons for migrating from high-end, shared-memory graphics architectures to commodity-based cluster architectures for VR applications running on multi-display systems are well-documented [12] [13]. Nonetheless, transitioning code from a shared-memory system to a cluster is not without its challenges. Regardless of architectural approach, drawing to multiple displays requires some mechanism of sharing information amongst each graphics subsystem. For a coherent image based on a consistent application state to be rendered across all displays, this sharing is necessary.

Combined with their specialty hardware, Silicon Graphics' IRIX ™ operating system solves this issue with an intrinsically shared memory model. Only for a few specific instances (e.g. OpenGL display list calls) must the programmer ever acknowledge the existence of multiple graphics cards; most often the mode of ensuring synchronization between pipes is transparent. In clusters, the burden of information sharing is shifted to the programmer. The explicit synchronization of information across all nodes is the major issue in adopting a cluster for multi-display rendering.

Our method seeks to easily adapt an event-based VR application originally implemented for the single shared-memory node case to a cluster environment. We accomplish this by duplicating instances of the application across all cluster nodes and inserting several stages into the event dispatching mechanism to distribute events to each application instance and to synchronize their consumption. These additional stages in the event pipeline are encapsulated within Message Passing Interface (MPI) based [8] code, which we then interpose in the original applications and which conceals the intricacies of MPI from the application implementation. Though this software is not yet available publicly, we hope that our description serves as a beneficial model for others attempting to implement event-based applications for VR clusters. It should be noted that a complete multi-display VR system has additional requirements, such as configuring viewing projections for individual tiles, that will not be treated here.

The structure of this paper is as follows. A discussion of alternative methods for synchronization in rendering clusters is given in Section 2. The description of our general

approach and implementation details is given in Section 3. Section 4 describes several applications that we adapted to the cluster successfully. Finally, our conclusions and ideas for future research are presented in Section 5.

2 Related Work

Several methods for synchronizing applications in a cluster graphics environment have been proposed. These range from an extremely low-level approach of transmitting rendering primitives across the cluster, to a master/slave approach whereby one master node controls the execution of many slave nodes, to a distributed scenegraph approach in which all or part of a scenegraph object is stored on each node and changes to this scenegraph are propagated synchronously across the cluster.

Chromium [7] takes the first of these tacks. It replaces the OpenGL library on a given node and forwards an application's GL calls to one or more nodes in the graphics cluster. This technique is astonishingly elegant as any arbitrary OpenGL-based program can theoretically be run across multiple displays. Unfortunately, the network bandwidth requirements for applications with large amounts of changing geometry are prohibitive. To address this, Chromium can employ sort-first parallelism whereby each primitive is sent (in the ideal case) only to the node(s) that ultimately must display it. While effective for spatially well-organized geometry, in the worst case, data may need to be sent to all nodes [13].

Distributed Open Inventor [6] and OpenSG [15] implement a scenegraph structure that can be distributed across the nodes of the graphics cluster. Once this data structure exists on the nodes of the cluster, network messages invoking scenegraph updates maintain its synchronization across all nodes. If the scenegraph changes are minimal, this approach is very efficient as network traffic is proportionally minor. Of course, such proportionality becomes a serious disadvantage if continuous and major changes are needed, especially if new geometric objects are often added and removed.

Adapting an existing event-based, shared-memory application to a cluster environment using Chromium or OpenSG eliminates the need to synchronize events across the cluster as the synchronization occurs at the level of graphics primitives instead (OpenGL streams and scene graph nodes, respectively). From the API users' point of view, application state is, in both cases, only maintained on one node. Nevertheless, the aforementioned bandwidth considerations render these approaches infeasible for some applications.

A good balance of transitional simplicity and synchronization efficiency is achieved in the master/slave model, implemented in both the VR Juggler [2] and Syzygy [12] APIs. VR Juggler is a complete VR system that includes device drivers, windowing support, a math library, runtime reconfiguration capabilities, and clustering support. The support for running in a cluster environment is provided by NetJuggler [1] in VR Juggler 1.0 and is built in to the second version of VR Juggler as Cluster Juggler [10]. The master/slave approach achieves synchronization by running an identical copy of the rendering application on each of the cluster nodes. The master node (which could also be a rendering node) is responsible for regulating these duplicated application instances, such that at any given time each instance is in the same state. In contrast to our framework, both APIs employ a polling-based model for supporting user interaction. User interface code periodically queries current values of device data. Thus, synchronization is accomplished by specifying a strict application loop employing network barriers to ensure that each duplicated application instance will have the same user-inputted device data available for polling prior to the rendering calls for the next frame. We borrow from these approaches by mandating synchronization of input at a specific point in the application loop. Where the polling-based methods use this step to update the referents of device data objects, during synchronization, our method dispatches any queued events for consumption by registered event listeners.

For the VR programmer who already has stable applications in a non-cluster rendering environment, the master/slave model represents the simplest transition. Running many copies of the same application on many nodes should be trivial for an application that runs on one node. The internals of the application require no changes other than the mechanisms to maintain input synchronization. Correct rendering only requires adjustments to the viewport transform based on the physical geometry and arrangement of the display tiles. Network traffic is very reasonable as typical input types needed for interactive VR applications have very small space requirements.

Several features unique to Syzygy deserve mention. In addition to a master/slave model, Syzygy also provides a distributed scenegraph, which runs on the same infrastructure, but maintains synchronization in a very different manner (as was discussed in regards to OpenSG). Syzygy can also robustly handle the expected or unexpected entry and exit of nodes into the cluster.

3 Method

3.1 Goals

Our primary goal was to transition event-based applications originally written for shared-memory, multi-pipe SGI™ architecture to a cluster of commodity rendering nodes. This overarching goal in turn produced several corollary goals:

1. Minimize changes to applications and underlying event framework

2. Support interactive VR applications

3. Robustly manage processes on all cluster nodes

3.2 Approach

In accordance with our first goal, we chose to employ the master/slave approach for synchronization. Because in the event-based model application state is controlled by event listeners, our problem became one of guaranteeing the synchronization of event notification in the duplicated application instances.

On an abstract level, an event-based application behaves as follows. A device handler executes asynchronously to the application and generates events whenever the state of a user-controlled input device changes. These events are stored in a queue that is accessible by the application. In an interactive VR application, devices continuously generate events (e.g. head tracking); hence we refer to an event stream that feeds into this queue. At some convenient point, the application transfers execution to an event dispatcher. The job of the dispatcher is to notify event listeners that have registered interest in the particular class of event being dispatched. These listeners in turn update the application state based on the event's information. For example, the viewpoint update code in a VR application would register to be notified of events generated by the head tracker device. Once registered, each event generated by a change in that tracker's state (i.e. position and/or orientation) would be dispatched to that code, and the viewpoint would be changed accordingly. Figure 1(a) depicts this event-based model for the single node case.

Fundamentally, our approach for adapting to the cluster is to interpose an event distribution scheme between the event dispatcher and the event listeners. To this end, the event stream is queued at one node as before (we refer to this as the master node). The master node executes an event dispatcher and registers a single event listener that listens for all events. On each event notification, the listener serializes the event and stores it in a buffer. Prior to rendering, this serialized event data buffer is broadcast to the multiple rendering nodes (we refer to these as client nodes). Each client receives the serialized event data, reconstructs the event objects from their serialized form, and dispatches these events locally. Because the applications on the clients are duplicates of the original single-node case, the same event mechanisms are present, and this dispatch will notify the same listeners and update the state in an identical manner to the application running on a single node. Figure 1(b) depicts the additional components we implement for the cluster.

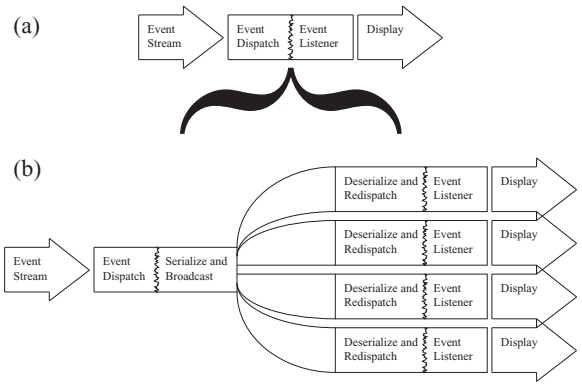

Figure 1. (a) Illustrates typical event based model. (b) The additions needed to this model for synchronization in a cluster. New pieces are interposed between the original event stream arrival and consumption points.

3.3 Implementation

As mentioned above, our implementation introduces several steps between the event dispatcher and event listeners of an existing shared-memory VR application. To achieve our first goal of minimizing changes to the existing application, we first implemented most of the code for these additional steps in a group of MPI processes running on all nodes. Second we created a new modular device driver for the original rendering application that communicates with the MPI client process via inter-process communication (IPC).

We chose to use MPI as it has become the *de facto* standard for cluster-based computation, and, moreover, is optimized to perform with very low latency when communicating with the Myrinet^{TM} [9] networking hardware in our cluster. This low latency is a requirement for our second goal of running VR applications interactively.

The MPI root process receives the event stream generated by the same device handler module used in the original application. This root process loads the same device configuration as the single-node case, and thus registers to listen for identical event classes. The primary sources of events for this handler are a VRPN [11] server and a custom interface to listen for TCP/IP network messages. When the MPI root process begins, it allocates a buffer into which its single event listener will serialize all events. After all available events are dispatched and subsequently serialized on the root node, the buffer is broadcast to each client and stored in IPC shared-memory.

After this broadcast, each MPI client process blocks until its associated rendering process is ready to dispatch new events. The readiness of a given rendering node is com-

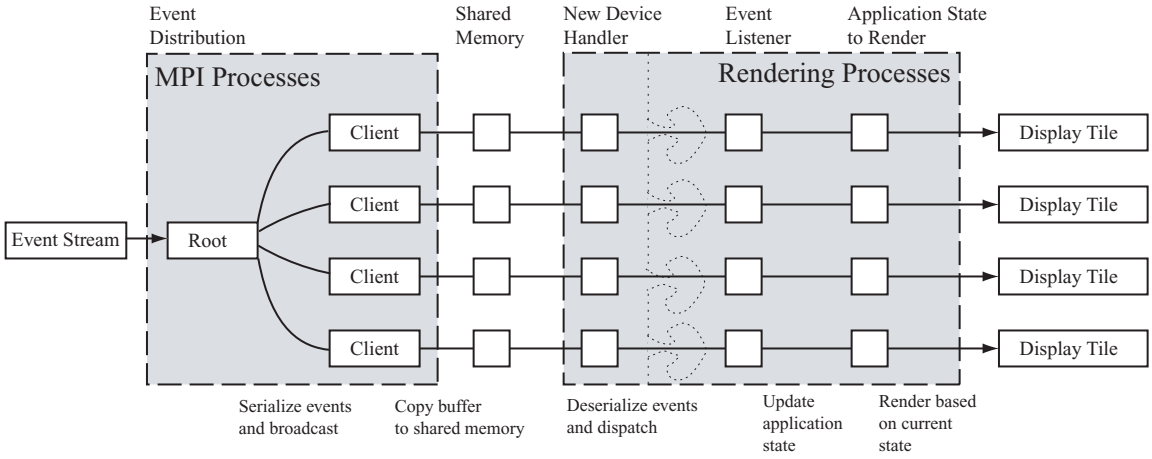

Figure 2. Our solution to event synchronization interposes MPI code to regulate the distribution of events to each rendering node. The event stream is received by the root MPI process, which serializes and broadcasts the stream to the MPI client processes. These processes supply the event stream along with synchronization barriers to the rendering processes via IPC. The dotted line represents the division between the original application and the new device handler piece we added to generate events from the shared-memory buffer.

municated via an IPC shared-memory segment storing a Boolean value, referred to as the "ready flag." The rendering process sets the ready flag to true immediately after its OpenGL buffer swap.

After this ready flag is set to true, the rendering processes block until the MPI clients pass an MPI barrier and set another IPC Boolean, the "all-ready flag," to true. At this point the serialized event data is "deserialized." The reconstituted event objects are then dispatched into the unaltered event distribution module from the original application. An analogous handshake using a pair of IPC Booleans (referred to as "done flag" and "all-done flag") combined with an MPI barrier forces each rendering process to finish the event receiving code, and thus, start the rendering code, simultaneously. These modifications are implemented by adding new device handler code. The new code replaces, but adheres to the same interface, as the original application's device handlers, thus making its inclusion transparent to the rest of the rendering program. Figure 2 depicts the entire system of the rendering and MPI processes.

3.4 Timing

As explained above, the MPI and rendering processes execute in lockstep while the rendering node is dispatching events from the serialized buffer. After this is completed, the rendering nodes make their graphics calls, and the MPI nodes serialize and broadcast new events. We assume that the rendering always takes significantly longer than the event broadcast for reasons described below. Figure 3 illustrates the timing of the system along with pseudocode for each step.

Serialization involves tightly packing raw event data into the broadcast buffer. Each serialized event begins with one byte signifying the type of event followed by the data specifying the particular event of that type. Tracking events, for example, comprise a four-by-four matrix, hence 16 doubles are stored in addition to a string indicating the specific tracker that generated the event. Most events require around 100 bytes of storage. Our tracking system updates at approximately 100 Hz. If we assume a minimal interactive framerate of 20 fps, then each tracker will generate about 5 events per frame. Thus, each event broadcast is on the order of 1000 bytes. Our Myrinet™ interconnects can transmit at 250 MB/s with a latency of 5 μs with the MPICH GM implementation of MPI [9]. Therefore, the total time needed for the synchronized distribution of events is of order 10 μs. Contrasted with an average time to render one frame of order 10 ms, the overhead of the cluster is negligible.

It should be noted that our synchronization scheme only ensures that rendering of the current frame begins simultaneously, not that the vertical retrace or OpenGL buffer swap of the rendering nodes is synchronized. To address these issues we respectively use the genlock and framelock features

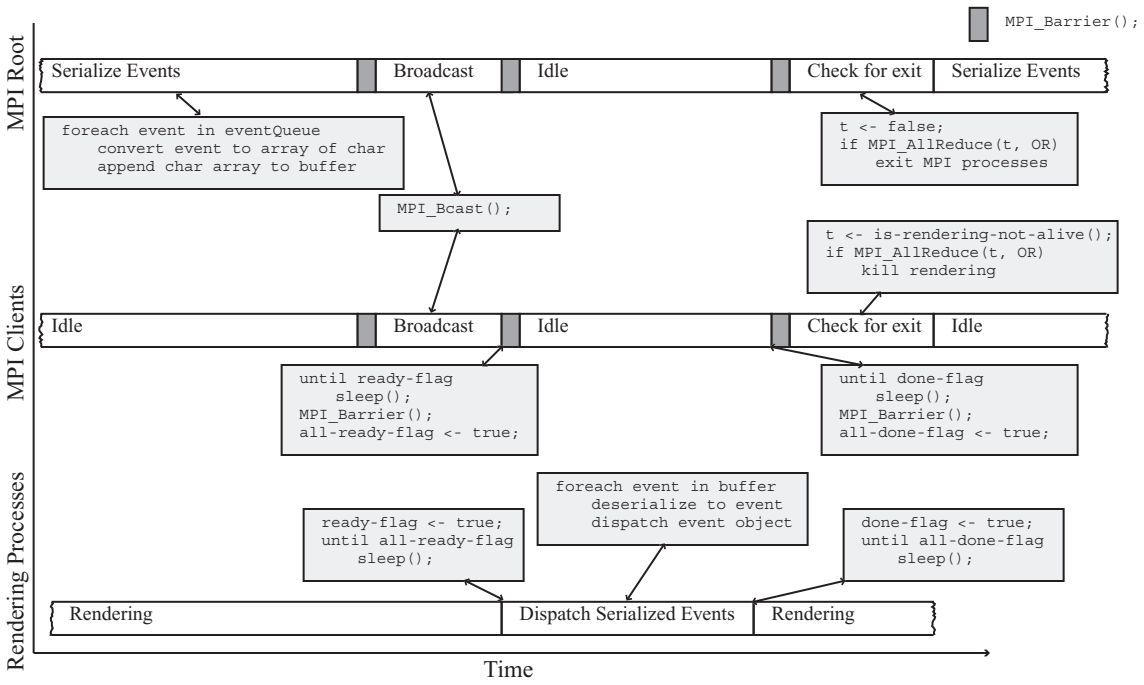

Figure 3. The serialization and broadcast of events occurs in parallel to the rendering of the scene. After rendering completes, handshaking using MPI barriers and IPC Boolean flags forces the rendering process to dispatch waiting events before the MPI Root is allowed to begin serializing and broadcasting again. Time is not to scale, as we assume rendering always takes much longer than event serialization and broadcast.

of our 3DLabs Wildcat III^{TM} 6210 graphics hardware [14].

3.5 Issues

Because our guarantee of image coherence is predicated on the fact that application state does not change based on anything other than the event listeners, several special cases are considered that could reasonably be ignored in shared-memory systems. As is discussed in [12] [13], using duplicated application instances requires that each instance have an identical conception of time and generate random numbers based on the same seed. We introduce a new event into our framework to explicitly deal with time. The root MPI node generates a time event at the beginning of each event broadcast. It is crucial for synchronization that any update to application state based on time use this event's information rather than querying an individual node's internal clock.

An additional complication we encountered in our transition was the proper management of processes and memory across the cluster nodes. We expect our cluster to be highly reliable, and we therefore did not attempt to robustly handle random failures as Syzygy does; after six months of use, this expectation has not introduced any problems. If a rendering process unexpectedly dies, we, at the very least, want every other rendering process to exit in addition to the MPI processes. Also, if the user wants to quit, the same must occur. After each broadcast the client MPI code checks to see if its associated rendering process is still alive. Using an MPI AllReduce with an "or" operation, we communicate this condition to all nodes. If the result of this global "or" is true, then each node signals its rendering process to exit.

4 Results

We employed our approach to adapt several existing VR research applications written for an SGI^{TM} Onyx2 IR 4-pipe machine to our new cluster of four Intel^{TM}-based machines with 3DLabs^{TM} Wildcat graphics hardware. There was no noticeable slowdown in running these examples on one node or all nodes of the cluster. In practice, the rendering of our scenes always takes much longer than the event broadcast, and because these are performed in parallel, the overhead of the cluster is only the deserialization and redispatch of the events, which is very fast.

4.1 Fluid Flow Visualization

Our driving application for adapting an existing VR program to cluster-based rendering was an air flow visualization around the wings of a bat. Many particles are added, deleted, and advected each frame, thus the master/slave model is ideal since it would be unrealistic to distribute geometric updates and maintain an interactive framerate. Instead, updates to the geometry on each rendering node are based on the elapsed simulation time. Thus, the effectiveness of the time event was crucial. We observed no discontinuity as particles flowed across different display tiles.

4.2 WorldToolkit™ Applications

We had previously integrated our event model into a WorldToolkit™ [16] shell such that we could utilize WTK for rendering and window creation. At the same time, we were still able to use our same event-based methodology. We were able to run duplicated instances of these WTK applications and use the event stream to update their state synchronously.

4.3 Trauma Surgery

A point-cloud dataset consisting of approximately 100,000 points had been collected to immersively visualize the hands of a surgeon practicing on a dummy. The size of the data made distribution of geometric changes infeasible. Synchronization relied heavily on the timing events which enabled each node to load the correct timestep of the data. Again we were unable to detect any discrepancy between nodes.

4.4 Glut Event Model

To prove the generality of our approach beyond our in-house event implementation, we adapted a simple Glut [5] desktop demo to run synchronously across five nodes. Glut employs an event-based model for desktop user-input. In Glut the programmer handles events by registering function callbacks that can update the application state. This adaptation was achieved by adding event serialization to the event callback functions on the master node, distributing the serialized event data via MPI, and dispatching the events on each client by invoking the same callbacks as on the master, but without the serialization additions. Our approach proved to be simple and effective, requiring the addition of approximately 50 lines of code and not showing any visually perceivable discrepancy between application state across the nodes.

5 Conclusions and Future Work

We have presented a framework for adapting an event-based VR application running in a shared-memory environment to a commodity cluster rendering system. Nevertheless, this framework could certainly be employed in new cluster-based implementations. Unlike VR Juggler or Syzygy, we do not present a full-featured VR system. Though we chose to work on in-house software for our initial attempts, given the open-source and extensible nature of the aforementioned implementations, future work should be considered in integrating an event-based application model into a more complete VR system. Other future work will include testing the scalability of our event distribution on much larger clusters.

Acknowledgements

This work was supported by LLNL Research Subcontract No. B527302 and NSF (CCR-0086065). This work used UNC's VRPN library, which is supported by the NIH National Research Resource in Molecular Graphics and Microscopy at the University of North Carolina at Chapel Hill.

References

[1] Allard, J., Gouranton, V., Lecointre, L., Melin, E., and Melin, E. "Net Juggler and SoftGenLock: Running VR Juggler with Active Stereo and Multiple Displays on a Commodity Component Cluster," in *Proceedings of IEEE Virtual Reality 2002*, IEEE, March 2002.

[2] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. "VR Juggler: A Virtual Platform for Virtual Reality Application Development," in *Proceedings of IEEE Virtual Reality 2001*, IEEE, March 2001, pp. 89-96.

[3] Cruz-Neira, C., Sandin, D., and DeFanti, T. "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," in *Proceedings of SIGGRAPH '93*, ACM Press, New York, August 1993, pp. 135-142.

[4] Foley, J., van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principles and Practice*. Addison-Wesley, New York, 1990.

[5] Glut: OpenGL Utility Toolkit, http://www.sgi.com/software/opengl/glut.html

[6] Hesina, G., Schmalstieg, D., Fuhrmann, A., and Purgathofer, W., "Distributed Open Inventor: A Practical

Approach to Distributed 3D Graphics," in *Proceedings of the ACM symposium on Virtual Reality Software and Technology*. ACM Press, New York, 1999, pp. 74-81.

[7] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P., and Klosowski, J. "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters," in *Proceedings of SIGGRAPH '02*, ACM Press, New York, August 2002, pp. 693-702.

[8] Message Passing Interface (MPI). http://www-unix.mcs.anl.gov/mpi/

[9] Myrinet Performance. http://www.myricom.com/myrinet/performance/

[10] Olson, E. "Cluster Juggler: PC Cluster Virtual Reality," Masters Thesis, Iowa State University, 2002.

[11] Russell, T., Hudson, T., Seeger, A., Weber, H., Juliano, J., and Helser, A. "VRPN: a Device-independent, Network-Transparent VR Peripheral System," in *Proceedings of the ACM symposium on Virtual Reality Software and Technology*. ACM Press, New York, 2001, pp. 55-61.

[12] Schaeffer, B. and Goudeseune, C. "Syzygy: Native PC Cluster VR," in *Proceedings of IEEE Virtual Reality 2003*, IEEE, March 2003, pp. 15-22.

[13] Staadt, O., Walker, J., Nuber, C., and Hamann, B. "A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering," in *Proceedings of the Workshop on Virtual Environments 2003*, ACM Press, New York, May 2003, pp. 261-270.

[14] 3DLabs: Genlock and Multiview. http://www.3dlabs.com/product/technology/Multiview.htm

[15] Voss, G., Behr, J., Reiners, D., and Roth, M. "A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters," in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, Eurographics Association, Aire-la-Ville, Switzerland, September 2002, pp. 33-38.

[16] WorldToolkit. http://www.sense8.com/products/