Combining a Multithreaded Scene Graph System with a Tiled Display Environment

E. Wes Bethel[a]
R3vis Corporation

Randall Frank[b]
Lawrence Livermore National Laboratory

J. Dean Brederson[c]
Scientific Computing and Imaging Institute
University of Utah

ABSTRACT

This paper highlights the technical challenges of creating an application that combines a multithreaded scene graph system for rendering with a software environment for management of tiled display environments. Scene graph systems simplify and streamline graphics applications by providing data management and rendering services. Software for tiled display environments simplifies use of multiple displays by performing such tasks as opening windows on displays, gathering and processing input device events, and orchestrating the execution of application rendering code. We explore technical issues in the context of an application that integrates both software tools, and formulate suggestions for the future development of such systems.

1. BACKGROUND

A ***tiled display environment*** is a single logical display composed of one more physical display devices. We use the term tiled display environment, or tiled display, to refer to surround-style systems, like the CAVE [1], and high-resolution projector arrays, such as the Powerwall [2]. In this discussion, we use the term ***host environment*** to refer to the software that performs tiled display device management. During our work, we explored the use of two such host environments, CAVELib[1] and the Virtual Display Library (VDL) [3]. In this study, we discuss both systems in general terms, and present a case study using VDL.

CAVELib and VDL share a similar design that make it easy to write reasonably portable applications that use multiple displays. Both use a configuration file that defines a mapping from logical displays to physical display devices. Both environments perform the mundane tasks of opening suitable drawables on the displays, and initializing OpenGL. When a frame needs to be rendered, the host environment computes the view transformation that corresponds to a given display, and invokes the application "draw function" to perform rendering. The host environment typically invokes all application draw functions in parallel in order to maximize frame rate, and synchronizes rendering across the tiled display.

A ***scene graph system*** refers to a set of data structures and associated operations that handle data management and rendering for graphics applications. Scene graph systems vary in features, types of primitives supported and in deployment environment. Some scene graph systems provide management of events, rendering contexts, and windows as an intrinsic (and sometimes inseparable) part of the system, while others rely on the application to perform these non-rendering tasks.

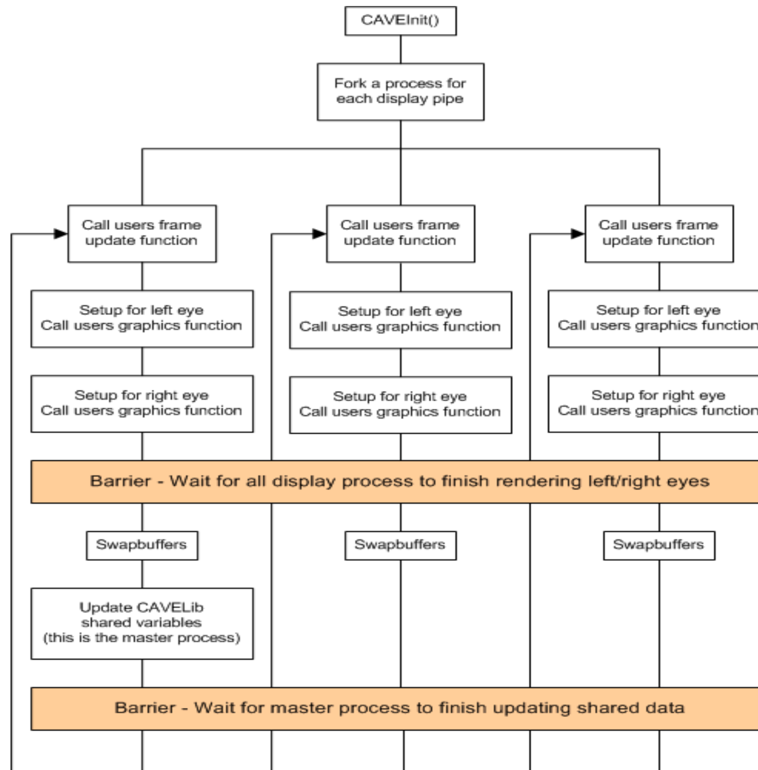
There have been similar efforts in the past to use scene graph toolkits for rendering management in tiled display environments. One effort [4] describes combining Performer [5] and CAVELib. These applications can be best described as Performer applications that use CAVELib to gather input device events. These applications do not use CAVELib to compute the view transformation, nor use CAVELib to invoke application rendering code. All view transformations are computed within the Performer scene graph hierarchy. The flowchart in Figure 1, from the CAVELib users manual located on the VRCO website, shows the logic of an application built using CAVElib. The flowchart depicts logic that includes global application initialization, calculation of per-view or per-display settings, invocation of application draw code, and post-rendering synchronization.

1. CAVELib is a commercial software package distributed by VRCO, http://www.vrco.com/.

a) E. Wes Bethel, R3vis Corporation, *wbethel@r3vis.com*
b) Randall Frank, Lawrence Livermore National Laboratory, *rjfrank@llnl.gov*
c) J. Dean Brederson, Scientific Computing and Imaging Institute, University of Utah, *jdb@cs.utah.edu*

A more recent effort [6] compares the use of Java3D [7] and Vega[2] as the underlying scene graph technologies for deployment in a tiled display environment. Frame rates for Java3D were reported to be an order of magnitude slower than the native mode Vega, but the Java3D application took substantially less time to develop. That study does not use an external host environment, such as VDL or CAVELib, but instead employs the inherent multiple-display capabilities of Java3D to perform tiled display management.

FIGURE 1. CAVELib Host Environment Application Architecture



Our goal is to use the host environment and scene graph systems to their fullest potential. The host environment will manage displays, gather input device events, and compute the view transformation for each display. As a side benefit, host environments can serve as a rudimentary processing framework for parallel applications, including the framework for synchronization amongst multiple rendering threads. The scene graph system will serve as a repository for graphics data and will perform rendering when invoked by the host environment. For the scene graph toolkit in this experiment, we used OpenRM Scene Graph, an open source, native mode, thread-safe, and multistage-parallel scene graph system ([8],[9]). Native implementations, as opposed to a platform-neutral bytecode and runtime environment, allow for reuse of existing custom computational components. Such components often provide a highly specialized feature not present in general purpose systems, and often are the result of substantial investment.

2. Implementation Details

In the discussion that follows, we focus on the interface between OpenRM and the host environment. The interfaces consist of resources that are shared between the two technologies. During our development, we encountered obstacles when combining host environments with the scene graph system into a single application. The obstacles stem from assumptions made by both concerning resource usage. We present an example that makes extensive use of features present in both systems.

2. Vega is a commercial software product from MultiGen-Paradigm used for creating visual simulation applications. See http://www.multigen.com/products/vega1.htm.

2.1 OpenRM and Host Environments

To begin our development, we first identified the "interfaces" between the host environment and OpenRM. We use the term interface in the sense of resources that are shared between the host environment and OpenRM, as opposed to API-style interfaces. These include the OpenGL rendering context, the drawable, the OpenGL transformation stack, and any special post-rendering activities, such as swapping buffers. These interfaces are summarized in Table 1.

Table 1. Interfaces Between Host Environments and OpenRM

Resource	Use
OpenGL context	The host environment creates and binds (make current) an appropriate context.
Drawable	The OpenGL-capable drawable (window) is created and mapped onto the physical display device by the host environment.
Transformation Stack	The host environment computes the view transformation that reflects the correct view transformation for a single viewer looking "through" a "window" on each display device.
Post-render actions	The host environment will swap buffers for double-buffered rendering contexts and may try synchronize all buffer swaps to update all display devices simultaneously.

The core of the integration problems stem from two design decisions central to host environments. The first is that the host environments make calls to the graphics API at all. Second, the calls that they need to make are near the start and at the end of the rendering process: to compute and load the view transformation, to invoke application rendering code, then to perform a buffer swap. The general tendency is that the host environments choose to control the rendering threads, and by proxy, the graphics contexts. The VDL and CAVELib designs follow this basic logic.

2.2 OpenGL Rendering Contexts

The OpenGL rendering context is the complete set of OpenGL state variables. The set of state variables includes a large amount of information, ranging from the size of the viewport, the depth of the framebuffer, the pixel thickness of lines, pixel color transfer modes, transformation matrix stack, textures, and so forth. When an application initializes the host environment, the host environment typically initializes each of the individual display devices. For each display device, the host environment locates and initializes a suitable OpenGL context, then opens a drawable on the display device.

According to the OpenGL ARB specification, an OpenGL rendering context can be active in only one thread at a time [10]. This means that one OpenGL context can be used by multiple threads, but only in strict round-robin fashion. To achieve maximum frame rates, most host environments open one OpenGL rendering context per display and then invoke application draw callbacks in parallel. In the portion of the OpenGL API that pertains to context management, there is a subroutine call that is used to either assert or relinquish ownership of the OpenGL context. When a thread needs to use the OpenGL context, it asserts ownership of the context using the appropriate subroutine call. Before some other thread can assert ownership of the context for use, the first thread must explicitly detach from the OpenGL context. Host environments do not perform this explicit detachment step because of fundamental design assumptions.

As part of the initialization process, the host environment obtains a suitable OpenGL rendering context from the system. When it is time to draw a frame, the host environment computes the view transformation for each display, and loads this matrix onto the transformation stack prior to invoking the application rendering callback. The application rendering callback then proceeds by issuing OpenGL rendering calls. Upon completion, execution control is returned to the host environment. Ownership of the rendering context is required to perform the swapbuffers function after rendering has been completed. Buffer swapping appears in the immutable lineup of tasks performed by the host environment. This fundamental host environment design implies that the rendering callback must exist in the same thread of execution as the host environment. This is not an optimal arrangement for high performance rendering applications that seek to keep the graphics pipeline full at all times.

2.3 Matrix Stack Management

Most scene graph systems are "uncooperative" in their approach to OpenGL resource management, including management of the matrix stack. When these systems render a frame, it is assumed that the scene graph is a completely self-contained descrip-

tion of geometry and other viewpoints. These systems often have a prescribed set of rules that must be followed in terms of scene graph organization. The rules often dictate that a view specification, such as a camera, is located at or near the root node of the scene graph. When the scene graph system begins to render, the OpenGL matrix stack is first initialized with the view parameters specified as part of the scene graph. In order to successfully use a scene graph system with a host environment, the scene graph system must be more cooperative in use and management of OpenGL resources. For example, the scene graph system might take into account that the contents of the matrix stack have been preloaded with transformations that should be used during rendering.

At the time we began to create investigatory applications that use OpenRM and host environments, we found that OpenRM required modifications in order to foster harmonious interaction with host environments, such as sharing transformations on the matrix stack. We added to OpenRM a mode of operation that can be considered adequately cooperative with respect to matrix stage management. The RMpipe object in OpenRM now has a two-state attribute that controls how the OpenGL matrix stack is initialized during rendering. In one state, OpenRM will initialize the matrix stack by loading the identity matrix prior to rendering and accumulating transformations in the scene graph. In the other state, the matrix stack is not initialized: it is assumed that the caller (the host environment) has pre-loaded values on the matrix stack, and that any transformations contained in the scene graph are concatenated to the values on the matrix stack. The developer must request the latter mode in order to use OpenRM with a host environment that preloads the view transformation on the matrix stack, as the former mode is the default when the RMpipe is created.

2.4 Who Owns the OpenGL Context?

Sharing an OpenGL context amongst multiple threads of execution can be a tricky proposition. In order to create an application in which one thread creates the context and then another thread draws to that context, special care must be exercised. The thread that creates the initial context must "yield" the context before another thread binds to it[3]. The host environment is a parallel application, and OpenRM is also multithreaded. In order to have both software systems harmoniously coexist, we need a clearly defined protocol concerning ownership of the OpenGL context.

OpenRM supports several modes of multistage, multithreaded rendering. In one such mode, one execution thread performs view dependent processing, such as frustum culling and level-of-detail model switching. The other thread performs render operations, and dispatches primitives not culled by the view stage to the graphics pipeline. Since the host environments in this study do not yield the OpenGL context prior to invoking the render callback, having the rendering work performed by a separate thread results in errors that stem from contention for the OpenGL context. However, we would like to benefit from multistage processing during rendering and use the host environment framework for display management.

In order to solve this problem, we implemented a new processing mode to the RMpipe object. This new mode places the view dependent processing in a separate execution thread, while the render processing remains in the same thread as the host environment. This is possible because OpenRM's view dependent processing does not require access to the OpenGL context: it only needs the initial matrices from the OpenGL matrix stack, which can be provided internally. This multistage rendering model introduces a one-frame latency typical of multistage rendering architectures. The view transformation specified by the host environment at frame N is not used by the application rendering callback until frame N+1.

2.5 Example: A Flyover and VDL

For this example, our goal was to combine a terrain flyover demonstration program with VDL, and to display the results in a tiled display environment. The scene graph created by the demonstration program contains a 3D camera: the position and orientation of the camera changes over time. The challenge was to combine VDL, which has its own view model, with the scene graph created by the application, which also includes a view model.

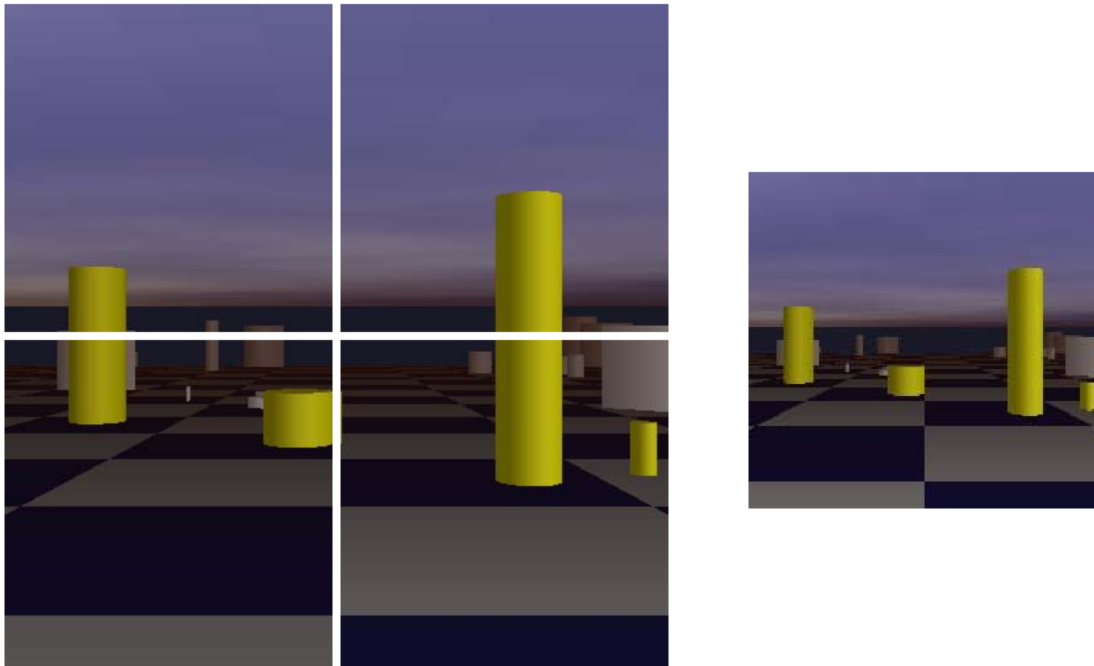
VDL's view transformation consists of two components: a frustum and a shear. The shear transformation aligns the frustum to a particular display window. When we specify an identify transformation for the frustum component, VDL preloads only the shear transformation onto the matrix stack. When the OpenRM rendering callback is invoked for each display device, OpenRM's view transformation is multiplied with the shear transformation on the matrix stack, thus producing the correct

3. OpenGL literature says nothing about "yielding" the context. We adopt use of this term in this context as a clarification.

view transformation. This design is somewhat more flexible than the CAVELib model, which contains an architectural bias that sets the viewpoint as well as the projection. Our flyover example uses extensive view-dependent operations to perform frustum culling and distance-based level-of-detail model switching [11], and for this reason, requires careful attention to the separation of model, view and projection transformations.

This flyover application uses OpenRM for data management and rendering and VDL for tiled display management. In Figure 2, the images on the left are purposefully separated to emphasize that the VDL display in this example is composed of four physically distinct windows. Each of these four images was created by a separate display device in a 2x2 array of display devices. The image on the right is the application "navigation" window. The viewer/3D camera is manipulated by mouse motions in this navigation window.

FIGURE 2. VDL/OpenRM Flyover Example



The shear transformation for each of the four VDL windows shown is first computed by VDL, and then the view transformation is computed and the scene is rendered by OpenRM. Performing view transformations within the scene graph is crucial for operations that occur in eye coordinates to be rendered correctly. Such operations include clipping planes, fogging, and certain types of texturing operations (our example demonstrates OpenGL fog).

During testing at Lawrence Livermore National Laboratory (LLNL), we ran this application using multistage, multithreaded render processing. OpenRM handles view processing in one thread, while render processing occurs in the same thread as VDL due to the vagaries of OpenGL rendering context ownership. This example demonstrates the use of view frustum culling as well as level-of-detail model switching. View frustum culling was independently applied to each of the four display channels, and only those scene graph objects that passed the view frustum cull test were passed on for rendering.

In the images shown in Figure 2, high resolution cylinders closer to the eyepoint are displayed in yellow, while low resolution cylinders further away from the eyepoint are displayed in white. This type of model switching occurs using a view-stage traversal callback that performs distance-based, level-of-detail model selection.

3. Conclusions and Future Work

Host environments serve to simplify resource management in tiled display environments. They typically assume full responsibility for initialization, which includes opening windows on each of the display devices. They perform per-frame tasks that

include computing the view and projection transformation for each display device, and they invoke user draw code once the view transformation has been loaded onto the OpenGL transformation stack. They synchronize amongst multiple display threads or processes at the end of the frame.

Scene graph systems serve the primary purpose of providing graphics data management services, and the means to render managed data. Many scene graph systems provide the support needed to initialize the graphics display environment, as well. During the process of rendering, scene graph systems will make changes to the OpenGL context. Such changes are necessary to implement changes in rendering parameters dictated by scene graph data. Examples include light sources, material properties, textures, and so forth.

Combining host environment and scene graph technologies is a logical choice for developers, for they are complementary technologies. In this paper, we have focused on the use of two specific host environments, VDL and CAVElib, and one specific scene graph API, OpenRM. The lessons we have learned are more broadly applicable, especially to future development of both types of technologies. Central to these lessons is the notion of management of the OpenGL rendering context.

As an example of how context mismanagement can adversely affect applications, we observe that both CAVELib and VDL store the view transformation on the OpenGL matrix stack. Due to the rules governing ownership of the OpenGL rendering context, use of the matrix stack as a communication mechanism precludes the use of a draw process that runs in a separate execution thread. In the future, host environments should consider approaches that allow for more flexible use of such constrained resources.

We propose that future host environments need not use the OpenGL rendering context at all. Host environments serve two primary purposes. First, they compute per-frame view transformations for each display device, and perform synchronization of multiple rendering processes. Second, and of less concern to this discussion, they make data from input devices available to applications. A rich set of host environment callbacks for initialization and post-rendering synchronization can be used to avoid direct use and management of the OpenGL context, and a default set of callbacks can be provided that serves to provide base-level functionality for those applications that do not require context sharing between host environment and scene graph rendering systems.

The work described was performed on SMP systems with multiple graphics pipes. Recent advances in parallel graphics APIs [12] facilitate use of distributed memory platforms for tiled display rendering by providing an abstraction of a graphics context which removes several noted OpenGL constraints (e.g. parallel primitive submission). In the future, we will explore deployment of host environments and scene graph toolkits on clusters.

4. Acknowledgement

This work was funded by the U.S. Department of Energy, Office of High Performance Computing Research in the Office of Science, through the Small Business Innovation Research (SBIR) program under contract number DE-FG03-00ER83083 and Lawrence Livermore National Laboratory under No. W-7405-Eng-48 (UCRL-JC-143204).

5. Bibliography

[1] C. Cruz-Niera, D. Sandin, T. DeFanti, Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE. In Computer Graphics, *Proceedings of SIGGRAPH 93*, August 1993.

[2] http://www.lcse.umn.edu/research/powerwall/powerwall.html

[3] D. Schikore, R. Fischer, R. Frank, R. Gaunt, J. Hobson, and B. Whitlock, “High-resolution Multi-projector Display Walls and Applications”, IEEE Computer Graphics and Applications, Vol 20, 4:38-44, Jul/Aug, 2000.

[4] D. Pape, pfCAVE CAVE/Performer Library (CAVELib Version 2.6), http://www.evl.uic.edu/pape/CAVE/prog/pfCAVE.manual.html.

[5] J. Rohlf and J. Helman, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Computer Graphics (Proc. ACM Siggraph 94)*, pp 381-394, August 1994.

[6] B. Christianson and A. Kimsey, Comparison of Java3D in a Virtual Environment Enclosure (2000). Master's Thesis, Naval Postgraduate School, Monetery, CA, March 2000.

[7] H. Sowrizal, K. Rushforth and M. Deering. The Java3D API Specification. Addison-Wesley, Reading MA, 1998.

[8] W. Bethel and J. D. Brederson, Hierarchical Parallelism in a Scene Graph. Submitted to Graphics Interface 2002.

[9] OpenRM Scene Graph, http://openrm.sourceforge.net/

[10] OpenGL 1.2 Specification, ftp://ftp.sgi.com/opengl/doc/opengl1.2/opengl1.2.1.pdf

[11] J. Clark, Hierarchical Geometric Models for Visible Surface Algorithms. Communications of the ACM, 19, 10, pp 547-554. October 1976.

[12] G. Humphreys, I. Buck, M. Eldridge and P. Hanrahan, Distributed Rendering for Scalable Displays. In *Proceedings of Supercomputing '00.* Dallas, Texas, November 2000.