A General Architecture for Distributed VR Interfaces

Thomas Batkiewicz
Iowa State University

Abhishek Seth
Iowa State University

Bryan Walter
Lockheed Martin

Jared Knutzon
Lockheed Martin

James H. Oliver
Iowa State University, oliver@iastate.edu

Follow this and additional works at: https://lib.dr.iastate.edu/me_conf
Part of the Computer-Aided Engineering and Design Commons, and the Graphics and Human Computer Interfaces Commons

Recommended Citation

This Conference Proceeding is brought to you for free and open access by the Mechanical Engineering at Iowa State University Digital Repository. It has been accepted for inclusion in Mechanical Engineering Conference Presentations, Papers, and Proceedings by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
A General Architecture for Distributed VR Interfaces

Abstract
This paper describes a software architecture useful in creating new virtual reality interfaces across a clustered network of computers. The architecture takes advantage of a concept common to desktop application development, the event loop, and provides a method by which it can be shared effectively between nodes in a cluster. This method improves upon the common replication approach to cluster synchronization by creating a single, simple object to share, as well as by supporting node specialization in a generic cluster. The method is designed to be easily added to existing applications as well as speed the development of virtual reality interface prototypes. An example application is discussed to demonstrate the current capabilities and limitations of the architecture. Future implications and improvements are also discussed.

Keywords
Virtual Reality Applications Center

Disciplines
Computer-Aided Engineering and Design | Graphics and Human Computer Interfaces

Comments

This conference proceeding is available at Iowa State University Digital Repository: https://lib.dr.iastate.edu/me_conf/109
A GENERAL ARCHITECTURE FOR DISTRIBUTED VR INTERFACES

Thomas Batkiewicz  
Department of Mechanical Engineering  
Virtual Reality Applications Center  
Iowa State University  
Ames, IA 50011  
twitz@iastate.edu

Bryan Walter  
Senior Research Engineer  
Lockheed Martin  
bryan.walter@lmco.com

Abhishek Seth  
Department of Mechanical Engineering  
Virtual Reality Applications Center  
Iowa State University  
Ames, IA 50011  
abhiseth@vrac.iastate.edu

Jared Knutzon  
Senior Research Engineer  
Lockheed Martin  
jared.knutzon@lmco.com

James H. Oliver  
Department of Mechanical Engineering  
Virtual Reality Applications Center  
Iowa State University  
Ames, IA 50011  
oliver@iastate.edu

ABSTRACT

This paper describes a software architecture useful in creating new virtual reality interfaces across a clustered network of computers. The architecture takes advantage of a concept common to desktop application development, the event loop, and provides a method by which it can be shared effectively between nodes in a cluster. This method improves upon the common replication approach to cluster synchronization by creating a single, simple object to share, as well as by supporting node specialization in a generic cluster. The method is designed to be easily added to existing applications as well as speed the development of virtual reality interface prototypes. An example application is discussed to demonstrate the current capabilities and limitations of the architecture. Future implications and improvements are also discussed.

Keywords: Virtual Reality, Human Computer Interaction, Networked Virtual Environments.

1.0 INTRODUCTION

Driven by the motivation to increase the resolution and field of view available to users, many modern virtual reality displays now operate on computer clusters rather than mainframe architectures. For both multiple-wall projection systems and tiled displays, each node in the cluster renders a single portion of the entire virtual reality display. These computer clusters have many advantages; a low cost that is driven by the use of commodity hardware, as well as a modular and scalable nature. [1]

However, these advantages come at the expense of the software complexity required. Synchronizing multiple displays is necessary for the immersive benefits of virtual reality to be fully realized, and it must be done for every rendered frame to prevent visual artifacts such as tearing. The contribution of the research presented in this paper is a new method for synchronizing these displays that allows for specialized nodes and provides a common protocol for inter-application communication.

2.0 BACKGROUND

Several frameworks have been developed to aid in the application development process for clustered VR applications. VRJuggler [2] provides an operating system abstraction layer that simplifies the process of creating cross-platform virtual reality applications. VRJuggler handles the management of the
view frusta for the nodes in the cluster; each scene is duplicated on all nodes such that only the perspective needs to be modified to render a portion of a multi-wall display. It also provides a basic method to synchronize objects by allowing developers to write a special serialization function that VRJuggler can use to distribute an object's state information to all the nodes in the cluster.

CAVELib [3] is another VR framework that has similar functionality to VRJuggler. By explicitly sharing certain data (device inputs) across the cluster, the state of all the nodes can be kept synchronized. Additionally, other data can be transferred between nodes in the cluster. Both of these frameworks provide a base level of abstraction for object serialization and data exchange between nodes in the cluster.

Several visualization scene graphs also support clustered rendering for multiple-projection displays. OpenSG [4], an open source graphics library provides a synchronization scheme at the rendering level instead of the application level described earlier. Using a master/slave paradigm, changes to the scene made on the primary rendering node are propagated to all the other nodes in the cluster [4]. Application developers can also specify other non-scene graph data types to be synchronized by OpenSG using field containers.

Syzygy [5] is a framework that implements both of the above approaches. Data can be exchanged either at the rendering level or application level using Syzygy. Similarly to VRJuggler, Syzygy provides a level of abstraction from the operating system and device drivers. However, using a technique similar to the scene graph approach discussed above, Syzygy can transfer information about the graphical objects to the nodes in the cluster [5].

Avocado [6], based on OpenGL Performer graphics library [7], is another distributed VR framework. Avocado provides an object field-container approach to data synchronization, as well as the device-abstraction layers. This approach is similar to the serialization approach; by packaging parts of an object's data into a chunk of memory, the framework can distribute changes to all the nodes in the cluster.

DIVE [8] is a framework that builds upon an abstract world model, and uses group-based multicast to handle network communication. With this peer-based approach, DIVE also utilizes proxy servers to handle communication between various applications that may be geographically distributed. Additionally, objects in the world model are not synchronized by transmitting the whole object at once; small updates are instead used, similar to passing changes in field containers above.

DEVA3 [9] is another framework, one that builds heavily on the use of sub-clustering objects, in which general entity behaviors are specified, with individual objects exhibiting a mixture of general and unique behaviors, some of which vary at runtime. Every node maintains a copy of all entities in the environment. Every entity is assigned a unique ID and modifications to an entity's behavior can be sent across the network in a peer-to-peer fashion using TCP/IP.

Many of these frameworks and solutions operate at a very simple level. Data is synchronized on the level of device inputs, or serialized objects/fields that are simply replicated across all the machines in the cluster. The use of such proxy objects only works if all nodes in the cluster are running exact duplicate applications, which may be difficult if cluster nodes serve unique purposes or do not have identical hardware. While the proxy objects form a basic foundation for intra-cluster communication, no attempt has been made to exchange data in a format that is meaningful at an application or user-interface level, which is advantageous when linking applications built using different languages or toolkits.

In this paper, we present a new framework design that facilitates distributed VR interfaces, and aims to provide an application-level exchange of data. A commonly used paradigm in traditional desktop application development, the event-loop, is often used both to transmit device interactions (i.e. mouse clicks) as well as application-specific information. An event-driven approach is given in Design Patterns, which splits a system into a series of subcomponents that communicate through specialized commands [10]. Building on this existing idea, we create a basic event loop structure, consisting of classes that interact to create a list of events that can be synchronized across the cluster. Using this method, the only object that needs to be serialized is this list of changes.

This gives us the advantage of allowing specialized nodes in the cluster underlying a virtual environment, and additionally makes networking in additional interfaces as simple and transparent as adding nodes to the cluster.

First, an example VR application is profiled in section 3, and gives a set of problems common to many VR applications with regards to cluster data synchronization. Section 4 provides a detailed description of the architecture used to solve these issues, with three C++ classes that form the backbone of the architecture. Section 5 covers an expansion on the application profiled in section 3, which was enabled by the use of this architecture over a wireless network. Finally, the impact of this architecture and future improvements are outlined in section 6.

3.0 THE VIRTUAL BATTLESPACE APPLICATION

In 2000, a research team at Iowa State University's Virtual Reality Applications Center (VRAC) began work with the Air Force Research Lab's Human Effectiveness Directorate and the Iowa National Guard's 133rd Air Control Squadron to develop an immersive VR system for distributed mission training called the Virtual Battlespace. The Virtual Battlespace is an immersive virtual battlefield environment that provides users with greater context and awareness of the status of the units under their control as well as the overall mission. By integrating UAV video feeds into the virtual environment can provide up-to-date access of the latest real time information from the vehicles. This virtual world, constructed from a mix of a priori information and real time sensor feeds, provides context to the operator. The result is a mixed reality system in which the real world video streams augment a dynamically
constructed virtual world. Using real world data to augment the virtual world is an inversion of the more typical paradigm of augmented and mixed reality where virtual information is used to enhance real world data and imagery.

The Virtual Battlespace integrates information about tracks, targets, sensors and threats into an interactive virtual reality environment that consolidates the available information about the Battlespace into a single coherent picture that can be viewed from multiple perspectives and scales [11]. Visualizing engagements in this way can be useful in a wide variety of contexts including historical mission review, mission planning, pre-briefing, post-briefing and live observation of distributed mission training scenarios.

Our work with the Virtual Battlespace inspired us to begin creating a cohesive virtual world representing the status of real time and a priori information about an engagement [12]. Figure 1 shows the Virtual Battlespace environment displayed on a six-walled stereo projection system, the C6 Cave, at VRAC. The C6 Cave is powered by a cluster of 48 commodity HP PCs running Linux. Each PC is responsible for rendering 1/8th of a wall, and is equipped with two graphics cards, one for each eye in stereoscopic mode.

![Figure 1. Battlespace Environment](image)

The Virtual Battlespace is designed to aid the operator in understanding the complex interactions of units on the battlefield and in making effective decisions. There are many different streams of information that provide support for battlefield decision making. Some of these include radar and other sensor feeds, satellite imagery, communication links and weapons information. The Virtual Battlespace is designed to fuse multiple information streams and make them centrally available to command and control personnel. The goal of this comprehensive presentation is to improve a user's ability to make effective and intelligent decisions [13, 14].

A stream of unit data to be displayed by the Virtual Battlespace can be generated by several diverse sources such as a simulated force generator like Joint Semi-Autonomous Forces (JSAF), or a live sensor such as a radar feed. The streams need not have a common format. JSAF created and monitored the bulk of the non-swarm battle participants in this research via a pre-recorded scenario. JSAF uses the Distributed Interactive Simulation (DIS) military communication protocol, and as a result, does not represent the most sophisticated military simulation software [15]. However, DIS is a sufficiently good format to create the proof-of-concept prototype of the Virtual Battlespace.

In the Virtual Battlespace, data streams are presented graphically to reduce the amount of textual information the commander is required to process. In other research, this reduction was found to allow operators to focus more time on critical decisions [16] [17]. The graphical elements used to display the units (alternatively called entities) generated by the data streams are a major component of the system. They not only portray the physical attributes of entities in the Virtual Battlespace, such as relative position, orientation, status and speed, they also portray derived attributes such as prior paths or sensor and threat ranges. To maintain the system's flexibility with respect to the format of the input streams, the display of the data streams is separated from their management and from the base application, allowing integration of completely new data streams without having to rewrite large portions of the code.

The operator interacts with the Virtual Battlespace and its unit proxies by using a wireless game controller. The controller used in this research was a Logitech Wingman equipped with a D-Pad, two analog sticks, four shoulder buttons (L1, L2, R1, R2), six buttons (A, B, C, X, Y, Z), a throttle slider and a start button. This input device was chosen for the Virtual Battlespace because game controllers are familiar to many people, especially young people. As these people are the likely users of a near-future installation of the Virtual Battlespace, it seemed like the ideal interaction device. The operator uses the controller to navigate and turn on or off graphical information features. An image of the controller is shown in Figure 2.

![Figure 2. Logitech Wingman Controller](image)
The architecture of this application before we applied our method was one common to most shared-memory architectures; objects maintained pointers to other objects and directly called the public methods of these objects to notify them or enact change. For example, when the user is navigating by pressing the joystick forward, the input manager would notify the scene graph to update the proper translation matrix. An example of this architecture is shown in Figure 3.

![Diagram of architecture](image)

**Figure 3. Shared memory architecture**

However, when moving to a distributed architecture, for this framework to stay intact all nodes in the cluster must be identical. This means that all objects must be replicated across the cluster, which precludes one of the benefits of clustering—specialized nodes. All nodes need to have a copy, even a redundant one, to prevent segmentation faults at runtime. Consider the simple case of the sound device; regardless of how many machines comprise the cluster, or how many walls the display has, typically only one set of speakers is needed with a single sound card on only one node.

### 4.0 SYSTEM ARCHITECTURE

The overall architecture follows the model of object-oriented programming, which allows for a modular and scalable design. Subcomponents often derive from a set of basic objects that are used extensively throughout the code.

Further, the architecture employs several open-source libraries that aid in the prototyping required in interface design. Use of these software foundations allows researchers to focus on the critical portions of the interface design and implementation, rather than spending their time building the entire infrastructure needed to support the application. The open-source libraries expose the underlying code which allows them to be modified to fit specific requirements. Existing design patterns were used to derive this architecture [10].

This research platform makes extensive use of object-oriented programming concepts. Objects serve a clear, unique purpose and encapsulate a very specific behavior. The goal was to create a simple, flexible architecture that minimizes object dependence and enables scalable interactions. Composed of three main components, this architecture implements a basic framework for inter-object communication that is easily extended to communicate between objects on remote computers or in different applications.

The first component is the command object which is the message passed between objects without requiring direct knowledge of what objects are loaded. The other components form a two-layer hierarchy, with the top layer being composed of a single object. This top layer is responsible for managing the second layer and synchronizing communication between components on the second layer. The top layer has a pointer to each member on the second layer, but only sees them as instances of a base class that implements the communication protocol. The objects on the second layer all derive from a base class, and are decoupled from one another; no pointers are exchanged among the second layer. The communication between these components is achieved by upwards propagation of the command-class messages to the top layer, where they are redistributed to all the components on the second layer.

### 4.2 Command Object

The command object is central to the distributed nature of the system's architecture. Communication between modules and networked applications relies upon this class. Each instance of a command encapsulates one instruction or update, and contains data relevant to execute that instruction or interpret the update. The command's simple structure allows considerable flexibility in expressing information that must be shared between components. The minimum size for each of these instructions is 16 bytes, which makes it suitable for both intra- and inter-application communication.

Commands include two top-level pieces of information, the directive and the qualifier, which together serve as the message header, and indicate the rest of the command's contents. The directive typically specifies what has changed in the application, and the qualifier is used to provide additional context or designation. These two fields are represented in the code as an integer that draws its value from a globally defined enumeration; a collection of named integer values. This simple underpinning keeps the size of the command structure small; 4 bytes each for the qualifier and the directive, while preserving the ability for user/programmer readability. Further, since the enumeration is a specific type, at compile-time all assignments of the enumeration are validated which can reduce time spent debugging.

Additionally, the command structure contains two dynamically-sized arrays for passing additional types of data—character data and numeric data. Character data is handled in the form of adding strings to the command. Numeric data is held in floating-point form, but integers and boolean values can
be cast into this type as well. In this manner, we are able to pass the common C++ types in a concise format.

4.3 Application Components

In the Battlespace applications, the components on the second-highest level are known as managers. Each manager encapsulates a specific functionality such as handling input devices, managing a scene graph, or updating vehicle dynamics. The managers are instances of C++ classes that can be dynamically loaded or unloaded by the application as needed. To allow communication between managers, each manager shares a common set of functionality by deriving from a base class, ApplicationComponent. This class provides a common interface that is used to share and synchronize updates between the managers, as well as providing a framework of handling the inputs from other managers. Each manager can register its own member functions to be automatically called when a command with a specific directive/qualifier is received through this architecture.

This simplifies the amount of work required for creating new components or modifying existing components. Interdependence between objects is minimized, since pointers to other components on the same layer are not necessary. Effectively, the managers are able to make function calls on other managers via this command communication, without hard-coded linkage between components.

4.4 Component Manager

A higher-level manager is responsible for synchronizing the communication between the other managers. This module, the ComponentManager, is similar to the event-processing system used in most modern GUI libraries. It is responsible for polling individual managers for their updates and disseminating this information to all the other managers. The functional cycle in which this process takes place is called the command loop. During each iteration, the ComponentManager calls the updateState virtual function implemented in each manager and inherited through the ApplicationComponent. This function passes back a list of commands that this manager has created since the last iteration. Once it has completed this process for each subcomponent, the command list is the collection of changes since the last iteration and provides all of the information required to bring other modules into sync.

Once a complete list of commands is assembled from each manager, the processCommands function is called on each manager. The managers can then react to these commands, or ignore them, based on their specific functionality. In this fashion, the managers are not required to know which other managers are loaded; they simply need to be able to process the commands related to their function and put out relevant updates for other managers. The relationship between the ComponentManager and the ApplicationComponents is shown in Figure 4.

![Figure 4. Architecture Hierarchy](image)

4.5 Networking

For the purposes of our research, the main Virtual Battlespace application must run on a distributed cluster of machines. The use of the command structure as the only mode of communication between managers greatly simplifies implementation on a distributed memory architecture as well as networking with other applications. After the command loop collects all the commands from the managers, it must simply share this list with each node in a cluster at a specific time to synchronize it. Since the command structure is so simple and is constructed out of basic types, serializing this structure to be sent across a network is trivial.

Furthermore, the process to network with other applications is greatly simplified. We define a new manager type, the NetworkManager, to encapsulate this behavior. This manager doesn't generate its own commands, nor does it process them; it makes the internal command list available to outside applications, and upon receiving commands from external sources adds them to the list to be processed internally. Since the command structure is already serializable, the NetworkManager can send and receive commands via socket communication.

This NetworkManager object makes it possible for a component on one node to send commands to a remote component as transparently as it would be if the component were co-located within the same node. Additionally, the NetworkManager acts as a filter for remote applications, by sending only the events that match the required "channels" needed by an application (Simulation events, UI events, etc...). This helps minimize the bandwidth required between applications/nodes. The flow of commands between networked applications is shown in Figure 5.
5.0 TASK MANAGER

The Task Manager is a desktop/tablet PC interface to the Battlespace research project that provides interactions and displays for supervisory control of unmanned aerial vehicles. Utilizing a north-up map display, the Task Manager provides a direct-manipulation interface to the units involved in an engagement. Used in two primary modes, the Task Manager can be used either in a planning/review mode that can be used to generate mission scenarios or a live-streaming mode that connects to a live Battlespace simulation via a network connection to edit and update path information on the fly. An image of this application is shown in Figure 6.

![Figure 5. Networking Diagram](image)

![Figure 6. Task Manager Application](image)

The goal of this research is to combine the precision of 2D mouse and pen-based interaction with the increased situational awareness provided by 3D battlefield visualizations like the Battlespace application. Combined use of these interfaces, either by a single operator or a small team of operators with task-specific roles, is proposed to produce a more favorable ratio of operators to units in field operations with superior decision-making capabilities due to the specific nature of the interfaces.

In the planning mode, the user can design an engagement for simulation. Planning can be done for both hostile and friendly troops, since modeling the engagement in its entirety is the primary goal. Adding unit and targets, planning unit paths, and setting time-on-target constraints are the primary user tasks in this mode. Users can manipulate paths by editing waypoint locations and altitudes. In live mode, the same interactions are available since the user is now interacting with a pre-generated mission plan, with the difference that the simulated units will have their behavior updated as changes are made. The user is able to view, edit, and change the engagement by altering waypoint paths, changing time-of-arrival and time-on-target information, and affect the simulation entities in real time.

The direct manipulation interactions in the live mode help compensate for the problems with precise spatial selection and manipulation in the 3D Battlespace application. Also, the use of this application grants the capability for keyboard-based input, whether using a standard keyboard or a virtual keyboard utility on a tablet PC. Furthermore, the ability for this same software package to be used in both mission planning and command and control will lower potential training times and costs since there is only one application for users to be trained with. It is important to note that this application is not what would typically be considered a virtual environment, but it is a crucial interaction device for the users.

The Task Manager is built using wxWidgets, a cross-platform, open source GUI framework for building desktop applications [18]. This set of libraries uses the operating system’s native widget set which means that applications built on it will adapt to whatever environment they are run in [18]. Abstraction away the mouse and keyboard devices, input is handled by mapping specific functions to be called on user-generated events such as mouse-clicks, mouse-motion, and keyboard press/releases. Since wxWidgets already contains this event-handling system, the ComponentManager and BattlespaceComponent classes are not used in the Task Manager. Instead, the incoming Commands from the simulation are merely wrapped inside the existing wxWidgets event class – which allows this application to act just like another node in the cluster. Furthermore, this application also generates Commands, allowing mouse-clicks (or tablet-clicks) to be quickly translated into application-level events that can affect the remote simulation and VR environment.

6.0 CONCLUSIONS & FUTURE WORK

As it stands currently, this architecture is very useful in synchronizing application-level information across either a clustered VR application, between networked applications, or as presented earlier, in both concurrently. Transmitting data at a level higher than that of simple button presses allows development of complex interfaces that can provide both the
immersive effects of VR with precise direct manipulation. This approach is very generalizable; the ApplicationComponent, NetworkManager, and Component Manager objects are completely reusable as is. The only specialization in this architecture lies in the decision of what set of events is required for inter-object communication, and the grouping of events into channels to ease their delivery to remote applications/nodes.

Scalability is an existing issue for this framework. Since the current network implementation is built on top of VRJuggler, we can be assured of its scalability for the VR application’s scalability, however, collaborative use by linking multiple VR systems or interfaces like the Task Manager likely has an upper limit. The use of multicast groups is one potential solution, however ensuring message delivery is crucial for some aspects of these networked interfaces. A hybrid approach, where full-state application data is sent over TCP/IP in a peer-to-peer fashion, with intermittent updates sent via UDP multicast is currently being developed.

Secondly, the architecture as shown is a flat tree structure; the ComponentManager acts as a root and the ApplicationComponents are its children. Supporting a deeper structure where even the ApplicationComponents can have their own children will be helpful in cases where certain objects must still be tightly linked (i.e. pointers to one another), without breaking the overall decoupled nature of the system.

Lastly, this system operates under a polling system; events aren’t processed until the next rendering frame comes around and the ComponentManager requests the list of changes from each component. Obviously, for general purposes this is acceptable, however it is not an optimal solution. The rate at which events are processed is capped at the rate at which the scene draws. Reworking the architecture into a system where the ComponentManager operates in its own thread, and allowing components to post & respond to events immediately is another task that is being examined. Naturally, this will require the pausing of the event loop momentarily for the rendering agent to have a locked state; however this will allow the events to propagate at a rate faster than the application’s frame rate.

7.0 ACKNOWLEDGEMENTS

The authors would like to thank the Air Force Office of Scientific Research (AFOSR) for their support of this project. Further, the following open source software packages were used in the development of the example applications.

1) VRJuggler Virtual Reality Toolkit [2]
2) OpenSG Open Source Portable Scene graph [4]
3) Demeter Terrain Engine [19]
4) wxWidgets GUI API [18]

8.0 REFERENCES

[3] CAVELib API
http://www.vrco.com/
http://www.sgt.com/products/software/performer/
[10] Design Patterns, Erich Gamma, et. al.
J. Knutzon, B. Walter, A. Sannier, & J. Oliver. NATO Conference, August 2003


[18] wxWidgets GUI API
http://www.wxwidgets.org/

[19] Demeter Terrain Engine
http://www.tbgsoftware.com/